Software Transactional Memory Building Blocks

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur
(Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Torvald Riegel
geboren am 1.3.1979 in Dresden

Gutachter:

Prof. Christof Fetzer, PhD
Technische Universität Dresden
Fakultät Informatik

Prof. Pascal Felber, PhD
Université de Neuchâtel
Institut d'informatique

Tag der Verteidigung: 13.5.2013

Dresden im Mai 2013

UNIVERSITÉ DE NEUCHÂTEL

Faculté des sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel - Suisse
Tél: + 41 (0)32 718 2100
E-mail: secretariat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

La Faculté des sciences de l'Université de Neuchâtel autorise l'impression de la présente thèse soutenue par

Monsieur Torvald RIEGEL

Réalisée en cotutelle avec

Technische Universität Dresden, D

Titre:

"Software Transactional Memory Building Blocks"

sur le rapport des membres du jury:

- Prof. Dr Pascal Felber, Université de Neuchâtel, co-directeur de thèse
- Prof. Dr Christof Fetzer, TU Dresden, D, co-directeur de thèse
- Prof. Dr Hermann Härtig, TU Dresden, D
- Dr Michael Hohmutz, Kernkonzept, D
- Prof. Dr Peter Kropf, Université de Neuchâtel
- Prof. Dr Rainer Spallek, TU Dresden, D
- Prof. Dr Heiko Vogler, TU Dresden, D

Neuchâtel, le 21 juin 2013

Le Doyen, Prof. P. Kropf

Software Transactional Memory Building Blocks

by

Torvald Riegel

Thesis submitted to
Technische Universität Dresden / Université de Neuchâtel
for the co-doctorate degree
Doktoringenieur (Dr.-Ing.) / Docteur ès Sciences

Advisers:

Prof. Christof Fetzer, PhD
Technische Universität Dresden
Fakultät Informatik

Prof. Pascal Felber, PhD
Université de Neuchâtel
Institut d'informatique

Submitted on: January 17, 2013

Contents

List of Algorithms

List of Tables

List of Figures

Acknowledgements

I am very grateful to my advisers, Christof Fetzer at Technische Universität Dresden and Pascal Felber at Université de Neuchâtel, for giving me the opportunity to carry out my doctorate studies in their research groups. I have learned a lot from them. They enabled me to get started quickly by giving me all the preparation and insights necessary to do research, provided many ideas and motivation, and supported me in later years of my doctorate studies when I was shaping my own research agenda. Thank you!

I also thank my coauthors, Martin Nowack, Jons-Tobias Wamhoff, Diogo Becker de Brum, Patrick Marlier, Etienne Rivière, Stephan Diestelhorst, Michael Hohmuth, and Martin Pohlack, without whom many of my publications would not have happened. We jointly worked towards many Samoa–time-zone paper submission deadlines. Without Stephan, Michael, and Martin Pohlack's work at AMD to make ASF publicly accessible and to provide the ASF-supporting simulator, my work on HTMs and HyTMs would not have been possible.

Furthermore, I am grateful to several people who acted as an external group of experts that I could lean on, and influenced my work and myself. Nir Shavit was a source of motivation to me, and he shared his understanding of the foundations of synchronization openly and always in a positive, inspiring, and energetic way. Dave Dice has very deep knowledge of the practical aspects of how to implement shared-memory synchronization, and is an incredibly friendly person at the same time. Mark Moir's rigorous thinking and great sense of humour make it a joy to argue as well as collaborate with him. Ulrich Drepper openly shared insightful perspectives on technologies such as TM and on the software industry in general with me. Ali-Reza Adl-Tabatabai was intent on giving feedback and spending time on discussing ideas and the state of the art. Thanks to all of you for your support.

Finally, I want to thank all other people who shared their knowledge, ideas, thoughts, or opinions with me.

Chapter 1

Introduction

Exploiting parallelism in programs has become a part of mainstream programming in recent years. This is a major research and educational challenge in software engineering because most application domains are affected and not just, for example, high-performance computing, and because parallel programs are highly different from the primarily sequential programs that were dominating previously.

This shift towards having to write parallel programs has been driven by limitations reached in general-purpose microprocessors. In the past, performance improvements in microprocessors were based on increasing microprocessor clock speeds and instruction-level parallelism. However, heat dissipation increases nonlinearly when increasing clock speeds, which decreases the benefit of this approach because it would make computing increasingly energy-inefficient. Second, instruction-level parallelism (i. e., transparent parallel execution of a sequential sequence of instructions by a microprocessor) is limited due to inter-instruction dependencies in typical sequential programs and because of the cost of the hardware required to (speculatively) execute large chunks of instructions in parallel (e. g., large pipelines and speculation buffers).

Therefore, most current general-purpose microprocessors instead rely on targeting thread-level parallelism by offering several processor cores on the same chip that each can execute a separate, large instruction sequence (e. g., different processes or threads). Such multi- or many-core CPUs do not increase performance by increasing the speed of a single core but rather primarily by putting an increasing number of cores on the same chip. Thus, the final performance of programs on new CPU generations only increases if the level of thread-level parallelism in the programs increases as well.

There are different kinds of parallelism that a program can try to use, which require a different amount of synchronization (i. e., coordination) between the parallel processes. For example, data parallelism typically requires little synchronization, but only if data can be partitioned by the programmer or the program so that each process operates on its own part of the data. However, if data cannot easily be partitioned or if there could be concurrent operations by different threads on the same part of data, then threads will have to synchronize their operations.

Synchronization is thus an important part of parallelization. Its effect on overall performance depends on the ratio of synchronization operations com-

pared to synchronization-free, data-parallel operations and on the level of parallelism that can be achieved in the synchronization parts of a program. For example, a program in which just 10% of the operations synchronize and in which the synchronization mechanism simply executes these operations sequentially will never execute more than 10 times faster than a sequential program according to Amdahl's Law [4], independently of how many cores are available.

The vision behind *Transactional Memory (TM)* is to allow programmers to only declare which operations synchronize without requiring them to actually implement and tune synchronization. TM enables them to use the database concept of a transaction for a set of operations on shared state. Informally, these transactions appear to execute atomically and isolated from other transactions. This is ensured by a generic TM implementation, which relieves programmers of a large part of the burden of having to think about and deal with possible low-level interleavings and interferences of concurrently executing operations. At the same time, the TM implementation can potentially execute transactions in parallel, for example by using optimistic speculative execution and automatic consistency checks.

With TM, transactions can consist of ordinary program code. Transactional operations are supposed to be no different from other operations in the program (e. g., no custom language like SQL for databases is expected). Also, transactions and nontransactional code should be able to operate on almost the same shared state (e. g., I/O might not be allowed in transactions but access to a program's global variables would be).

Different programming languages or environments might support shared state differently, ranging from shared memory to message passing and state machine replication. However, most current general-purpose microprocessors provide cache coherence for a system's main memory, and shared main memory is the primary way for threads to communicate on current systems. Therefore, TM is considered to typically be a form of shared-memory synchronization, especially in programming environments where state is shared among threads by sharing an address space (e. g., multi-threaded C/C++ programs). TMs for message passing environments that run on current systems would likely synchronize and communicate using shared memory as well, but their implementations would differ a lot from the former.

Note that some people prefer to let the term "TM" refer to some set of shared memory TM implementations and algorithms and to refer to the TM abstraction offered in a programming environment as atomic blocks or memory transactions. I think that this distinction is neither necessary nor useful. There is such a wide range of possible implementations of TM that the only significant differentiator of such a set would be that those implementations can be used to build a TM abstraction.

In this thesis, my focus is on TM for programming environments in which threads share state via shared memory and a shared address space. Such TMs are widely applicable in current systems (e. g., in C, C++, and Java programs) and do not depend on a particular parallelization approach or programming language.

TM also has wider goals than other implementation-centric mechanisms for shared-memory synchronization—in particular, most forms of locking and custom concurrent algorithms—because of its focus on generic implementations for synchronization declarations. Therefore, it promises more than these other

mechanisms but might make use of them in the TM implementation at the same time. TM's first advantage is that programmers do not have to implement synchronization anymore and think about whether their implementation composes with the implementation of other operations' synchronization in a program (if those other operations use TM as well). Second, a program that uses transactions is not tied to a particular TM implementation, and the TM can pick an implementation that works efficiently for this particular program, potentially even during the execution of this program based on the actual workload and hardware being available (e. g., in a non-platform-specific application). In turn, there is no reason why TM implementations should magically perform better than custom implementations of synchronization if programmers are willing and able to spend enough development effort on synchronization (e. g., when developing an operating system kernel). In the trade-off between performance and development costs, the goal for TM is to achieve sufficient or better performance while requiring less development effort. In summary, because this trade-off is different for each program, TM's performance goal is to perform well enough while still being convenient enough for programmers to be the overall better synchronization mechanism in many programs.

The most important disadvantage of the other mechanisms for shared-memory synchronization is that they are not as composable as TM when developing software (a more detailed comparison using an examples appears in Section 2.3). When using locking, two separate operations protected by several locks cannot be easily composed into a single atomic operation because the lock acquisition order in the composite must be compatible with the acquisition order in the whole program. Second, the locking scheme of the two operations must be exposed to the composite, which breaks information hiding. Custom concurrent algorithms typically cannot be easily composed as well because current microprocessors support synchronization instructions that are limited to a single memory location, and indirection or complex algorithms are required to build concurrent operations that access several pieces of shared state atomically.

In contrast, transactions are composable because they can be nested within each other and because programmers can use ordinary program code within transactions (with potentially some restrictions, such as I/O). To allow this kind of composability and to achieve usability, transactions must be integrated into programming languages. Otherwise, programmers would have to modify transactional operations to use the TM explicitly, which would make programming with TM a lot more costly in programming languages such as C where accesses to shared state are represented as loading from and storing to memory locations.

This required integration of TM with programming languages already shows that supporting TM either affects several system layers and components or is influenced by them.

- TM support within compilers is required to transform language-level transaction declarations into code that uses a TM implementation. Also, many TM-specific optimizations can only be performed during compilation because only there is all the information contained in the source code available and it is decided how the TM implementation is to be used by the application.

- TM implementations have to work correctly and efficiently for a wide

range of transactions. For example, incrementing a single counter in a transaction is a very different workload compared to a large transaction that navigates through several data structures. Yet, both should execute efficiently. The compiler might be able to provide hints about which kind of workload a transaction is, but this is not always possible. Furthermore, the performance of a TM implementation depends a lot on the hardware (e. g., the relative cost of cache misses) and on libraries and the operating system (e. g., in which patterns a process is allocating memory). New hardware extensions for TM also have to be investigated and used effectively.

- To be adopted, TM has to also compose with existing software infrastructures. This not only affects the programming language integration, but TM has to also work within current software environments, legacy source code and libraries must be usable from within transactions, and TM must not interfere with nontransactional code or require changes in such (legacy) code.

A closer inspection of the TM problem space shows that the problem is indeed cross-cutting and that there is strong coupling between the problems and solutions on the individual layers. For example, which TM properties to guarantee is an important question for the programming language integration but also affects achievable performance in the sense that different language-level decisions will yield highly different objectives for a TM implementation. In turn, a particular guarantee might be so costly to implement in terms of performance that it might not make sense to provide it to programmers because they likely would not use it anyway due to the overhead.

Therefore, during research and development of TM systems, one has to consider all the affected system layers and TM aspects. If not, it is unlikely that the TM performance goal can be reached, and one will not be able to find out whether the vision behind TM can be put into practice.

Contributions. In my work, I have focused on investigating and implementing the *building blocks* that are required for a *high-performance, practical, and realistic software TM*.

The building blocks host several novel algorithms and optimizations for TM implementations and shared-memory synchronization, both for current hardware and potential future hardware extensions for TM. As I have explained previously, the final TM use cases and workloads are not yet known, so I aimed at constructing algorithms and optimizations that are widely and robustly applicable in TM systems, and can serve as useful building blocks for future TM systems. They have been incorporated into TM implementations by other research and industry groups, and have initiated further research.

At the same time, these building blocks compose into a *full TM implementation stack*. The two main implementation components are TinySTM++, a software TM implementation of several TM algorithms, and the Dresden TM Compiler (DTMC), a C/C++ compiler with TM support. This TM stack is ready to be used today and contains all the features necessary for early TM adopters. Providing such a stack is essential because it enables feedback from programmers about TM, which in turn directs future TM research and helps to

break the chicken-and-egg problem that research encounters that proposes fundamental changes in programming abstractions. Second, validating the building blocks in a full implementation stack shows opportunities for cross-block optimizations and ensures that assumptions of blocks about the environment remain realistic.

To keep the development effort bounded, I focused on a particular class of programming environments and hardware.

- I chose to not expect that programmers would follow a specific parallelization approach but to instead assume explicit threading (i. e., programmers manage threads explicitly and use transactions to synchronize between threads). This too is a specific choice, but it covers what would happen in other parallelization environments (e. g., using transactions to synchronize between OpenMP tasks).

- I focused on userspace applications as opposed to operating system kernels as TM usage environments because composability matters most for applications, which are typically authored by several programmers with different sets of skills, and because building custom synchronization code might be beneficial in a kernel, but not in platform-independent applications. Nevertheless, many of the building blocks would be applicable in a kernel as well. I also expect no custom operating system support for TM because this would make TM adoption more difficult.

- C and C++ as programming languages are a useful TM research environment because they are widely used and are sufficiently low-level (e. g., in comparison to a managed environment such as a Java virtual machine, the TM implementation gets validated against potential low-level issues and constraints). Furthermore, C/C++ applications do not reside in a homogeneous environment but often have to deal with a large number of legacy libraries and constraints, which is a good testing ground for TM composability. Just-in-time compilation is not common for C/C++ programs.

- The early TM adoption will likely happen on server and desktop multicore microprocessors (e. g., on architectures like x86-64), so this should also be the targeted hardware. When considering potential future hardware extensions, these should be realistic proposals for first-generation TM support.

Roadmap. In Chapter 2, I will provide background information about the basics of shared-memory synchronization and about transactions and concurrency control from a database perspective. Also, I will show a few use cases of TM and how it compares to other synchronization mechanisms, and give a brief overview of the history of TM to put my thesis' results into context with the overall development of TM ideas.

In Chapter 3, I will describe my building blocks approach to TM by first discussing the TM problem space in more detail, including TM requirements, a structure for TM-based synchronization and the associated performance costs, and refined focus and assumptions about the TM stack. A summary of the building blocks I focus on appears in Section 3.2.1. Also, I will briefly describe

the TM implementation stack that I built and the commonalities in the setup of the performance experiments.

Next, in Chapter 4, I will describe a specification of how transactions can be integrated into C/C++ and will explain what this means for a programmer who uses the new transactional language constructs. Based on this specification, I will then derive requirements for TM implementations in both compilers and TM runtime libraries.

The subsequent chapters focus on specific building blocks. First, I will present STM algorithms and implementations in Chapter 5, in particular the highly efficient atomic snapshots in time-based STM, and performance trade-offs in implementations of such STMs.

Chapter 6 is about TM optimizations within compilers. I will describe how automatic partitioning of application data, a divide-and-conquer approach, enables two TM optimizations: per-partition tuning in partitioning-aware STMs, and how to colocate application data and TM synchronization metadata.

In Chapter 7, I will describe how to integrate a potential future hardware extension for synchronization—in particular, AMD's Advanced Synchronization Facility—into the TM implementation stack, and will present efficient hybrid software/hardware TM algorithms, which can concurrently execute transactions that use the new hardware extension and transactions that do not.

Finally, I will conclude in Chapter 8. To increase locality, the discussions of related work are embedded into the matching individual chapters.

Chapter 2

Background

In this chapter, I will first provide background information about the two main areas that TM is based on: shared-memory synchronization, and transactions and concurrency control as used in databases. The latter provides the idea of transactions as an abstraction for the synchronization of concurrent accesses but does not provide solutions or implementations that can be used as-is in a TM setting. In contrast, the former provides the foundations for TM implementations, but it did not really offer a programming-language–level abstraction as powerful and composable as transactions until TM became a hot research topic.

As explained in the previous section, I do not focus on a specific parallelization paradigm in this thesis, except that I assume that there is thread-level parallelism and threads will have to synchronize concurrent accesses to shared data (as opposed to parallel accesses to disjoint data). Thus, parallelism and concurrency are not the same thing: There can be threads that execute in parallel yet never synchronize, and there can be threads that execute concurrently and have to synchronize but will never offer any parallelism. In practice, most parallel threads will have to synchronize at some time (e. g., at least to signal that they have completed their chunk of work). Still, we want threads that synchronize as efficiently as possible, which often means that they should synchronize as little as possible because synchronization can result in runtime overheads, either due to sequential execution at some level (e. g., when CPUs communicate) or just because enabling concurrent execution for a piece of code requires the execution of more CPU instructions. Amdahl's law [4] gives an upper bound on the program speedup S one can achieve:

$$S = \frac{1}{(1-P) + P/N}$$

P is the proportion of the program that can execute in parallel and N is the number of CPUs. In other words, one can obtain a speedup that grows with the number of CPUs only if the fraction P can be made sufficiently large; any sequential execution caused by synchronization will be an obstacle and decrease the maximum speedup.

In the second part of this chapter, I will illustrate the potential benefits of TM with a few example uses, and will also give a brief history of TM.

2.1 Shared-Memory Synchronization

In what follows, I will give a summary of the most important points that are necessary to understand the basic challenges of shared-memory synchronization. This is not a complete tutorial about this topic, and in order to understand the concurrent algorithms presented in this thesis in detail, I would highly recommend reading a good textbook such as The Art of Multiprocessor Programming [56] by Herlihy and Shavit.

One key point necessary to understand the challenges of synchronization is that communication and coordination are not for free in distributed systems, especially in asynchronous systems. First, it takes time to deliver a message between two processes, so this can potentially result in one process having to wait for another. Second, in asynchronous systems, it is not known (or not precisely known) how long delivering the message will take and how long the other process needs to process the message. Thus, the sending process will have to either wait for a confirmation message from the receiver, or be very conservative and wait for an upper bound of the processing time (if it exists).

Multiprocessor systems (e. g., multicore CPUs) are distributed systems, and they are also asynchronous if we consider just the execution of instruction sequences at individual CPUs. In general, a thread running at one CPU cannot estimate which instruction another thread is currently executing at another CPU, unless they synchronize. Even then, a thread can only observe the messages (e. g., updates to a memory location) sent by another thread but does not know that the other thread is currently doing. Note that to keep discussions general, I will talk about CPUs only and will not distinguish between CPUs in different sockets, cores in a CPU, or hardware threads in a core, unless this is necessary.

Caches. Main memory in most current computers is physically separated from the CPUs, so reading values from or writing to main memory is rather costly compared to CPU-internal communication (e. g., due to smaller bandwidth to the memory and longer round-trip times of messages). Therefore, CPUs use several levels of caching, where caches typically get smaller but faster the closer they are located to the CPU's core logic. The data in these caches is typically kept coherent to ensure that threads reading from a cache see the most recent values and that updates of values in a cache are eventually propagated to main memory. Note that the hardware's (i. e., the CPU's or platform's) memory model specifies the details of the *cache coherence* guarantees, which I will explain later. There are several ways to implement cache coherence in hardware, and I will not explain any of these protocols here.

To understand the concept, it is sufficient to imagine a simple protocol where a CPU accessing a memory location would first ask the nearest cache (i. e., the first-level L1 cache) whether it holds the most recent copy of the value of a memory location; if it does not, then the CPU asks higher-level caches; if the value is not in any cache, then it fetches the value from main memory. An access to the L1 cache is the fastest way to fetch the value, and usually takes just a few CPU cycles. In contrast, fetching a value from main memory can require hundreds of CPU cycles or more. Because of this difference in speed and the limited capacity of caches (they require quite some space on the CPU die, and energy during runtime), the efficient use of caches is essential to achieve good

performance. CPUs can hide some of the cache and memory access latencies with techniques such as out-of-order execution of instructions, but this has limits as well.

Another effect that can decrease performance is *contention*, which in general means that some resource is so contended in terms of requests that it has to process that it becomes a bottleneck. In the case of caches, contention can arise when different CPUs fight about which CPU's cache a memory location is cached in; if these CPUs go back and forth, they will constantly suffer from *cache misses*, and each cache will have to constantly transfer the most recent value from the other cache to resolve the cache miss. This will increase the memory access times at both CPUs.

Cache coherence does not operate on the basis of individual bytes of memory but rather at the granularity of *cache lines*, with each line representing one continuous 64-byte chunk of main memory, for example. In practice, it is therefore often necessary to separate data that is frequently updated by different threads into different cache lines to avoid any cache contention overheads due to *false sharing* at the cache line granularity.

It is also important to consider that main memory access paths can be different between individual CPUs, which is called *nonuniform memory access* (NUMA). One example for this are systems were each CPU socket handles access to a part of main memory, and accessing a certain memory location from one CPU might involve asking the memory controller in another CPU to fetch the value. Thus, it is not only faster to synchronize for CPUs that are closer to each other (e. g., two cores on the same CPU socket) because they have a closer shared cache, but it is also faster for a CPU to access main memory attached to the same CPU socket. Similarly, cache line contention between cores on different CPU sockets can be especially costly because of higher communication costs between those sockets.

Overall, memory access latencies (both for main memory and caches) can vary significantly on a single machine as well as between different machines. This complicates the performance tuning of synchronization code a lot, and can also lead to completely different concurrent algorithms being optimal on different machines. More information about memory access costs can be found in a tutorial [35] by Ulrich Drepper.

CPU instructions for synchronization. When threads synchronize via shared memory, they have to access the same memory locations to exchange information. Because those threads execute asynchronously, memory accesses can happen concurrently, so threads have to consider the guarantees that hardware provides for concurrent memory accesses.

Individual loads from and stores to memory locations are usually *atomic* on general-purpose CPUs (i. e., the individual bits comprising the location are guaranteed to be accessed by a single virtually indivisible operation). This typically holds for accesses of up to machine word size (e. g., 64b on x86-64) but not for larger accesses nor for accesses by more than one CPU instruction.[1] Note that atomicity of loads and stores does not guarantee any specific ordering (e. g., program order might not be preserved). Ordering guarantees are instead

[1] There are often other hardware-specific restrictions such as no atomicity with respect to instruction fetching by a CPU. Details can be found in the respective CPU manuals.

specified by the hardware's memory model, which I will describe later.

Because atomicity is not guaranteed for more than one instruction, it is easy to see that threads cannot safely modify memory locations without possibly overwriting concurrent writes to the same location. Because the overall system is asynchronous, some other thread might always have a *pending write* that it is about to execute and that can overwrite previous writes by other threads. Second, no thread can read from some location and make an update under the assumption that the former location still has the same value. Informally (and simplified), this means that threads that only use ordinary loads and stores are not able to agree on something because they cannot make a decision (i. e., write to memory) while still knowing that their former view of the world still holds.

Thus, what we need is a way for threads to execute an update atomically with the check of some invariant (i. e., a load from some location). This will allow a group of threads to reach *consensus* on the value of a piece of shared state because no thread will jeopardize a prior decision anymore by a pending, uncontrolled write.

CPUs offer this functionality by providing a *compare–and–set (CAS)* operation, which is a special atomic CPU instruction that updates a memory location if and only if the same location has the value that the thread expects it to have:

```
1 bool CAS(machine_word* location, machine_word expected, machine_word new_val)
2 {
3   if (*location == expected) { *location = new_val; return true; }
4   else return false;
5 }
```

This instruction is also called compare–and–swap or compare–and–exchange, which are basically the same but return the previous value of the memory location instead of just a boolean flag. The load-linked–store-conditional instruction offered by some CPU architectures has a similar purpose and applicability. In pseudo-code, I will also use the following notation for CAS:

$$\mathsf{cas}(\mathit{location} : \mathit{expected} \rightarrow \mathit{new\_val})$$

The CAS operation is universal in the sense that it can be used to implement consensus for arbitrary operations on objects larger than a machine word [52]. However, the instruction itself really only offers atomic access to a single machine word, so even with this universal tool, concurrent programming is still very complex because everything has to be implemented with atomic accesses to single memory locations (e. g., using techniques such as indirection, versioned state, operation logs, etc.). This leads to many concurrent algorithms being complicated enough to be publishable results, which also highlights the need for simpler abstractions such as TM.

Some CPU architectures (e. g., x86) offer a few other instructions that execute read–modify–write operations such as fetch–and–increment or fetch–and–or atomically on a single memory location. These operations are typically slightly faster than implementations that loop until an equivalent CAS operation succeeds. CAS operations for two (DCAS) or more separate nonadjacent memory locations have been invested but have not appeared in mainstream hardware so far. DCAS [33] was reported to provide not enough benefit to justify hardware support, whereas limited first-generation HTM was reported to make concurrent algorithms significantly simpler in some cases [25].

Memory models. A CPU architecture's memory model specifies the allowed outcomes when different CPUs are concurrently executing instructions that access memory. While executing a single thread in isolation will give this thread the impression that its instructions were executed in program order (i. e., the order that they have in the executed instruction sequence), concurrently executing instructions on other CPUs will not necessarily observe these outcomes in the same order. General-purpose CPUs typically do not provide *sequential consistency*, which would guarantee that all memory accesses are applied in a global total order that is compatible with the individual threads' program orders.

Memory models thus specify the ordering and atomicity guarantees that are provided by CPUs when they execute concurrent memory accesses. For example, on x86 CPUs, pairs of stores will become visible to other CPUs in program order; however, for a store followed by a load (in program order, both to different memory locations), the load might be reordered and fetch a value before the store becomes visible to other CPUs. Programmers can disallow such reordering by inserting *memory barriers* (also called *memory fences*) between memory accesses in the instruction sequence, which are special CPU instructions that prevent certain reorderings of these accesses. Different barriers are sometimes called by the reorderings that they prevent (in our example, a store–load barrier could be used). Some instructions have implicit memory barrier semantics (e. g., CAS on x86 is a full barrier and prevents any kind of reordering across it). The complexity of hardware memory models differs (e. g., x86 [102] is simpler than PowerPC [100]), but they are generally rather complex and require great care when programming if the code is supposed to be somewhat efficient. Also note that these potential reorderings are effects that programmers have to pay attention to in addition to the interleavings that result from having no atomicity for several memory accesses.

Several programming languages also have memory models because many compiler optimizations are based on the reordering or merging of the memory accesses contained in the program source code. Thus, programmers need to know which executions and outcomes are allowed when threads execute concurrently, which they can reason about based on the language's memory model. Compilers then have to generate code that implements the language's memory model on top of the memory model of the target architecture.

Section 4.1 gives an overview of the C++11 memory model. In pseudocode based on it, I will use the notations outlined in Table 2.1; for brevity, I will not distinguish atomically accessible variables from other state (e. g., as necessary in C++11 using atomic<> types). To understand how an algorithm operates, it is sufficient to assume that all memory accesses are full barriers. However, knowledge of the C++11 memory model is required to understand whether the memory barriers used in a particular algorithm are actually sufficient.

Linearizability. The following explanations will have made it obvious that instruction sequences are not a convenient way to reason about the concurrent behavior of operations, and that we instead need less complex correctness criteria. Linearizability [58] is such a criterion; informally, an implementation of a concurrent data structure is linearizable if (1) all its operations get virtually executed at one indivisible step at some point in time (i. e., the linearization point) during the invocation of the respective operation, and if (2) the outcomes of all

Notation (example)	Description
$a \leftarrow b$	Load b (relaxed MO), store into a (relaxed MO).
$a \leftarrow_{acq} b$	Load b (acquire MO), store into a (relaxed MO).
$a \leftarrow_{rel} b$	Load b (relaxed MO), store into a (release MO).
$a \leftarrow \mathsf{inc\text{-}and\text{-}fetch}_{rel}(b)$	Atomically increment and fetch b (release MO) and store into a (relaxed MO).
$\mathsf{cas}_{acqrel}(a : b \rightarrow c)$	CAS a (acquire–release MO) from b to c.
fence_{rel}	Memory barrier (fence) with release MO.

Table 2.1: Memory order (MO) notations. MO semantics are as in the C++11 memory model, but accesses default to being atomic and with relaxed MO.

operations are equal to the outcomes of executing the operations sequentially in the order given by the linearization points (i. e., like what a sequential implementation would produce). Invocations are like function calls, so one can model them as invocation and response events with timestamps (e. g., from wall clock time); this real time order then constrains when operations can take effect. Note that it is only necessary to be able to associate each invocation at runtime with a linearization point that leads to an equivalent sequential execution; it is not required that the operation takes effect atomically at exactly this point in real time. For a precise definition and explanation of linearizability, please see the previously recommended textbook [56].

Linearizability is widely used to describe and reason about the correctness of concurrent algorithms. Because it is based on ordering in real time (which is somewhat ubiquitous), it is fairly straight-forward for programmers to reason about compositions of invocations to linearizable objects, and linearizability can be investigated for objects in isolation, irrespective of what other objects are doing (provided that there is information hiding and objects are only accessed using their operations). For example, CAS on x86 can be roughly considered as a linearizable operation when ignoring other loads and stores because it is atomic and its implicit full memory barrier will prevent reordering with other instructions. In turn, we can implement a concurrent linearizable counter using such a CAS instruction:

```
1 machine_word fetch_and_increment(machine_word* counter)
2 {
3   machine_word value;
4   do { value = *counter; } while (!CAS(counter, value, value + 1));
5   return value;
6 }
```

For this code, we know that the invocation of CAS that returns true will be between the invocation and response events of fetch_and_increment. This particular CAS invocation also produces both the return value of the operation and the only update to the counter's value. Because CAS is linearizable, our counter is linearizable too if fetch_and_increment is the only operation that our counter offers.

Progress. Complementing correctness criteria such as linearizability, progress conditions specify whether concurrent operations that are invoked by different threads will eventually finish (i. e., make progress). The best explanation of

progress that I am aware of is in a paper by Herlihy and Shavit [57], which explains the relationship between progress conditions, progress for some or all operations, and the guarantees provided by schedulers that control the execution of threads (e. g., operating system schedulers). For the purpose of this thesis, it is sufficient to consider just four conditions. In *blocking* implementations, operations can be prevented from making progress by stopping the execution of operations by one or several other threads, because in these implementations operations might block until another one has finished first. In *nonblocking* implementations, stopping the execution of some operations cannot prevent progress, but note that this does not guarantee progress in general. There are three important nonblocking conditions. First, *obstruction-free* implementations ensure progress for every thread that is executed for a sufficiently long time in isolation (i. e., without interference by other threads). Second, *lock-free* implementations guarantee that there will always be some progress eventually, independently of how or whether threads are executed, even though some operations might never finish. Finally, *wait-free* implementations guarantee eventual progress for all operations that are invoked. Thus, wait-freedom is the strongest progress condition but also the most complex to implement.

Locks. One way to prevent concurrent threads from interfering with each other is to use *mutual exclusion* to ensure that only one thread is executing. Locks are implementation mechanisms that ensure mutual exclusion. Threads have to *acquire* a lock before executing a mutually exclusive operation and *release* it afterwards (this region is then a *critical section*). If another thread wants to enter the critical section by acquiring the lock, it will have to wait until the first thread has released the lock again (therefore, locks are blocking implementations). Note that there are also types of locks that allow more than one thread to enter the critical section, or where only some operations block other operations from entering (e. g., reader–writer locks). Also, there are several ways to implement locks (e. g., spinlocks, queue locks, and others), which all have different performance characteristics and often ensure different progress conditions (but are still blocking). Finally, lock elision has been proposed [86], which allows the speculative execution of critical sections and thus allows threads to execute logically mutually exclusive instead of mutually exclusive in time. Its implementations are thus similar to TM in spirit but with locks as fixed fallback solution. Hardware support for lock elision is only starting to appear in mainstream CPUs, and software solutions basically at least face the same issues as STMs.

Despite those differences, all kinds of locking suffer from two major drawbacks. First, they *rely on conventions* in the sense that different parts of a program (and thus likely different programmers as well) have to agree on which locks protect which objects or state. Disagreement about this will lead to uncontrolled concurrent accesses to state, which can easily result in value errors (e. g., imagine the CAS in the previous fetch_and_increment example to be not atomic and rather consist of separate read and write access). Thus, this convention has to be documented and followed when programming, which requires more effort by programmers and is error-prone. Furthermore, objects either have to expose locks that are stored in the object, or callers have to manage locks for components that they use; both breaks information hiding. Even worse, locking

conventions do not only have to specify the state–to–lock mapping, but there also has to be a globally agreed-upon order in which locks are acquired to prevent *deadlocks*. For example, imagine that there are two locks A and B and two threads that try to acquire both; if the first thread acquires A and the second thread subsequently acquires B, then both threads will deadlock because when they next try to acquire the respective other lock, they will wait for each other and will not make progress anymore.

Second, locks are an implementation mechanism, not just a declaration about mutual exclusion. Even if the lock implementations can be exchanged later on, programmers usually have to decide how many locks protect which data. For example, they can use *coarse-granular* locking where each lock protects a lot of state; this will keep the locking convention simpler, but the achievable parallelism might be low. Or they might use fine-granular locks, which potentially allows for more parallelism but will result in a much more complex locking convention.

Both drawbacks decrease the composability of programs that use locking. In the first case, correctness is at stake because the composition of two locking conventions must still be deadlock-free, for example. Changing the use of locks within a component can require a lot of programming effort. In the second case, there is a potential lack of perfomance composability. A library programmer often can not anticipate in which kind of workload the library code will be used in; if using coarse-granular locking, the library might cause a performance bottleneck, whereas fine-granular locking that has to be exposed on the interface will make the library harder to use and might also be unnecessary in terms of performance[2].

Summary. To summarize, shared-memory synchronization can be very difficult, especially when using the low-level synchronization operations offered by current mainstream CPUs. Existing general-purpose abstractions (or mechanisms) for synchronization such as locking also have their pitfalls, and often require programmers to make the difficult trade-off between ease-of-use and good performance. This is one reason why researchers and developers have been investigating new abstractions such as TM.

2.2 Transactions and Concurrency Control

The concept of transactions has its origins in the database field. Today, transactions are used in a variety of systems besides databases, such as in messaging systems, some file systems, and of course TM. For many databases, the guarantees that transactions provide roughly follow the so called ACID properties: Atomicity, Consistency, Isolation, and Durability. However, different databases serve different purposes—therefore, these properties are also provided to different extents. For the other systems that make use of transactions, this applies even more (e. g., durability might not be considered at all). For example, whereas atomicity is associated with fault tolerance and recovery in databases,

[2] In some cases, depending on the scheme of fine-granular locking and the implementation of locks, the performance of fine-granular locking can actually be worse than coarse-granular locking because of lock acquisition runtime overheads and of the space overhead of the locks that have to be managed.

TM sees atomicity more as a means to manage the indivisible execution of concurrent multi-step operations. Nevertheless, the basic concept of a transaction is present in all those systems.

Thus, it makes sense to know about at least the basic understanding of transactions in the database field. In this section, I will give a very brief overview of the parts of database theory that are most relevant for TM. To get a real overview of this field, I would recommend to read the textbook by Weikum and Vossen [119]. With a few exceptions, I will not summarize database algorithms here because TM implementations are more inspired by shared-memory synchronization techniques, even though there are strong similarities between some TM algorithms and classical database algorithms such as two-phase locking on a conceptual level.

Serializability. The main correctness criterion for database transactions is serializability, which roughly expresses that a concurrent execution of transactions has to be equivalent to executing these transactions in some serial order. However, there are several kinds of serializability, whose definitions vary significantly regarding which concurrent executions are deemed serializable. Also, the database field has evolved over quite some time and so has serializability theory, which becomes visible when comparing older definitions of serializability with recent ones.

Transaction executions are modeled in the form of schedules that give a total order in which the individual steps of transactions are executed. Operations include begin, commit, and abort actions as well as the actual operations performed by a transaction such as reads from and writes to individual data items in the database. A serializability definition then defines whether a certain schedule is serializable and thus to be allowed for execution by a database. In a database implementation, a scheduler performs *concurreny control* by deciding which schedules are correct (i. e., serializable) and potentially aborting transactions to allow for creating a correct schedule.

Note that from the perspective of programming-language memory models such as the one of C++11, such schedules and the definitions for many kinds of serializability are at a too high level of abstraction, make a few implicit assumptions, and are rather coarse when it comes to progress properties; however, there are well-suited to let database users and implementers reason about the properties that a database provides when executing transactions, so they fulfill their purpose.

Among the different kinds of serializability, *conflict serializability* (CSR) is probably the most interesting one in the context of TM. Roughly, operations conflict if they target the same data item and the order in which they are applied has an effect on either the resulting state or the results returned from those operations (e. g., reads do not conflict with reads, but writes conflict with reads and writes). CSR requires conflicting operations by different transactions to be ordered in the same way in both the concurrent schedule and the assumed serial execution of those transactions; if one would map each conflicting pair of operations to a directed edge between the respective transactions, the schedule is conflict serializable iff the resulting graph is free of cycles (and any topological sort of this graph is then a possible equivalent serial execution of those transactions). *Commit order-preserving conflict serializability* (COCSR)

further restricts CSR by requiring that the order of conflicts between transactions also determines the order in which transactions commit (i. e., the order of their commit actions in the schedule). COCSR allows for schedules that would also be linearizable because roughly, the order of commits represents the linearization points.

CSR and COCSR are interesting because they are efficient to check and can also be efficiently created by schedulers that use locks to protect concurrent access to data items. In particular, when using *two-phase locking*, a scheduler would first acquire all locks that are required in the transaction (the growing phase), and would only later start releasing them but not acquire any further locks (the shrinking phase). As a result, at some point the transaction will hold all the locks that are necessary to execute all operations of the transaction. The resulting schedules will be conflict serializable. *Strong two-phase locking* is a form of two-phase locking in which all locks are held until the transaction commits, which creates schedules that satisfy COCSR. If using locks, TM implementations typically do not release locks prior to transaction commit because the set of accessed memory locations is in general not known in advance and a new location could be accessed at any time (i. e., it is unknown whether the shrinking phase can start until there actually is a commit); thus, such TM implementations would guarantee COCSR and linearizability for transactions.

Page vs. object model. The operations that transactions can execute can be categorized as either following a page model or an object model: The former considers just read and write accesses to data item, whereas the latter can deal with arbitrary operations on data objects (e. g., inserts and removes from sets). Database theory covers both areas and there are object-model versions of correctness criteria and schedulers (e. g., conflicts for arbitrary operations can be defined based on commutativity properties). Object-model concurrency control is also called *abstract concurrency control*.

Operations on an abstract level can in turn be implemented as separate transactions on a lower-level representation of the respective object (e. g., with plain reads and writes). This is called *multilevel concurrency control* and can include several layers of abstraction. This is a very useful approach, both for transactions for distributed systems and for potentially improving performance. Conflicts at a low level of abstraction might not necessary represent conflicts at a higher level of abstraction, so this decreases concurrency and thus transaction throughput (e. g., inserting two different elements into a set does not conflict on an abstract level but could require modifying the same low-level state depending on the implementation of the set). This same approach can also be applied in a TM context: Transactional Boosting [53] combines abstract concurrency control with efficient linearizable base implementations of concurrent data structures to allow for high-performance transactional accesses to those data structures.

Recovery. Databases have to potentially abort transactions, so their schedulers must only allow transaction schedules in which uncommitted transactions can indeed be safely aborted. Inverse operations can be used to model this (and also to implement this) because they can undo previous operations (e. g., for a write operation, we could write back the previous value).

To be prepared for failures and for recovering from them, schedulers can

always consider *expanded schedules* which are basically the original schedules appended with inverse operations for all operations of uncommitted transactions (in reverse order). If an expanded schedule is correct, then this also means that we can recover from any error while executing this schedule. Note that this does not just include database failures but also cases like having to abort due to encountering a deadlock.

This should already illustrate why database recovery theory is also interesting from a TM perspective. TM implementations in general do not know which operations a transaction might execute and cannot calculate a correct schedule before executing a transaction; thus, they have to be prepared to abort some of those transactions on demand.

One criterion that is important when reasoning about correctness of expanded schedules is *reducibility*: Informally, schedules are reducible if they can be transformed into a serial schedule by (1) reordering nonconflicting adjacent operations (i. e., exploiting commutativity and conflict definitions much like in CSR) and unordered commutative operations, (2) removing an operation and its inverse if both appear one after the other in the schedule, and (3) removing reads of uncommitted transactions. If all prefixes of a schedule are reducible, the whole schedule is *prefix reducible* (PRED) and, informally, serializable even when taking failures into account.

However, trying to find a sequence of transformations that show inclusion in PRED is not quite practical at runtime, so simpler conditions that also guarantee PRED would be useful. *Rigorousness* is such a condition and requires (1) that data items are not read or overwritten if an uncommitted transaction has most recently written the same item (i. e., only committed values are read or overwritten), and (2) that data items are not overwritten if they have been previously read by uncommitted and still active transactions. Rigorousness is stricter than PRED, but strong two-phase locking generates exactly the class of rigorous schedules.

To summarize, many concepts from database theory can be applied to TM even if TM has to consider further constraints such as the memory models of programming languages. For example, TMs that perform strong two-phase locking also produce serializable schedules even if they have to abort transactions.

2.3 TM Usage Examples

The following examples are based on C++ transaction statements, which are explained in detail in Section 4.1. To understand these examples, it is sufficient to assume that the compound statement following the **__transaction_atomic** keyword is executed as a single transaction. For example, the following code's first transaction will increment counter atomically, whereas the second transaction will increment counter only if it has a value smaller than 10:

```
1 __transaction_atomic counter++;
2 __transaction_atomic { if (counter < 10) counter2 = counter++; }
```

Both transactions could be executed concurrently by different threads but would still appear to run virtually atomically and without interleaving with any other transaction (e. g., counter2 will always remain less than 10).

Likewise, we can express a CAS using a transaction:

```
1 bool CAS(machine_word* location, machine_word expected, machine_word new_val)
2 {
3   __transaction_atomic {
4     if (*location != expected) return false;
5     *location = new_val;
6   }
7   return true;
8 }
```

This example also shows one of the benefits of proper integration of transactions and programming languages: In the failure branch, we can just return from the function containing the transaction, and the transaction will still be properly committed.

The following (contrived) example shows that we can call functions from both transactions and nontransactional code (e. g., the compiler will ensure this for Set :: insert , creating a transactional version of this function if necessary). This makes it easier for programmers to use transactions because they can transform existing sequential code (e. g., Set :: insert) into synchronizing code just by *declaring* the transaction. Furthermore, transactions can be nested, even dynamically (whether there is nesting in this example depends on whether key is already contained in multiset):

```
1 void addNumbers(int key)
2 {
3   set = new Set();
4   set->insert(5);
5   set->insert(23);
6   __transaction_atomic {
7     if (!multiset.insert(key, set)) {
8       addNumbers(key + 1);
9       delete set;
10    }
11  }
12 }
```

To illustrate the difference compared to locking, let us look at one final example:

```
1 template <typename T>
2 void swap(T& a, T&b) { T temp = a; a = b; b = temp; }
3
4 int counter1, counter2;
5
6 // Thread 1:
7 __transaction_atomic {
8   swap(counter1, counter2);
9 }
10
11 // Thread 2:
12 __transaction_atomic {
13   if (counter1 < 10) {
14     swap(counter2, counter1);
15     counter2++;
16   }
17 }
```

Using transactions, it is easy to see that both threads' operations can safely execute concurrently because all accesses to the two counters are declared to be atomic using transactions.

However, how would we implement this with locks? We cannot simply acquire locks in swap for both a and b in some static order because the threads will call swap with each a different counter as the first argument, so doing that would result in possible deadlocks. Second, Thread 2 needs the access to counter2 to

be atomic with the swapping, so it would have to acquire a lock early anyway. Third, swap might get mostly called from sequential code, so acquiring locks there would be unnecessary overhead.

If we instead choose to acquire locks before the call to swap, then both threads need to agree on the order in which they have to acquire locks and on which ones (i. e., follow a common locking convention). Additionally, they have to know which data in swap needs to actually be protected by locks, which is fine in this tiny example but breaks information hiding and can get much more complex for nontrivial functions.

Performance can also be a concern: If we use just a global lock for both counters, correct lock acquistion becomes much easier, but performance might be bad if Thread 2 mostly runs with counter1 being less than 10 and some other thread accessing just counter2. Using reader–writer locks would also be difficult because Thread 2 might have to upgrade its lock for counter1; to avoid potential deadlock, we would have to acquire the strongest set of locks potentially necessary.

Overall, even this small example illustrates that TM provides a much easier to use programming abstraction because the compiler and runtime system will care about how to implement the synchronization declared by the programmer. In turn, this can allow for a higher level of composability than when using several locks because atomic transactions will provide atomicity even if combined or concurrently executed with other atomic transactions.

2.4 A Brief History of TM

In what follows, I will provide a brief timeline of how TM has developed both in terms of research and industry adoption. This is not intended to list all results and steps but rather to show the overall direction of TM and how my thesis fits into this time-wise. More information can be found in the sections about related work that follow (e. g., Section 3.3), and a book by Harris et al. [50].

TM research started almost 20 years ago. In 1993, Herlihy and Moss first proposed TM support as a hardware feature [55]. In 1995, Shavit and Touitou first proposed STM [103]. While earlier work already established the concept of transactions [119], including transactions integrated with programming languages [75, 38, 24, 107], both these two papers are now widely considered to represent the start of TM research.

Then, it took 10 years until 2003, when the first dynamic STMs were presented: Harris and Fraser's Java-based STM and early TM compiler support [49], and DSTM by Herlihy et al. [54]. In contrast to prior work, those STMs did not require transactions to declare all memory locations that they would access at the start of a transaction; this is important because, for example, it allows transactions to navigate pointer data structures, which makes them more widely applicable.

In 2006, time-based STM was first proposed in a paper that I coauthored [92]. This is a STM technique that allows to reduce the runtime overheads significantly when transactions do not make their read operations visible to other transactions (see Section 5 for details). This work sparked a lot of related research, and the technique is used in production-level TM implementations till today [83, 44].

Also, several people such as Ennals [37] argued that STMs should not strive to provide nonblocking progress guarantees and rather use locks internally in the STM implementations to reduce runtime overheads. Dice et al. first proposed the combination of time-based STM with a lock-based implementation in TL2 [27].

Furthermore, the first TM software stacks for Java that integrated TM runtime libraries with programming-language integration and compiler support were also presented in 2006 [1, 51] (see Section 3.3).

Soon after that, around 2007, several authors presented TM programming-language integration and first compiler optimizations for C/C++ programs [41, 118, 5, 23, 83]. While the way in which programmers would declare transactions differed in these approaches, all relied on compilers to bridge the gap from C/C++ declarations to TM runtime libraries, instead of relying on TM library interfaces or simple source-to-source transformations. Among these, my work on Tanger [41] resulted in the first publically-available TM support for a C/C++ compiler (which is now a part of DTMC, see Section 3.4).

Other TM research focused on topics such as TM theory and correctness criteria, hardware support for TM, conflict resolution policies between transactions, integration of I/O operations, nesting semantics, distributed TM, and programming-language–integration issues.

While likely in development for a longer time, TM support in mainstream CPUs has been announced only fairly recently. Sun's Rock CPU [12] featured a rather restricted HTM, but was ultimately canceled. AMD published a proposal for an HTM [2], and both Intel and IBM have announced the availability of HTMs [64, 117] in processors scheduled to be available in 2013 or 2012, respectively.

Transactional language constructs for C++ [63] have been recently proposed for standardization; this draft specification has been the result of a collaboration between several companies since 2008, and is since 2012 being developed under the umbrella of Study Group 5 of the ISO C++ committee [66]. GCC, arguably the most important open-source C/C++ compiler today, supports most of this draft specification since its 4.7 release in 2012 [44].

Chapter 3

TM Building Blocks

In this chapter, I will discuss the TM building blocks approach that I followed in my research in detail. As outlined previously, TM is a problem that cuts through several layers of a typical system stack and, in turn, is affected by how TM aspects are implemented on each of those layers.

Therefore, I will first analyze the TM problem in detail: What are the requirements for TM abstractions and implementations precisely? What are the potentially relevant factors in TM-based synchronization, and how can we structure them so that we better understand their interaction? What are the performance challenges, and how can we address them and with which optimization attempts?

Second, in Section 3.2, I will motivate and demarcate the scope of my work: What are the basic assumptions about the problem, what should I focus on in my research, and which building blocks are necessary? While not all of the building blocks have been open research questions, my work contributed to the state-of-the-art in most of the major layers that affect TM (e. g., synchronization algorithms, compiler-based optimizations, and exploiting new hardware features).

Finally, I will describe related work that investigated full TM implementation stacks (Section 3.3), as well as the software prototypes that I developed for the building blocks and the experiment setup (Section 3.4).

3.1 Problem Analysis

In the following analysis, the starting assumptions about the system and the use of transactions are intentionally unspecific in order to provide a birds-eye view of TM and the contexts it might be used in. This will get refined in Section 3.2. For now, let us just assume that (1) there is some application with transactions declared by a programmer, (2) a compiler transforms it to executable code, (3) this code runs on top of or merged with other code such as libraries or an operating system, and (4) all of that is executed on some piece of hardware.

3.1.1 TM Requirements

TM's high-level goal is to provide a generic implementation for synchronization declarations supplied by programmers. Its performance has to be high enough to justify using TM declarations instead of concrete implementations (e. g., using locks) and tuning of synchronization. This leads to three main requirements for TM: *usability*, *composability*, and *performance*. Note that composability contributes to usability but is an advantage that other synchronization mechanisms do not offer to the same extent and thus is important enough to be considered a distinct requirement.

Usability. The development effort that TM requires when developing a program that uses it is important because it is one part of the benefits that TM can offer. Achieving usability means keeping this effort low by, for example, making programming with TM intuitive, easy to reason about, and not error-prone.

Integrating TM into programming languages is a TM feature that is essential for usability.

- If there would be no language integration and code transformation by a compiler, programmers would have to use TM like a library and transform every language-level access to program state into a call into the TM library. Using existing source code in transactions would then require a costly refactoring of this code. In turn, the calls to the TM library would make the code harder to read, and harder to be used outside of transactions.

- The semantics of programming languages intersect with what a TM provides. For example, a language's memory model specify when accesses to state happen and how a program is supposed to behave in a multi-threaded setting. A TM must be able to roll back and restart transactions, which can interfere with exceptions and other kinds of control flow transfer in the language.

Second, the TM semantics also have a large influence on usability:

- It must be easy to understand for the programmer and should be intuitive for someone who is familiar with the respective programming language. The correctness of a program with transactions should be simple to reason about, at least in comparison to other synchronization mechanisms.

- It must be useful in a sufficiently large number of use cases. A wider applicability of TM increases usability because getting familiar with TM then amortizes over a potentially larger number of applications.

Composability. TM is not a free-standing solution but rather embedded into existing systems, driven by parallelism becoming a more important aspect in theses systems. TM has to compose well with the other components it is being combined with, especially if incremental adoption of TM is to become possible:

- It must be possible to use TM within existing systems and programming languages, which requires building TM in such a way that it makes sense to use it in these environments. For example, there should be no dependence on new kinds of hardware support for TM.

- Reusing existing code, libraries, and other components must be possible. Programming language integration and compiler support is necessary for this. If executing the existing code transactionally is not possible, there should be fallback solutions (e. g., for I/O).

- Parts of the systems that do not use TM should not be affected by the use of TM elsewhere. This applies to aspects such as language semantics and performance.

- Transactions must compose with each other. On the level of programming languages, constructs such as transaction statements can provide this in a straightforward way (see Section 2.3 for an example). However, this also requires common language-level and TM library interfaces to enable cross-vendor and cross-compiler composability.

Performance. Scalability and single-thread overheads are the two major aspects of TM performance:

- Single-thread overheads refer to how much slower the transactional execution of a piece of code is compared to the sequential execution of this code. This class of overheads is not caused by the interference of concurrently executing transactions but instead by the TM just preparing the code for such interference (e. g., adding checks that abort a transaction on conflicts). Keeping single-thread overheads low is especially important in workloads with low contention and a lot of parallelism, or when only few threads are executing concurrently.

- Scalability characterizes how much of the parallelism in a workload a TM can exploit when the level of concurrency increases (e. g., when more threads execute transactions concurrently).

Note that the optimal trade-off between scalability and single-thread overheads is specific to the respective workload and application. Similarly, applications might have further requirements regarding fairness or latency of the executions of transactions, but a simple liveness property of the whole TM might be sufficient in other cases. Energy efficiency might be a concern in some environments, too.

3.1.2 TM-Based Synchronization: Time and Space

To give an overview about the problems associated with TM-based shared-memory synchronization, I will next discuss two major dimensions of the problem: Time and space.

On both dimensions, a large number of factors decide which low-level shared-memory synchronization actually happens in a system. The TM typically controls only a subset of these factors. For example, an application using the TM decides which operations are performed in transactions, allowing the TM to control only how these operations are performed. Second, it might be feasible to not integrate TM with a certain component to avoid strong coupling between components (e. g., the TM and the operating system). These trade-offs are a large part of the decision which building blocks a TM should consist of.

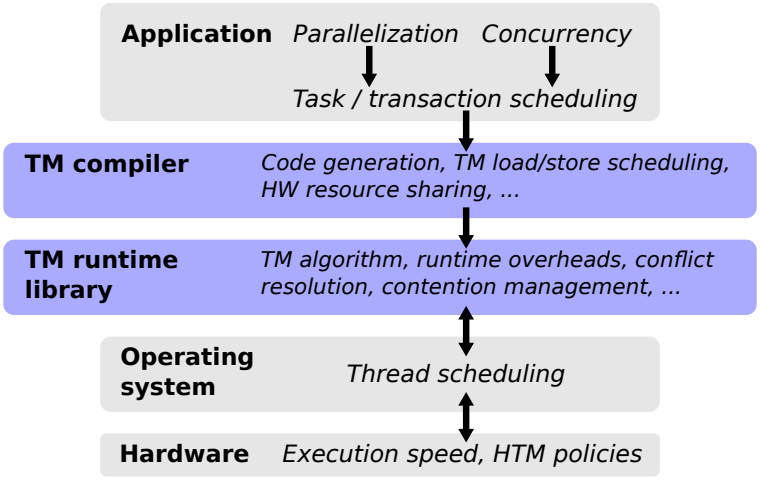

Figure 3.1: An overview of the temporal aspect of TM-based synchronization.

Note that the following discussion highlights the possibilities and is meant as an overview; I will present my selection of blocks later in Section 3.2 and discuss further details in the following sections.

The TM even cannot know about all factors, or it is not feasible or too inefficient to determine (e. g., the average execution time of a certain piece of transactional code). Thus, it will likely have to deal with a lot of uncertainty regarding the system it operates in, and especially regarding the transaction workload.

Time. The time at which operations are performed determines whether they actually execute concurrently (i. e., whether their executions overlap in time) and thus how they need to be synchronized. For shared-memory synchronization operations, the temporal behavior has a large influence on performance (i. e., it can influence the level of contention on a resource, see Section 2.1 for details). The scheduling of higher-level operations can also change whether operations execute concurrently, and thus whether they have to synchronize at all on lower levels.

Figure 3.1 illustrates the components that contribute to determine the temporal behavior of an application with transactions. First, the application follows a certain approach to parallelization (e. g., explicitly multi-threaded), which determines when the high-level tasks and transactions in the application execute, and on which and on how many operating-system threads.

The TM then has a large effect on the temporal behavior of the transactional code in the application. The compiler and how its TM support is implemented generate the code for transactions and thus determine when (transactional) loads and stores are scheduled in the code and thus when they happen during the execution of a transaction. Different TM algorithms and implementations have different runtime overheads, affecting transaction execution time. The TM also decides how to proceed when transactions conflict (e. g., aborting a particular transaction and using back-off). The application code generated by

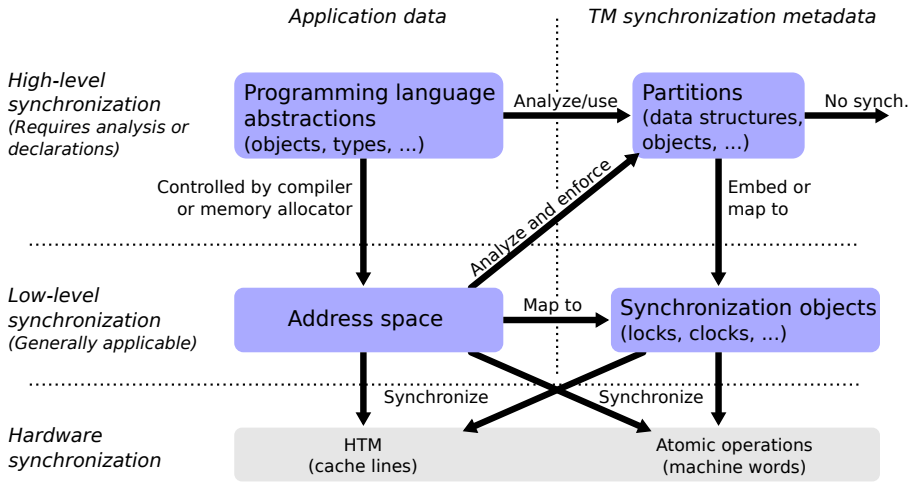

Figure 3.2: An overview of the spatial aspect of TM-based synchronization.

the compiler and the TM implementation share resources, so there is further potential interference.

The operating system then controls when threads execute and on which hardware resources (e. g., when a thread executes on a certain CPU compared to threads of another application). The execution speed of the thread's code depends on the properties of the hardware it is executed on and can vary widely due to concurrent execution of other code (e. g., due to contention or cache miss latencies). Furthermore, if the hardware also contains support for TM (HTM), then its properties such as conflict resolution policies will also impact the temporal behavior.

These effects and influences on the temporal behavior of the application are not isolated. As I have mentioned previously, a different parallelization approach can potentially run transactions in a different order leading to less conflicts between transactions; in turn, this could allow the TM to choose a different TM algorithm with more coarse-grained low-level synchronization leading to less runtime overheads and faster execution of the transactions. As another example, consider how to best tune exponential back-off that gets triggered on contention on locks used to synchronize tasks or transactions. The application-level scheduling can affect the contention on these locks, but the back-off parameters can also influence the high-level scheduling because back-off changes the executing time of application-level transactions.

Space. To synchronize, TM has to ultimately use hardware synchronization operations on some locations in shared memory. The decision where to synchronize in this space is the second important dimension of the TM problem space. Figure 3.2 shows an overview of the possible strategies.

The hardware level is shown at the bottom of the figure, representing shared memory and the standard atomic instructions (e. g., CAS) or HTM. The data residing in shared memory is either application data (on the left side of the figure) or TM synchronization metadata (i. e., data used exclusively by the TM).

Furthermore, we can classify data as allowing for either low-level or high-level synchronization.

Low-level TM synchronization strategies do not depend on high-level information such as language-level information but just need the address and length of an access, as well as whether the acces is a load or store:

- An HTM can synchronize directly on the application data by using the hardware's support for transactions. Hybrid software/hardware TM often accesses TM metadata such as locks using both HTM and standard atomic operations.
- Pure STM typically synchronizes on application data and TM metadata using atomic operations.
- STMs operating on this lower level (e. g., word-based STMs, see Chapter 5) associate TM metadata with application data by mapping addresses in the address space to metadata objects such as locks. This mapping is a key component of such an STM because it has a significant effect on transaction performance.

TM can also try to make use of high-level information, which is either already available for the transactions or can be inferred through automatic analysis. Information that is available and can be exploited (e. g., by a compiler) includes programming language constructs such as types or objects in the high-level source code of the application, or in general any kind of synchronization-related declaration provided by a programmer.

A TM can exploit this information by using it to infer a high-level partitioning of the application data and associating TM metadata with these partitions. For example, in a programming language based on objects, a TM can synchronize per object by embedding TM metadata in each object, or by mapping metadata to objects. This can be extended to any kind of partitioning. Partitions, in turn, can be derived by various other forms of analysis. For example, pointer analysis can cluster memory allocations for a single instance of a data structure into one partition.

I will discuss using partitions in detail in Chapter 6, but its main role is already shown in the figure: It represents an alternative way to associate TM metadata with language-level items such as objects or variables. If a TM does not make use of high-level information but just considers accesses to an address space, it will have to depend on a low-level mapping from the address space to TM metadata, and will additionally be affected by how the compiler and the memory allocator lay out language-level items in the address space. For example, an allocator might place an effectively thread-local and frequently updated variable in the same cacheline as a shared but mostly-read variable; when mapping cachelines to locks, this would result in more conflicts between transactions compared to when putting the two variables in different partitions. In general, finding data for which synchronization is not necessary is an important special case of partitioning (e. g., stack frames created within transactions, or the effectively thread-local variable in the previous example).

A TM can also potentially try to analyze or predict partitions based on low-level information. However, such an analysis would likely be more difficult and imprecise, and thus might have to be paired with enforcing the prediction or catching false predictions in some other way.

3.1.3 TM Performance: Workloads and Optimizations

In general, the performance of TM implementations can be a very complex topic because of the wide variety of workloads (i. e., how an application uses transactions, when they are executed, etc.), the many factors that can affect performance, and the equally many possible approaches to optimizing performance. For example, a TM implementation can be as simple as using a single global lock, which can perform close to optimal on some workloads (e. g., those without any available parallelism) but performs miserable in other situations (e. g., whenever there is disjoint-access parallelism).

Workloads. The amount of available parallelism in a workload is a major factor, obviously. First, the parallelism that the underlying hardware supports is limited (e. g., by the number of CPUs). Second, the application itself will limit parallelism (e. g., by running only a certain number of threads or using algorithms with limited parallelism). Furthermore, synchronization in the application can limit parallelism too, so the level of concurrency also matters (e. g., if using locks, if threads often have to wait because of mutual exclusion with another thread already holding the requested lock).

For TM, we additionally have to consider that transactions can be executed speculatively, rolled back, and restarted. Concurrency control can be optimistic rather than just pessimistic as with typical forms of locking. This can allow for a higher level of concurrency but also requires the TM to make more choices and avoid further performance pitfalls.

The probability of conflicts between transactions can often be used as a metric for the level of concurrency because conflicts require one transaction to be virtually executed before another. However, what eventually counts as a conflict depends on the level of abstraction.

On a high level of abstraction, this is determined by the expected semantics of the transactions; for example, two increments of a value might not conflict when using abstract concurrency control (see Section 2.2). The compiler or execution environment might be able to infer the semantics, but often this requires additional annotations by the programmer; in turn, however, this raises the development effort and thus decreases usability.

Low-level conflicts are usually accesses to the same data item, where at least one of the accesses is a modification. From this perspective, the data access patterns of transactions matter: which application data they access, with loads or stores, and when in the transaction this does happen (early, or close to commit).

Furthermore, the application logic and the additional code required for transactional synchronization share hardware resources during execution. Therefore, for example, properties of the application code such as the size of the cache working set are a factor too because this determines whether usage of additional cachelines by the TM implementation will potentially decrease performance.

Finally, for an application, not all transactions might be equal in terms of performance. For example, are some transactions on the critical path of a parallel execution and should the TM thus give them higher priority and execute them faster than others if possible? Is the application more interested in general throughput of transaction commits or in overall small latencies or fairness? At which resource-usage cost should the TM try to execute transac-

tions? Is reducing energy consumption more important than throughput? The quality–of–service requirements and utility function for TM performance, and thus the answers to the previous questions, are application-specific. Similarly to the transaction semantics, the TM might often not be able to infer those requirements automatically.

Possible optimization approaches. In general, we want to optimize the execution costs, so keeping the additional resource usage required for transactional synchronization low is usually good. For example, this includes computational overhead due to instrumentation of transactions by a compiler, cache footprint for TM metadata, memory access and synchronization costs (see Section 2.1), and wasted work (i. e., transactions that have to be rolled back and restarted). However, if there is or could be parallelism, we also want to exploit that by choosing a scalable execution method and running transactions in parallel. We usually do need to spend hardware resources and energy for that, either for synchronizing memory accesses (e. g., a CAS is more expensive than just writing to the same location) or for optimistic or speculative execution.

Similarly to the variety in workloads, there are plenty of optimization opportunities and workload aspects that can be targeted by optimizations. Therefore, I cannot give a complete overview here but will instead illustrate some possibilities by listing a few dimensions of the design space.

First, which workload scenario and performance utility function do we optimize for? As discussed previously, the latter might not be precisely known. Likewise, automatically detecting which kind of workload will be or is being executed can be quite tricky for the TM at both compile time and runtime.

Second, when do we optimize? Statically at compile or link time, or dynamically at runtime of the transactions? The earlier we attempt to optimize, the less likely it is that we have enough information about the workload, the hardware, and the rest of performance-influencing factors. Consider compilation, for example: We cannot always do just–in–time compilation and without it, we cannot adapt at runtime anymore; however, we also cannot generate custom code for all potential scenarios. In contrast, a TM runtime library can adapt at runtime but it then also has to spend resources to try to infer information about the workload that might still have been easily available at compile time. Additionally, there is a runtime overhead associated with flexible execution (e. g., branches, indirect calls, more bookkeeping) that we would not have if we had code generated specifically for the particular scenario.

Third, which parts of the execution environment do we optimize? Which parts have to contribute TM building blocks? This obviously includes the compiler and a TM runtime library, but do we also need to or try to optimize other libraries or parts of the operating system?

Fourth, which parts of application do we optimize? We obviously have to touch at least some part of the transactions but, for example, do we also want to optimize code surrounding transactions (e. g., to aid speculative execution of transactions)?

Finally, do we choose optimistic or pessimistic approaches of concurrency control? This often is associated with a choice between risk but potential gains versus no risk but being stuck with a fixed and perhaps too pessimistic approach. This also applies to speculative execution: It can be very beneficial but

can also go wrong, and there are overheads associated with containing misspeculation. Speculative execution is also often limited in practice (see Section 4.2.2) because generally, it is typically not safe to communicate speculative values to components that are not under control of nor integrated with the TM.

3.2 Focus and Assumptions

Like any other research project, my thesis was motivated by the state-of-the-art, and evolved in interaction with how the state-of-the-art evolved. When I started work on my thesis, TM was definitely a promising idea but it was not clear whether the TM vision would actually be achievable. Would TM be useful to programmers? Is it an important mechanism compared to other approaches? Can the TM implementation reach its goals (see Section 3.1.1)? This kind of context determined my research focus and the assumptions that I made.

Thus, the most important TM research objective was to study what remains of the potential of TM when it is being translated from a vision to an implementation in a realistic system environment. TM is supposed to make synchronization easier for programmers, so the best test for this is ultimately to provide a TM implementation to programmers and to evaluate the feedback. This also answers what should classify as a realistic system environment: We need to focus on current systems because only there we have a large number of active users and programmers who can provide feedback soon. This will not necessarily reveal the full potential of TM nor every possible show-stopper in future systems, but it will give a reasonable first evaluation. Another aim of this first exploration of the TM vision is of course to learn about the problem space and the relation to other research areas and problems, and to steer future research.

For this reason, my focus was to build a full TM stack that is a best-effort implementation of the TM vision for current systems. This research has both a breadth-first and a depth-first search component. First, what does such a TM stack have to include, which ingredients are necessary? Second, how far towards the TM goals do selected important ingredients bring us? Are they sufficient to reach a goal such as performance, or are they insufficient?

In my thesis, I help answer these questions by first arguing which *building blocks* a first-generation TM stack should consist of, and by implementing and evaluating such a stack. Second, I derived the requirements for TM compilers and TM runtime libraries in such a stack (Chapter 4), investigated how to improve STM performance (Chapter 5), which divide-and-conquer optimizations compiler support can enable (Chapter 6), and how realistic first-generation hardware support for TM can be integrated and used to improve performance (Chapter 7). To keep the required development effort bounded, I focused on a particular class of environment and made a few assumptions, which I will explain next.

Userspace applications. Whereas developers of operating system kernels, standard libraries, or managed environments might be among the first users of HTM, they might not be the first to make use of a general-purpose TM stack because it would come with too many new features to rely on (e. g., language integration). These developers typically can spend more effort per line of code

than what would be reasonable for an ordinary userspace application. Building custom synchronization code might be beneficial or required in a kernel but it often just unnecessarily increases development costs in a platform-independent application.

Thus, focusing on applications will likely yield a wider audience and a better testbed for usability features such as TM semantics and language integration, and for the general-purpose solution that TM is supposed to be. Composability might also matter more for applications and their use of libraries, which are typically authored by several programmers with different skill levels. Nevertheless, the insights gained when building an application-focused TM will also be applicable in kernels and similar environments.

C/C++ instead of managed environments. Validating a TM implementation against potential low-level issues and constraints can yield useful insight into the potential of TM. For example, what are the implications for TM if the application uses manual memory management, or if it calls native code (e. g., in third-party libraries) that accesses the same data that is accessed by transactions? C and C++ allow such low-level interactions in programs, and thus TM specifications and implementations for C/C++ will have to deal with this.

In contrast, languages such as Java do not support such low-level, close-to-the-hardware actions (or hide them behind interfaces such as JNI). The respective managed environments (e. g., a Java virtual machine) typically have a large part of the application under full control (e. g., standard libraries are available in non-platform-specific code and can thus be transformed by the managed environment). Just-in-time compilation is not common for C/C++, so we have to build TMs for C/C++ that must not rely on it to improve performance. TM research results for C/C++ (e. g., TM algorithms) will still be useful for managed environments, even though there might be additional constraints such as requiring integration of TM and automatic memory management.

Standard hardware and first-generation HTM. TM will likely be first adopted on server and desktop multicore microprocessors, so this should be the targeted hardware. In particular, I chose x86-64 because it is most-widely available and supported. PowerPC and SPARC would have worked as well but were either not supported as well in the software I used (i. e., the compiler that I extended with TM support) or I did not have access to such hardware. They can run the same operating systems and toolchains as used for x86-64 though, so the main difference for TM would be differences in the hardware (i. e., memory models and synchronization runtime overheads). ARM is not yet common on servers. The most important consequence of the choice of 64b for TM implementations is that standard atomic operations then can deal with more bits (see Section 2.1), so, for example, counters in TM metadata might never have to deal with overflow. Nevertheless, the software prototypes I built (see Section 3.4) also work on 32b x86.

When considering potential future hardware extensions, these should be realistic proposals for first-generation TM support. Even though it might take rather long before this support is available in real hardware (compared to only on a simulator) and to a large number of programmers, focusing on first-generation hardware is better than expecting advanced and costly hardware support. Inves-

tigating the former can provide valuable insight for hardware manufactures and can help to make TM hardware available, because it faces a chicken–and–egg problem as well.

Compiler support. Integrating TM into programming languages is necessary to achieve usability. Extending C/C++ with the possibility to mark compound statements as transactions (see Section 2.3 for examples), and adding support for this to a compiler is the best choice for this integration. Approaches that do not extend the language do not work well in C/C++ because they are either not transparent (e. g., requiring each transactionally accessed item to be derived from a custom class for transactional state), not powerful enough (e. g., preprocessor macros), or do not compose well with other language features such as exceptions. Regarding the semantics of this language extension, I followed the consensus that emerged in a part of the TM research community (see Section 4.1 for details).

Transforming the memory accesses in transactional code using (dynamic) binary instrumentation is possible [120, 41] but does not provide full integration of TM into the language. Furthermore, this approach likely leads to a larger runtime overhead of the transformed code because native code is more difficult to analyze and rewrite than the intermediate code that a C/C++ compiler uses internally, and a lot of information contained in the source code will have been lost after the initial compilation to native code.

Only basic support for external actions. TM has to compose with existing software such as legacy code and libraries. One important case is that it must be possible for transactions to call functions that have not been transformed by the compiler (e. g., `malloc` or `sqrt`). Some of these functions can be executed as is (e. g., because they do not access any shared state). Others can have wrapper functions that make them compatible with a transactional execution (e. g., `malloc`). If applicable, these properties are expected to be provided along the declarations of such functions. In any other case, the TM has to fall back to nontransactional execution (i. e., serial–irrevocable mode). These options are the basic support that I focused on.

The class of functions that is more difficult to handle is code that communicates with external components (e. g., file I/O functions). A TM would have to provide at least some transactional guarantees for these external actions. For example, it would at least have to roll back updates by a transaction that is to be aborted, even if it cannot guarantee isolation from the effects of external updates. How this can be supported depends on the semantics of the external component.[1] In principal, one has to rely on techniques known from distributed databases and other distributed systems that provide atomic snapshots and distributed commits. I have built such a framework to support external actions (similar work was published concurrently by Volos et al. [115]), and such a framework is sufficient for many of the functions provided by the standard C library. However, supporting actions on more complex external components quickly becomes highly dependent on these components (e. g., the operating system or conventions regarding external resources), and some actions just cannot be rolled back or executed in isolation without extending the features of

[1] Therefore, dynamic binary instrumentation alone would not be sufficient.

the external component. Furthermore, classic forms of error handling (e. g., returning error codes from functions) are a problem because they come from a sequential perspective that does not consider distributed or speculative execution. Thus, new ways of error handling [91] might be necessary for a proper language integration of transactions and external actions.

No assumptions about the kind of parallelization. The use cases of TM that will matter in the future are not yet known. There are several ways of parallelization, which differ in how they use synchronization and thus the transaction workload that they request the TM to execute.

For current TM research, focusing on explicit threading as the kind of parallelization used in applications promises to yield the most generally applicable results. In this case, threads can execute transactions whenever they want. Thus, a TM can be exposed to any kind of workload, provided that the available TM applications (e. g., benchmarks) actually are different. The TM cannot expect that it has additional information about the workload available (e. g., transaction priorities, or whether a transaction will execute the same operations when it is restarted). It also cannot expect to control the workload beyond simply delaying execution (e. g., it can execute only one thread's transaction but it cannot change the order in which a thread executes transactions).

Making none of these assumptions about the kind of parallelization ensures that the breadth of the research on the potential of TM is not restricted. For example, if TM works well with explicit threading, it is likely to also work well when used inside of OpenMP tasks. Of course, this will not reveal the perhaps extended potential that TM might have in particular use cases, nor how TM would compare against other custom synchronization solutions for these cases.

General-purpose performance optimizations. Even without any assumptions about the workload and kind of parallelization, one has to select which kinds of workloads should be the first targets for performance optimizations. Based on how synchronization is used currently, it is nevertheless reasonable to expect that a few properties are common:

- Transactions often load from many more memory locations than they write to. Therefore, transactional loads represent a large part of the runtime overhead of TM and should be as efficient as possible.
- Typically, accessing disjoint data in parallel results in less overhead than accessing the same data. Thus, a TM should try to keep up this benefit and exploit disjoint-data parallelism in workloads. This also extends to avoiding overheads for concurrent read accesses to the same data.
- Synchronization is associated with runtime overheads, so if programmers try to optimize transactions, they likely will make them rather small than large in terms of the number of operations.

As I explained in Section 3.1.2, there are many factors which have an influence on the performance of synchronization, and they are typically not isolated from each other but interfere in various ways. Therefore, my focus was on finding performance optimizations that are fairly robust towards variations in these factors and that promise to be useful in future TM implementations as well.

One further consequence of not having information about the kind of parallelization is that the TM does not know the precise performance utility function (see Section 3.1.3). The performance goal that I focused on is that the TM should target high throughput while trying to avoid starvation of individual transactions. Blocking TM implementations are allowed, transactions do not have different priorities, and increasing fault tolerance [116] is not an objective.

Without this information, it is difficult to build generally useful and applicable optimizations related to the temporal aspect of TM-based synchronization (see Figure 3.1). Therefore, I chose to not focus on the temporal aspect (except minimizing runtime overheads) because it seems to be better to investigate this in scenarios in which more is known about the parallelization approach. Furthermore, if the TM should also be able to control scheduling outside of the TM library (e. g., thread scheduling in the operating system [76]), it has to depend on custom support in the operating system or the application. Even just pinning threads to different CPUs could interfere with application-level assumptions in unexpected ways.

With respect to the spatial aspect of TM-based synchronization (Figure 3.2), I focused on both low-level and high-level approaches. The former is necessary because one cannot expect the compiler to always be able to analyze a C/C++ program. Neverheless, trying to exploit high-level information can be very beneficial, as explained in Section 3.1.2.

I chose to not expect programmers to provide detailed programming-language–level hints aimed at performance optimizations because this decreases TM usability.

I also do not provide any models for workloads or TM performance that would be sufficiently precise to be the base for robust automatic tuning and adaption at runtime. Without any assumptions about the kind of parallelization and no programmer-supplied hints, it is difficult to correctly analyze what kind of workload is being executed and to predict which effects a certain tuning decision would have. Applying rule–of–thumb tuning and heuristics is of course still possible and can be beneficial [108], but has not been a major part of my research focus. Proper models for more advanced decisions seem to have to be somewhat complex, and thus it is likely better to investigate them after more widely use of TM has shown which kinds of workloads are actually realistic and matter in practice.

3.2.1 Building Blocks: Summary

To summarize, the focus for this thesis are (1) C/C++ userspace applications, (2) standard hardware and realistic first-generation HTM, (3) explicit threading as an approximation of no assumptions about the kind of parallelization used in the applications, (4) general-purpose performance optimizations including compiler support, and (5) no custom support for TM by other parts of the execution environment such as the operating system and no advanced programmer-supplied optimization hints.

Thus, the most important system layers for the TM stack are the compiler, the TM runtime library, and the hardware that is used to execute the applications. I built software prototypes for the compiler and the TM runtime (see Section 3.4), whereas the hardware is just used by the former two layers and not modified.

The TM stack consists of the following main building blocks, which will be discussed in detail in the subsequent chapters:

Programming-language integration. The compiler has to map transactions in C/C++ source code to code that calls a TM runtime library, which raises the following questions: How do we map language-level semantics to an implementation? Which part of that is the responsibility of the compiler, and which part of it has to be handled by the TM runtime library? What are the precise guarantees that the TM runtime library has to provide? These questions will be answered in Chapter 4.

STM algorithms and implementations. The TM runtime library needs to implement the guarantees that are expected by the compiled code that is using the library. This has to be efficient, and run on current hardware. Both algorithmic and implementation aspects of this will be covered in Chapter 5, with a special focus on making transactional read operations efficient.

Compile-time optimizations. While standard compilation targets a TM runtime library that synchronizes on the lower level of the spatial aspect of TM-based synchronization (see Figure 3.2), I will explain in Chapter 6 how compilers can use compile-time analysis of the program code to provide the library with high-level information about the transaction. This extends the standard library interfaces discussed in Chapter 4 and allows the library to execute transactions more efficiently.

HTM integration. It should be possible to use first-generation hardware support for TM once it becomes available. Therefore, it is necessary for the other building blocks to compose well with HTM, which I will discuss in Chapter 7 based on an HTM proposal by AMD. We also want to achieve good performance despite the potential limitations of such early HTM implementations, so I will also present efficient hybrid STM/HTM algorithms.

Together, these building blocks comprise a working first-generation TM stack, which provides powerful programming-language constructs for transactions and shows decent performance on current hardware and likely on hardware with first-generation HTM as well. All of these building blocks except the compile-time optimizations are essential for a general-purpose TM system for the set of assumptions that I have chosen, either for usability or for performance reasons.

Of course, there are other features that could be good to have in a TM, especially to improve performance: For example, better scheduling of transactions (e. g., contention management), a tighter integration with parallelization frameworks, and automatic tuning and adaption at runtime. Nonetheless, these features would also require the building blocks I have selected, at the very least to exploit their full potential.

3.3 Related Work

Only few research projects have investigated a full set of TM building blocks (i. e., as outlined in Section 3.2.1), especially when compared to the overall

amount of TM-related research in the past years. Many investigations cover only individual pieces such as specific STM or HTM algorithms; while these are valuable contributions, they can only partially reveal how TMs need to be designed to be deployed as general-purpose programming tools for production-level code.

The related work that is the closest to my set of building blocks are two C/C++ TM stacks that essentially provide the same programming language constructs for transactions and use roughly the same ABI. Both are implementations by industry and aimed at production-level deployments (even though the implementations are still considered experimental).

The first of the two is Intel's work on TM support for C/C++ as part of their compiler prototypes [83], which has evolved over the years, concurrently with my research. They had a large influence on the shape of the C/C++ TM specification (see Section 4), and the ABI used by my software prototypes and by GCC (see below) was initially proposed by them. They use a time-based STM based on the algorithms that I describe in Section 5. The major contributions that my work provides compared to theirs are a detailed description of the semantics of the ABI including the split of responsibilities between compilers and TM runtime libraries (see Section 4.2), whole-program compiler optimizations (see Section 6), and more advanced HyTM algorithms as well as a detailed study of HTM integration (see Section 7). See Sections 5.4 and 6.4 for more information about their work.

The second of the two similar stacks is the TM support by the GNU Compiler Collection (GCC) [44]. It consists of support for the C/C++ TM specification in the C and C++ compilers and a TM runtime library, libitm. The former was significantly influenced by the explanations of the requirements on compilers that I present in Section 4.2 (e. g., how to not violate publication safety). The latter essentially follows TinySTM++'s design (see Section 3.4.2): It implements a subset of the same TM algorithms (e. g., Algorithm 3) but with a few minor changes aimed at production-level deployments (e. g., it can dispatch ABI calls to different TM algorithm implementations at runtime). DTMC (see Section 3.4.1) uses a port of the TM support in the frontend of GCC's C/C++ compiler to parse TM constructs such as transaction statements.

Outside of C/C++, early work by Intel [1] and Microsoft [51] was the first to investigate TM building blocks for Java. Both show that the overhead of object-based STMs in managed enviroments can be quite small. Their compilers employ STM-specific optimizations and the STMs are tightly integrated with the respective Java runtime environments. However, because of relying on a managed environment and using a language like Java, they can use optimizations that are not practical in C/C++ environments (e. g., not guaranteeing the consistency of a transaction during its runtime but only when it commits).

Microsoft's STM.NET is another set of TM building blocks for a managed environment [112], with the notable distinction that it also tried to integrate other transactional services with the transactions support at the programming-language level. STM.NET does not seem to have been made available as part of any product so far.

The Velox TM stack [3] is the collection of the outcomes of a larger research project that I participated in. Some of the building blocks and software prototypes that I present in this thesis are part of the Velox stack (e. g., Section 7.2 or DTMC).

To improve readability, I will discuss related work specific to certain building blocks along with the presentation of those building blocks (i. e., in Sections 4.3, 5.4, 6.4, 7.2.5 and 7.3.3).

3.4 Software Prototypes

To be able to validate TM building blocks both in terms of performance properties and the overall design (e. g., interface design decisions), it is necessary to build prototypes of a full software stack containing those building blocks. The software that I built is such a stack; while it might not have all the polish that would be required for production-level deployments, it is sufficiently close to that to ensure that the findings will continue to apply to implementations ready for production-level deployments. This is important because otherwise, studies like the full-system HTM evaluation discussed in Section 7.2 would not be meaningful.

Therefore, I will next give an overview of both the TM support for C/C++ compilers (DTMC) and the TM runtime library (TinySTM++) that I built. Both are availabe under an open-source license.[2]

In Section 3.4.3, I will also describe the aspects of the experimental setup that are common across the different evaluations in the subsequent chapters.

3.4.1 DTMC

The Dresden TM Compiler (DTMC) relies on both the GNU Compiler Collection (GCC) and the LLVM compiler infrastructure [71] to compile C/C++ code with transactional language constructs as described in Section 4.1 into executable code that targets the TM runtime library ABI described in Section 4.2.1. This is a multi-pass process.

First, LLVM's compiler front-end for C/C++ (`llvm-gcc`) parses and transforms source code into LLVM's intermediate representation (IR). To support transactional language constructs, Martin Nowack ported the TM support code in the former TM branch of GCC to `llvm-gcc`; this translates constructs such as transaction statements into blocks of transaction code that are demarcated from nontransactional code by calls to special marker functions. In the initial version of DTMC [41], which just consisted of the the LLVM transformation pass described next, the programmer had to provide those marker calls instead of being able to rely on transaction statements[3]. In either case, the output of the compiler's front-end is thus LLVM IR in which transaction statements exist as demarcated blocks of code.

Second, transactional code is transformed so that it uses the TM runtime library. This happens as one transformation pass in LLVM's middle-end (i. e., embedded within the sequence of LLVM's general-purpose optimization passes on LLVM IR). LLVM can optimize whole programs by first transforming all compilation units (i. e., individual C/C++ source code files) into LLVM IR and then merging these into a single LLVM IR representation of the whole program. This is very convenient for TM transformations and optimizations, so DTMC's

[2] The source code can be obtained at `http://se.inf.tu-dresden.de`, for example.

[3] For example, instead of **__transaction_atomic** { x++; }, the programmer had to write DTMC_BEGIN(); x++; DTMC_COMMIT();

compiler driver tool (`tmlink`) runs the TM transformation pass on the whole program's IR code.

This pass then first analyzes the control-flow graph of the program and finds all basic blocks that could be executed from within transactions: (1) basic blocks lexically contained within transaction statements, (2) functions marked as transaction_safe, and (3) all functions that could be called from transactional code. The latter functions get cloned so that these transactional clones can contain the instrumentation required to use a TM runtime library; however, if such functions are marked with the transaction_pure or tm_wrapper attributes, then they do not get cloned and can either be used unmodified in transactional code or have a specialized transactional version, respectively. As second step, all transactional code is transformed by (1) replacing memory accesses with calls to matching load and store functions in the TM runtime library, (2) redirecting function calls to the transactional clones, and (3) adding the calls to TM runtime library functions that are necessary to start or commit transactions.

DTMC can create several code paths for each transaction, whose purpose is to have different kinds of TM instrumentations available for each transaction. For example, for HyTMs (see Section 7.3), it is beneficial to have one code path for transactions that execute with help of the HTM and another code path for transactions that run the STM fallback. While this would also be possible with dispatching between both ways at runtime during each call to the TM runtime library, having specialized code paths available reduces runtime overheads significantly and allows the compiler to optimize more aggressively. The instrumentations on each path do not differ except that they target differently named sets of the base ABI functions (e. g., the hardware transaction code path calls functions whose name is prefixed with "hytm", whereas the STM code path uses the default ABI names). The compiler informs the TM runtime library which code paths are available when starting a transaction (using a parameter to _ITM_beginTransaction), and the library then chooses which code path should be executed (using bits in the return value of _ITM_beginTransaction). Figure 4.4 on page 51 shows an example. The library can make this choice on each attempt to execute the transaction, including on restarts. The user can decide which code paths should be generated; by default, `tmlink` uses a configuration file provided by TM runtime libraries to make this selection.

Finally, `tmlink` links the TM runtime library statically with the transformed program if the former is available as LLVM IR. This happens right before the LLVM code generator transforms LLVM IR into code executable on the target architecture, so whole-program optimizations can be applied to the composit of program and the TM runtime library. For example, this allows inlining of the library's load and store functions, which in combination with having specialized code paths, can lead to highly optimized code despite the TM being available just as a library. The example in Figure 7.7 on page 178 shows that his can result in code of the same quality as if the compiler inserted the TM instrumentation code directly.

3.4.2 TinySTM++

TinySTM++ is a framework for TM runtime libraries that eases experimenting with different STM and HyTM implementations. It contains all the functionality required to build ABI-compliant TM runtime libraries (see Section 4.2.4) and

allows implementations of TM algorithms to make use of generic components as necessary (e. g., there is generic support for serial-irrevocable mode as explained in Section 5.2). It is written in C++ and uses templates extensively to allow for a high degree of code re-use while at the same time guaranteeing that no unnecessary runtime overheads are introduced despite the large flexibility of the framework. For example, the same implementation of Algorithm 3 can be instantiated as a pure STM runtime library as well as a HyTM when combined with an implementation of Algorithm 10. TM runtime libraries can support several code paths as explained in Section 3.4.1 (e. g., Algorithms 3 and 10 could be used by different code paths in a HyTM).

TinySTM++ hosts the implementations of the algorithms presented in Section 5.2 and Chapter 7. The implementations of the algorithms presented in Chapter 6 are based on a older version of TinySTM [42].

There are a few implementation properties that are worth pointing out. First, the implementation tries to avoid costly cache misses due to false sharing on cache lines (e. g., TM metadata is split into thread-private and globally accessible parts, and padding is used to ensure that independent but frequently accessed TM metadata parts are on separate cache lines). Second, all STMs do not use any kind of contention management or exponential back-off on transaction abort; instead, transactions are restarted immediately. This is certainly a limitation of the implementation, but falls into the temporal aspect of TM-based synchronization and is thus out of the scope of my work (see Section 3.2). However, all ASF-based TMs (1) use a simple back-off scheme on aborts on the hardware transaction code path[4], (2) switch a transaction to the fall-back execution method (i. e., software transactions or serial-irrevocable mode) after 100 aborts due to conflicts with other threads[5], and (3) ignore the first abort due to exceeding HTM capacity if only serial-irrevocable mode is available as fall-back execution mode. Similarly, with the LSA STMs evaluated in Section 5.2.2, transactions switch to serial-irrevocable mode after 100 aborts to avoid potential livelocks. Note that the STM implementations used in Chapters 6 and 7 do not perform such a switch to a fall-back execution method. With the exception of the partitioning-aware STMs in Section 6.2, all orec-based STMs map from memory locations to orecs using fixed hash functions and hash function parameters (i. e., the parameters are constants in the executable code), but arrays of orecs are allocated dynamically (i. e., the TM runtime library has to load the pointer to the array on each transactional memory access).

Finally, TinySTM++ uses only spinlocks (i. e., locks that never block using facilities of the OS but instead just poll the lock's value in a loop until it is not acquired anymore). Note that this is not a general limitation in the algorithms but rather an implementation issue that would be important in production-level deployments; although the decision whether to spin or block using OS facilities could take properties of the TM into account, the underlying problem is independent of TM and is an issue that all lock implementations face. Therefore, I consider this to be out of the scope of my work.

[4] Before restarting, hardware transactions wait for a certain amount of time chosen randomly from the range of zero to the time the transaction tried to execute unsuccessfully so far.

[5] This counter is maintained per thread and reset whenever a transaction commits.

Benchmark	Update transactions	Average number of elements	Element value range
SkipList-Large	20%	4096	8192
SkipList-Small	20%	512	1024
RBTree-Large	20%	4096	8192
RBTree-Small	20%	512	1024
LinkedList-Large	20%	256	512
LinkedList-Small	20%	14	28
HashTable	100%	64000	128000

Table 3.1: IntegerSet benchmark configurations.

3.4.3 Notes about the Experimental Evaluation

The experiments conducted for the performance evaluations in Chapters 5 to 7 have several things in common, which I will describe in this section (e. g., the selection of benchmarks). However, the experiments differ in other aspects such as the hardware they have been executed on, which I will described in the sections about the respective experiments.

The lack of available TM benchmarks is a big problem for TM research in general. The STAMP TM benchmark suite [11] is basically the only set of C/C++ benchmarks that is freely available, used widely by the TM community, and is not meant to test pathological workloads or corner cases. Given that TM has so far seen little use by C/C++ programmers—or at least that those programmers have not made their uses public, it is also not possible to just take existing non-benchmark programs that use transactions and transform them into benchmarks. There are other applications with transactions that have been used by research groups as experimental base for publications, such as those used in a study [84] by Pankratius and Adl-Tabatabai, but those have not been made publicly available. Besides the benchmarks that try to resemble real programs, there also exist microbenchmarks that typically test TM performance when transactions are used to synchronize access to shared data structures. A frequently used group of such microbenchmarks tests concurrent operations on different implementations of sets of integers. Therefore, and due to the lack of better candidates, I use both the STAMP and the integer set benchmarks.

IntegerSet benchmarks. The transactional workload that is created by these benchmarks is a sorted set of integer values that is accessed and modified by several threads. Each thread continuously runs a transaction that either inserts a new element into the set, removes an element from the set, or tests whether a certain element is contained in the set. This set is not a multiset, so a new element is only inserted if it is not yet present in the set; thus, insert and remove operations are not guaranteed to perform transactional write operations.

Each of the IntegerSet benchmarks implements the set either using a skip list, a red-black tree, a sorted linked list, or a hash table. The skip list uses at most 8 levels. The hash table uses open hashing, 2^{17} buckets, and a multiplicative hash function; each element in the hash table resides in a separate list node

Benchmark	Comments	Benchmark parameters
Genome	16M segments	`-g16384 -s64 -n16777216`
	Sim: 16K segm.	`-g256 -s16 -n16384`
Genome-4M	4M segments	`-g16384 -s64 -n4194304`
Genome-8M	8M segments	`-g16384 -s64 -n8388608`
KMeans-Lo	64K input	`-m40 -n40 -t0.00001 -i random-n65536-d32-c16.txt`
	Sim: 2K input	`-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt`
KMeans-Hi	64K input	`-m15 -n15 -t0.00001 -i random-n65536-d32-c16.txt`
	Sim: 2K input	`-m15 -n15 -t0.05 -i random-n2048-d16-c16.txt`
Vacation-Lo	4M transactions	`-n2 -q90 -u98 -r1048576 -t4194304`
	Sim: 4K txns.	`-n2 -q90 -u98 -r16384 -t4096`
Vacation-Hi	4M transactions	`-n4 -q60 -u90 -r1048576 -t4194304`
	Sim: 4K txns.	`-n4 -q60 -u90 -r16384 -t4096`
Vacation-1M	1M transactions	`-n10 -q90 -u80 -r65536 -t1048576`
Vacation-2M	2M transactions	`-n10 -q90 -u80 -r65536 -t2097152`
Vacation-2M-Lo	2M transactions	`-n2 -q90 -u98 -r1048576 -t2097152`
Vacation-2M-Hi	2M transactions	`-n4 -q60 -u90 -r1048576 -t2097152`
SSCA2		`-s20 -i1.0 -u1.0 -l3 -p3`
	Sim:	`-s13 -i1.0 -u1.0 -l3 -p3`

Table 3.2: STAMP benchmark configurations. The configurations annotated as "Sim" in the second column are used for experiments executed in a simulator (see Chapter 7), all other configurations are used by non-simulated executions.

referenced by one of the buckets, so adding an element to the table requires one call of malloc to dynamically allocate memory.

Table 3.1 shows the benchmark configurations that I use. The operations performed by each transaction are chosen randomly such that the potentially updating transactions with insert and remove operations occur with the probability shown in the second column of the table; insert and remove operations always occur with the same probability. Note that as discussed previously, the probability of transactions that actually modify state might be lower (but element lookups are always read-only transactions). The values of elements used as arguments to operations are chosen randomly in the range of zero to the value shown in the right-most column of the table. During benchmark initialization, the integer sets are populated with half as many elements as the upper bound of the range from which their values are picked; therefore, the number of elements in the sets will remain roughly constant during the execution of the benchmark.

Each benchmark is executed for five seconds except if a simulator is used to execute experiments as in Chapter 7, in which case the benchmarks perform a fixed and lower number of operations to keep simulation time reasonable. Finally, the implementations of the IntegerSet benchmarks used to evaluate TM metadata colocation differ slightly (see Section 6.3.2 for details).

STAMP benchmarks. The STAMP benchmarks that I selected for my experiments are Genome, KMeans, Vacation, and SSCA2. Genome's transactions access a mix of pointer data structures and a large number of character strings.

Note that STAMP's manual instrumentation of memory accesses in transactions does not treat the frequent string comparisons in the benchmark's transactions as transactional accesses because these strings do not change in the respective phase of the benchmark. Thus, this is a programmer-supplied optimization, which is not available to TM compilers such as DTMC. This increases the number of transactionally accessed memory locations in each transaction compared to what has been reported for STAMP originally. Genome also uses almost 1.5GB of memory on 64b systems with its default non-simulator configuration. KMeans mostly operates on arrays of primitive types, has rather short transactions that access few memory locations, and spends only little of its total execution time in transactions. Vacation simulates a reservation system; its transactions mostly operate on a couple of red-black trees and linked lists. SSCA2's transactions are used to build graph data structures in parallel that are implemented using arrays; it also spends only little of its total execution time in transactions.

Table 3.2 shows the benchmark configurations that I use. The configurations annotated as "Sim" are used for the experiments in Chapter 7; they have shorter execution times, which keeps simulation time reasonable.

The implementation of these benchmarks is based on STAMP version 0.9.6 but contains a few modifications and bug fixes. First, the original STAMP used map data structures with 32b integer keys and values also to map from or to pointers—these data structures were changed to instead use pointers as keys and values. This allows the STAMP benchmarks to also work correctly when compiled as 64b-pointer programs, and improves the quality of the compiler analyses used in Chapter 6. Second, the barriers that control when application threads start to execute transactions have been changed to a spinning implementation, which allows for more precise measurements of the execution time when used with a simulator; the barriers now also work with thread counts that are not a power-of-two value. Finally, some of the entry points to the main data structures in the benchmarks were placed on separate cache lines to avoid unnecessary hardware transaction false conflicts.

Other benchmark, software, and execution parameters. The specific hardware or simulator used for the experiments in Chapters 5 to 7 are different and described in these chapters. Notwithstanding, benchmarks are never executed with more threads than logical CPUs provided by the hardware because TinySTM++'s implementation only uses spinlocks; as explained previously, this is not a general limitation, nor is it a case that would be very important in practice. Threads are pinned to logical CPUs in the STM experiments (see Section 5.2.2 for details) and the HTM experiments (see Section 7.4; threads are pinned to cores of the simulated CPU).

All performance measurements have been executed several times, and the data shown is the average of the individual measurement results.

As mentioned previously, the TM runtime libraries are statically linked to the benchmarks, and link-time optimizations are enabled (using the default LLVM optimization passes). Chapters 5 and 7 use the same DTMC version (i. e., based on LLVM 2.8) and the most recent TinySTM++ and 64b benchmark applications. The benchmarks in Chapter 6 use an older DTMC version based on LLVM 2.1, an older TinySTM version, and are 32b programs. The glibc

version used to execute experiments is 2.15 in Chapter 5, 2.10 in Chapter 7, and 2.3.6 in Chapter 6. All benchmarks use glibc's default memory allocator except the HashTable experiments of Chapter 7, which use the Hoard memory allocator [8].

Chapter 4

Integrating TM with C/C++ Programs

Language integration is essential for TM to provide usability and composability. Requiring programmers to manually instrument the program code to use TM like a library would be an obstacle to reaching those goals. Instead, programmers should be able to just demarcate which regions in the code are supposed to be transactions, and have the compiler automatically transform these code regions. A simple option to demarcate regions are calls to special marker functions:

```
DTMC_BEGIN(); x++; DTMC_COMMIT();
```

This is fairly straightforward to implement in a compiler and also was what I had chosen to do in early versions of DTMC. It does not require extensions to the programming language but has shortcomings exactly because of that. First, the compiler would still have to be aware of the program's source code during the transformations because otherwise, it cannot precisely detect what is transactional code (e. g., if transforming intermediate code created by other parts of the compiler). Second, it still needs to be specified how transactions interact with other language components such as exception handling. Overall, the compiler's frontend has to support TM and we need an extended language specification.

Thus, one can also extend the language in the first place because it would need a similar kind of compiler support and level of understanding by the programmers. The resulting lack of compatibility with older compilers and related tools is likely to be outweighed by the increased clarity in both syntax and semantics of programs with transactions.

The central TM language construct in C/C++ are transaction statements, which are executed as a transaction and consist of either a single statement or a block of statements (see Section 2.3 for examples). To specify their semantics and to implement support for them, we need to also consider other specifications and interfaces, which are shown in Figure 4.1.

The programming language's memory model, which defines how programs access memory and how multi-threaded programs synchronize concurrent accesses, has to be extended with the semantics of transactions. An extended memory model then specifies the orderings guaranteed by transactions and how they are related to the rest of the model. One particular TM concern is how

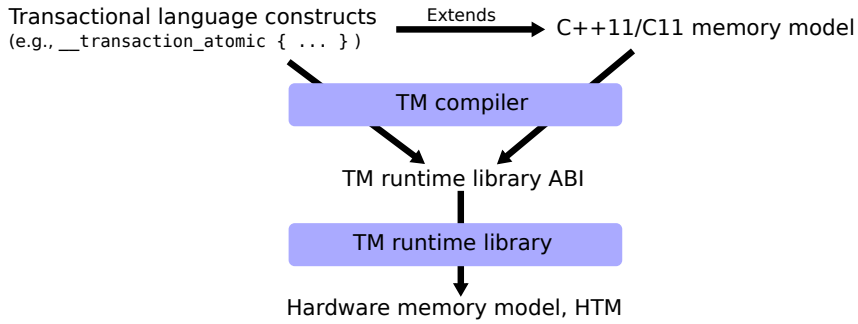

Figure 4.1: Relation of the TM specification to other interfaces.

transactional memory accesses interact with nontransactional memory accesses, covering questions such as strong versus weak isolation or publication and privatization.

For reasons further explained in Section 4.2, we want the compiler support to be fairly independent of actual TM implementations, and rather let the compiler target a common Application Binary Interface (ABI) as intermediate interface. TM runtime libraries then implement this ABI for some set of hardware architectures, which also provide different hardware memory models. The compiler and the TM runtime libraries thus have to jointly implement the C++ memory model extended with transactions, on top of the memory models provided by hardware architectures (e. g., by using hardware instructions with memory barrier semantics to ensure certain orderings required at the programmming language level).

4.1 Specifying TM for C/C++

In this section, I will explain a draft specification of transactional language constructs for C++ [63] ("specification" for short), which is a joint effort by several companies (HP, Intel, IBM, Oracle, and Red Hat) and individuals, including people working on the major C++ compilers. It is based on and extends the C++11 standard [65]. I will also describe the extensions to the C++11 memory model [6, 10] that are necessary for the additional TM language constructs. I will then, in the following section, derive what this specification actually means in terms of requirements on a TM compiler and runtime library. Finally, I will compare this specification to other approaches for modeling the correctness of concurrent operations in Section 4.3.

Overview of the C++ memory model. To put it simply, the memory model uses per-thread program order together with the synchronization relations present in the program to derive a *happens–before* relation, which then also describes how memory locations are modified and which values are read by load operations. Programs have to be free of data races (i. e., all accesses must be properly ordered by *happens–before*); if they are not, then the behavior of the program is undefined. Thus, the memory model defines which executions are

allowed for a given program and how a program's threads can synchronize with each other.

From a formal perspective [6], the model consists of several relations that are either derived from a program's source code (and thus are fixed for a given program and control flow path), or can be chosen freely to represent the different executions a program might have (i. e., to model the indeterminism that arises when executing a multi-threaded C++ program).

First, the *sequenced–before*, *data–dependency*, and *additional–synchronizes–with* relations are determined by the program's source code and the assumed control flow in the program. *sequenced–before* basically is the per-thread order of operations in the program, and thus takes a central role. The other two relations are not as important to understand how TM fits into the memory model (e. g., *additional–synchronizes–with* models orderings such as between a child thread and the operation in the parent thread that created them).

Second, *reads–from*, *modification–order*, and *sequentially–consistent* are "witness relations" that represent the relation or ordering between accesses by several threads. The relations are the witnesses of some chosen execution and are used to enumerate all possible executions. *modification–order* orders operations that modify the same location, and *reads–from* defines which modification to a location is observed by a reading operation to the same location. Locations can either be nonatomic (normal program state), atomic (accessible by atomic operations such as CAS), or mutexes. Atomic operations accept an additional *memory order* modifier that affects the strength of its synchronization, including which other operations it potentially synchronizes with (e. g., memory_order_relaxed does not impose an order, whereas memory_order_seq_cst is stronger). *sequentially–consistent* orders sequentially consistent operations (e. g., locking and unlocking a mutex).

Together, these six relations can be used to enumerate the possible candidate executions of a program. The model also uses them to derive other relations, of which *synchronizes–with* and *happens–before* are the most important ones from the perspective of our discussion. *synchronizes–with* contains the order that is enforced by synchronization operations in the program: for example, two atomic operations in different threads on the same memory location, one a store with memory_order_release and the other a load with memory_order_acquire, result in a *synchronizes–with* edge from the store to the load (see Table 2.1 for the notations I use in algorithms). *happens–before* is, roughly speaking, the transitive closure of *synchronizes–with* and *sequenced–before* (it is not transitive though for chains including a certain kind of memory order modifier on atomic operations). Thus, *happens–before* is the top-most specification of ordering in some execution of a program, but it is not a total order.

Next, these six relations and the derived relations are used to determine which candidate executions are consistent. The precise consistency constraints are described in the formal model and I will not explain them in detail. However, they are pretty intuitive and basically constraints on the various relations and combinations of them; for example, *reads–from* must be consistent with *modification–order* and *happens–before* in that reads observe the most recent value written to a location according to the orderings specified by *happens–before* and *modification–order*. Any inconsistent candidate executions are not further considered.

Finally, if *any* of the consistent candidate executions contains some form of

race condition (e. g., data races due to conflicting accesses by different threads that are not ordered by *happens–before*), then the behavior of the program is undefined; otherwise, behavior will be equal to one of the candidate executions. Note that this also highlights the "catch fire" semantics of race conditions and undefined behavior. In particular, if there can be a case where a program is not race-free, then the program's execution enviroment (e. g., the compiler or the TM runtime library) does not even need to ensure that all other candidate executions would execute correctly.

Programmers thus have to essentially use the right amount of synchronization in their programs to prevent data races and other incorrect yet race-free behavior. They can do that using ordered atomic operations, locks, or also transactions as we will see later. The compiler and all other parts in the C++ execution enviroment, including TM runtime libraries, are responsible for translating race-free source programs into native code that is also race-free and only yields executions that are equivalent to consistent candidate executions in the model. This can restrict compiler transformations (e. g., code movement across operations that contribute to *synchronizes–with*), and the compiler's code generator and the runtime libraries have to use suitable native code (e. g., memory barriers) that correctly implements the C++ memory model on top of the memory model provided by the targeted hardware platform.

Please also note that the memory model relies on ordering as expressed in *happens–before* and not on linearizability, which is commonly used to reason about concurrent data structures (see Section 2.1). Furthermore, the C++ standard is rather vague regarding progress guarantees (e. g., it allows I/O operations to actually finish after the respective I/O function has returned, which would not be a linearizable operation). However, if the operations of a concurrent data structure are indeed linearizable, then ordering in *happens–before* also becomes straightforward.

TM Language Constructs. The main transactional language construct are *transaction statements*, consisting of either the **__transaction_atomic** keyword or the **__transaction_relaxed** keyword followed by a compound statement (see Section 2.3 for examples). Alternatively, transactions can also have the form of *transaction expressions* or *function transaction blocks*. The former make parenthesized expressions transactional, whereas the latter execute whole function bodies as a transaction; both can be expressed with transaction statements, so I will not consider them further.

Informally, atomic transactions (using the **__transaction_atomic** keyword) can be thought of as executing instantaneously (i. e., atomically and in isolation from other threads) if there are no race conditions with other nontransactional operations. These transactions are called *atomic transactions* and can only execute code that can execute safely in an atomic transaction or can be transformed so that it is *safe code*.

Alternatively, transactions can also be annotated as *relaxed transactions* (using the **__transaction_relaxed** keyword) and can then execute unsafe code. Examples for unsafe code are accesses to volatile memory locations or C++ atomic variables, file I/O, or functions in libraries only available as native code. Thus, relaxed transactions do not provide full atomicity but are atomic only with respect to other atomic or relaxed transactions; in contrast to atomic transactions,

they can communicate with other threads from within the transaction via unsafe code. In a typical STM implementation, those relaxed transactions that execute unsafe code will use mutual exclusion to prevent any other transaction from running concurrently, which can result in significantly less scalability. Note that what the specification defines as unsafe code is not necessarily inherently unsafe for atomic transaction, but rather a trade-off influenced by several factors such as implementation concerns and performance overheads for nontransactional code.

Programmers can declare code as safe or unsafe using the transaction_safe and transaction_unsafe attributes, and the compiler can declare code implicitely safe if it can analyze that it is indeed safe. The specification also allows other implementation-defined mechanisms that can declare code as safe (e. g., the transaction_pure and tm_wrapper attributes discussed later). Safe and unsafe declarations have to be used consistently, and compile-time errors will be raised if this not the case or if atomic transactions could potentially execute unsafe code.

Therefore, while atomic transactions provide stronger guarantees to programmers because of the compiler checking that they will never execute unsafe code, relaxed transactions allow all code to be potentially executed from within a transaction, which can aid usability and composability.

The specification describes a few further language constructs. First, transactions can be *canceled* using the **__transaction_cancel** statement, which rolls back the current transaction and continues execution after the transaction (i. e., it basically skips execution of the transaction). A programmer can additionally request that an exception should be thrown after canceling the transaction. Second, programmers can declare whether a transaction is allowed to throw exceptions or not. To preserve existing exception semantics, throwing exceptions from within a transaction will not automatically cancel this transaction.

Transactions can be nested within other transactions. One can distinguish between flat, closed, and open nesting: Flat nesting treats the outer-most transaction and all nested transaction as a single unit, committing or aborting all of them together. With closed nesting, nested transactions can or should roll back without rolling back enclosing transactions. Open nesting allows nested transactions to commit separately from the enclosing transactions, which makes possible semantics more difficult. The specification focuses on flat nesting and does not make use of open nesting. It indicates that closed nesting should be used when transactions are canceled (unless specified otherwise), but does not explicitely forbid flat nesting or spurious aborts (i. e., no explicit progress requirements).

TM Extensions to the C++ Memory Model. To understand what it actually means if a transaction appears to execute instantaneously in the absence of race conditions, we need to look at both the definition of C++ race conditions and the TM-specific ordering guarantees that yield instantaneous execution.

The specification introduces additional transaction begin ("StartTransaction") and commit ("StopTransaction") operations to demarcate transactional code. These operations are purely used to express transactions as part of the C++ memory model (e. g., in *sequenced–before*), and are only associated with outermost transactions but not with nested ones. Code sequenced before a

```
// Thread 1

__transaction_atomic {
  temp1 = x;
  temp2 = y;
}
```

```
// Thread 2
y = 1;

__transaction_atomic { x = 1; }


y = 2;
```

Figure 4.2: An example with two race conditions.

```
// Thread 1
// y is initially zero


__transaction_atomic {
  temp1 = x;
  if (temp1 == 1)
    temp2 = y;
}
```

```
// Thread 2
// y is initially zero
y = 1;
__transaction_atomic { x = 1; }




__transaction_atomic { x = 2; }
y = 2;
```

Figure 4.3: Publication and privatization without race conditions.

transaction (i. e., before any operation that is part of this transaction) is sequenced before StartTransaction, which in turn is sequenced before the transactional code (likewise for StopTransaction). This determines per-thread ordering.

Second, the specification defines a *transactional synchronization order* (TSO), a total order over all StartTransaction and StopTransaction operations in all threads. Different transactions must not overlap in this order (i. e., one transaction's StartTransaction operation must not be ordered between the transaction begin and commit of another transaction).

Transactions then contribute to the *synchronizes–with* relation in that StopTransaction operations synchronize with the StartTransaction operation of the next transaction according to TSO. Thus, transactions do not synchronize with other C++ operations (e. g., locks or atomics) but contribute to *happens–before*. The only constraint on TSO is that it has to be consistent with *happens–before* and must not lead to a cyclic *happens–before* relation. I will discuss in the next section how a TM implementation can pick a suitable TSO.

As a result, the interaction between transactions is straightforward to implement and to understand for programmers. However, the interaction with nontransactional operations is more involved because only race-free C++ programs have defined behavior. There are essentially two cases that we can distinguish, *publication* and *privatization*, depending on whether a transaction is meant to make previous operations visible in a race-free way (by synchronizing with other transactions) or to make data thread-private again.

Figure 4.2 shows a code example with two race conditions: The access to x is properly synchronized, but Thread 2's nontransactional assignments to y are not ordered in *happens–before* with respect to Thread 1's transactional access to y. Note that the race conditions are not due to the order of the transactional operations of Thread 1. Because the program is not race-free, behavior is undefined and the program could crash or return incorrect results.

Figure 4.3 shows correctly synchronized code. Considering TSO, Thread 1's

transaction could be ordered before, in between, or after Thread 2's transactions. In the first case, before publication, Thread 1 will read the initial value of x and will not access y. In the second case, after publication, it will read Thread 2's assignments to both x and y, which is okay because the assignment to y is sequenced before the Thread 2's transaction and thus happens before Thread 1's transaction (*happens–before* is roughly the transitive closure of *sequenced–before* and *synchronizes–with*, to which TSO contributes). In the last case, after privatization, Thread 1 will read the value 2 assigned by Thread 2's second transaction and will not access y, again preventing the race condition.

For programmers, this means that they have to be careful to produce race-free code. This is similar to what is required when using locks but the programmer does not have to manage mutexes (or locks) and deal with potential deadlocks.[1] Transactions can rather be compared to using a single global lock.

However, even though transactions might synchronize similarly to a global lock in the memory model, we do want transactions to synchronize more efficiently on the implementation level and execute concurrently if possible. For the TM implementation, this results in additional requirements that I will discuss in detail in the next section (e. g., the compiler must not introduce race conditions when optimizing transactional code).

There are no additional progress or liveness guarantees in the specification, which makes sense because the C++ standard itself does not require strong guarantees but rather states that this is implementation-defined. Programs should terminate eventually to be practical though.

4.2 Requirements for TM Implementations

In this section, I will derive what the TM specification means in terms of requirements on a TM compiler and runtime library. I will discuss the implications that the specification has for TM implementations and how language-level requirements in the memory model can be translated into orderings and guarantees that are required on the level of the TM runtime library ABI. Such an understanding is necessary to properly split responsibilities between TM compilers and runtime libraries. The TM compiler has to translate source code based on the specification into code that expresses these language-level requirements in terms of calls to the ABI and the guarantees offered by the ABI. TM runtime libraries have to implement this ABI on top of the memory model provided by the targeted hardware architecture. Therefore, the ABI must be reasonable to implement.

While the specification and the focus of what follows are on C++, TM support for C can be handled in a similar way. The current C standard does not define a memory model but the upcoming version, C1x [67], specifies a memory model very similar to the one of C++11. The language-level constructs for C will be similar despite perhaps some differences in the language syntax. Thus, one can expect that large parts of the TM support can be shared between C and C++ implementations, and that TM runtime libraries and the ABI can be reused without changes for C programs.

[1] Note that deadlocks can occur in relaxed transactions that block and wait for external events to happen.

It is beneficial to have a common ABI that separates the TM compiler support from TM runtime libraries because of several reasons. First, such a separation allows for compatible compilers and libraries, which is important to be able to link components that use TM but have been compiled using different tools (e. g., linking an application compiled with Intel's compiler with system libraries compiled by GCC). Second, this also allows plugging in different TM implementations by different vendors if the ABI is generic enough to work with different TM algorithms (e. g., switching to a new TM implementation when workloads change or if hardware TM support becomes available, without having to recompile all applications). Third, keeping most of the TM implementation in libraries makes compiler support simpler to implement.

There is a trade-off between performance and flexibility related to pluggable TM implementations. Dynamically linking TM runtime libraries gives the most flexibility but can incur runtime overheads because all interactions with the TM implementation require a function call (e. g., every transactional memory access). However, TM runtime libraries can also be statically linked. This binds an application to a certain TM implementation at link time[2] but enables link-time optimizations such as inlining of the TM implementation code into the application and other whole-program transformations. This can reduce or often even eliminate these overheads (see Section 7.2.3 for example code targeting an HTM). Thus, reyling on a common ABI is not an unavoidable cause of potential overheads, unless it structures code in such a way that important link-time optimizations are prevented.

The ABI that I am considering here has been initially proposed by Intel [62] and is now being maintained by Intel, Red Hat, and others. It is designed to be generic enough to be compatible with different kinds of TM algorithms and implementations. I am focusing on the Linux version of this ABI with some changes applied that are not yet reflected in the most recent ABI specification. I will first give a brief overview of the ABI to show its basic structure and to illustrate how a compiler would map a transaction's code to the ABI. However, the ABI does not yet specify the detailed synchronization-related guarantees and requirements for both the compiler and TM runtime. Therefore, I will later extend the ABI specification so that the requirements for compilers and runtime libraries become clear.

4.2.1 Overview of the TM Runtime Library ABI

The ABI consists primarily of functions for starting and committing transactions and accessing memory transactionally. Figure 4.4 shows an example transaction together with a simplified version of the code that a compiler would create for this transaction.

Transactions are demarcated using calls to the _ITM_beginTransaction and _ITM_commitTransaction functions. The former is similar to the setjmp function in that it must take a snapshot of callee-saved registers and can return several times. This allows the TM runtime library to restart a transaction by implementing and using a matching longjmp-like function. The compiler also generates code that checks the return code of _ITM_beginTransaction and either

[2] Unless the execution environment supports just-in-time compilation or linking during the runtime of a program, which is not common for C/C++.

```
1  extern long cntr;
2  void increment() {
3    __transaction_atomic {
4
5
6
7      cntr = cntr + 5;
8
9
10
11
12   }
13 }
14
```

```
1  extern long cntr;
2  void increment() {
3    ret = _ITM_beginTransaction(...);
4    // Omitted: save or restore of stack slots
5    if (ret & a_runInstrumentedCode) {
6      long l_cntr = (long) _ITM_RU8(&cntr);
7      l_cntr = l_cntr + 5;
8      _ITM_WU8(&cntr, l_cntr);
9    }
10   else if (ret & a_runUninstrumentedCode)
11     cntr = cntr + 5;
12   _ITM_commitTransaction();
13 }
14
```

Figure 4.4: An example of how source code with a transaction statement (left) is transformed to code that targets the TM runtime library ABI (right). Note that some additional code around _ITM_beginTransaction has been omitted for brevity.

saves (or restores) stack slots that might be (or might have been) modified by a future (or previous) attempt to execute the transaction. Alternatively, the compiler can suspend the reuse of stack slots that are live into the transaction during the transaction. Together with the rollback of transactional memory accesses provided by the TM runtime library, the setjmp-like transaction begin and the stack rollback make restarting a transaction transparent to the transactional code (the details will be discussed in Section 4.2.2).

Compilers can create several code paths with different kinds of instrumentation for each transaction. The example in Figure 4.4 shows two of these, the first with standard instrumentation and the second with uninstrumented code. The compiler informs the TM runtime library about which code paths are available (using arguments to _ITM_beginTransaction). The library is then free to choose (via bits in the return code) the code path that it suspects to yield the best performance; for example, when transactions are only infrequently executed, running uninstrumented code together with mutual exclusion for full transactions might be most efficient.

The instrumented code path shows how operations that access memory are split into calls to the transactional load and store functions in the library (e. g., _ITM_RU8 and _ITM_WU8). There are separate functions for accesses of different size than the 8-byte integers shown in the example. Also, there are further versions of these functions that tell the library that the same location was already previously accessed by another transactional load or store. In our example, the compiler could also use _ITM_RfWU8 and _ITM_WaWU8; this would allow a blocking write-through STM library to obtain a write lock right away at the first read ("Read–for–Write"), which in turn allows a very efficient "Write–after–Write" variant to be used for the second access to cntr.

Table 4.1 shows an overview of all functions in the ABI in its most recent version for Linux (the most recent official version of the ABI specification [62] provides no memory management functions and treats exception handling differently).

For transactions that run in serial–irrevocable mode, the TM runtime library guarantees that no other transaction is running concurrently (i. e., mu-

Category (example)	Description
Begin/commit (_ITM_beginTransaction)	Starts and commits transactions.
Load/store (_ITM_RU8)	Individual transactional loads and stores of integers, floats, vectors, and complex numbers of different sizes.
Undo logging (_ITM_LU8)	Undo-logging for write-through stores that need no synchronization or isolation.
Bulk data transfers (_ITM_memcpyRtWt)	Transactional versions of memcpy, memmove, and memset, with different variants for combinations of transactional and nontransactional access to the source and destination memory regions.
Memory management (_ITM_malloc)	Variants of malloc, calloc, free, **new**, and **delete** that are safe for use in transactions.
Exception handling (_ITM_cxa_throw)	Transactional wrappers for the standard exception-handling functions.
Serial–irrevocable mode (_ITM_changeTransactionMode)	Requests a different mode of execution for the current transaction.
User commit/undo actions (_ITM_addUserCommitAction)	Registers custom actions to be run on transaction commit or abort. Used by transactional wrappers of library functions (tm_wrapper annotation).
Miscellaneous	Initialization and finalization, aborting transactions, ABI version querying,...

Table 4.1: An overview of the functions that comprise the ABI.

tual exclusion) and that the transaction will never have to be aborted (i. e., it is irrevocable). This allows a transaction to execute TM-unsafe code that might be encountered in relaxed transactions, or the uninstrumented code path if this might be more efficient.

While the ABI is specified for only the x86 and x86_64 architectures, most of the functions are fairly independent of a specific platform (e. g., x86 on Linux). The platform-specific parts are mostly about aspects such as which calling conventions have to be used or which data types have to be supported for transactional loads and stores, which are platform-specific anyway.

4.2.2 Constraining Speculation

While the current specification of the ABI precisely defines some aspects of the interface such as the input and output of functions or which code to use for exception handling, it does not define the required behavior under concurrent execution. For example, which accesses to shared state by a runtime library are allowed or required for a given sequence of calls to the ABI functions, and how

does a compiler express language-level requirements in terms of ABI calls? As a result, it is unclear for both compiler and runtime library implementers how to use or implement the ABI correctly.

Looking at the C++ TM specification, the basic problem to be solved is that a runtime library has to pick a TSO (Transaction Synchronization Order) for transactions expressed via the ABI, but that the TSO choice must be consistent with the rest of the *happens–before* order, including the relation between accesses in different transactions. Also, the compiler must express *happens–before* constraints by using the ABI, and how this can happen needs to be specified.

We want the runtime library to be able to choose TSO because we want to keep the implementation of concurrency control in the library. We also do not want to require the library to choose TSO early (i. e., when starting a transaction) because this would disallow optimistic concurrency control and significantly limit the TM algorithms that can be used.[3] Thus, a runtime library might have to *speculatively* execute transactions and abort and restart them when the tentative TSO choice changes.

TM and the C++ as–if rule. However, speculative execution must be harmless and invisible in terms of the allowed behavior of a program. For C++ programs, this requires equality to the behavior of an abstract machine executing the program according to what is summarized as the as–if rule in the C++ standard.

Code that is unsafe according to the TM specification (e. g., accesses to volatile variables, or I/O) often cannot be speculatively executed because it represents actions of the abstract machine and cannot be rolled back or made invisible under the as–if rule.[4] For such code, the runtime library then has to select a position in TSO pessimistically and let the transaction become irrevocable. Unsafe code is typically not instrumented by the compiler, so it is also hard to isolate other code that is running concurrently. Together, these are the reasons for the existence of the serial–irrevocable mode in the ABI.

Implementing the as–if rule correctly is partially specific to the implementation of the environment that is executing the program. For example, it depends on this environment which actions performed by a TM implementation actually count as visible side effects. In our case, it primarily restricts the possibilities for speculative execution: The fewer side effects the environment allows to be contained, the less options for speculative execution the TM implementation has because otherwise side effects would become visible (i. e., according to what the environment defines to be visible) that would not occur when executing a C++ abstract machine.

To illustrate the difficulties associated with a practical definition of what constitutes a visible side effect in an environment, let me discuss one example in more detail: segmentation faults that occur due to misspeculation. Especially in STMs, validating that the memory accesses performed by a transaction

[3] We also cannot expect that all memory accesses of a transaction are known when starting the transaction because control flow and subsequent memory accesses might be data-dependent. Therefore, a TM cannot choose an optimal TSO early for such transactions.

[4] The compiler and runtime library can potentially defer the execution of these parts of code to the commit of the respective transaction. However, this only works in some scenarios and might require complex analysis of the transaction by the compiler (e. g., it can work if there is only output but no input and all output actions can be executed atomically).

represent an consistent, atomic snapshot can be quite costly. It could thus be beneficial to allow the TM runtime library to return speculative results of load operations to transactional code. However, if these results are pointers, dereferencing an inconsistent value (e. g., a null pointer) can lead to a segmentation fault, which is, on the operating systems that I consider, translated into a signal that is delivered to the signal handlers installed by the program. If the program did not install a signal handler, the program will terminate before it finished execution, which would be incorrect behavior. If it did install a signal handler, then incorrect behavior would likely result as well because arbitrary signal handlers cannot be expected to be aware of speculation inside of the TM implementation and thus would not know how to handle this segmentation fault. Therefore, the TM would have to install its own signal handler and mask segmentation faults that might have occurred due to misspeculation. Furthermore, the TM would have to prevent the application or other libraries from installing a different handler, leading to potential conflicts with these other components. However, even when controlling the userspace signal handling, the segmentation faults would still be visible at the kernel level, for example as part of page fault statistics or perhaps to intrusion detection systems. Such observers are unlikely to know or care about TM misspeculation.

This example shows that misspeculation is difficult to handle once side effects become visible to parts of the system that are not anymore under control of the TM runtime library or the compiler. Trying to contain the visibility of such side effects (and similar effects like nontermination or exceeding resource usage) in C/C++ environments requires solutions [116, 21] that are rather invasive, complex, and tightly coupled with other components in the environment. Whereas this might not matter that much in managed enviroments (e. g., a Java virtual machine), it does not seem to be beneficial for C/C++ and first-generation TM support because it would make it harder to provide TMs that are practical on a wide variety of systems.

Constraining speculation. Therefore, it seems to be more beneficial to constrain speculation and thus limit the effects of misspeculation, at least in the case of first-generation C/C++ TM implementations. The TM runtime library still selects TSO dynamically at runtime, but with restrictions.

To specify the restrictions on the implementations, we have to look at the code that the compiler creates from the transactional source code. The compiler-generated code consists of TM-pure operations, unsafe operations, and calls to TM ABI functions. *TM-pure operations* are all code and instructions that are either annotated as `transaction_pure` or which the compiler can detect to be safe and not need transactional protection from the TM (e. g., control flow instructions or arithmetic operations on CPU registers). TM-pure operations can thus be speculatively executed (I will define what this means precisely in Section 4.2.3). *Unsafe code* is all the code that is neither TM-pure nor supported by the ABI, and is always preceded by an ABI call that requests the TM runtime library to switch to an execution mode that supports unsafe code (i. e., serial–irrevocable mode and pessimistically choosing TSO).

When ignoring unsafe code, an execution of the compiler-generated code for a transaction is thus a sequence of TM-pure operations interleaved with calls to TM ABI functions. Figure 4.5 shows a simplified example execution of the

```
1  // TM−pure code                          // ABI calls
2  ret =
3                                          _ITM_beginTransaction(...);
4                                          // ret = a_saveLiveVariables | ...;
5  // Save stack slots (not shown)
6  long l_cntr = (long)
7                                          _ITM_RU8(&cntr);
8                                          // Abort and restart.
9                                          // _ITM_beginTransaction() returns a second time.
10                                         // ret = a_restoreLiveVariables | ...;
11 // Restore stack slots (not shown)
12 long l_cntr = (long)
13                                         _ITM_RU8(&cntr);
14 l_cntr = l_cntr + 5;
15                                         _ITM_WU8(&cntr, l_cntr);
16                                         _ITM_commitTransaction();
```

Figure 4.5: An example execution of the transaction of Figure 4.4 with one abort in the transactional read. TM-pure operations are shown on the left and ABI calls on the right.

code that a compiler would generate for the transaction shown in Figure 4.4. The sequence always starts and stops with calls to the ABI begin and commit functions, respectively. Transactions can be aborted and restarted only within calls to ABI functions (see Section 7.2.4 for a discussion of the problems caused by aborting within TM-pure operations). On a transaction restart, control flow is modified so that _ITM_beginTransaction returns again, which will reexecute the transaction's code from the beginning.

TM-pure operations and the TM runtime library have to work together (via the ABI) to execute a transaction. To reduce coupling, we do not want to require them to be aware of the specifics of the as–if rule on the both sides of the ABI. Instead, we want to enable the compiler and TM runtime library to separately reason about as–if on their respective side of the ABI.

Table 4.2 shows the high-level guarantees that enable separate reasoning about as–if by the compiler and the TM runtime library. In particular, these guarantees enable the compiler to reason about which code is TM-pure (or can be made TM-pure and how to implement this), without having to know how the TM runtime library picks TSO. In turn, the TM runtime library can reason about the as–if requirements for memory accesses executed by it without having to consider TM-pure code in detail. These guarantees are conservative choices that restrict speculation, but the gained decoupling makes the potential loss in performance worthwhile. If necessary, a higher level of coupling can always be introduced in a later revision of the ABI (e. g., by providing more information about TM-pure code to the library).

Let us now look at the guarantees in detail. Guarantee L1 in Table 4.2 is essential in that it requires the TM runtime library to stick to a valid TSO during the execution of a transaction. Values returned by the library (e. g., results of transactional loads) are input to TM-pure code, so the TM-pure code will see a valid execution that could have happened even with a sequential execution of all transactions.

The counterpart to L1 is C1, which requires TM-pure code to be independent of a specific TSO choice. This allows the TM runtime library to change TSO during the execution of a transaction because the TM-pure code's semantics or

	Compiler
C1	TM-pure must be independent of TSO.
C2	Preserve *sequenced–before*/*happens–before* of memory accesses in race-free code.

	TM runtime library
L1	Pick a valid TSO dynamically and only return values consistent with TSO. Change TSO without abort only if change is transparent to TM-pure code.
L2	TSO and memory accesses must be consistent with *happens–before*. No race conditions must be introduced.

Table 4.2: High-level guarantees provided by a compiler and a TM runtime library, respectively.

safety are guaranteed to not be affected. This is important because otherwise, transactions could not be executed optimistically; they would either have to abort if another transaction commits, or the TM would have to select a TSO a priori and would have to know about the tentative updates of previous transactions, which is not possible for all possible code. However, TSO is only allowed to change if the TM runtime library would have returned the same values from previous operations for the newly chosen TSO (i. e., the change must not be observable by TM-pure code).

Furthermore, C1 also requires that TM-pure code is race-free if one would ignore the TSO contributions to *synchronizes–with*. While this is obvious for code accessing no shared state (e. g., only accessing the thread's stack), it requires other code accessing shared state to be properly synchronized and to not conflict with any synchronization internal to the TM runtime library.

The first part of L2 is a straightforward requirement that is also part of the language-level specification (Section 4.1). It restricts which TSO choices are valid. With the current ABI, the added restrictions are relatively strong because the compiler only instruments transactional code, it only communicates *happens–before* via the order of calls to the ABI functions, and because the TM runtime library is only active during the execution of transactions. This means in turn that the TM runtime library has to assume that nontransactional code before a transaction synchronized with other threads (and thus expanded *happens–before* with more relations than those resulting from TSO). This also applies to the nontransactional code executed after returning from a transaction's commit function. Therefore, the TM runtime library has to ensure that all operations before the start of the transaction (including previously committed functions in other threads) are visible to transactional memory accesses and to TM-pure code (i. e., publication safety). Likewise, after returning from a commit function, the TM runtime library's TSO choice must be final because subsequent nontransactional code could rely on this choice and could communicate it to other threads. If the library would know that nontransactional code would be free of synchronization and side-effects, it could choose TSO more freely. However, this information is not provided by the current ABI, so the

choice has to be conservative.

Finally, together with the second part of L2, C2 ensures that the speculative execution of source code without race conditions is still race-free and consistent with *happens–before*. This is a joint responsibility of the compiler and the library because the former has to properly communicate the language-level memory accesses to the latter. Both have to ensure that no potential race conditions are introduced (e. g., by requesting or making accesses to data that would not be accessed by the abstract machine). This puts restrictions on the implementations of concurrency control algorithms in the library and the transformations by a compiler (e. g., reordering and prefetching).

I will provide more details about the high-level guarantees in Table 4.2 in what follows, but one can already see that together, they roughly ensure that a TM implementation adheres to the C++ TM specification: Active transactions execute as if in isolation, TSO and individual executions are consistent with *happens–before*, TM-pure code is not affected by a dynamic selection of TSO at runtime, and race-free code is executed in a race-free manner. Of course, this does not make the speculative execution completely transparent (e. g., because of a potentially higher resource usage than with sequential execution), but both the compiler and the library are allowed to separately make reasonable implementation choices that satisfy the as–if rule.

4.2.3 Compilers

A large part of the TM support in a C/C++ compiler is straightforward to implement (e. g., analyzing which code is transactional or potentially called from transactions, and cloning transactional code and instrumenting it so that it uses the TM ABI for memory accesses). However, two issues deserve more attention: (1) TM-pure code and (2) the compiler transformations that are allowed (or required) for accesses to shared memory and how to express *sequenced–before* with these accesses.

TM-pure code. When instrumenting transactional code, the compiler has to decide which operations in the code are unsafe, TM-pure, or handled by the TM runtime library. The latter case is straightforward to handle because the operation just has to be replaced with a call to the associated TM runtime library function as specified by the ABI (see Table 4.1). Unsafe code has to be prefixed with a call to the ABI's function that requests serial–irrevocable mode.[5] However, unsafe code cannot be executed concurrently with other transactions, so the compiler should not treat TM-pure code as unsafe code.

Therefore, the compiler has to determine whether code is trivially TM-pure or whether it can be made TM-pure by the compiler. This decision is somewhat implementation-defined and subject to the as–if rule, but does not depend on the TM runtime library and can be handled by the compiler on its own (if adhering to the rules described next). Compiler implementations will typically have to consider both intermediate forms of transactional code (where most of the instrumentation is likely to happen) and the native code that will be

[5] If the operation is already dominated (in terms of control flow) by another call to this ABI function in the same transaction, then a new call instructions does not need to be inserted.

Category	Remarks
Register-only CPU instructions	Must be safe and use only CPU registers (e. g., most control flow instructions but not system calls).
Stack accesses	Write-through. Loads and stores are TM-pure. Compiler-generated code rolls back stack slots on transaction restart or does not re-use them.
Accesses to thread-local data	Write-through. Loads are TM-pure. To make stores TM-pure and ensure rollback, the compiler inserts calls to undo-logging functions (_ITM_L) before the first store to a location.
Loads from immutable data	No side effects, no synchronization necessary (e. g., loads from virtual method tables).

Table 4.3: Examples of TM-pure code.

generated for potentially TM-pure operations (e. g., built-in functions used to implement complex operations).

The majority of TM-pure operations are those which (1) do not result in visible side effects in terms of the C++ abstract machine (e. g., I/O or volatile memory accesses), (2) do not contribute to *synchronizes–with* and are race-free, and (3) are idempotent (wrt. restarts of a transaction) or rolled back by some mechanism (e. g., compiler-generated code).

Table 4.3 shows examples for such TM-pure operations. Loads that need no synchronization in nontransactional code are usually TM-pure because they target immutable, existing variables (otherwise, the original code would not be race-free, in which case transactions have undefined behavior as well). Operations that modify thread-local state can still be TM-pure because of available rollback mechanisms. CPU state (e. g., registers or floating-point state) gets rolled back by the setjmp-like behavior of _ITM_beginTransaction. Stack slots potentially modified in a transaction get either actively rolled back by code inserted by the compiler at the start of a transaction (triggered by a bit in the return value of _ITM_beginTransaction, see Figure 4.4), or the compiler can instruct its code generator to not re-use stack slots that are live into a transaction and rather use new stack slots to store modifications within this transaction. Finally, the ABI provides undo-logging functions (see Table 4.1) to log the previous values of modified memory locations (without protecting them from accesses by other threads) and undo any changes on transaction restart. A location can only be accessed by a TM-pure operation in a transaction if all accesses in this transaction are by TM-pure operations or unsafe code.

The compiler's code generator might use built-in functions to implement complex operations from intermediate code. Those functions, even though they might communicate with other threads or might have to synchronize their accesses to shared state, can still be TM-pure if they are idempotent and fulfill the additional requirements discussed next. Note that these guarantees also explain the high-level guarantee C1 in Table 4.2 in more detail.

Race-free even without transaction atomicity. TM-pure operations must be properly synchronized and race-free even if discarding TSO contributions of currently active or future transactions to *synchronizes–with*. Intuitively, TM-pure operations must either not need to synchronize at all, or must be independent of any transactional synchronization because this is the sole responsibility of the TM runtime library and only well-defined at the language-level but not at the implementation-level that TM-pure operations are a part of.

No dependence on TSO choice or changes. TM-pure operations have to be independent of a particular TSO choice or change in this choice. They can expect and observe all orderings visible via *happens–before* to the associated thread and transaction when the transaction was first started. However, they must not depend on or be affected by later additions or changes to TSO (e. g., commits of other transactions). They can also expect that values returned from the TM runtime library during the current execution attempt of the transaction (e. g., since the most recent restart of this transaction) are consistent with the current TSO choice. For example, if a transaction did not abort since an earlier TM-pure operation, this does not mean that no other transaction committed in the meantime, nor that the current transaction might still be able to successfully commit.

Must be self-contained. Because transactions can be aborted during every invocation of a function of the ABI, and because the compiler can interleave TM-pure operations with calls to ABI functions, TM-pure operations must be self-contained in that they do not expect subsequent operations (TM-pure or ABI) to be executed as well. However, executions of individual TM-pure operations will not be interrupted or aborted. This might also restrict the transformations of transactional code that the compiler is allowed to do (e. g., inlining a TM-pure operation and then moving ABI calls into the inlined code can be a fault).

No interference with TM-internal synchronization. TM-pure operations can synchronize with other threads, but this must not create deadlocks or any other kind of conflict when combined with the TM-internal synchronization. Self-contained TM-pure operations are important for this requirement as well (e. g., if such an operation acquires a lock it must also release the lock before it can possibly be aborted). TM-pure operations are allowed to block on other operations except anything related to TM (e. g., operations that execute transactions or any ABI function). However, they must not wait for or depend on the execution of other operations that have not been started yet. For example, blocking on a lock acquired by another TM-pure operation is allowed because the acquired lock shows that the other, self-contained operation is already running. However, waiting for another transaction to finish or depending on another transaction to not abort is not allowed.

These requirements are basically also sufficient for functions annotated as `transaction_pure` to be indeed correct TM-pure operations. Additionally, such code has to ensure that it remains self-contained despite potential compiler optimizations (e. g., a programmer could have to add a `noinline` attribute to

the function). Transactional wrapper functions (associated with functions annotated with the tm_wrapper attribute) are sequences of TM-pure operations interleaved with calls to ABI functions, so the previous discussion also serves as a guideline for how to correctly implement such wrappers.

Compiler transformations. All operations in transactional code that are not TM-pure or unsafe have to be transformed into calls to the TM runtime library's functions. For individual memory accesses, these transformations are straightforward because the compiler just has to transform the access (e. g., in intermediate code) into a function call (e. g., to _ITM_RU8). However, transforming or reordering several accesses in a piece of transactional code requires more care because the compiler must not introduce additional race conditions that were not present in the C++ source code. Furthermore, the TM runtime library observes the sequencing of accesses (i. e., *sequenced–before*) at the language level through the order of the compiler-generated library calls, so the compiler must not lose important sequencing information when reordering accesses (high-level guarantee C2 in Table 4.2).

The first requirement for the compiler is to instruct the library to access exactly the same locations as accessed on the language level (i. e., by an abstract machine running the program). Accesses can be split or merged if necessary but must not touch other locations because these could be concurrently accessed by other code, leading to race conditions that do not exist in the source program. Using whole-program analysis, the compiler could potentially detect that some locations (e. g., bytes between two variables that have been added to align those variables) are only accessed by loads and stores in the TM runtime library and are always accessible. However, the potential benefit in terms of TM performance is probably rather small, so just accessing exactly the same locations seems to be sufficient.

Note that this applies to loads from memory as well. Loading from memory will not change the results of other accesses to the same location but the location could not be accessible, which could raise segmentation faults that are visible side effects. Some architectures such as SPARC provide nonfaulting load instructions but we cannot expect that such instructions are generally available.

As a second requirement, the compiler must not access locations speculatively, as it would happen when predicting that a certain branch is taken and prefetching the values that would be accessed in the predicted execution. In the case of misspeculation, the additional access could lead to a race with other concurrent code. For such data-dependent accesses, the compiler must not place them before the other access (and call to the TM runtime library) whose result determines whether the access would be executed on the language level.

Third, the compiler is allowed to reorder two accesses in a single transaction if it can prove that the later access (in terms of control flow) would happen in any case when the first access would execute in this transaction. The reason for this is that (1) transactions do not remove race conditions in the transaction's source code and that (2) the C++ standard allows undefined behavior resulting from race conditions to occur before the execution of the code that contains the race condition (see §1.9.5 in the standard [65]). Thus, the TM can assume that every transaction must be race-free even if executed atomically in any possible interleaving with nontransactional code. Performing an access earlier

in the execution of the transaction must thus still be race-free, provided that this access is guaranteed to be executed (in contrast to a speculative execution as explained previously) and the reordering would be allowed in a sequential execution of the transaction. If this would instead lead to a race condition, then the race condition could also be triggered in valid execution without reordering, and the reordering just leads to an earlier exposition of the race condition.

Finally, there are restrictions regarding the reordering of code across transaction boundaries. Basically, TM-pure operations can be moved into and out of transactions because they are independent of TSO. Unsafe operations must not be moved into transactions because this might invalidate liveness properties that exist in the source program, and they also must not be moved out of transactions because this might result in race conditions. Other code must not be moved out of transactions but can be moved into transactions because it does not synchronize (it would be unsafe code otherwise). If a transaction can potentially be canceled (see Section 4.1), then no code can be moved into or out of the transaction, unless the code is TM-pure and its execution cannot be detected by the program.

These previous requirements also show why publication safety is mostly the responsibility of the compiler and the transactional program. To avoid race conditions, the program's source code must first load the data that determines whether the published data is available (e. g., flags or pointers) before accessing the published data itself. Thus, accesses to published data are data-dependent on the former data, and the compiler must not reorder these accesses so that they happen speculatively before the data dependency. Given these guarantees, the TM runtime library is free to select or even change TSO during the runtime of a transaction that might read published data; the nontransactional accesses to the published data by the publisher and the observer will always be synchronized by the intermediary transactional accesses to the publication flags.

4.2.4 Runtime Libraries

The purpose of the high-level guarantees that the library has to provide (L1 and L2, see Table 4.2) is to split responsibilities between the compiler and the library. To ensure those guarantees and to fulfill the C++ TM specification, there are several requirements that the library has to fulfill in its implementation of the ABI.

Next, I will describe these requirements, starting with how transactional accesses have to be ordered with respect to each other. The interaction of transactional and nontransactional memory accesses further restricts which executions are allowed. The library also has to pay attention to how memory accesses are implemented (e. g., in software or by special TM hardware support) and that this integrates well with the C++ memory model. Finally, there are further requirements regarding aspects such as progress.

Transactional accesses and TSO. High-level guarantee L1 requires the library to pick a valid TSO and to only return values to TM-pure or unsafe code that are consistent with the current choice of TSO.

There are different ways for a library to manage and determine TSO, which are typically based on some form of isolating transactions and synchronizing

some representation of TSO. Examples for such representations are several objects with timestamps, a set of plain locks, or a single global lock with a version number. The library is free to choose any representation as long as it allows implementing the requirements described in what follows (see Section 5 for a discussion of STM algorithms).

Isolating transactions allows the library to run an uncommitted transaction with a tentative position in TSO (e. g., it could share its tentative position with another uncommitted transaction). This tentative position can change dynamically. Iff a particular change would be observable by TM-pure or unsafe code based on the values returned by previous calls to ABI functions, the library has to abort and restart the transaction before such a change becomes observable.[6] Basically, only the ABI's functions for transactional loads from memory return values, so the library has to make sure that the snapshot taken of the memory accessed in the transaction is consistent in that it looks like one valid TSO choice to TM-pure and unsafe code in this transaction. As explained in Section 4.2.2, this allows for library-independent TM-pure code and a separation of responsibilities between the compiler and the library.

If considering only the transactional accesses (i. e., ABI load and store calls), the library has to at least guarantee something equivalent to the notion of conflict serializability used in databases, which requires all memory accesses within a transaction to be consistently ordered with conflicting memory accesses in other transactions, yielding an acyclic order of those transactions that is consistent with TSO (see Section 2.2). The reason for this is that (1) TSO is a total order and contributes to *happens–before*, (2) that memory accesses inside transactions contribute to the *reads–from* (store–load access pairs) and *modification–order* (store–store access pairs) relations, which are part of *happens–before* as well, and (3) that those memory accesses must still fulfill the other requirements of the memory model (e. g., that loads read from the most recent store according to *happens–before*, see the formal model [6] for details). Because *happens–before* must be acyclic, the ordering of conflicting memory accesses (*reads–from* and *modification–order*) within different transactions must be consistent with the ordering of those transactions in TSO.

A transaction executed in serial–irrevocable mode trivially fulfills this requirement because no other transaction is running concurrently and it has (or will have) a *synchronizes–with* edge to the previous (and the subsequent) transaction.

Transactional accesses and as–if. However, conflict serializability is not sufficient because executions that are allowed by the library (by a certain choice of TSO and *reads–from* and *modification–order* relations) must adhere to the as–if rule. This is not just necessary to be able to install correct behavior when committing transactions but also to make speculative execution of transactions safe (e. g., for TM-pure code).

Therefore, the TM library must not return values that have been read from uncommitted updates to shared state because it is unclear whether the updating transaction has already finished its updates or whether a partial update and thus inconsistent data would be returned. This restricts the allowed executions in

[6] If it is only observable based on values returned in a nested transaction, then it is sufficient to roll back this transaction and subsequently executed code (i. e., closed nesting is allowed).

practice, adding to the base requirement of conflict serializability.

Transactional memory accesses as implemented by the library must touch exactly the same bytes in memory as the original access in the source code. The compiler tells the library which bytes these are (by transforming the source access into a call to a certain ABI function). The library has no additional information about other memory locations, so it has to assume that accesses to, for example, adjacent bytes could create race conditions. An exception are HTMs that can rely on hardware to isolate memory accesses and only have to care about interference with nontransactional accesses by the same thread (see Section 7.2).

Likewise, the library must not execute accesses speculatively but rather has to follow the sequence of ABI calls to load and store functions (i. e., a memory access must not happen before the respective call). Only the compiler knows whether potential subsequent accesses are going to happen or might not happen because they are data-dependent. The call sequence also communicates *sequenced–before* to the library, so the library has to assume that all accesses are ordered in *sequenced–before*.

As explained from the perspective of the compiler in Section 4.2.3, accesses that are known to happen anyway can be reordered. For the library, this means that it is free to touch the individual bytes of an access in any order (e. g., in _ITM_memsetW). Also, for those ABI load and store functions that communicate some information about future accesses (the read–for–write variants, see Section 4.2.1) or past accesses (e. g., write–after–read), the library is free to reorder those accesses (e. g., writing an idempotent value early in read–for–write) because the compiler has determined that these accesses are going to happen and are not data-dependent.

Nontransactional accesses, TSO, and *happens–before*. So far, we have just considered transactional accesses performed by the library. However, there are also nontransactional memory accesses that affect *happens–before* or can observe it, and that therefore yield additional restrictions on the executions that can be allowed by a library.

Nontransactional accesses within transactions have to be either TM-pure or unsafe operations, and are thus straightforward to handle. Unsafe operations might access the same memory locations as transactional accesses but will only happen after the transaction has been switched over to serial–irrevocable mode; therefore, the library just has to ensure that in this mode, all transactional accesses are indeed regular memory accesses and prior transactional stores have been applied as regular stores to the memory locations (i. e., transactional accesses must be visible to unsafe operations). TM-pure operations are guaranteed to be separated from transactional accesses in terms of the accessed memory locations, so they do not need to be handled by the library.

Nontransactional accesses outside of transactions do create new requirements for a library, primarily regarding privatization safety and which TSO choices are allowed. The library has no control or information about code outside of transactions, so it has to assume that this code is immediately synchronizing with all other threads that are not running transactions currently[7]. When such

[7]Currently executing transactions cannot synchronize with nontransactional code in other threads because (1) this is essentially forbidden in TM-pure code (it is forbidden to observe

synchronization happens, TSO can be observed by nontransactional code, which can lead to dependencies on TSO being final and stable at this point. Otherwise, invariants in the code might be violated, which in turn means that the as–if rule would be violated as well. The high-level guarantee L2 in Table 4.2 is related to this.

Therefore, a transaction must never get a position in TSO that might order it in *happens–before* before the nontransactional code sequenced before it. This does not necessarily require transactions to be linearizable operations, but library implementations must make sure that the earliest allowed position in TSO is consistent with *happens–before*. In turn, when a transaction has committed and returns to nontransactional code, its position in TSO cannot be easily changed anymore; similar to the requirement aimed at consistency of TM-pure code, TSO changes can only happen if they would not have been observable by any committed transaction.

However, the library must also ensure that the speculative execution of transactions is still safe according to the as–if rule, even when nontransactional code becomes active (after a transaction commit) and relies on a certain TSO choice. In particular, the library has to be prepared for privatization: Let us assume that a committed transaction P performed a write operation that introduces a conflict on some location with another active transaction O that read from this location. Subsequently, P accesses some data nontransactionally, assuming that its write operation notified other transactions that this data is now private. O, the observer, would access this data only if it is not private. Therefore, when P commits, the library is forced to finalize its TSO choice because this is what the nontransactional code relies upon. If O still operates with a tentative position in TSO that is before P, then there is a chance that O performs inconsistent operations (potentially resulting in illegal side effects) as soon as the nontransactional code after P gets executed.

Preventing such behavior is called *privatization safety*, and there are different ways for a library to implement this. First, it can delay the execution of the nontransactional code after P until all other potential observers (e. g., O) have realized P's commit and moved to a later position in TSO (in our example, O would have to abort because P's write has changed O's snapshot). Second, the library can try to implement transactional accesses in such a way that O's inconsistent operations are still a safe form of speculative execution under the as–if rule. Inconsistent operations can be transactional stores and loads because the nontransactional code might change the operating system's memory protection flags for the privatized data, so even a load might raise a segmentation fault and lead to a visible side effect. Possible implementations of privatization safety are further discussed in Section 5.2.

Note that a read-only transaction P cannot privatize data because they cannot force an ordering between P and O. Without this ordering, O might always be ordered after P and would still access presumably privatized data. Thus, the source code would not be race-free and the library would be allowed to do anything. Programmers could erroneously try to let P detect (with loads) whether any O is running, but this code would not be race-free either.

---

or reveal TSO), (2) the library itself only synchronizes internally, and (3) unsafe code can synchronize but will run in serial–irrevocable mode only.

Memory access implementation requirements. A library has to enforce the ordering requirements discussed previously in a way that is compatible with the implementation of the C++ memory model in the nontransactional parts of a program. Thus, it either has to use suitable native code (e. g., atomic CPU instructions and memory barriers) or employ C++ atomic operations (i. e., so that the compiler takes care of the translation to native code). The latter is particularly useful if library functions can get inlined into the program because then the compiler can more easily optimize across atomic operations.

There are basically three kinds of orderings that the library has to ensure: (1) between transactional memory accesses of different transactions, (2) between transactional and nontransactional memory accesses, and (3) TSO for unsafe memory accesses in transactions (i. e., serial–irrevocable mode). To do that, implementations have to either enforce edges in *synchronizes–with* between operations in different threads or have to rely on *sequenced–before* and existing edges in *synchronizes–with*. Note that adding such an edge to *synchronizes–with* can be costly because it typically requires some sort of memory barrier in the matching native code (see Section 2.1), so it should only be used if it is indeed necessary.

To order transactional memory accesses properly, a library has to make sure that there are edges in *synchronizes–with* that order conflicting memory accesses in the same order that the associated transactions have in TSO (i. e., so that all *reads–from* and *modification–order* relations are properly backed with a *synchronizes–with* edge). Depending on the TM algorithm, the number of edges that are necessary can range from one edge per memory access to one edge per transaction.

However, if two transactions are not in conflict (i. e., their memory accesses would be race-free if executed as ordinary code by two different threads), then no edge in *synchronizes–with* is necessary for this pair of transactions. This is because the source program cannot know how these transactions are ordered in TSO[8]. In turn, detecting the order in a race-free way would require additional synchronization operations that already include the necessary *synchronizes–with* edges. For example, to detect which of two updating transactions to disjoint memory locations executed first, other transactions would have to take several snapshots of the modified locations; likewise, nontransactional synchronization code would have to communicate the observed order, which requires *synchronizes–with* edges as well. Not having to inject *synchronizes–with* edges for nonconflicting transactions is important for the performance because otherwise, all transactions would have to synchronize with each other irrespective of whether the TM algorithm and the transaction workload require this or not.

The ordering of transactions and nontransactional memory accesses outside of transactions has to also be considered. TM-pure operations do not need special care because they must be independent of TSO and their internal synchronization must not interfere with TSO. As discussed previously, ensuring privatization safety is mostly about preventing unsafe speculative execution and proper synchronization between transactional memory accesses. For publication safety, it needs to be ensured that all nontransactional accesses that are sequenced before a transaction become visible no later than the commit of this

[8]TM-pure code must not observe it, and the transactions do not read from each other so their results cannot reveal the order

transaction. Therefore, there needs to be a *synchronizes–with* edge between this transaction and all other transactions that read from this transaction (e. g., via a release operation in the first transaction and a matching acquire operation in all readers).

Unsafe memory accesses in a transaction will only be executed if this transaction has switched to serial–irrevocable mode before (i. e., it is guaranteed to be executed mutually exclusive with other transactions). Thus, the library has to use some mechanism for mutual exclusion (e. g., a lock) to order the unsafe accesses according to TSO. However, transactions that are not in serial–irrevocable mode do not have to synchronize with each other, only serial–irrevocable transactions have to synchronize with transactions in any mode. This distinction is important because in the expected common case, transactions only infrequently switch to serial–irrevocable mode, and it allows a library to implement this mode without having to suffer from cache misses in the common case even though an additional memory barrier is necessary on typical hardware during the start of each transaction.

Also note that transactions and in particular transactional memory accesses do not need to be linearizable as long as they are still consistent with TSO and *happens–before* (e. g., if they correctly handle the *sequenced–before* and *synchronizes–with* relations). If accesses would have to be linearizable, library implementations would have to be conservative and add costly memory barriers to each transactional load, for example. This also applies to transactions in general, even though transactions in most practical library implementations will often be linearizable (or close to being linearizable).

Of course, this requires the library and how it implements TSO and *happens–before* to be compatible with how nontransactional code—as generated by the compiler—implements the C++ memory model. This is likely to be the case for most STMs, but might require more care in HTMs, depending on how well hardware transactions integrate with the rest of the memory model of the specific architecture.

Miscellaneous. There are a few other things that a library has to take care of, which will be explained next.

The ABI allows registering handlers during a transaction that will be executed on abort or on commit of this transaction. These handlers can be used by transactional wrappers for functions annotated with the tm_wrapper attribute to implement transaction support for external resources (e. g., the abort handler of the malloc function would release the previously allocated memory region). While this is not specified in the ABI, it makes most sense to treat *commit handlers* as unsafe or nontransactional code that can assume that the transaction has already committed. Thus, commit handlers have to be executed after TSO is agreed upon by all transactions, which implies that privatization safety has also been established for this transaction before commit handlers are run. Because transaction aborts in HTMs can be asynchronous (see Section 7.1 for why this can be beneficial), *abort handlers* have to assume that they will be run after the transaction has been rolled back already, so it is best to require them to always be run after any other TM-internal rollback has happened and the transaction has been restarted, but before transactional code is executed again.

Besides the requirements related to the as–if rule explained previously, the

library should also strive for *operating as transparently as possible*. For example, the excessive use of resources such as memory might lead to a program not being able to run as expected by the programmer. However, many components of a typical execution environment for C/C++ allocate memory, so it does not seem useful to set up any specific constraints regarding resource usage, at least for general-purpose TM implementations.

C++11 requires that all threads that are actually executed by the operating-system scheduler eventually make *progress*. However, this requirement is quite informal and OS schedulers are expected to be "reasonable" (e. g., it seems not quite clear whether the lock-free atomic operations in C++11 indeed need to be lock-free or just obstruction-free). Instead, implementations of C++11 are expected to provide practical guarantees for the users of this implementation. This position is understable if one considers that differences in OS scheduler properties can affect how to implement progress guarantees and at which cost (this also holds if just considering liveness). From the TM perspective, this means that progress guarantees for transactions are likely implementation-specific too; however, practical implementations will probably ensure that at least some transactions make progress eventually (based on each implementation's assumption of what exactly constitutes a reasonable OS scheduler).

Finally, it seems unreasonable to require that TMs would work across process boundaries (e. g., for another thread's data that has been mapped into the adress space of the progress that is executing the transaction); STMs just rely too much on virtual addresses to distinguish between data items, and considering any remapping would likely increase runtime overheads on the fast paths. Likewise, synchronization via TM will not be address-free (see §29.4 in the C++11 standard).

4.3 Discussion and Related Work

In what follows, I will discuss the similarities and differences of the C++ TM specification with other approaches for modeling the correctness of concurrent operations.

Linearizability. Linearizability (see Section 2.1) is widely used as correctness criterion for concurrent data structures. One could try to use it for TM by treating the full application state as the data structure and the individual transactions in the application as the data structure's operations. However, this is the external view of a data structure, whereas with TM, we have to put more attention on what happens internally during the execution of transactions. For linearizable data structures, this is treated as an implementation aspect and typically not further considered, whereas for TM we want to specify when such implementations are correct for arbitrary transactions (e. g., by considering the as–if rule as explained previously).

Second, linearizability uses the real-time order of operation invocation and response events to restrict the order of operations, which allows data-structure–local reasoning about linearizability. However, strict real-time order is stronger than C++11's *happens–before*; because we have to consider the full application state anyway, local reasoning is less important. Note that most STM implemenations indeed guarantee linearizability for transactions at least with respect to

other transactions, whereas this might not strictly hold for all kinds of HTM.

Serializability. When considering TSO in isolation, it is similar to serializability in that a single global order is established over all transactions (see Section 2.2). A majority of the current STMs that conform to the C++ TM specification rely on algorithms that are similar to some form of strong two-phase locking (see Section 5 for examples). This also applies to HTMs if they abort transactions immediately whenever there is a conflicting access by another thread. As a result, these TMs will basically have serializable (and linearizable) transactions.

Furthermore, database recovery theory [119] also tells us that those TMs are rigorous too because of relying on strong two-phase locking. This means that uncommitted values will never be read or overwritten by other transactions, and that values that have been read by uncommitted transactions are also never overwritten. Thus, transactions are operating under isolation and do not see partial updates of other transactions, for example. This, together with serializability, is already a large part of what a TM has to ensure to enforce TSO and comply with the as–if rule.

However, databases do not consider other aspects that matter for transactions in the TM context. For example, progress is not explicitly discussed but it is rather just assumed that schedulers will always accept schedules of a certain form. The TM specification does not discuss progress explicitly either but just inherits the informal C++11 progress guarantees, which at least provide some more details about what one can expect from reasonable implementations.

The most important difference however is that databases do not need to consider nontransactional operations. In contrast, the C++11 memory model makes a clear distinction between memory accesses that are atomic and contribute to *synchronizes–with* in some extent, and ordinary memory accesses that do not synchronize and might not even be atomic. Transactions are embedded via TSO into the language's memory model, and *happens–before*—including TSO—then completely specifies the order of transactions and nontransactional memory accesses. As a result, TMs have to consider aspects such as privatization, publication, and how to implement transactions in a data-race-free way.

Similarly, linearizability too only considers the relation between linearizable operations, but most nontransactional operations are typically not linearizable (which depends on the underlying (hardware's) memory model); thus, like serializability, linearizability is not really sufficient to reason about TM correctness.

Strong isolation. TMs are said to provide strong isolation [9] (also called *strong atomicity*) if they do not rely on data-race freedom and instead guarantee that individual nontransactional memory accesses and transactions are atomic with respect to each other. While this initially might sound convenient for a programmer, it comes with a couple of disadvantages compared to the *weak isolation* that the C++ TM specification requires.

First, in most proposals for strong isolation, it is not clear what language-level code should form atomic operations. For example, is the C++ statement "x++" a single memory access? A novice programmer might assume it is but it may be or may not be a single access in native code. Even with a language-level specification of what constitutes atomic accesses, programmers would have to

remember these rules. Furthermore, goarse-granular race conditions between transactions and sequences of nontransactional accesses would still be possible, so programmers have to consider these cases anyway; declaring the atomicity intent using transactions for all synchronizing memory accesses could be considered to be just proper documentation (or declaration) of this intent.

Second, on current standard hardware, ensuring strong isolation decreases the performance of nontransactional code because it disallows common compiler optimizations for non-concurrent code (e. g., common-subexpression elimination), and requires additional memory barriers to enforce that nontransactional operations become visible in program order. This can be avoided if the compiler can automatically infer data-race-freedom (with respect to transactions). However, this requires pointer analysis, which is rather difficult in C/C++ programs. Thus, with strong isolation, an incomplete analysis by the compiler would result in a direct performance decrease, not just in a missed optimization opportunity.

Overall, the potential programmability benefits of strong isolation are not convincing enough to justifiy the likely performance decrease of nontransactional code. This especially applies to programming languages which rely on data-race freedom already. Tools for automatic data-race detection might be a better means for programmers to gain confidence in the correctness of their programs because it will find executions where atomicity violations have not yet been considered by the programmer.

TM language extensions for C++. In a discussion of options for TM language extensions for C++ [18], Crowl et al. already identified several language constructs and design decisions that appeared later in the C++ TM specification. Most importantly, they propose adding transaction statements as the fundamental way of demarcating transactions. They were looking for practical language extensions that are usable for normal programmers, allow for incremental adoption of TM, and do not limit efficiency or scalability. This lead them to make design decisions that are very similar to what the specification proposes, such as choosing weak atomicity and always providing privatization safety. I/O and other operations with side effects that cannot be easily made transactional are not allowed in transactions, but different ways to support such operations are discussed (i. e., basically the transaction_pure and tm_wrapper cases). Furthermore, they also discuss exception semantics but do not choose a default (i. e., whether to abort or commit transactions in which exceptions are thrown). However, Crowl et al.'s focus is on exploring the design space, so the details of proposed features are not fleshed out (e. g., there is no distinction between atomic and relaxed transactions), or important features are missing (e. g., there are no transactional versions of functions); most notably, it is not discussed how to integrate transaction semantics with the programming language's memory model.

Single global lock atomicity for Java. Menon et al. describe four weak-atomicity TM correctness conditions [82] that are all based on the idea that synchronization with transactions should be similar to using a single global lock (which is used only for transactions and nothing else). These correctness conditions are aimed at Java, whose memory model is somewhat different to the C++11 memory model. Unlike the C++11 catch-fire sematics for data races,

Java requires minimal guarantees even for programs with data races. Therefore, a TM implementation cannot rely on transactions to be of a certain form (e. g., to be race-free even when there is concurrent publication, see Figure 4.3 for an example); instead, the TM has to provide semantics that behave like a global lock to ensure these minimal guarantees.

This is what *SGLA*, the strongest condition described by Menon et al., guarantees. TMs implementing SGLA have to basically perform concurrency control between transactions (like for serializability or TSO) and ensure privatization safety like explained previously for C++, but they also have to use global coordination to ensure publication safety. Menon et al. propose to use "start linearization" to implement this, which enforces that transactions do not commit until other transactions that were started early than this transactions have also committed. This does result in higher runtime overheads for the TM. Furthermore, it relies on racy transactional read accesses to return reasonable values (e. g., to be atomic). This can work in Java but is not suitable for C++ (e. g., because compilers expect nonsynchronizing code to be data-race-free, and would have to skip some optimizations otherwise).

Menon et al. also describe three other conditions that are weaker than SGLA, of which only *encounter-time lock atomicity* (ELA) is of interest for us here. It is defined to be equivalent to a lock-based execution that acquires potentially multiple locks for each accessed data item but before any access to this data item (i. e., like two-phase locking). ELA is then shown to provide privatization safety, but publication safety only for data-race-free publication. TM runtime overheads for ELA will be smaller than for SGLA because global coordination is not necessary anymore due to being able to rely on data-race-free publication.

ELA is thus similar to what is required by the C++ TM specification, in the sense that programmers must use only data-race-free publication and compilers must not introduce additional data races (e. g., by hoisting loads to before other loads that they are data-dependent on, or by using accesses of larger granularity than what is specified by the source program). Note that the specification therefore does not ignore publication safety but rather distributes the responsibilities differently between the programmer, the compiler, and the TM runtime library to allow a higher performing implementation (in contrast, SGLA puts the sole responsibility on the library).

Single global lock atomicity for C/C++. Shpeisman et al. subsequently investigated single global lock atomicity in the context of C/C++ [104]. They observe that single global lock semantics in race-free programs do not provide full atomicity guarantees because transactions that contain synchronizing operations (e. g., with C++ atomic operations) do not represent data races but can still communicate with other threads (like in the case of locks). For similar reasons, lock-based semantics are not sufficient to allow transactions to be safely canceled.

To provide full atomicity, Shpeisman et al. define Race-Free Atomicity (RFA), another correctness condition that guarantees atomicity for all data-race-free transactions even if those include synchronizing operations. RFA disallows any interleaved execution of RFA transactions and conflicting synchronization actions (i. e., basically actions that would constitute a data race if they

were ordinary nonsynchronizing actions). However, RFA cannot be practically implemented, so Shpeisman et al. additionally propose to provide a separate kind of atomic transactions that are not allowed to contain synchronizing actions; this allows such transactions to guarantee RFA[9]. Atomic transactions can then coexist with so-called critical transactions that continue to provide lock-based semantics and can contain synchronizing code.

This split into atomic and critical transactions is equivalent to the distinction between atomic and relaxed transactions in the C++ TM specification, only that the specification uses the notion of unsafe code instead of synchronizing code.

Shpeisman et al. focus on the TM semantics at the programming language level. They do not discuss how to implement these semantics nor how to define a TM runtime library ABI and distribute the implementation responsibilities between compilers and libraries.

Opacity. Guerraoui and Kapalka define another TM correctness condition that they call opacity [47]. It models TM as a black box, and executions as a series of invocation and response events that represent calls to TM operations (e. g., reads and writes, or transaction starts and commits). Nontransactional operations are not considered at all (nor are related aspects such as publication or privatization). Therefore, opacity does not capture an aspect of TM that is very important in practice, and so it is not sufficient for compiler or runtime implementors, nor for programmers that want to use TM. As I have explained previously, we want TM to be integrated into programming languages (and thus into those languages' memory models too), so a nonintegrated black-box definition is not optimal (e. g., the previously discussed differences between SGLA for Java and ELA for data-race-free C++ also highlight this).

Basically, executions are said to be opaque if there exists a sequential execution of the transactions that (1) is equivalent to a completed version of the original execution, (2) respects the real-time order of transactions, and (3) preserves the sequential semantics of all operations. In a complete version of an execution, all live transactions are chosen as either being committed or aborted.

Executions are equivalent if they contain the same operations for each transaction with the same responses, and if the order of operations within each transaction is preserved (but a reordering between operations of different transactions is allowed, so this is similar to conflict serializability's notion of equivalence). Assuming that opacity is supposed to be applied at the programming language level (this is not specified by the authors), the C++ TM specification is somewhat more permissive in that it allows the reordering of operations by the compiler or a TM runtime library if this would be allowed by the as–if rule (including the constraints related to publication safety). Note that the current TM runtime library ABI also assumes an implicit total order of operations though.

Related to that, the authors claim that letting transactions operate on inconsistent values (e. g., dirty reads) would always be incorrect. However, this is rather determined by the TM implementation and its capabilities to, for example, isolate and mask potential failures. This concern is therefore better captured under the umbrella of the as–if rule.

[9]If atomic transaction do not contain synchronizing actions, then there cannot be any conflicting synchronizing actions in other threads if the program is data-race-free.

Transactions are supposed to be ordered in real-time order if one transaction's commit event happens earlier in real time than the other transactions start event. This is similar to linearizability, and tries to capture the externally observable ordering of transactions without relying on a full-program happens-before relation (i. e., C++11's *happens–before*). However, this could become problematic because it basically requires transactions to be a full memory barrier, which is not implicitly necessary for all HTMs. The C++ TM specification is better in that regard in that TSO is not constrained by real-time order but instead by *happens–before*, which captures all the ordering constraints that are actually requested by a program (or present in a certain execution).

Guerraoui et al. also define *parametrized opacity* [45], which is a version of opacity parametrized by a memory model. However, this condition aims at providing strong isolation as specified by the model, whereas C++11 and the C++ TM specification rely on data-race-freedom and weak isolation.

Conclusion. As the previous discussion shows, the biggest advantage of the C++ TM specification compared to other correctness conditions—with the exception of the work by Shpeisman et al. [104]—is really its tight integration with the C++11 memory model and the C++ language in general. While TSO is conceptually quite similar to the ideas behind serializability and other conditions, the way in which it relies on and contributes to *happens–before* allows programmers to think about synchronization using transactions as just another part of the language and to understand the interactions of transactions with other parts of a program. The choice of weak isolation for transactions cleanly follows C++'s choice of relying on data-race-freedom; both choices are motivated by unacceptably high performance overheads for stronger models (i. e., strong isolation and sequentially consistent hardware memory models, respectively). The distinction between atomic and relaxed transactions is also a notable difference; it provides programmers with a choice between transactions that execute atomically with respect to all other code and transactions that can execute similar to critical sections protected by a global lock.

Neither the C++ TM specification nor the work by Shpeisman et al. cover how TM implementations can actually implement the proposed semantics. While implementation concerns definitely influenced the choice of semantics, the specifications do not give detailed guidance to implementers.

Such guidance is what my work contributes (see Section 4.2). I have discussed the trade-offs that implementations face, and derived a split of implementation responsibilities between TM compilers and TM runtime librarys. This also provides clear semantics for the TM runtime library ABI, whose previous specification defined just the sequential functionality of the functions that comprise the ABI. As a result, the ABI can then also serve as a precise specification of what TM runtime libraries for C/C++ have to provide, and the TM implementations and algorithms presented in Chapters 5 and 7 are built according to this ABI's requirements.

Like the C++ TM specification, the ABI benefits from a tight integration with the C++11 memory model and other properties of C++ environments (e. g., the as–if rule) because this allows for a more straightforward extension of existing implementations with TM support.

Chapter 5

Software TM Algorithms and Implementations

STMs implement TM without relying on any special hardware support for TM. Until HTMs are widely available and able to execute the vast majority of transactions, STMs are thus the primary means to implement TM. Therefore, until then, TM performance is basically determined by STM performance (or by hybrid software/hardware TMs assuming wide availability of first-generation best-effort HTMs, see Section 7.3).

There is a large number of possible STM implementations, for example using just a single global lock, two-phase locking on multiple locks, or various nonblocking algorithms. Nonetheless, for the workload assumptions described in Section 3.2, there seems to be one sweet spot for general-purpose STMs: Using optimistic concurrency control for transactional read operations while still allowing concurrent but nonconflicting update transactions to commit in parallel. Furthermore, we can expect transactions to likely execute many more transactional read than write operations, so the performance of taking an atomic snapshot of the data accessed in the transaction is essential for good overall performance.

However, efficiently checking the atomicity of snapshots is difficult in a TM setting because the data items that a transaction will access are not known until the transaction is actually executed; with every new read operation executed by a transaction, the STM has to validate that the snapshot is still atomic. Validating this by just reading all previously accessed data items again leads to increasingly larger runtime overheads when the size of the snapshot grows.

Nonetheless, this was the standard approach until early 2006 when Pascal Felber, Christof Fetzer, and I first published [92] how to take atomic snapshots more efficiently by relying on a *global time base* shared by all transactions.[1] In this class of algorithms, which we later called the *Lazy Snapshot Algorithm* (LSA) [90], validating snapshot atomicity for a newly executed read operation has only a small constant runtime overhead in the common case. I will explain this class of algorithms on an abstract level in Section 5.1.

Next, in Section 5.2, I will discuss an implementation of LSA that is tailored

[1] Note that even though this paper focuses on Snapshot Isolation [7] as consistency condition, it also presents the fully atomic version of the algorithm.

for C/C++ environments. Contrary to our first Java-based and nonblocking implementations of LSA, this implementation uses pessimistic concurrency control for write operations and keeps only a single version of each transactional data item. Such an implementation—and LSA—is still one of the most important STM implementation variants for C/C++ besides just one or two other major variants.

Given that the STM is operating on top of an asynchronous system, the STM has to explicitly establish a global time base: It can either implement it using shared memory synchronization (e. g., as a linearizable integer counter), or can rely on other time bases provided by the operating environment. The former is simple to implement and works well in small-scale systems but suffers from synchronization runtime overheads and contention in larger systems. However, LSA can be modified to use imprecisely synchronized real-time clocks as time base, which removes the single global synchronization bottleneck in the algorithm and which I will explain in Section 5.3.

Finally, in Section 5.4, I will conclude with a discussion of related work.

5.1 Time-Based STM

Obtaining atomic snapshots of transactional data with low runtime overhead is critical for STM performance. According to our workload assumptions described in Section 3.2, loads are much more common than writes in transactions, and concurrent reads to the same data items should not result in unnecessary runtime overheads. In this section, I will explain how STM algorithms can achieve this with the help of a global time base shared by all transactions.

For the following discussion, it is sufficient to consider a simple definition of transaction and snapshot atomicity: Similar to database transactions and serializability (see Section 2.2), we want to allow concurrent execution of STM transactions but those transactions have to virtually execute in a total order and in isolation; thus, an atomic snapshot has to virtually execute without interleaving execution steps of any other transaction. Furthermore, snapshots get built incrementally when executing a transaction because it is not known a priori which data items a transaction will access. Finally, a snapshot's position in the total order of transactions must be consistent, which basically means that any change to it must be invisible to the transaction.

A more precise definition of the atomicity and consistency requirements for C/C++ environments is discussed in Section 4.2 and will be used as requirements for an implementation of time-based STMs as discussed in Section 5.2. Nonetheless, providing snapshots that are always atomic and consistent is an essential requirement for C/C++ STMs, which is why the algorithms described next are used in those implementations.

Visible vs. invisible reads. When implementing snapshots, the first major choice is whether a transaction makes its read operation visible to other transactions or not, leading to the distinction between visible reads and invisible reads.

With *visible reads*, transactions either peek into the metadata that the STM maintains for each transaction to determine whether other transactions are reading a certain transactional data item, or transactions announce their start and

end of read operations at metadata associated with each data item. Usually, the latter is preferred because scanning through other transactions' read sets is costly. Announcing a read operation can then be implemented by, for example, incrementing a shared counter before reading an item, and decreasing the counter of all items read in the transaction when the transaction commits or aborts. Update transactions are then only allowed to store to a data item if there are no active readers.

Altogether, this approach is straightforward to implement and basically equal to the two-phase locking scheme explained in Section 2.2. The shared read counters are typically parts of a set of reader–writer locks that data items are mapped to, and transactions acquire the necessary locks on-demand during their execution.

The disadvantage of this approach is that while reads of the same data item can be executed concurrently by different transactions (i. e., two reads do not conflict with each other), doing so requires updates to shared memory, and performance will suffer due to contention and cache misses that arise when different threads modify the same piece of STM metadata (e. g., a shared read counter or a reader–writer lock, see Section 2.1 for further explanations).

When using *invisible reads*, the STM does not have to modify shared metadata for each read operation but must instead check that no other update transaction interfered with the atomicity of the snapshot and the transaction. This process is called *validation*, and can also be understood as checking that the transaction could have held a read lock since the first access to the data item; validating all data items accessed by the transaction then basically guarantees atomicity of the snapshot.

Because snapshots have to be always consistent (see Section 4.2), the STM needs to validate the snapshot before returning any value to the program, so during each transactional load operation. A simple way to do this, called *incremental validation*, is to just validate all previously accessed data items. With this approach, performance will not suffer from overheads that visible reads suffer from but instead from quadratic validation costs: for the n-th read access, a transaction has to validate the previous $n - 1$ data items.

Another potential performance disadvantage compared to visible reads is that with invisible reads, concurrency control is forced to be optimistic: Update transactions cannot know whether their commit will force other transactions to abort. However, this is mostly relevant in workloads with frequent read–write conflicts between transactions, in which case the STM could still use visible reads when necessary (e. g., after a transaction has aborted a couple of times). As I have explained in Section 3.2, optimizing the temporal aspect of TM-based synchronization is beyond the scope of my work, which includes this aspect of optimistic versus pessimistic concurrency control.

The difference in performance between visible and invisible reads with incremental validation can therefore be roughly characterized by the relative speed of updating shared data (e. g., cache misses) compared to normal computation and reading shared data.

Figure 5.1 illustrates this trade-off with a simple benchmark: 8 threads execute read-only transactions to the same set of data items, except for the results marked with "disjoint accesses", in which each transaction accesses its private set of data items. The figure shows the runtime overhead of a transaction divided by the number of read operations in the transaction, for transactions

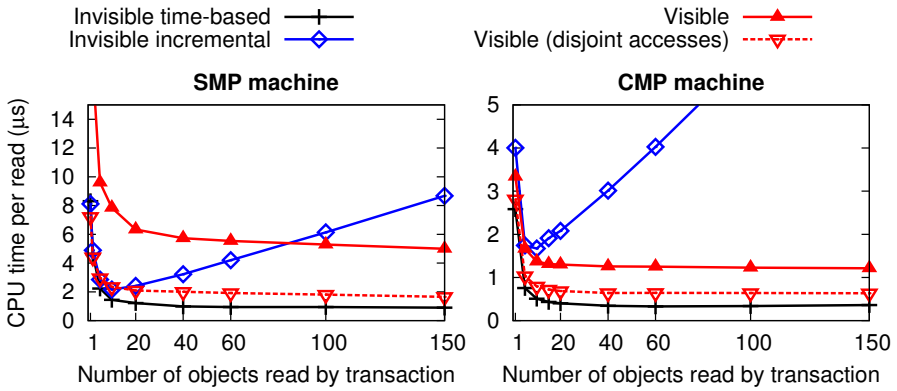

Figure 5.1: Performance of visible versus invisible reads on both an SMP and a CMP machine.

of different read set size (i. e., how many separate data items each transaction accesses). The results were obtained using a Java STM that either uses visible reads, invisible reads with incremental validation, and invisible reads with time-based validation.

These results highlight the principal differences in performance, even though STM snapshot performance is determined by several more factors. First, comparing the overheads within each of the plots in the figure shows (1) the quadratic overhead of invisible reads with incremental validation, where larger transactions suffer from an increasingly large per-read overhead, (2) the large decrease in performance of visible reads when transactions access the same data, due to contention and cache misses on the shared STM metadata, and (3) the low and constant overhead of time-based validation. Comparing both plots in the figure, we can see that incremental validation is much more costly on the CMP machine[2], whereas non-disjoint visible reads are more costly on the SMP machine[3]. This can be explained by the difference in the hardware architectures: the CMP cores are not very fast (the incremental validation loop hurts) but are close to each other on a single chip (cache misses are relatively cheap), whereas the CPUs on the SMP machine are rather powerful on their own but sit on distant sockets (i. e., communication between CPUs is more time-consuming). Interestingly, time-based validation performs well on both of these architectures.

Benefits of a global time base. The performance advantage that these figures show for time-based validation is based on tagging updates to transactional data with timestamps from a discrete global time base that is shared by all transactions. Progress from the perspective of the transactional data happens whenever an update transaction commits, and when that happens our global time base advances (i. e., it ticks). The new time (or timestamp) can then be

[2] A Sun Fire T2000 server with a single socket and an 8-core 1.2-GHz UltraSPARC T1 processor; each core handles four hardware threads concurrently.

[3] A symmetric multi-processor system with four sockets holding Xeon CPUs and enabled hyperthreading (i. e., eight logical CPUs).

used to identify the commit, and we thus call it the *commit time*.

Based on that, transactions can take snapshots virtually at a certain time (called the *snapshot time*) by reading from the most recent update transaction that updated the respective data item prior to the snapshot time. Because of the global time base, updates to different locations can be easily ordered and compared: When a transaction reads a new data item, it can determine the atomicity of the whole snapshot by just comparing timestamps (which explains the "time-based" attribute); it does not need to validate previously accessed items as with incremental validation. Also, read operations do not need to modify shared data. Therefore, time-based validation can avoid both these runtime overheads.

5.1.1 Lazy Snapshot Algorithm (LSA)

In the following, I will describe LSA on a higher level of abstraction. This makes it easier to understand and avoids having to dive into the low-level implementation details. Most notably, for the pseudo code below, let us assume that all STM operations (e. g., individual loads and stores executed by a transaction) execute atomically and in a sequentially consistent manner. I will present a complete concurrent implementation for C/C++ on the basis of the C++11 memory model in Section 5.2, which does not rely on these simplifying assumptions.

LSA handles transactional accesses to shared data items, which can either be higher-level constructs or just plain memory locations (see Figure 3.2 and the discussion in Section 3.1.2). For now, let us assume a set of abstract shared objects $\mathcal{O}$. Each object $o \in \mathcal{O}$ traverses a series of versions $o_1, o_2, \ldots, o_i$. Every time an object is written by a committed transaction, a new version of this object is created and the previous one becomes obsolete. The STM may—but does not need to—keep multiple versions of an object at a given time; only the latest version is necessary.

The discrete logical global time base of LSA is designated by *clock*. It can be implemented using a simple shared integer counter, which update transactions can increment atomically to acquire a unique commit timestamp.

Let us also assume that objects are only accessed and modified within transactions. Hence, we can describe a history of an object with respect to the global time base *clock*. $\lfloor o_i \rfloor$ denotes the time when version i of object o has been written, and by $\lceil o_i \rceil$ the last time before the next version is written. Let us call the interval between these two bounds the validity range of the object version and we denote it simply by $[o_i]$. If o_i is the latest version of object o, then $\lceil o_i \rceil$ is undefined (because we do not know until when o_i will be valid), otherwise $\lceil o_i \rceil = \lfloor o_{i+1} \rfloor - 1$. For convenience, $o_\star$ denotes the most recent version of object o.

The sequence $\mathcal{H}(o) = (\lfloor o_1 \rfloor, \ldots, \lfloor o_i \rfloor, \ldots)$ denotes all the times at which updates to object o are committed by some update transactions. $\lfloor o_1 \rfloor$ is the time when the object was created. Sequence $\mathcal{H}_i$ is strictly monotonically increasing, i.e., $\forall o_i \neq o_\star : \lfloor o_i \rfloor < \lfloor o_{i+1} \rfloor$.

Algorithm 1 shows the multi-version, write-through variant of LSA as pseudo code. Functions prefixed with stm− are the functions that comprise the external interface of the STM: Starting a transaction (stm−start), loading from and storing to transactional data items (stm−load and stm−store), and committing

Algorithm 1 Lazy Snapshot Algorithm (write-through, multi-version variant)

```
1:  Global state:
2:    clock ← 0                                          ▷ Global time base (shared integer)

3:  State of thread p:
4:    st: snapshot time (range)
5:    r-set: set of read object version
6:    w-set: set of written objects

7:  stm-start()_p:
8:    st ← [clock, clock]                                ▷ Initial snapshot time bounds
9:    r-set ← w-set ← ∅

10: stm-load(o)_p:
11:   if o ∈ w-set then
12:     return o_⋆                                       ▷ Read previous write
13:   if ⌊o_⋆⌋ > ⌈st⌉ then                               ▷ Is latest version too recent?
14:     extend(clock)                                    ▷ Try to extend (optional)
15:   if ⌊o_⋆⌋ ≤ ⌈st⌉ then                               ▷ Can use latest version?
16:     st ← [max(⌊st⌋, ⌊o_⋆⌋), ⌈st⌉)]                   ▷ Yes, use latest version
17:     r-set ← r-set ∪ {o_⋆}
18:     return o_⋆
19:   else
20:     if w-set = ∅ ∧ (∃o_i : ⌊o_i⌋ ≤ ⌈st⌉ ∧ ⌈o_i⌉ ≥ ⌊st⌋) then   ▷ Can use older version?
21:       st ← [max(⌊st⌋, ⌊o_i⌋), min(⌈st⌉, ⌈o_i⌉)]
22:       r-set ← r-set ∪ {o_i}
23:       return o_i
24:     else
25:       abort()                                        ▷ Cannot find valid version

26: stm-store(o,val)_p:
27:   if o ∉ w-set then                                  ▷ Need to acquire on first write to o?
28:     if ⌊o_⋆⌋ > ⌈st⌉ then                             ▷ Is latest version too recent?
29:       extend(clock)                                  ▷ Try to extend
30:     if ⌊o_⋆⌋ ≤ ⌈st⌉ then                             ▷ Can use latest version?
31:       st ← [max(⌊st⌋, ⌊o_⋆⌋), ⌈st⌉]
32:       o_⋆ ← new()                                    ▷ Create/acquire a new latest version
33:       ⌊o_⋆⌋ ← ∞
34:       w-set ← w-set ∪ {o}
35:     else
36:       abort()                                        ▷ Cannot find valid version
37:   o_⋆ ← val                                          ▷ Write through to acquired latest version

38: extend(t)_p:
39:   ⌈st⌉ ← t                                           ▷ Try to extend snapshot up to t
40:   for all o_i ∈ r-set do
41:     if o ∉ w-set then                                ▷ Recompute validity unless object already acquired
42:       ⌈st⌉ ← min(⌈st⌉, ⌈o_i⌉)

43: stm-commit()_p:
44:   if w-set ≠ ∅ then                                  ▷ Nothing to do if read-only transaction
45:     ct ← (clock ← clock + 1)                         ▷ Unique commit time (atomic increment)
46:     if ⌈st⌉ < ct − 1 then                            ▷ Must validate if others committed in the meantime
47:       extend(ct − 1)
48:       if ⌈st⌉ < ct − 1 then                          ▷ Are snapshot and commit time adjacent?
49:         abort()                                      ▷ No, commit would not be atomic
50:     for all o ∈ w-set do                             ▷ Make new versions accessible to other transactions
51:       ⌊o_⋆⌋ ← ct                                     ▷ Validity starts at commit time

52: abort()_p:
53:   for all o ∈ w-set do
54:     delete(o_⋆)                                      ▷ Remove acquired latest version
```

a transaction (stm−commit). A transaction completes successfully if it executes the algorithm until commit without executing the abort function (after whose execution the transaction will be restarted from its beginning).

For each thread executing transactions, the STM maintains a read set *r-set*, a write set *w-set*, and a snapshot time range *st*. The read set tracks all the object versions read by a transaction, whereas the write set tracks the objects written by the transaction (we always write the most recent version).

When starting a transaction, the STM clears both read and write set. It also sets the snapshot time range so that it is valid starting at the current time (i. e., value of the global time base *clock*), which I will explain in detail later.

Write-only transactions. Let us start by considering just transactions that only write objects. If stm−store gets called for an object that the transaction has not written to yet, it first checks whether the most recent version of the object is accessible and tries to extend the snapshot time if necessary (line 29). We can ignore this for now because the read set *r-set* will always be empty for write-only transactions, but I will revisit the snapshot time maintenance aspects later. In our case, $\lceil st \rceil$ will always be set to *clock* (line 29 and line 39), which is the current time. We will thus be able to access the most recent version $o_\star$ (line 30) unless, as we will see next, another transaction is already writing to this object.[4] To write to the object, we (1) first update our snapshot time (explained later), (2) create a new most recent object version $o_\star$, (3) set its validity range to start at ∞, and (4) add the object to our write set (lines 31–34). The third step is precisely what makes the most recent object version inaccessible to other transactions because the current time will always be smaller than ∞. Thus, if another transaction has installed such a $o_\star$, we will just abort (line 36). Finally, if we did not abort, we modify the value of $o_\star$ (i. e., *write through* to memory).

Thus, writing an object is visible to other threads and prevents them from writing. This exclusive access can either be implemented like locking so that other writers simply have to wait until the existing writer transaction has finished. Alternatively, this also allows nonblocking implementations, where the exclusive access is handled more like a write marker that requires another writer to first abort the existing writer transaction. The first Java-based LSA implementations [92, 90] where indeed nonblocking but we will assume a lock-based approach in what follows.

By acquiring and writing to the most recent object version, we thus essentially lock the object, write to it, and the updates will become visible as new most recent object versions $o_\star$. If we have to abort we simply remove $o_\star$, which will both undo our updates and make the previous most recent version accessible again to other transactions (lines 53–54 in stm−abort).

When stm−commit is called, the update transaction is ready to commit: It (1) acquires a unique commit time *ct* from the global time base *clock* on line 45, (2) validates (lines 46–49), which will always succeed for write-only transactions for similar reasons as in stm−store, and (3) makes the new most recent object versions created by this transaction accessible by letting their validity ranges start at the commit time (lines 50–51).

[4] Remember that we assume that STM functions execute atomically, so there cannot be a committed version whose validity starts at a most recent time than the time we obtained from *clock* previously.

Overall, updates thus resemble a two-phase locking scheme: we acquire locks before writing to an object, and never release the locks until commit. We also obtain the commit time when we have acquired all the locks. Therefore, all updates of a transaction are atomic and the commit times will be consistent with the order of committed object versions by different transactions: two-phase locking serializes conflicting transactions, which also enforces serialized acquisition of commit times.

Atomic snapshots. Now that we know how atomic updates and the validity ranges of object versions work, we can also understand how we can maintain atomic snapshots. Because those validity ranges are all constructed using the global time base, and because uncommitted updates are inaccessible due to the object locking being used, we can just combine the ranges to reason about the order of a transaction's operations with respect to other transactions' operations.

The transaction uses the snapshot time range st to keep track of the validity ranges of the objects that it read: Each accessed object version's validity range incrementally constrains its lower bound $\lfloor st \rfloor$ and its upper bound $\lceil st \rceil$ further (i. e., st becomes the intersection of st and the validity range). st is a range because the same snapshot might be valid at different adjacent positions in the serialization order of transactions; if the range becomes empty, the snapshot is not atomic anymore, and the transaction has to abort.

Considering read-only transactions first, lets us now look at how snapshots are constructed. In stm−start, we set the snapshot time range to the current value of the global time base *clock*, which thus represents the current time.[5] This ensures that we do not read from out-dated object versions and is required to ensure linearizability for transactions, for example. Also, we never speculate about what might be committed in the future, and thus set $\lceil st \rceil$ to a time that is never higher than the current time *clock*.

In stm−load, we first try to access the most recent object version $o_\star$ and try to extend st if necessary, which I will explain later. Then, if we can access $o_\star$, we intersect st with $o_\star$'s validity range, add $o_\star$ to the read set, and return its value (lines 16–18). Otherwise, we look for another object version o_i whose validity range intersects with st and use it instead (lines 21–23), only using the existing upper bound $\lceil o_i \rceil$ to further constrain st. If no suitable object version exists, we have to abort. Note that if we read an object several times, this will also ensure that we always read the same object version (and thus keep the snapshot atomic) because of how we constrain $\lceil st \rceil$.

Up to now, we have only considered constraining $\lceil st \rceil$ (e. g., by validity ranges or by the current time in stm−start). However, we can also check whether our snapshot would still be valid at a more recent snapshot time and not capped by more recently committed object versions. We call this a *snapshot extension* and it is performed by the function extend, which iterates through the read set and recomputes a new upper bound for st based on the object versions' validity ranges (line 42) and a caller-supplied upper bound t (line 39). Note that t is always less than or equal to the current time, so we never speculate about what might nor might not happen in the future (lines 14, 29, and 47).

[5] We could also set it to $[clock, \infty]$ but would then need to use a different representation of locked object versions than ∞. For brevity, Algorithm 1 constrains st right away.

The validation of the readset during snapshot extensions is thus similar to what we would do when using incremental validation; however, we only need to extend *on demand*, which also motivated the name *Lazy* Snapshot Algorithm. We only need to extend if another conflicting update transaction committed concurrently, but not in the assumed common case; even if we have to extend, we will be on somewhat of a slow path anyway because we are reading a fresh update by another thread and thus will suffer from cache misses. Overall, this leads to the small and typically constant runtime overhead for stm−load that leads to the performance advantage of LSA shown in Figure 5.1.

Update transactions. While we can commit read-only transactions right away (line 44) because their snapshot is always kept atomic and never earlier than the transaction started, we have to ensure for update transactions that the snapshot time range and the commit time overlap.

Therefore, an update transaction must have only the most recent versions of each accessed object in its snapshot. This is ensured by the checks on line 30 in stm−store and line 20 in stm−load. In turn, we rely on this on lines 12 and 41 (i. e., if we have written an object, there is no other object version ordered between the version in our snapshot and $o_\star$).

Update transactions commit by first acquiring a new commit time ct from the global time base (line 45). Next, they ensure that $\lceil st \rceil$ is equal or larger than $ct - 1$, attempting a snapshot extension if necessary, and aborting if it cannot be ensured (lines 46–49). Thus, st and ct are adjacent on the global time base; because ct is a unique timestamp acquired by this transaction, st and ct essentially overlap and there cannot be another transaction whose commit would prevent our snapshot from being atomic with our updates and the commit time acquisition. This works again similar to two-phase locking, only that our reads are invisible—but using time-based validation, we ensure that we could have held read locks until after acquiring ct during commit.

The only remaining step is to release all write locks, which we can do by setting the lower bound of the validity ranges of the written object versions to ct (lines 50–51).

Essential ordering constraints. The correctness of LSA relies on a few ordering and atomicity constraints ensured by the algorithm. First, on transactional loads, we obtain an atomic snapshot of both the data and the commit time associated with this data. While this is simply assumed to be the case in Algorithm 1, concrete implementations have to enforce this (see Algorithm 3). As a result, we can reason about snapshot consistency based on just the timestamps.

Second, a transaction makes its updates visible to other transactions (e. g., by installing tentative versions as in Algorithm 1) *before* obtaining a new commit time. Thus, the commit time is greater than any value that the global time base had when any of these updates were not yet visible to other transactions. This ensures that any other transaction making a snapshot at or after the updating transaction's commit time will also see the updates (or, alternatively, that the updated objects are still inaccessible or locked because the updates have not finished committing yet). Updates are made accessible only after it has been ensured that the transaction is actually allowed to commit.

As a result, updates are virtually committed atomically with obtaining the commit time (i. e., right between the commit time and the previous timestamp). Also, this ensures that reading transactions will observe the effects of other update transaction completely or not at all.

To ensure that a transaction's updates and its snapshot are consistent, updating transactions check that their snapshot is still valid at the time of the commit. While there exist different variations of this (see below), the underlying key principle is that if such a check succeeds, we know that the snapshot is atomic with respect to obtaining the commit time; because the updates are atomic with respect to the commit time too, both the snapshot and the updates are atomic with respect to each other.

Thus, the global time base and how transactions link their updates and snapshots to timestamps from this time base effectively orders transactions. Note that this is not necessarily a total order; for example, transactions that do not have data conflicts with each other (e. g., two read-only transactions accessing the same object) need not be ordered.

Algorithm 3 enforces a total order for all update transactions by letting them all acquire a commit time on their own, but this is not strictly necessary. It is required that all updates of a transaction have the same commit time (to make snapshots work), but updates having an equal commit time does not necessarily imply that those updates were by the same transaction.

Essentially, concurrent update transactions with no data conflicts can share a commit time. This is possible as long as the commit time they obtain is still greater than any value of the global time base when their updates were not yet fully visible to other transactions (see above). However, if transactions do share commit times, they cannot skip validating their snapshot if the snapshot time is right before their commit time (lines 46–49 in Algorithm 3). Instead, they always have to validate that their snapshot is right before their commit time and no other transaction has tentative conflicting updates pending; those updates could be by other transactions that use the same commit time. The LSA variants using a real-time clock as time base rely on this approach (see Section 5.3), as does related work (see Section 5.4).

Write buffering. Algorithm 1 shows a write-through variant of LSA, but an STM can also buffer writes until commit as outlined in Algorithm 2 (the functions ending in −wb then comprise the external STM interface). This *write-back* variant really just defers making writes visible until right before commit; the allowed externally visible behavior of a transaction and the essential ordering guarantees do not change. Note that we could also defer writing back real values to the to-be-committed object version until after validation in commit as long as this happens before we release the write locks (line 50 in Algorithm 1, not shown in Algorithm 2 for brevity). Furthermore, we could also acquire locks early as in the write-through variant but never write data updates until we are sure to commit, leading to a variant called *encounter-time locking* [42].

Discussion. In what follows, I will conclude with a discussion of some performance-related properties of LSA and time-based STMs in general.

First, one might expect that LSA needs to perform extensions frequently when there are concurrent updates. In fact, however, LSA is quite independent

Algorithm 2 Lazy Snapshot Algorithm (write-back variant, extends Algorithm 1)

```
 1: State of thread p:                                   ▷ extends state of Algorithm 1
 2:    w-buffer: buffered writes per object

 3: stm-start-wb()_p:
 4:    stm-start()
 5:    w-buffer ← ∅

 6: stm-load-wb(o)_p:
 7:    if ∃w-buffer_o then
 8:       return w-buffer_o                              ▷ Read buffered write
 9:    return stm-load(o)

10: stm-store-wb(o,val)_p:
11:    w-buffer_o ← val                                  ▷ Buffer store to o

12: stm-commit-wb()_p:
13:    for all o : ∃w-buffer_o do   ▷ Acquire objects with buffered writes before trying to commit
14:       stm-store(o, w-buffer_o)
15:    stm-commit()
```

of the speed in which concurrent transactions commit and thus increase the value of the global time base. If there are no concurrent updates to the objects that a transaction accesses, the most recent object versions do not change and no extension is required for obtaining a consistent snapshot. This is the case, in particular, if the value of clock has not changed since the start of the transaction. If *clock* has been increased concurrently and the transaction is an update transaction, then one extension is required during commit. Extensions are thus only required due to concurrent conflicting commits, and thus after cache misses. Also, at most one extension is performed per accessed object, but this worst case is extremely rare in practice because it requires very specific update patterns.

Accesses to the global time base might become a bottleneck when update transactions execute frequently due to cache misses and contention on *clock*. In practice, however, the number of accesses to *clock* remains small. All transactions must read the current time once when they are started, and update transactions must additionally acquire a unique commit time. Further accesses are not required for correctness. For example, if an update transaction needs to access a version more recent than its current validity range, it can extend the snapshot's upper bound up to any time at which the version was valid, not necessarily up to the current time (as shown in Algorithm 1). Time information gathered from the accessed objects can thus be used instead of reading the global time base. Alternatively, *clock* can also be replaced by more scalable alternatives such as approximately synchronized clocks (see Section 5.3), and update transactions can share commit times as discussed previously.

LSA can but does not need to maintain multiple object versions. The number of versions that are kept can have an influence on the likelihood that a read-only transaction can commit successfully, but it also leads to additional runtime overheads.[6] Thus, other implementations such as the one for C/C++ environments that I will describe in Section 5.2 do not maintain multiple object versions.

[6]The potential of keeping multiple versions is further discussed elsewhere [40].

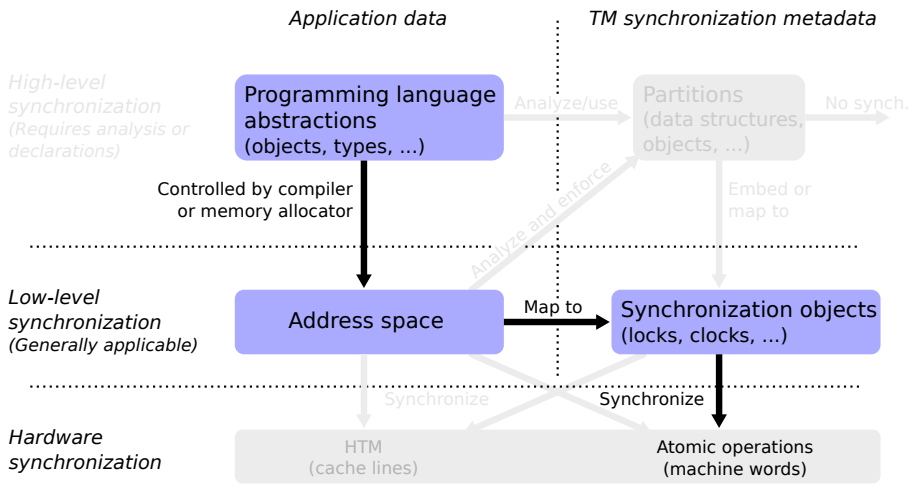

Figure 5.2: TM-based synchronization: STM.

5.2 Time-Based STM for C/C++ Environments

In this section, I will describe how to implement LSA (Section 5.1) in a C/C++ environment and so that it fulfills the requirements for TM runtime libraries laid out in Section 4.2.4.

The most important difference to the abstract LSA presented previously is that we cannot rely on having a set of discrete shared objects anymore because C/C++ programs can access arbitrary memory locations using pointers. While we could try to find the object associated with a memory location using a runtime lookup, the runtime overhead of this would be prohibitive. Therefore, the STM is *word-based* and maps low-level memory locations to STM metadata using a hash functions that cover the whole address space; Figure 5.2 highlights the relevant part of the spatial aspect of TM-based synchronization (see Section 3.1.2 for a description). In Chapter 6, I will describe how to use compile-time techniques to also exploit high-level mappings in C/C++ environments.

Low runtime overheads are essential for a TM implementation. Most importantly, if the programmer already optimized a program's transactions so that they conflict infrequently (i. e., disjoint-data parallelism, see Sections 3.1.3 and 3.2 for further discussion), we want to execute those transactions as quickly as possible. LSA already targets this scenario (e. g., it preserves disjoint-access parallelism by synchronizing on individual objects instead of a single global lock), so the implementation should also be optimized toward it.

A *blocking* STM implementation is easier to implement and can also perform better when aborts are infrequent because the transaction does not need to handle being involuntarily aborted by other transactions or other kinds of interference; after acquiring a lock, the associated data belongs to the transactions and it can access it without requiring further synchronization.

A *write-through* approach for transactional stores helps to keep overheads low because transactions do not need to buffer writes and look up previous

writes in the buffer on read–after–write situations.[7] Instead, the STM can just acquire locks for transactional data that is to be updated and modify the memory locations in place without a need for any software indirection layer.

Maintaining only a *single version* of each memory location instead of keeping additional older versions of the data is also more efficient for update transactions (e. g., no preservation of old data, no garbage collection for obsolete versions, etc.). Only read-only transactions can benefit from older versions when using LSA, but only the most recent version is required to be available; for a write-through, word-based STM this simply is the value at the particular memory location.

For these reasons, the STM that I will present next is a word-based, blocking, write-through, and single-version implementation based on LSA.

5.2.1 LSA on C++11

The LSA-based STM implementation that I will discuss in what follows is both for C++11 and based on it: First, it provides the guarantees that the C++ TM specification requires for TM runtime libraries (see Section 4.2.4), and can be used to implement the respective ABI. Second, the implementation itself is based on C++11 in that it uses C++11's memory model and atomic operations to implement the synchronization between transactions.

The high-level requirements on TM runtime libraries are shown in Table 4.2 on page 56. I will just give an overview of these requirements for now, and discuss them in detail after describing the TM algorithm.

The first requirement, L1, essentially states that transactions need to be totally ordered, called Transaction Synchronization Order (TSO). The way how LSA and time-based STMs in general use snapshot and commit times from a global time base achieves such an ordering. In particular, TSO is consistent with the ordering of commit times and snapshot times of transactions; whenever those times are not ordered, the respective transactions do not conflict and the program also cannot observe that, due to the data-race freedom requirement of C++11.

Snapshot times are a tentative TSO choice. Trying to extend a snapshot to the current time from the global time base (i. e., validating that the snapshot has not changed in the meantime) checks whether the transaction can change its position in TSO without this being visible to TM-pure or unsafe code.

The second requirement, L2, states that TSO and the transactional memory accesses must be consistent with *happens–before*, and that data races must not be introduced. The former essentially requires the TM runtime library to preserve *happens–before* relationships established in nontransactional code, and I will discuss later how the algorithm ensures that. The latter forces the library to ensure privatization safety and to access only exactly the data that the transaction would access if executed by the C++ abstract machine.[8]

[7] Note that in a different workload scenario in which transaction aborts are frequent, a write-back approach can be better due to a smaller window of interference between concurrent conflicting transactions.

[8] For example, when rolling back writes of an aborted transaction, the TM must not undo those writes at a coarser granularity than the original memory accesses because this could overwrite adjacent memory objects that could be concurrently accessed by other threads.

Ownership records (orecs). Unlike in the LSA version that I presented previously, we do not want to maintain multiple versions of each memory object; instead, transactions modify objects in the program's address space directly. Transactions use *ownership records* to synchronize with each other, which are essentially custom locks carrying a timestamp. Before updating a certain memory location, transactions first acquire the associated orecs; when committing an update, orecs are released and the timestamp is updated so that reader transactions are aware of the update.

An orec is a machine-word–sized data structure whose most-significant bit serves as a lock bit: If it is set, the orec is acquired and the remaining bits identify the transaction that acquired it (e. g., a pointer to the per-thread metadata that the TM runtime library maintains). If the bit is not set, the remaining bits are split up between a timestamp from the global time base (i. e., the commit time of this update, as we will see later) and an *incarnation number* in a few least-significant bits.

The incarnation number serves a different purpose than the timestamp: It does not represent when the data was committed but can instead be used to decrease runtime overheads of aborted update transactions and transactions conflicting with those aborted transactions. An incarnation number is optional and only useful in write-through designs, but can increase performance in exchange for reserving a few bits.

Putting the lock bit, timestamp bits, and incarnation numbers bits in this order from most significant to least significant allows for small optimizations in the TM implementation. The value of an orec with a set lock bit will always be larger than any (non-overflowing) snapshot time, so a simple comparison can check two conditions; likewise, an appropriately shifted value of a snapshot time can discard incarnation numbers in a comparison between a value of an orec and a snapshot time.

Mapping memory locations to orecs. Orecs are kept separate from the memory locations that transactions access. Thus, when a transaction accesses memory at a certain address, it must map from this address to the orec associated with this memory location. All transactions must obviously use the same mapping function because they synchronize with other transactions based on the orecs; using different mappings would prevent proper synchronization.

There are no fundamental constraints on the the number of orecs and the nature of the mapping, but there are practical constraints. Calculating which orec an address maps to is on the fast path of transactional loads and stores with the current ABI, so complex functions are likely to lead to runtime overheads. Likewise, using a larger number of orecs can make false conflicts between transactions less likely (because fewer memory locations map to the same orec) but can also result in higher memory requirements and cache footprint.

These concerns motivate the use of simple hash functions, for example splitting the address space into equally-sized stripes and mapping those to entries in an array of orecs. I will discuss these trade-offs further in Section 5.2.2. For what follows, it is sufficient to assume some deterministic mapping function that is used by all transactions (denoted `hash` in the algorithms).

Algorithm 3 Lazy Snapshot Algorithm (C++11-based)

```
 1: Global state:
 2:   clock ← 0                                          ▷ Global time base (shared integer)
 3:   orecs: array of word-sized ownership records, each consisting of:
 4:     locked: bit indicating if orec is locked
 5:     owner: thread owning the orec (if locked)
 6:     time: commit timestamp (if ¬ locked)
 7:     inc: incarnation number (if ¬ locked)

 8: State of thread p:
 9:   st: snapshot time (only upper bound)
10:   r-set: read set of tuples ⟨addr, time⟩
11:   w-set: write set of tuples ⟨orec-index, orec-value⟩
12:   undolog: undo-logging data (sequence of tuples ⟨addr, val⟩)

13: stm-start()_p:
14:   st ←_acq clock
15:   r-set ← w-set ← undolog ← ∅

16: stm-load(addr)_p:
17:   orec ←_acq orecs[hash(addr)]
18:   if orec.locked then
19:     if orec.owner ≠ p then
20:       abort()                                       ▷ Orec owned by other thread
21:     return *addr                                    ▷ We own the orec; just read through
22:   if orec.time > st then                            ▷ Need to extend snapshot?
23:     extend()                                        ▷ Aborts if validation fails
24:   val ←_acq *addr
25:   if orecs[hash(addr)] ≠ orec then                  ▷ Load again and compare with previous load
26:     abort()                                         ▷ Data at addr was perhaps modified concurrently
27:   r-set ← r-set ∪ {⟨addr, orec.time⟩}               ▷ Add to read set
28:   return val

29: stm-store(addr,val)_p:
30:   orec ← orecs[hash(addr)]
31:   if orec.locked then
32:     if orec.owner ≠ p then
33:       abort()                                       ▷ Orec owned by other thread
34:   else
35:     if orec.time > st then                          ▷ We may have read from addr before, so...
36:       extend()                                      ▷ ...abort if validation should fail
37:     if ¬ cas_acq(orecs[hash(addr)] : orec → ⟨true, p⟩) then   ▷ Try to acquire orec
38:       abort()
39:     fence_rel                                       ▷ Memory barrier with release memory order
40:     w-set ← w-set ∪ ⟨hash(addr), orec⟩
41:   undolog.push(⟨addr, *addr⟩)                       ▷ Log previous value of *addr
42:   *addr ← val                                       ▷ Write through to memory

43: stm-commit()_p:
44:   if w-set ≠ ∅ then                                 ▷ Nothing to do if read-only transaction
45:     ct ← atomic-inc-and-fetch_acqrel(clock)         ▷ Unique commit time (atomic increment)
46:     if st < ct − 1 then                             ▷ Must validate if others committed in the meantime
47:       extend()                                      ▷ Aborts if validation fails
48:     for all ⟨orec, orecval⟩ ∈ w-set do
49:       orecs[orec] ←_rel ⟨false, ct, 0⟩              ▷ Release orecs

50: extend()_p:
51:   st ←_acq clock
52:   for all ⟨addr, time⟩ ∈ r-set do                   ▷ Are orecs free and timestamps unchanged?
53:     orec ← orecs[hash(addr)]
54:     if (orec.locked ∧ orec.owner ≠ p) ∨ (¬ orec.locked ∧ orec.time ≠ time) then
55:       abort()                                       ▷ Inconsistent snapshot
```

Algorithm 3 Lazy Snapshot Algorithm (C++11-based, continued)

```
56: abort()_p:
57:   undolog.rollback()                                  ▷ Undo previous writes in reverse order
58:   ct ← 0
59:   for all ⟨orec, orecval⟩ ∈ w-set do
60:     if incarnation-left(orecval.inc) then             ▷ No incarnation number overflow?
61:       orecs[orec] ←_rel ⟨false, orecval.time, orecval.inc + 1⟩  ▷ Release orec (new incarnation)
62:     else
63:       if ct = 0 then                                  ▷ Acquire new "commit" timestamp
64:         ct ← atomic-inc-and-fetch_rel(clock)
65:       orecs[orec] ←_rel ⟨false, ct, 0⟩                ▷ Release orec (new timestamp)
```

Description of the algorithm. Algorithm 3 shows the C++11-based version of LSA. Even though I still show it in terms of pseudo-code, it is based on the memory model of C++11. Unlike for Algorithm 1, functions are not assumed to be atomic anymore. However, all individual memory accesses to global state (including application data) are assumed to be atomic and with relaxed memory order as default. Atomic operations that require stronger memory orders are annotated with the respective order (see Table 2.1 on page 12).

I only show load, store, start, and commit functions in Algorithm 3, but the other functions that are part of the TM runtime library ABI are either straightforward to implement, or are load or store variations. Also, I will focus on the differences to Algorithm 1 in the following description, and will discuss why certain memory orders and barriers are required afterwards; for now, it is sufficient to assume that all atomic operations are sequentially consistent.

The stm−start function is similar to Algorithm 1, except that the snapshot is now characterized by just a single value—and not an interval—which is initially set to the value of the global time base when the transaction starts (line 14); we do not keep multiple versions for memory objects, so we really are only interested in the upper bound of the interval.

Transactional stores first map the target address to an orec, and then load the value of this orec (line 30). If the orec is locked by some other transaction, we abort the transaction. If the orec is not locked, we have to acquire it before we can write through to memory (line 37); using CAS makes sure that lines 30–37 are atomic with respect to other modifications of the orec[9]. However, if the transaction has read from memory mapped to the same orec before, then we need to make sure that no other value has been committed in the meantime; given that update transactions need to extend the snapshot at commit anyway, we can also try to extend the snapshot right away (line 35). Note that unlike in Algorithm 1, unsuccessful snapshot extensions abort the transaction. After successful orec acquisition, we issue a release memory barrier[10] and add the orec to the write set. Finally, we perform undo logging and write through to memory (lines 41–42).

When a transaction commits, it follows essentially the same steps as in Algorithm 1. However, we release the orecs that we have acquired (lines 48–49), instead of making the most recent memory object versions accessible (we already have written updates trough to memory in stm−store). When releasing an orec,

[9] See the stm−commit and abort functions for how those prevent ABA issues.

[10] Instead of the barrier, we could also require release memory order for all the stores to the application data, but this is likely to be more expensive (e. g., if there is more than one write per orec).

we set its timestamp to the transaction's commit time, making it accessible only to transactions with a sufficiently large snapshot time. Also, the incarnation number is set to zero because we use a new timestamp. Note that even though the stm−commit function is not atomic, it provides the essential ordering (see Section 5.1.1) between updates being inaccessible (i. e., orecs locked), acquisition of a new commit time (line 45), snapshot time extension if necessary (line 47), and making updates accessible again by releasing the orecs.

Aborting a transaction is slightly more complex because of incarnation numbers, but essentially we just have to use the undolog to roll back previous updates to memory (line 57) and then release the orecs. For the latter, we need to notify reading transactions about potentially inconsistent reads of data by making sure that the orec's value after being released differs from the value it had before we acquired it. The first way to achieve this is to acquire a new timestamp from the global time base (line 64) and then release the orec as if we would when committing the transaction (line 65). To other transactions, this kind of rollback will look like just another write-only transaction that committed but did not change any data; this is safe because we have acquired all the orecs for the data we updated. However, this requires accessing the global time base and potentially acquiring a new commit time from it, which can increase contention on the time base and thus reduce performance. Alternatively, we can keep the timestamp of the orec unchanged but instead increment the incarnation number (line 61) as long as it does not overflow. This will also let readers detect potentially dirty reads, as I will explain next.

Transactional loads are somewhat different than in Algorithm 1 because we cannot assume that stm−load executes atomically. We first map the target address to an orec, and then load the value of this orec (line 17). If the orec is locked, then we either abort if it is locked by some other transaction or we have a read–after–write situation and can thus just read the data (line 21). If the orec is not locked, then we must try to extend the snapshot if our snapshot time is not recent enough to form an atomic snapshot (line 23); we do not have multiple object versions available as with Algorithm 1, so we must read the data that has been committed most recently. Next, we can read the data from the target address (line 24). Note that this read can be pending in the sense that it is not atomic with the previous load of the orec (and thus the checks of the orec's value); the orec can change in the meantime. Ensuring privatization safety, which I will discuss below, also ensures that such pending reads are harmless.

We can make reading the orec and the data effectively atomic by reading the orec's value again after reading the data and aborting if the orec's value has changed since the first read (line 25). The value could have changed if either a new update has been committed in the meantime (i. e., *orec.time* changed) or if we potentially read uncommitted data (i. e., *orec.inc* changed or the orec is now locked). When transactions change data, they always change some part of the orec's value (lines 37, 49, 61, and 65), and in a way that avoids any ABA issues. Thus, reading the orec's value twice is like validating the single data load, and will allow us to detect any inconsistencies; checking atomicity of the whole snapshot of the transaction is still based on time-based validation.

Snapshot extensions validate similarly to per-load validation. We first read the current value of the global time base (line 51), which becomes our new snapshot time if the snapshot extension succeeds. After this, we check that all orecs in the read set are either locked by us, or not locked and their timestamp

has not changed (line 54). Note that changes just to the incarnation numbers are fine; we had consistent reads for each orec initially, and different incarnations—but with the orec not locked—are just concurrent aborted transactions that have been already rolled back. Besides reduced contention on the global time base, this is another advantage of using incarnation numbers: They can reduce the number of extensions and aborts caused by concurrent yet aborted transactions. If validation of any orec in the snapshot fails, the transaction aborts.

For update transactions that have also read data, a successful snapshot extension thus also checks that the snapshot is still valid after the commit time has been acquired.

Also, some smaller optimizations are not shown in Algorithm 3. For example, instead of aborting immediately, transactions can also spin for a while if an orec is already acquired, in the hope that the other transaction might commit or abort soon.

Privatization safety. Privatization refers to a transaction making some data inaccessible to other threads, and the TM runtime library has to ensure that this can be safely done (see Section 4.2.4 for details). The privatizing transaction thus must be an update transaction; it has committed and thus fixed its position in TSO, but this does not immediately make other transactions aware of this. In particular, other transactions are not aware of the privatization iff their snapshot time is less than the privatizing transaction's commit time.

If privatization is not safe, there are a few things that can go wrong. First, transactions can use private and thus potentially inconsistent data or can write to private data. This will lead to either transactions or nontransactional code operating on inconsistent data, which in turn can lead to incorrect behavior of the program. Such behavior cannot be easily contained, especially in a typical C++ implementation. STM algorithms such as NOrec (see Section 7.3.2) that rely on a centralized commit phase implicitly prevent this first kind of incorrect behavior but at the expense of less scalability.

Second, STMs that use invisible reads will have pending loads (see the previous discussion of stm−load), which can target privatized data. This is also the case for the NOrec STM (see Sections 7.3.2 and 5.2.2), for example, which will never return the value of such a load to the transaction but will read privatized data. While the data race that the load causes is benign on typical architectures, accessing data for which the privatizing thread has changed the memory protection properties (e. g., by releasing memory or re-mapping the respective memory page as read-only) is not benign and can lead to memory protection faults.

Some architectures, such as SPARC, provide hardware instructions for non-faulting loads; using those for data loads together with an STM like NOrec can avoid the privatization problems. However, non-faulting loads are not available on many other common architectures such as x86.

Another way to make the pending loads harmless would be to try to mask protection faults caused by transactions. However, this either requires custom support in the operating system, or custom signal handlers and enforcing that the TM runtime library's signal handlers are always the first to be called for a memory protection fault, which in turn likely requires custom standard library support.

We can also ensure privatization safety purely at the level of the TM runtime library by letting potentially privatizing transactions never return to nontransactional code until all other concurrent transactions are aware of the privatizing transaction's commit. Informally, privatizers thus wait for quiescence of older snapshots.

To achieve that, all transactions publish their snapshot time regularly in per-thread variables. They have to do so at least when starting a transaction and after committing or aborting, marking the transaction as inactive in the latter two cases. They can also update their published snapshot time after successful snapshot extensions. Update transactions[11] then have to wait until all concurrent transactions either became inactive or have a snapshot time that is equal or larger than their own commit time. This ensures that there is global consensus on TSO (up to the update transaction's commit time) before it is exposed to nontransactional code (e. g., code changing the privatized data). Algorithm 3 does not show this privatization safety implementation, but it is straightforward to build using atomic operations and release and acquire memory orders.

The disadvantage of this approach is that it introduces a delay before update transactions can return to nontransactional code, either due to having to wait for older transactions or because of suffering from cache misses on the published snapshot times of other threads. However, the big advantage in practice is that this works across all architectures and without having to rely on custom support in either the operating system or the standard libraries.

Correctness. In what follows, I will show that Algorithm 3, including the privatization safety implementation as described previously, satisfies the requirements on TM runtime libraries (see Table 4.2 on page 56).

The timestamps used in LSA directly correspond to the ordering of transactions in TSO, and Algorithm 3 ensures that transactional data loads and stores are consistent with this order. It does so by establishing *synchronizes–with* relationships through pairs of release and acquire memory orders (MOs) on certain atomic operations that are in a reads-from relationship.

First, release MO when releasing orecs (lines 49, 61, and 65) synchronizes with acquire MO loads from those orecs (line 17). Thus, data stores that happen before releasing an orec (i. e., commits or rollbacks) happen before data loads associated with the same orec. Those data loads could read from data stores that did not happen before the release of the orec but later (e. g., uncommitted data stores), but this case will be detected by the second orec load (line 25), as we will see later.

Similarly, release MO when releasing orecs also synchronizes with acquire MO on the CAS that acquires an orec (line 37). This is like ordinary locking, so data stores overwrite the most recent updates of the data, and read–after–write situations are also covered.

Second, the release MO memory barrier sequenced before data stores (line 39) synchronizes with acquire MO on data loads (line 24) to the same memory location. Therefore, the acquisition of the orec (line 37) before the data store happens before the second load of the orec (line 25) following the matching data load. This means that if the data load reads potentially uncommitted data, then the second load of the orec will either observe the orec to be locked

[11] Read-only transactions cannot privatize data.

by the other transaction, or have a different commit timestamp or incarnation number. In either case, the values of the first (line 17) and second load of the orec will differ; they will only be equal if the data load read from the most recent data store before the orec was released. Thus, transactional loads will abort if the loads of the orec and the data were not effectively atomic.

Third, release MO when incrementing the global time base (lines 45 and 64) synchronizes with acquire MO on loads of the global time base (lines 14 and 51).[12] Thus, changes to orecs that happened before incrementing the global time base will happen before anything that a transaction does after deciding to work on a particular snapshot time. In particular, snapshot extensions and transactional loads will observe all finished or unfinished commits that have commit times equal or less than this snapshot time. For example, assume an update transaction that has just acquired commit time t but failed validation and is now about to roll back: A concurrent conflicting transaction building a snapshot also at time t is guaranteed to observe the orecs still being acquired by the update transaction, and will not build an inconsistent snapshot. This also shows why orec acquisition must happen before acquiring a commit time.

Considering all those three constraints on *happens–before* together guarantees that Algorithm 3 follows the same abstract synchronization scheme as Algorithm 1. Transactions build atomic snapshots that contain the most recent data at this snapshot time according to TSO, and update transactions are also totally ordered by TSO. Concurrent read-only transactions can have the same snapshot time, but this does not invalidate the total ordering requirement of TSO because a data-race–free program cannot observe the difference; thus, this is a valid application of the as–if rule (see Section 4.2).[13]

Besides establishing a TSO and making transactional memory accesses consistent with TSO, Algorithm 3 has to satisfy further requirements to be correct.

First, changes in tentative TSO positions of transactions must be invisible to the program, and indeed they are: A transaction can extend its snapshot time only if none of the orecs in its read set changed. Thus, it is guaranteed that the data did not change either in a data-race–free program, so executing the transaction at the new snapshot time—and thus a different position in TSO—would have returned the same values.[14]

Second, TSO contributes to *happens–before* simply because the TM synchronizes based on the same memory model, so *happens–before* relationships will be established by transactions. Likewise, TSO is also consistent with *happens–before*: If synchronizing actions in nontransactional code establish *happens–before* relationships, then these will affect and constrain which snapshot time a transaction starts at.

Third, publication via transactions also works correctly because a correct

[12] Algorithm 3 also shows acquire MO for the increment of the global time base during commit (line 45). Technically, this is just required when commit uses an optimized version of the extend function that just validates but does not read the global time base with acquire MO.

[13] Note that this reasoning holds just for atomic transactions that cannot communicate with nontransactional code. However, relaxed transactions might need to execute unsafe code to communicate with other threads, in which case they will switch to serial-irrevocable mode first as I will explain subsequently.

[14] Again, this expects atomic transations that cannot communicate with other threads except through the TM runtime library. Relaxed transactions that do communicate will switch to serial-irrevocable mode first.

publishing transaction happens after the nontransactional operations on the published data, and because the compiler preserves the ordering of accesses in transactions (see Section 4.2.3).

Finally, data races are not introduced because transactions access only the data that the abstract machine would access for an execution with the particular (tentative) TSO choice (i. e., there are no access-granularity data races). Transactions do make tentative TSO choices and thus can have pending loads and stores; however, those target only data that could have been validly accessed by a transaction in some execution. Privatization safety implemented as outlined previously ensures that a transaction that potentially privatizes data does not return to nontransactional code before all concurrent active transactions reached consensus on TSO up to the where the privatizing transaction committed. Thus, after this consensus, the concurrent transactions cannot introduce data races on the privatized data based on a tentative TSO choice anymore.

Serial-irrevocable mode. Relaxed transactions (see Section 4.1) can execute unsafe code for which we cannot guarantee isolation from other transactions. Therefore, relaxed transactions enforce serialized execution with respect to other transactions before executing unsafe code (i. e., they switch to serial-irrevocable mode). Once they execute in isolation, they try to commit what has been executed so far and if this is successful, continue executing the rest of the transaction without using the TM algorithm. Serialized execution of transactions trivially satisfies the correctness requirements for a TM runtime library.

Code to provide serial-irrevocable mode is not shown in Algorithm 3 but can be implemented independently of it using a multiple-reader–single-writer lock: Transactions that need to execute in isolation try to acquire the lock as a writer, and all other transaction execute as readers and synchronize using a TM algorithm. We can optimize this lock for non-serial execution being the common case by using Dekker-like synchronization: Every thread has a flag that states whether a non-serial transaction is or wants to become active, and there is a global flag that threads that want to execute in isolation try to acquire. In the absence of serial-irrevocable–mode transactions, threads thus can use such a readers–writer lock without suffering from cache misses or contention.

5.2.2 Performance Trade-Offs and Evaluation

In this section, I will evaluate the performance of LSA for C++11 (Algorithm 3). The focus will be on scalability (i. e., how much performance a TM can provide when an increasing number of threads execute potentially conflicting transactions) and runtime overheads (i. e., how much overhead the TM introduces for transactional code compared to a nontransactional execution of the same code). However, how the TM maps from memory locations to orecs (i. e., the hash function used in Algorithm 3 as well as the size of the array of orecs) can have a very large influence on performance. Therefore, I will investigate the performance trade-offs for this mapping and suitable hash functions first before looking at scalability and runtime overheads.

The machine used for the performance measurements in this section is a two-socket x86-64 NUMA system with two two six-core CPUs[15] with hyper-

[15]Intel Xeon X5650 running at a clock speed of 2.67GHz.

threading enabled. Thus, it provides 24 logical CPUs in total to the operating system. Each socket represents one NUMA node. Benchmark threads are pinned to logical CPUs, alternating between NUMA nodes and first filling up CPU cores before making use of hyper-threading.[16] This scheme will lead to NUMA-related slowdowns to be visible as soon as benchmarks are executed with multiple threads, and will not lead to hyperthreading-induced slowdowns until higher thread counts, when the likely higher level of contention causes memory to become more of a bottleneck.

The benchmarks that I will consider are Genome, KMeans-Lo, KMeans-Hi, Vacation-Lo, Vacation-Hi, and SSCA2 (see Table 3.2) as well as all the IntegerSet microbenchmarks (see Table 3.1). Note that I will show execution times for STAMP (i. e., less is better) but transaction throughput for IntegerSet (i. e., more is better).

The STM implementations that I evaluate here are part of TinySTM++. See Section 3.4.2 for further details about the implementations.

Memory–To–Orec Mapping

With orec-based STMs such as LSA, the detection of conflicts between transactions happens on the orecs and not on the original memory location. Thus, the memory–to–orec mapping affects every transactional memory access and thus the performance of the whole TM.

When choosing a mapping, we face several trade-offs. To keep the runtime overheads of transactions as low as possible, a coarse-granular mapping would be best so that a transaction has to acquire or validate as few orecs as possible. However, a fine-granular mapping can help to avoid *false conflicts* between transactions (i. e., conflicts that exist on the level of orecs but not when considering the actual accesses to application data), so that the TM does not provide less scalability than available in the workload. But if the workload does in fact not scale, then it can again be better to use a coarse-granular mapping so that transactions pass the bottleneck as quickly as possible.

To establish a coarse-granular mapping, we can either (1) map several memory locations to the same orec or (2) increase the *minimum granularity* of memory accesses considered for conflict detection. Using the latter to some extent is probably always required. For example, if most memory accesses in a transaction target pointers, then we certainly do not want to acquire or validate a separate orec for each byte of each pointer; instead, when using machine words as minimum granularity, we only have to deal with one orec per pointer and still will not increase false conflicts if pointers are always aligned to words. Using a smaller granularity than machine words could help in some applications, but is probably not useful in general because many concurrent data structures are based on pointers or machine-word-sized integers (e. g., types such as `size_t`). Also note that the granularity affects the other components of the mapping too because it changes the input values for those, as we will see later.

Another trade-off exists regarding the *number of orecs* that the TM should use. Using a large number of orecs allows for a potentially more fine-grained

[16] For example, thread 1 is on the first core of the CPU that comprises the first NUMA node, thread 2 is on the first core of second NUMA node CPU, thread 3 is on the second core of the first NUMA node CPU, etc.; thread 13 shares the same CPU core as thread 1, thread 14 the same as thread 2, and so on.

concurrency control because the TM needs to map fewer memory locations to the same orec. However, a large number will also increase the space overhead, and can—depending on the hash function—increase a transaction's footprint in caches and other hardware resources such as TLBs; especially the latter can also affect the performance of nontransactional code. On the other hand, a small number of orecs will less likely cause these problems but instead forces the hash function to try harder to avoid false conflicts.

Finally, the computational overhead of the hash function itself is also an important factor because it is executed once for each transactional memory access.[17]

This list of trade-offs already shows that choosing—or finding—a good hash function is nontrivial. What makes this even more difficult is that the input values to the hash function (i. e., the addresses of memory objects of the application) are controlled by entities outside of the control of the TM runtime library: Compilers, linkers, memory allocators, and the memory use patterns of the application itself. When the TM runtime library has to perform a transactional memory access, it cannot associate this access with a high-level memory object anymore (see Figure 5.2; alternatives based on compile-time analysis are discussed in Chapter 6).

Furthermore, the layout of application data in the address space affects the performance of more than just transactional code. For example, the memory allocator in the GNU C library (glibc) lets each thread allocate memory in different regions of the address space, which reduces the synchronization-related runtime overhead of memory allocation and can also lead to lower NUMA-related memory access costs by allocating the memory pages for those regions on the same NUMA node as the region's thread. While this scheme can be a challenge for TM hash functions as we will see later, it leads to faster execution of the whole program. Thus, even if the TM could influence the memory allocation scheme, it would have to trade off benefits for transactions against benefits for everything else in a program. Second, it is not clear whether a tight integration between the TM and other system libraries provides enough benefits to justify the additional complexity and engineering costs. Therefore, I will assume that TM runtime libraries for the usage scenarios that I focus on cannot rely on being able to influence an applications use of the address space, and will just have to handle the current memory allocation schemes.

Hash functions. Algorithm 4 shows the hash functions that I will consider: Simple, Mult, and Mult32. All three have two common parameters: *Shift* (line 2) and the *number of orecs* (lines 3–4). The number of orecs must be a power-of-two value, and all hash functions return an index into the array of orecs (i. e., they return values in the range $[0, \text{orecs})$). The Shift parameter represents the minimum access granularity discussed before; Shift controls how many of the least-significant bits of an address are discarded before the address is actually hashed (e. g., see line 13).

The *Simple* hash function then just uses the least-significant bits (of the remaining bits after shifting) to index into the array of orecs (line 14). While this

[17] This is to some extent a result of how the TM runtime library ABI is designed: Each load or store function receives the absolut address or the target memory location, and not a relative address, for example.

Algorithm 4 Hash functions for mapping memory addresses to orecs on a 64b system.

```
1  // Global constants
2  unsigned shift;                    // The number of least-significant bits to discard
3  unsigned orecs_bits;               // The number of bits required to index into all orecs
4  unsigned orecs = 1 << orecs_bits; // The number of orecs must be a power of two
5
6  // Odd random numbers. The examples given are the numbers used in the evaluation.
7  uint64_t randomMult = (11400714818402800990ULL >> shift) | 1;
8  uint32_t randomMult32 = 81007;
9
10 // The simple, default mapping
11 unsigned hash_Simple(void* addr)
12 {
13   uintptr_t a = (uintptr_t)addr >> shift;
14   return a & (orecs - 1);
15 }
16
17 // Multiplicative hashing
18 unsigned hash_Mult(void* addr)
19 {
20   uintptr_t a = (uintptr_t)addr >> shift;
21   unsigned pointer_bits = sizeof(uintptr_t) * 8;
22   return ((a * randomMult) >> (pointer_bits - orecs_bits - shift)) & (orecs - 1));
23 }
24
25 // Multiplicative hashing, 32b variant
26 unsigned hash_Mult32(void* addr)
27 {
28   uint32_t a = (uintptr_t)addr >> shift;
29   return (a * randomMult32) >> (32 - orec_bits);
30 }
```

is fast to compute, it does not consider any higher-order bits at all, which will result in two addresses being mapped to the same orec whenever their distance is a multiple of $2^{orecs\_bits+shift}$. This seems to happen rather often in practice. For example, the per-thread regions in glibc's memory allocator discussed previously are aligned to power-of-two addresses. If threads then have similar allocations patterns for shared data (as with the IntegerSet benchmarks), the probability of false conflicts rises the lower the number of orecs or access granularity.

The *Mult* hash function aims at avoiding the collision issues of the Simple hash function by using multiplicative hashing [68]: Instead of using just the least-signficant bits, it multiplies the address with a random number and uses the most-significant bits of this product (line 22). Note that it still discards the least-significant bits first, which reduces the number of bits that it can take into account (i. e., the number of bits by which to shift the product to the right is reduced by the Shift number of bits). The random number must be odd and in the range of possible inputs (i. e., $[0, 2^{64-shift})$ in a 64b system); the number used in my implementation is based on the $(\sqrt{5}-1)/2$ recommendation by Knuth [68] (see line 7). The class of multiplicative hash functions with such constraints is universal in the sense that the probability of a hash collision for two different input values is bounded by $2/2^{orecs\_bits}$ [32]. In practice, this means that the TM can just pick a different random number—possibly at runtime—if it suspects the current level of false conflicts to be too high, and that doing so will with high probability lead to a better mapping eventually (if the first random number was indeed an unfortunate choice). Thus, compared to the Simple hash, Mult can avoid the 2^n-difference collisions, can steer away from other collisions

by using a different random number, and is still relatively efficient to compute with just one 64b multiplication and a few additional bit-wise operations. It is also faster than other universal classes of hash functions (e. g., computing the address modulo a prime number).

The *Mult32* hash function is a 32b variant of Mult and uses only the lower 32 bits of the address after discarding the access-granularity bits (line 28). This allows it to use just a 32b multiplication, which might be more efficient than Mult's 64b multiplication on some hardware. The choice of a small random number for Mult32 (line 8) is not completely arbitrary: it represents a contrast to the large factor used in Mult, yet the value is large enough so that low-order bits of addresses will still make a difference.[18]

Interestingly, all three hash function result in about the same single-thread runtime overheads for transactions. Running the benchmarks with a Shift value of 6 and just 8 orecs (i. e., the whole array of orecs fits into a single cache line) on the machine described previously shows that the performance difference for any pair of hash functions is not more than 5% and typically around 2%. There is no clear winner, but Mult32 often performs best and Mult worst. The same holds when increasing the number of orecs to 2^{12}. This might be different on less powerful CPUs.

Other candidates for hash functions could be xor-based functions that incorporate knowledge about common layout patterns in the address space to mix input bits with low collision probability (e. g., to avoid the issues of Simple). Open hashing (i. e., keeping a linked list of several orecs per index when false conflicts happen) probably has a too large runtime overhead because of the indirection and thus additional cache misses and footprint. Closed hashing and linear probing might be worthwhile, but I have not investigated this further.

Hash functions parameters. When we keep the random numbers used in Mult and Mult32 constant, all three hash functions are parametrized by Shift and the number of orecs. If we do not want to have to rely on automatic tuning of these parameters at runtime (which might be difficult), then we need to select parameters a priori that will hopefully provide decent performance.

I have measured the performance of LSA with each the Simple, Mult, and Mult32 hash functions, Shift ranging from 3 to 10, and the number of orecs ranging from 2^{12} to 2^{26} on the STAMP and IntegerSet benchmarks with 4, 12, and 24 threads, respectively. The performance of HashTable and SSCA2 is not significantly affected by the choice of hash function or hash function parameters, so I will not consider them in detail.

To be able to select good parameters, we first need to prune parameters that are unlikely to provide good performance; this would also be helpful if we could rely on automatic tuning.

Let us start this pruning process by looking at large numbers of orecs first; specifically, whether 2^{24} or 2^{26} orecs provide any benefit over 2^{22} orecs or less.

[18] Assuming that the smallest value of orecs_bits is 16, then the 16 low-order bits of the multiplication product will be discarded. For the least-significant bit of the address (after discarded access-granularity bits) to have a significant impact, the minimum value of the random number needs to be larger than 2^{16}; the value on line 8 is the product of $(\sqrt{5}-1)/2)$ and 2^{16}. Note that for the experiments with 12 orecs_bits , a few further least-significant bits of the address thus have less influence; nonetheless, the performance of Mult32 is still good in such a setting.

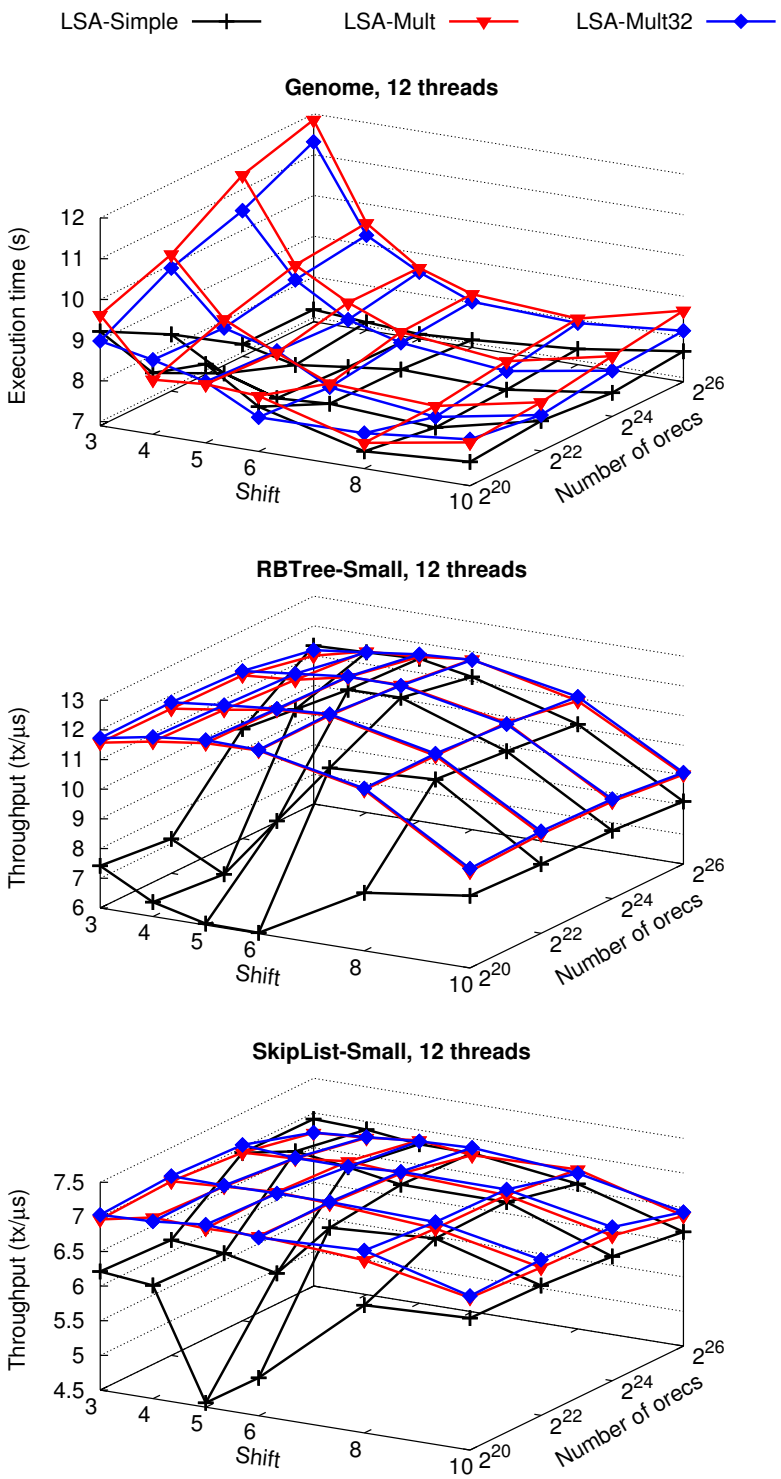

Figure 5.3: Performance of LSA with different hash functions and hash parameters for large numbers of orecs.

Figure 5.3 shows three interesting cases. In Genome, Simple can benefit only slightly from more than 2^{22} orecs, and just if Shift is less than 6. With the same benchmark but 24 threads (not shown), it can benefit a bit more and also with larger values of Shift. However, in all other STAMP benchmarks Simple does not benefit from large numbers of orecs, or is even slowed down by them (e. g., in Vacation). Mult and Mult32 never benefit or are slowed down, especially with small values of Shift as shown for Genome. The reason for the latter is likely that multiplicative hashing returns values that are distributed more randomly across the whole array of orecs, in contrast to Simple which translates locality in accesses to locality in the array of orecs; The result of such behavior is that Mult and Mult32 access a wider range of memory, and then probably suffer from the higher TLB footprint, for example. A small access granularity (i. e., a small Shift value) increases this problem because transactions then need to touch more orecs. A similar effect is visible with the IntegerSet benchmarks, but less pronounced; overall, Mult and Mult32 already reach peak performance with 2^{22} orecs or less.

In contrast, Simple can benefit from large numbers of orecs on the IntegerSet benchmarks, as shown by the RBTree-Small and SkipList-Small examples in Figure 5.3. RBTree-Small suffers especially with a small Shift, which indicates that this is caused by false conflicts for pairs of addresses that have a power-of-two distance; LinkedList-Small and LinkedList-Large show similar patterns. SkipList-Small does not show this problem but is slowed down if Shift has a value of around 5 (RBTree-Large and SkipList-Large are similar); however, with a different Shift value, 2^{22} orecs can also result in decent performance.

Overall, performance does not increase when using more than 2^{22} orecs. Second, the space overhead would just be too large to be practical, as 2^{22} orecs already require 32MB of memory on a 64b system. Therefore, I will not consider more than 2^{22} orecs.

In contrast, 2^{12} orecs result in a space overhead of just 32KB, which might be sufficient even on systems with a small amount of memory. However, do they allow for decent performance and infrequent false conflicts? Figure 5.4 shows a few examples.

For STAMP, Mult32 provides good performance in comparison to the other hash functions: It is always better or as good as the others in Genome and Vacation, but struggles with large Shift values in KMeans (but not significantly except in KMeans-Hi with 12 or 24 threads, where Mult is better). With IntegerSet, Mult32 is always better than Simple. Mult is somewhat better than Mult32 in LinkedList but there is no clear winner among the two (see Figure 5.4): Whereas Mult32 is typically better when Shift is less than 5, Mult performs equal or better starting at Shift values larger than 5. Overall, for 2^{12} orecs, Mult32 with Shift set to 4 provides good performance compared to Mult or Simple with any Shift setting. Mult with Shift set to 6 also performs well, but struggles in Vacation. Simple with Shift set to 6 works well in STAMP, but has very low performance in IntegerSet.

Notwithstanding, 2^{12} orecs never provide a performance advantage over 2^{16} orecs. The latter allow for at least 10% more performance on basically all benchmarks, they are less prone to slowdowns associated with certain Shift values (especially with Mult32 on IntegerSet and Mult on STAMP), and their space overhead of 512KB is probably still practical for many applications. Therefore, I will not consider 2^{12} orecs further.

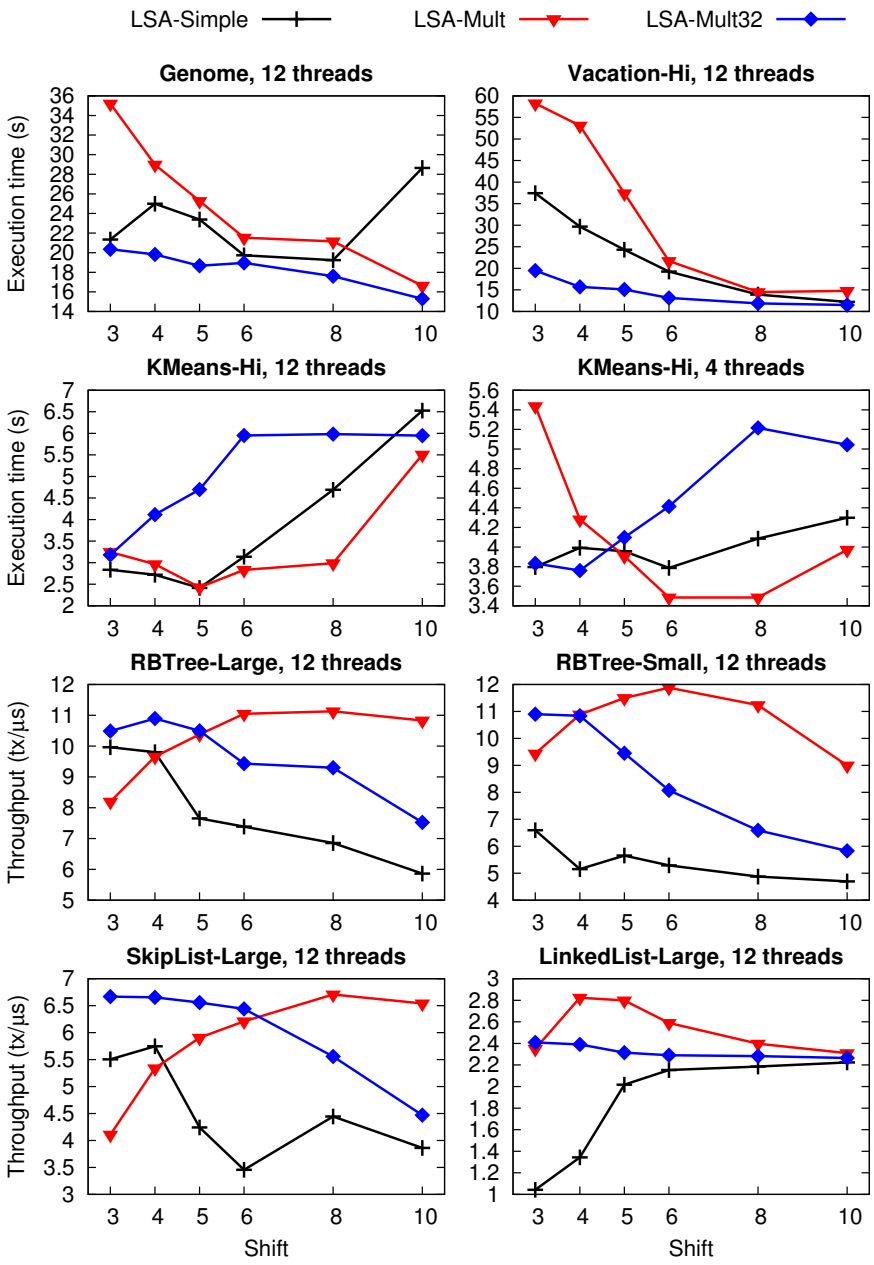

Figure 5.4: Performance of LSA with different hash functions for 2^{12} orecs and different values of Shift.

For the Shift parameter, we do not need to consider a value of 10 because it never provides a performance advantage except a very small one in Vacation with a larger number of orecs, it does not perform well in many of the IntegerSet benchmarks, and it performs very poorly in KMeans. Likewise, we can discard a Shift value of 3: It is never beneficial with Mult32, performs badly with Simple in IntegerSet, and only provides a slight benefit in a few other benchmark configurations (e. g., Simple on four-thread RBTree-Large, or KMeans-Hi with 24 threads).

After this pruning of certain parameter values, we can consider just a Shift parameter ranging from 4 to 8 and 2^{16} to 2^{22} orecs. Because we do not want to assume automatic tuning and want to find a general-purpose memory–to–orec mapping, we need to select one set of parameter values for each hash function that provides good performance across all benchmarks.

Figures 5.5 and 5.6 show the performance of LSA with the three hash functions on selected STAMP benchmark configurations. Genome benefits from a large number of orecs, especially when Simple is used. Vacation-Lo and Vacation-Hi show performance characteristics similar to the case shown in Figure 5.5 independently of the number of threads. In contrast to Simple, Mult and Mult32 get slowed down by large numbers of orecs and small values of Shift, which is a similar effect as in Genome with more than 2^{22} orecs (see Figure 5.3 and the previous explanation of this effect). The results for KMeans-Lo do not show any clear trends. In contrast, performance of KMeans-Hi suffers significantly if the minimum access granularity is too large (see Figure 5.6), especially with Simple. However, with less contention as in the case of 4 threads, the most significant slow-down for Mult and Mult32 again happens with a combination of many orecs and a small Shift value.

Figures 5.7 and 5.8 show the performance of selected IntegerSet configurations[19]. Hash function parameters do not have a large influence on Mult and Mult32, which indicates that both are good hash functions in that they map evenly across the array of orecs and thus are less likely to cause false conflicts; the two exceptions are a small slow-down in case of many orecs and a small Shift value, and LinkedList-Large, which seems to slightly favor a small access granularity. In contrast, Simple is very sensitive to the hash function parameters, to the point where certain parameters lead to just half the throughput compared to others. This indicates that Simple is more prone to hash collisions in the case of certain memory access patterns.

Across all STAMP and IntegerSet benchmarks, Mult32 seems to perform best with a Shift value of 5 or 6 and 2^{16} orecs. A larger number of orecs provides only small performance benefits in primarily Genome as well as LinkedList and well as RBTree-Small with 4 threads, but not in other cases.

For Mult, the sweet spot seems to be at a Shift value of 6 and 2^{20} orecs. A larger number of orecs would help in Genome but would decrease performance in Vacation. 2^{16} orecs would be a decent choice too; Mult32 is somewhat better for this number of orecs in Genome and Vacation, but performs slightly worse in some of the IntegerSet configurations.

For Simple, the choice is more difficult due to its sensitivity to the hash function parameters. 2^{16} orecs do not allow Simple to provide good performance

[19]SkipList-Large shows similar characteristics as RBTree-Large except that Simple performs even worse compared to the other hash functions. LinkedList-Small is similar to RBTree-Small. SkipList-Small resembles characteristics of both RBTree-Small and RBTree-Large.

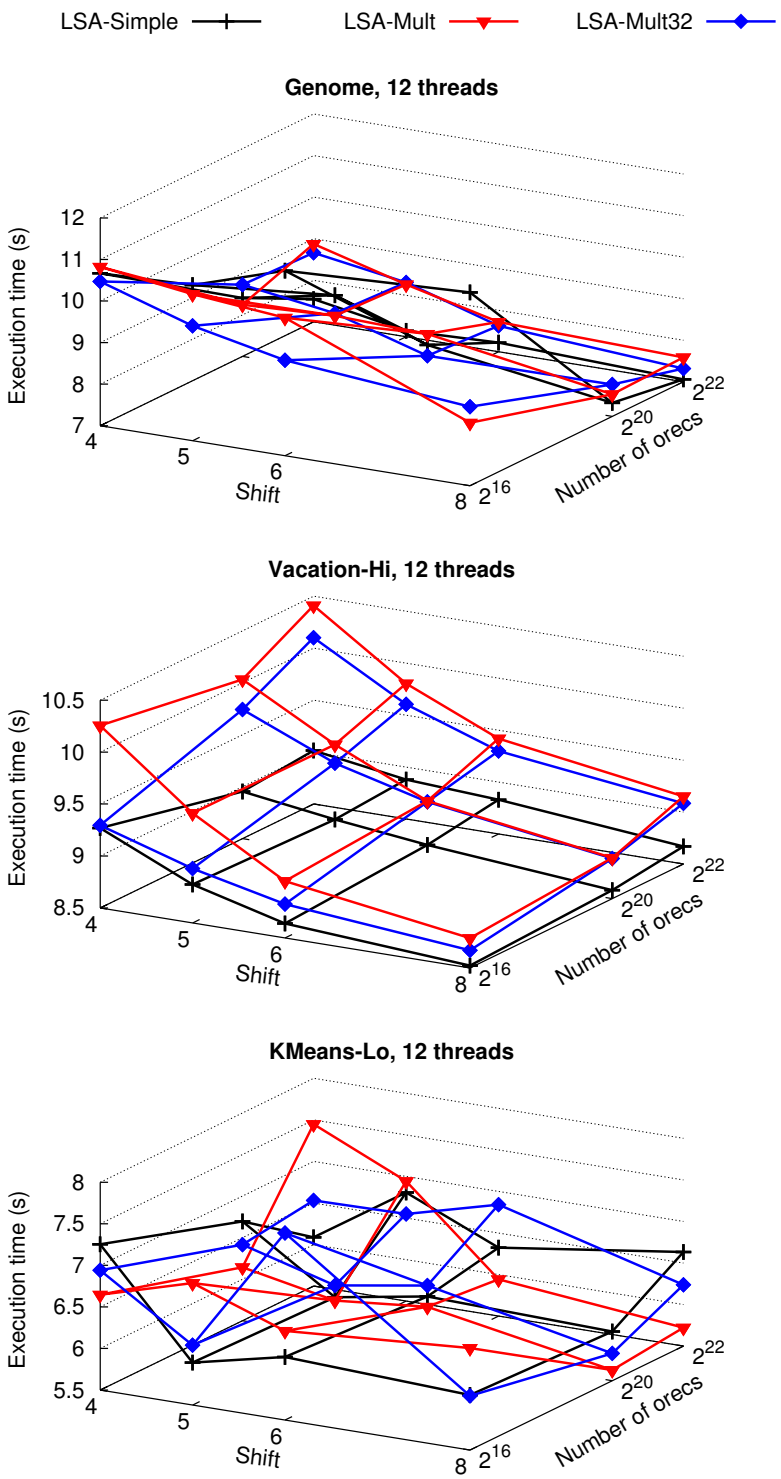

Figure 5.5: Performance of LSA with different hash functions and hash parameters on selected STAMP benchmarks.

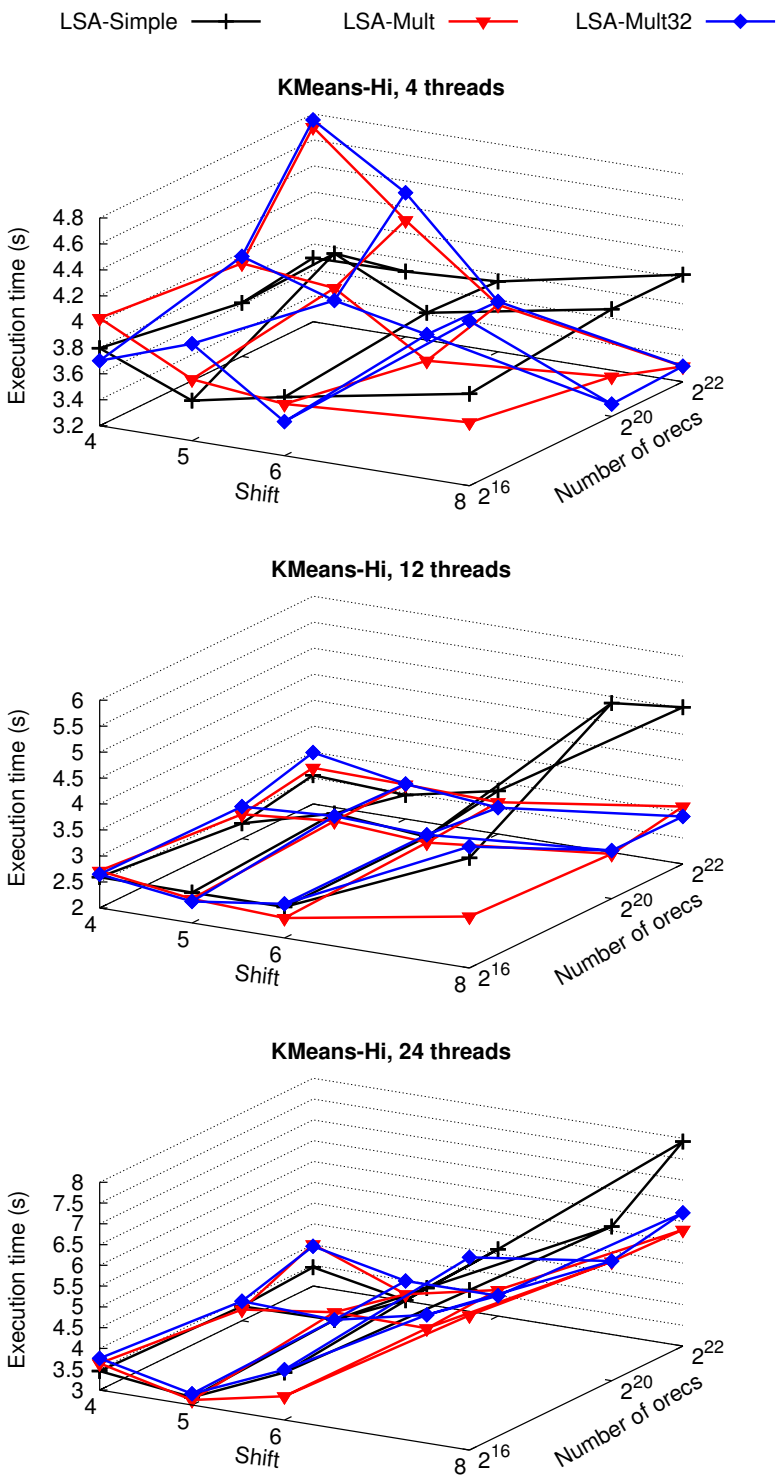

Figure 5.6: Performance of LSA with different hash functions and hash parameters on KMeans-Hi.

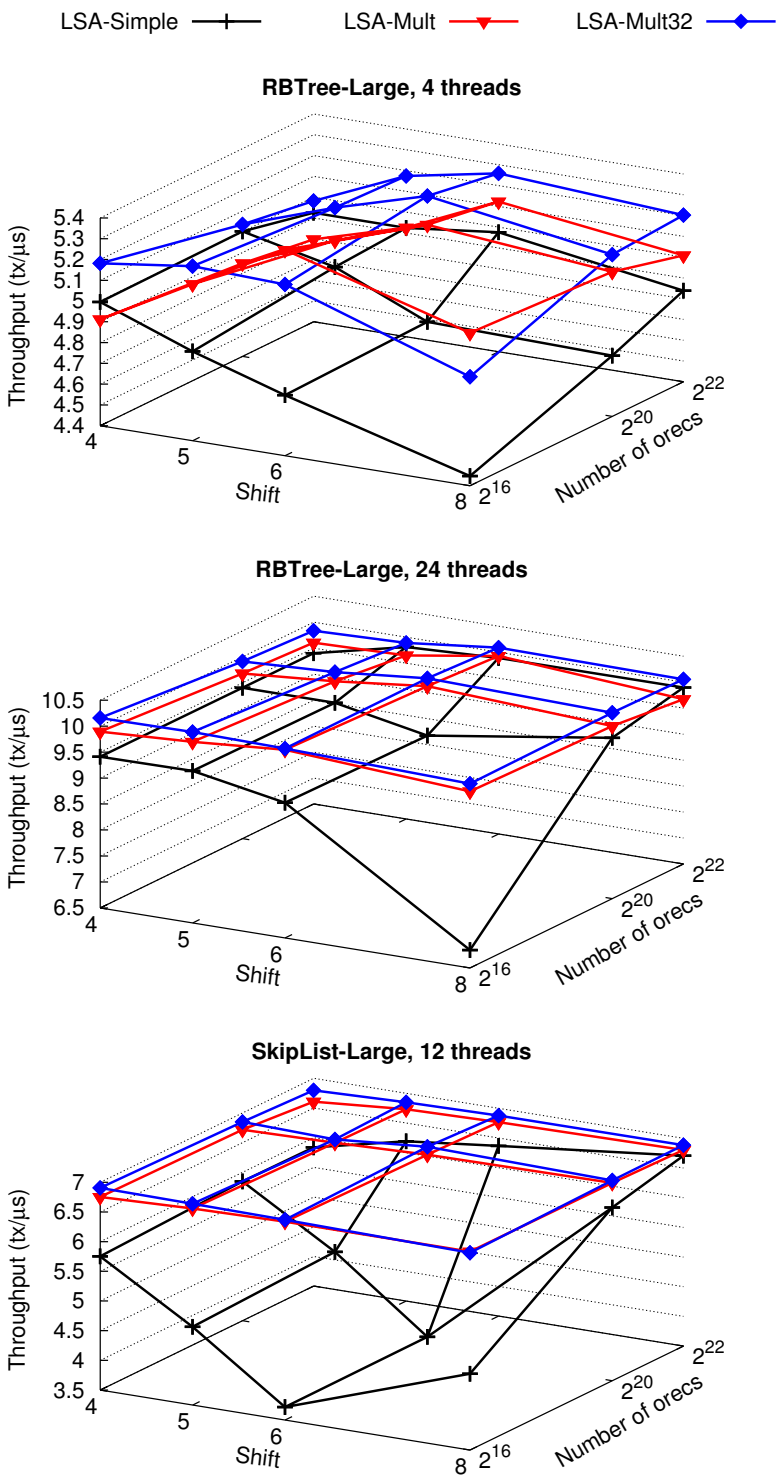

Figure 5.7: Performance of LSA with different hash functions and hash parameters on selected IntegerSet benchmarks (1).

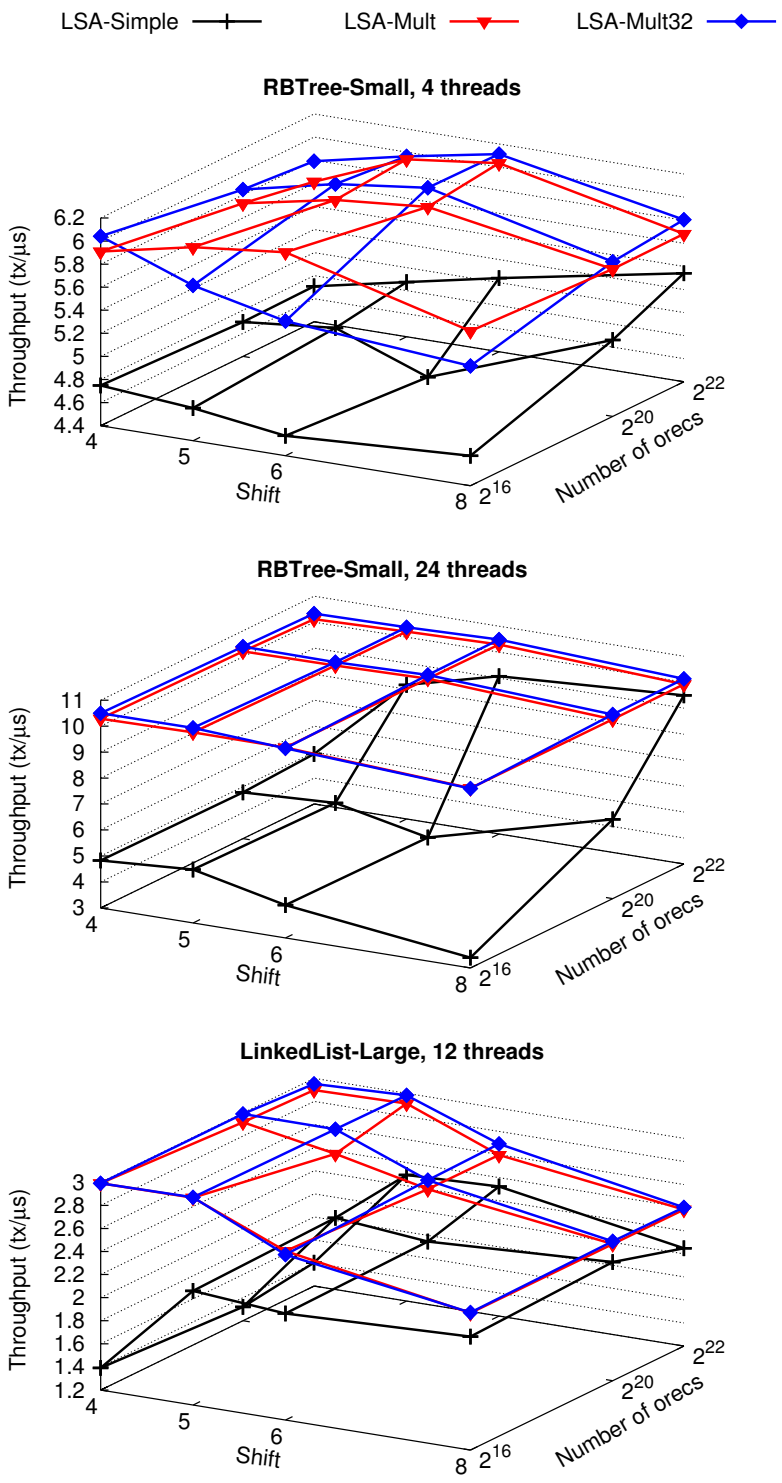

Figure 5.8: Performance of LSA with different hash functions and hash parameters on selected IntegerSet benchmarks (2).

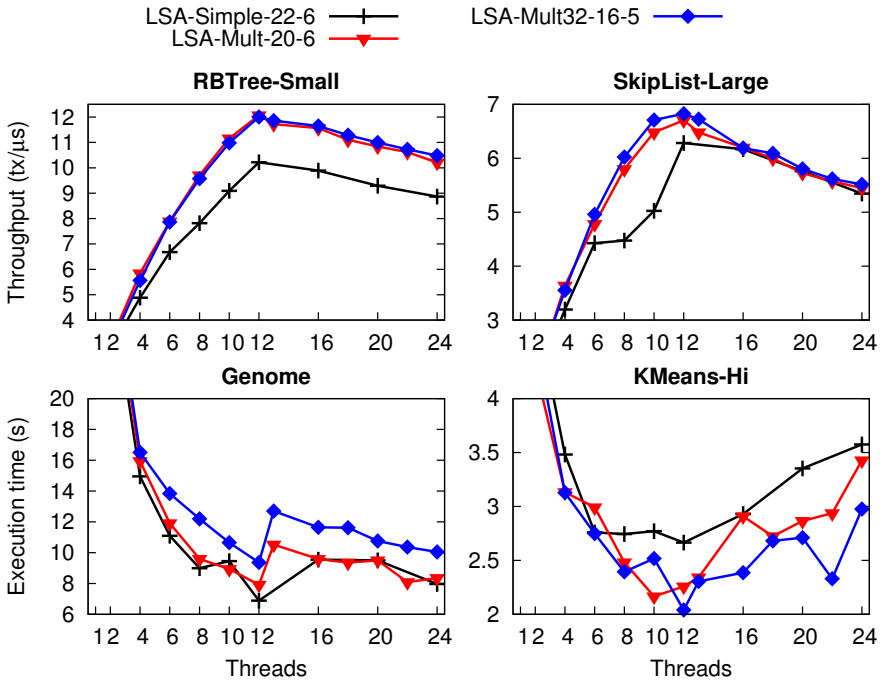

Figure 5.9: Examples for the scalability allowed by the Simple hash function compare to the Mult and Mult32 (e. g., LSA-Mult32-16-5 uses 2^{16} orecs and a Shift value of 5).

in IntegerSet and Genome. With 2^{20} orecs, Simple can still suffer from slowdowns, especially with small IntegerSet configurations and a Shift value of 6, or in Genome. A Shift value of 4 or 5 does not perform well in LinkedList and RBTree-Small; in turn, a value of 8 leads to very bad performance in KMeans-Hi. Thus, the best choice seems to be to use 2^{22} orecs and a Shift value of 6 for Simple.

Of those three candidations for the memory–to–orec mapping, I will not consider Simple further. First, while it provides a slight performance advantage over Mult and Mult32 in Genome and Vacation, it scales significantly worse to a larger number of threads in KMeans-Hi and most of the IntegerSet benchmarks (see Figure 5.9 for examples). Second, it has to use 2^{22} orecs to not decrease performance further, which results in a space overhead of 32MB, compared to the 8MB or 512KB of Mult with 2^{20} orecs or Mult32 with 2^{16} orecs, respectively. Finally, Simple is not robust as the hash functions based on multiplicative hashing, which can provide a good mapping for all kinds of access patterns by just trying a sufficiently large number of different random numbers.

Mult and Mult32, with the random number parameters that I chose, do not result in significantly different performance, and should rather be considered as two examples of multiplicative hashing. The Mult32 example shows that requiring just 2^{16} orecs can still provide good performance, but roughly similar performance would be possible with Mult and 2^{16} orecs. Even though these hash

function parameters have been evaluated just on the STAMP and IntegerSet benchmarks, multiplicative hashing forms a class of universal hash functions and thus should provide good memory–to–orec mappings with other workloads as well.

Runtime Overheads and Scalability

After the previous selection of memory–to–orec mappings for LSA (i. e., Mult with 2^{20} orecs and Shift set to 6, and Mult32 with 2^{16} orecs and Shift set to 5), we can now investigate the performance of LSA in terms of runtime overheads and scalability. I will compare LSA to the NOrec STM (Algorithm 12) and to a global lock used to synchronize otherwise transactional code.

NOrec is a write-back STM design with invisible reads based on a single orec (see Section 7.3.2 for a detailed description). The orec also serves as time base, and NOrec uses time-based validation as a fast path; additionally, it employs value-based validation once time-based validation fails so as to prevent aborts due to commits of nonconflicting update transactions. NOrec thus never needs to access several orecs (which decreases its cache footprint) and does not depend on a well-tuned memory–to–orec mapping. However, commits of all update transactions are serialized using the single orec and prevent concurrent execution of transactional accesses by other threads, which can become a bottleneck once update transaction commits are frequent or take a long time. The serialization of updates allows NOrec to implicitly avoid most ways in which privatization safety could be violated. However, it can still read application data that has been privatized, which can lead to protection faults if the memory protection level of the privatized data is changed (e. g., if it is unmapped); thus, NOrec does not provide full privatization safety as required by the C/C++ TM specification (see Chapter 4). Nonetheless, masking such violations would be easier than for other kinds of violations (e. g., writes to privatized data when LSA rolls back a transaction). Therefore, I have not added enforcement of full privatization safety to NOrec. Finally, note that the NOrec implementation does not fall back to execute transactions in serial-irrevocable mode if they abort more than a hundred times; unlike LSA, NOrec does not suffer from potential livelocks because a transaction only aborts if another update transaction finishes committing.

Using a global lock to provide mutual exclusion between regions of code is semantically equivalent to using relaxed transactions for this code (see Chapter 4). It would also provide a similar ease of use to programmers, although it does not guarantee the same level of safety as atomic transactions. Furthermore, a TM runtime library could also use a global lock internally to synchronize transactions[20], which leads to a TM implementation with no scalability but very low runtime overheads. In particular, in my implementation the global lock is glibc's implementation of a normal POSIX Threads mutex and acquired and released automatically by the TM runtime library. The library has been stripped down to avoid most unnecessary runtime overheads (e. g., CPU registers are not saved when a transaction is started, although stack slots are still saved) and always executes transactions using the uninstrumented code path. After link-time optimizations, this results in similar code as if the programmer had added explicit

[20] In the current C/C++ TM specification, this even holds for atomic transactions that will never get canceled, due to the restrictions on TM-safe code.

mutex acquire and release operations to the application code instead of using transaction statements.

I will not compare LSA against STM designs based on multiple orecs and visible reads because a prior study [83] has shown that they perform inferior to invisible reads. The authors of this study used a bit vector in each orec to let transactions publish transactional read accesses, an STM very similar to LSA (Simple hash function, 2^{20} orecs, and Shift set to 6), and measured the performance of the STAMP benchmarks on similar hardware (a two-socket, quad-core Intel Xeon). They found that visible reads never provided a performance benefit with either one or eight threads, but insteaded resulted in signficantly worse performance in a few benchmarks such as Genome (see Figure 11 in the study). Visible reads are thus probably a useful mechanism with some workload patterns but not an approach that is beneficial in general, especially when several concurrent transactions read from the same application data.

Other STM designs that have been proposed either have obvious performance disadvantages in the general case (e. g., invisible reads with incremental validation) or do not provide the consistency guarantees that we need in the C/C++ context (see Section 5.4). NOrec and the global lock are good representatives of the current range of single-orec or single-lock concurrency-control mechanisms.

Figure 5.10 shows the single-thread runtime overhead of LSA, NOrec, and a global lock compared to sequential code. Note that the STAMP benchmarks execute a mix of transactional and nontransactional code in each thread, whereas IntegerSet executed almost only transactions.

The STMs result in a significant runtime overhead compared to sequential code, especially when transactions execute many memory accesses as in Genome and LinkedList-Large. Besides memory accesses, other components of these overheads are the costs of starting and committing transactions, allowing for the restart of transactions (e. g., saving CPU registers and stack frame contents), and other minor costs such as the maintenance of TM-internal performance statistics. The compiler also influences these overheads by, for example, the quality of its inlining decisions or how well its register allocator can cope with the increase in code size due to instrumentation.

The runtime overhead of the global lock is often much smaller but is still significant especially in IntegerSet benchmarks like HashTable that run very short transactions. A part of this overhead could be avoided by further optimizations of the global lock TM runtime library, but the overheads associated with acquiring and releasing the lock will remain. Nonetheless, a global lock is—in terms of single-thread runtime overhead–the most efficient general-purpose way to synchronize code, and thus should be the baseline that STM is compared to.

In general, it is important to keep in mind that the data shown in Figure 5.10 represents a snapshot of the performance of a prototype implementation; while single-thread overheads of STM will likely remain significant in the general case, there is also room for further optimizations both in the compiler and in the TM runtime library.

Even if STMs suffer from a significant single-thread runtime overhead, they can still provide good performance if their performance scales with the number of threads executing transactions concurrently (i. e., if STMs can exploit parallelism in the transactional workload). Figures 5.11, 5.12, 5.13, and 5.14 show performance scalability for the IntegerSet and STAMP benchmarks. Note that

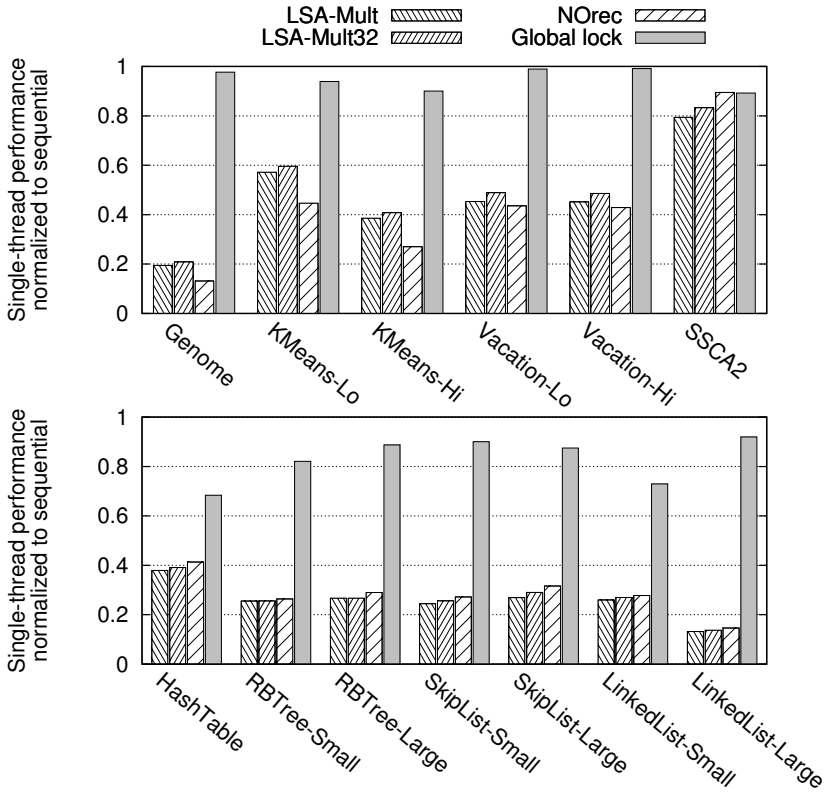

Figure 5.10: Single-thread runtime overhead of LSA compared to NOrec and a global lock, shown as performance in single-threaded benchmark runs normalized to the performance of equivalent sequential code containing no synchronization at all (i. e., higher values represent less runtime overhead).

to increase clarity, the figures do not show results of LSA-Mult if it scales very similar to LSA-Mult32[21].

Because a quiescence-based privatization safety implementation can limit scalability heavily, I also show performance of variants of LSA that do not ensure privatization safety (named LSA-Mult-NP and LSA-Mult32-NP). While not ensuring privatization safety would violate the C/C++ TM specification, other implementations than those based on quiescence would be conceivable (see Section 5.4). Furthermore, this helps to distinguish scalability limitations caused by the core algorithm of LSA from limitations in the privatization-safety implementation.

The cost of memory accesses also has a large impact on performance. Cache misses get the costlier the further away the most recent value of a memory location is (i. e., in main memory or in another CPU core's cache). As explained previously, the benchmarks pin threads to logical CPUs; the scheme used so far alternates between CPUs on different sockets (and thus different NUMA nodes), which exposes the performance effect of NUMA as soon as more than one thread

[21]This also applies to the no-privatization-safety variants described next.

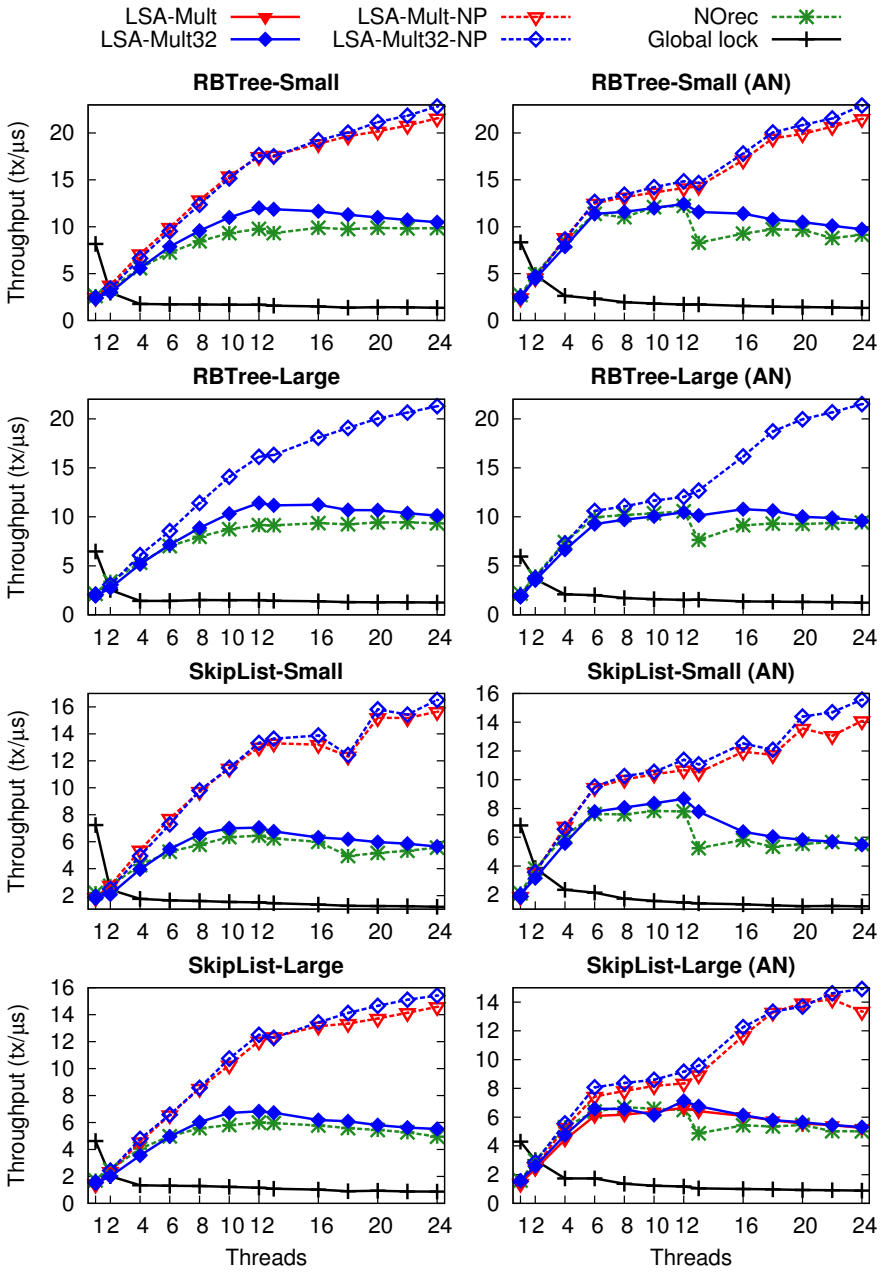

Figure 5.11: Scalability of LSA, NOrec and a global lock with RBTree and SkipList.

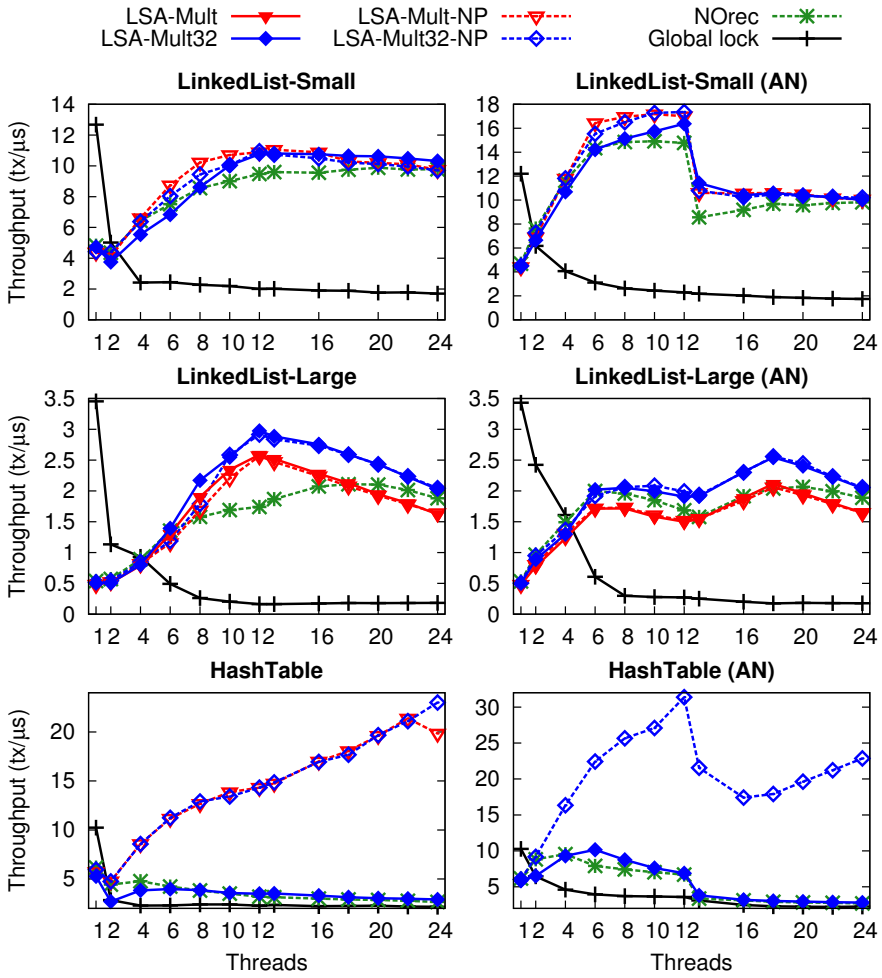

Figure 5.12: Scalability of LSA, NOrec and a global lock with LinkedList and HashTable.

is used. To highlight the significance of these effects, I also show measurements for a different pinning scheme, denoted by AN ("Avoid NUMA"). With this scheme, benchmarks first use all logical CPUs on the first socket (including those provided by hyperthreading) and only after that pin threads to CPUs on the second socket.[22] Thus, we can expect to see the effects of hyperthreading starting at more than 6 and more than 18 threads, and NUMA effects to kick in at more than 12 threads.

The global lock does not provide any scalability beyond a single thread in all benchmarks except KMeans and SSCA2 AN; even in these exceptions, scalability fades away once the number of threads becomes large or NUMA-

[22]That is, for 1–6 threads, each thread runs on its own core on the first socket; with 7–12 threads, all threads are still on the first socket but thread 7 shares a core with thread 1, etc.; starting with thread 13, the second socket is also used in a similar way.

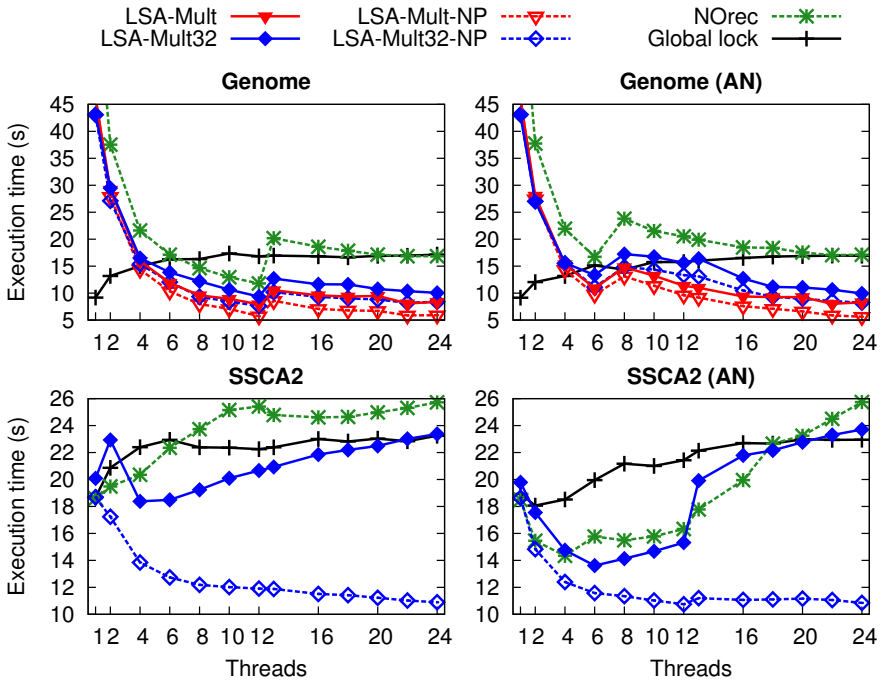

Figure 5.13: Scalability of LSA, NOrec and a global lock with Genome and SSCA2.

related overheads are present. In most cases, performance drops significantly as soon as more than one thread is used. While this performance drop could be partially reduced by a more suitable tuning of the implementation of normal mutexes in glibc[23], a certain part of this slow-down is inherent and due to contention on the lock (e. g., the performance drop in LinkedList-Small from 1 to 2 threads is smaller with the AN pinning scheme than with the default scheme because the former keeps the first two threads on the same NUMA node).

RBTree and SkipList seem to allow for a high level of parallelism (see Figure 5.11), but do not benefit much from hyperthreading (e. g., see RBTree AN at 6 to 12 threads). The performance of LSA itself scales well, but the privatization safety implementation limits the overall performance of the LSA-based STMs (i. e., the NP-variants of LSA, which do not guarantee privatization safety, provide much more throughput at higher thread counts). This limitations is more severe in the presence of NUMA-related memory-access overheads (e. g., compare RBTree-Small with RBTree-Small AN at 1 to 6 threads, or see the drop in throughput in SkipList-Small AN with more than 12 threads).

The privatization safety implementation is also the major reason for lack of scalability in HashTable but has comparatively little influence in LinkedList

[23]The mutex implementation in the current version of glibc does not use any spinning, so threads trying to acquire the lock will immediately block using an operating-system kernel mechanism when they find out that another thread has already acquired it. There is also no back-off or similar measures to reduce memory contention.

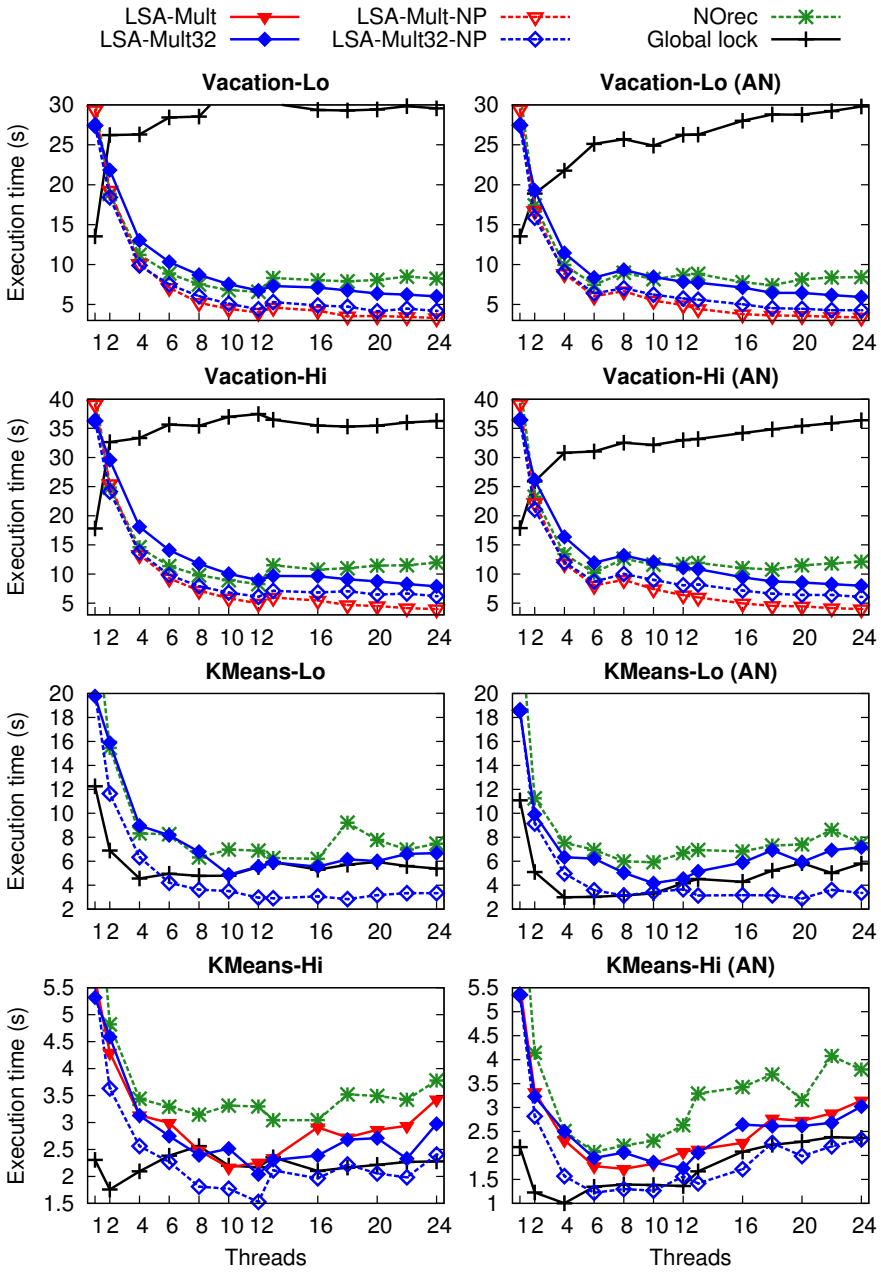

Figure 5.14: Scalability of LSA, NOrec and a global lock with Vacation and KMeans.

(see Figure 5.12). Even without privatization safety, HashTable is very prone to NUMA-related memory-access overheads as shown by the throughput drop from 12 to 13 threads in HashTable AN; the cause of this could be either the global time base of LSA (which is implemented using an integer counter accessed by all threads) or scalability bottlenecks elsewhere in the system (e. g., the memory allocator). However, such a throughput drop is never visible in all other benchmarks except LinkedList-Small (which has also very short transactions); this indicates that a global time base implemented as a shared counter is often not a significant obstacle to scalability.

NOrec never scales better than LSA with privatization safety for larger thread counts, and provides slightly more throughput typically only with less than 6 threads. Compared to LSA without privatization safety, it only yields a very small performance benefit with 1 or 2 threads. In general, NOrec is more sensitive to NUMA overheads than LSA due to having to access the single orec on each transactional read operation (e. g., see the throughput drop from 12 to 13 threads on RBTree AN and SkipList AN).

LSA provides performance equal to the global lock as soon as more than one thread is used, and better performance at 4 or more threads; the only exception is LinkedList-Large, which executes long transactions with little available parallelism. Even if we would assume an ideal lock without any slow-downs under contention and no cache miss costs (i. e., it would always provide as much throughput as with a single thread), LSA could still provide better performance in some benchmarks with at least 4 to 8 threads: On RBTree, on SkipList except SkipList-Small with the default thread pinning scheme, and on LinkedList-Small AN. LSA without privatization safety provides significantly more throughput in all benchmarks except LinkedList.

Regarding the memory–to–orec mapping, Mult and Mult32 result in very similar performance except in LinkedList-Large.

The STAMP benchmarks show similar performance characteristics (see Figures 5.13 and 5.14). Vacation has a lot of available parallelism, which allows the STMs to scale well and LSA to perform better than the global lock with 2 to 4 or more threads. The Mult hash function provides better scalability when no privatization safety is ensured. NOrec performs somewhat better than LSA with privatization safety up to 8 to 12 threads but scales worse beyond this point.

In Genome, LSA suffers from a high single-thread runtime overhead but scales well and thus still manages to perform better than a global lock with 6 or more threads. However, the STMs do not benefit from hyperthreading (e. g., see Genome at 12 versus 13 threads and Genome AN at 6 versus 8 threads). As in Vacation, Mult provides better performance with higher thread counts, which is probably caused by Mult being able to use 2^{20} orecs instead of 2^{16} orecs that Mult32 has available. NOrec has even higher single-thread overheads than LSA and does not scale better.

The privatization safety implementation again limits scalability, especially in benchmarks with short transactions (i. e., SSCA2 and KMeans). But even with privatization safety, LSA performs better than the global lock in SSCA2.

In contrast, KMeans is a challenging workload for the STMs. The global lock scales up four threads, which shows that the benchmark also executes a lot of nontransactional code in parallel. Only LSA without privatization safety can perform better than a global lock, starting at about 6 threads.

Conclusion

Judging based on the measurements presented previously, the biggest obstacles for LSA's performance are (1) the scalability decrease imposed by the quiescence-based privatization implementation and (2) the single-thread runtime overhead that can only be countered by good scalability. In contrast, implementing the global time base using a global shared integer counter seems to not be a significant obstacle to scalability except potentially with very small transactions such as in the HashTable and LinkedList-Small benchmarks.

There are several possible approaches to increasing the scalability of privatization safety. Besides the optimizations proposed by related work (see Section 5.4), even just using techniques like combining [56] in the quiescence implementation could decrease the number of cache misses and thus overall performance. However, this cannot work around the inherent overhead of this kind of privatization-safety implementation, namely that potentially privatizing transactions have to wait for other transactions to extend their snapshot time or commit. Therefore, avoiding to ensure privatization safety for transactions that do not actually privatize could provide greater gains; if possible, this should happen automatically (e. g., based on compile-time analyses) to not burden the programmer with having to provide error-prone annotations.

The single-thread runtime overhead is to a large extent a result of the steps required to perform transactional accesses; for example, maintaining a read set or an undo log requires just a few CPU instructions but still represents a significant overhead when compared against the single memory access that would be executed in sequential code. However, DTMC does not perform several optimizations such as the read–for–write and write–after–write memory access variants already considered in the ABI (see Section 4.2.1). Further optimizations would likely be possible but also require a more complex ABI; it would be easier to justify the engineering effort once TM has more users and it becomes clearer to which extent first-generation HTM support can take pressure off of STM performance.

Despite these two performance obstacles, LSA typically allows for better performance than provided by a global lock once several threads execute transactions concurrently; on our benchmarks and hardware, LSA starts to provide better performance at somewhere between 2 and 6 threads. Comparing LSA to the performance of a global lock is a sensible comparison because a global lock is as easy to use as transactions—the performance goal of TM is to provide programmers with a useful trade-off between ease of use and performance. The IntegerSet benchmarks illustrate that this would still hold in scenarios in which one lock protects a whole data structure.

Beating the performance of sequential code is more difficult for LSA, requires higher thread counts than necessary for beating the global lock, and also a workload with enough available parallelism. Nonetheless, the performance of LSA without privatization safety shows that LSA itself scales well enough to make this possible.

NOrec has in several benchmarks a small performance advantage over LSA with privatization safety on small thread counts, but provides less scalability; however, compared to LSA without privatization safety, it looses this advantage and scales much worse. Whether NOrec provides a useful alternative for small thread counts thus depends on how practical NOrec's incomplete privatization

safety guarantee is in the target environment; if it is not feasible to implement sandboxing for its pending reads and thus the implementation has to ensure privatization safety explicitly, then NOrec is unlikely to provide any performance benefit over LSA with privatization safety.

Nonetheless, even though none of the global lock, NOrec, or LSA perform best on all thread counts, they can all be used to implement TM in a way that is transparent to programmers. TM implementations can even switch at runtime between a global lock and LSA, for example, depending on how frequently transactions execute concurrently (see Section 5.4); this is one of the major advantages when using TM as a programming abstraction for synchronization.

Further investigating such tuning decisions, in particular how to automatically and efficiently categorize a workload at runtime, should be a worthwhile topic for future research. For example, it would be interesting to better understand when visible reads could provide a performance benefit, and to find a way to let the TM apply them only for the transactions or data where this would increase performance.

Likewise, tuning of everything related to the temporal aspect of TM (see Section 3.1.2) also promises to provide performance benefits, but is out of the scope of my work. The LSA implementation uses no contention management at all (i. e., it restarts transactions immediately after they have aborted) but prevents livelocks using the mechanism described in Section 3.4.2. Contention management mechanisms could perhaps also decrease the negative performance impact of false conflicts caused by unfortunate memory–to–orec mapping parameters.

However, one major roadblock to better STM tuning is the currently little use of TM by mainstream programmers, which in turn leads to a lack of benchmarks that would represent these use cases properly. Without more certainty about which workloads and utility functions to tune for, making meaningful tuning decisions is difficult and error-prone. Thus, future work will have to continue to investigate and evaluate TM performance trade-offs.

5.3 Scalable Time Bases for Time-Based STM

As we have seen previously, the global time base plays a central role in time-based STM algorithms like LSA. So far, we have only considered a simple time base that is straightforward to implement: A global shared integer counter. This counter works very well in smaller-scale systems. However, when the number of concurrent threads grows or the machine itself becomes bigger and thus communication between threads becomes more costly, a shared counter can quickly become a performance bottleneck. While transactions read the counter only infrequently (i. e., when starting, and optionally on-demand when snapshot extensions are necessary), every update transaction has to increment the counter. This leads to contention on the counter and cache misses for all concurrent transactions. A more scalable time base would thus be very beneficial.

With that in mind, what could be a more scalable time base? An interesting observation is that in a very large system with many committing update transactions, the counter would be updated very frequently. Each update transaction represents progress and lets the global time advance; From the perspective of a transaction, it will look like time is always advancing, pretty much like a wall clock that ticks independently of what its observer does. Furthermore, more

clock ticks (or counter increments) will not affect a transaction unless those ticks where due to conflicting concurrent transactions.

This analogy suggests that we can use real-time clocks as time base. I will show in what follows that this is indeed possible and how it can be achieved. I will consider two kinds of clocks: *perfectly synchronized clocks* and *externally synchronized clocks*. Perfectly synchronized clocks (conceptually) give all threads access to one global clock without any reading error. The reading error is the difference between the value read and the correct value. Typically, such a perfectly synchronized clock would need to be implemented in hardware but software implementations with lower accuracy are also possible. An externally synchronized clock also provides access to a global clock but with some reading error that might vary.

Common to both kinds is that they can be more easily parallelized than simple logical counters. Furthermore, the global clock does not actually need to be a real-time clock: Neither its speed nor its value needs to be approximately synchronized with real time. However, having a global real-time clock typically simplifies the implementation of an externally synchronized clock (because local clocks with a bounded drift rate can be used to approximate real time). In particular, this reduces the overhead and error if the synchronization is done in software.

Next, I will first explain how to make LSA ready to be used with real-time clocks (Section 5.3.1) and discuss perfectly synchronized (Section 5.3.2) and externally synchronized clocks (Section 5.3.3). I will conclude with a case study whose experimental results show that using real-time clocks can indeed improve the scalability of time-based STM on large systems (Section 5.3.4).

5.3.1 LSA on Real-Time Clocks

Algorithm 5 shows the Lazy Snapshot Algorithm (LSA) modified so that it can be used with real-time time bases, which I will subsequently call *LSA-RT*. It extends the single-version, write-through, blocking variant of LSA implemented on top of C++11 (Algorithm 3); a multi-version variant is presented in the original paper [93].

LSA-RT requires just four changes compared to Algorithm 3; while we have to change how we deal with timestamps and access the global time base, both the basic algorithm and the global and per-thread state can remain unchanged.

First, we access the global time with the new function `get_time`, both when starting a transaction (line 2) and when trying to extend the snapshot time (line 38).

Second, we delegate all comparisons between timestamps obtained from the global time base to a new function, `possibly_more_recent`. In single-version LSA, we only compare timestamps when we check whether a transaction's snapshot time is more recent than the commit time of the data being accessed. If the latter might *possibly* be later than the former, we have to extend the snapshot and validate (which explains the name of the new function).[24] This comparison takes place when reading and writing transactional data (lines 10 and 23).

Third, we now obtain a commit time using another new function called

[24]Remember that unnecessary snapshot extensions are harmless and only lead to some unnecessary runtime overheads.

Algorithm 5 LSA modified for real-time time bases (LSA-RT base, extends Algorithm 3)

```
1:  stm-start()_p:                                          ▷ Replaces same function of Algorithm 3
2:    st ← get-time()
3:    r-set ← w-set ← undolog ← ∅

4:  stm-load(addr)_p:                                       ▷ Replaces same function of Algorithm 3
5:    orec ←_acq orecs[hash(addr)]
6:    if orec.locked then
7:      if orec.owner ≠ p then
8:        abort()                                           ▷ Orec owned by other thread
9:      return *addr                                        ▷ We own the orec; just read through
10:   if possibly-more-recent(orec.time, st) then           ▷ Need to extend snapshot?
11:     extend()                                            ▷ Aborts if validation fails
12:   val ←_acq *addr
13:   if orecs[hash(addr)] ≠ orec then                      ▷ Load again and compare with previous load
14:     abort()                                             ▷ Data at addr was perhaps modified concurrently
15:   r-set ← r-set ∪ {⟨addr, orec.time⟩}                   ▷ Add to read set
16:   return val

17: stm-store(addr,val)_p:                                  ▷ Replaces same function of Algorithm 3
18:   orec ← orecs[hash(addr)]
19:   if orec.locked then
20:     if orec.owner ≠ p then
21:       abort()                                           ▷ Orec owned by other thread
22:   else
23:     if possibly-more-recent(orec.time, st) then         ▷ May have read from addr before, so...
24:       extend()                                          ▷ ...abort if validation should fail
25:     if ¬ cas_acq(orecs[hash(addr)] : orec → ⟨true, p⟩) then   ▷ Try to acquire orec
26:       abort()
27:     fence_rel                                           ▷ Memory barrier with release memory order
28:     w-set ← w-set ∪ ⟨hash(addr), orec⟩
29:   undolog.push(⟨addr, *addr⟩)                           ▷ Log previous value of *addr
30:   *addr ← val                                           ▷ Write through to memory

31: stm-commit()_p:                                         ▷ Replaces same function of Algorithm 3
32:   if w-set ≠ ∅ then                                     ▷ Nothing to do if read-only transaction
33:     ct ← get-commit-time(acqrel)                        ▷ Commit time
34:     extend()                                            ▷ LSA-RT must always validate (and abort if validation fails)
35:     for all ⟨orec, orecval⟩ ∈ w-set do
36:       orecs[orec] ←_rel ⟨false, ct, 0⟩                  ▷ Release orecs

37: extend()_p:                                             ▷ Replaces same function of Algorithm 3
38:   st ← get-time()
39:   for all ⟨addr, time⟩ ∈ r-set do                       ▷ Are orecs free and timestamps unchanged?
40:     orec ← orecs[hash(addr)]
41:     if (orec.locked ∧ orec.owner ≠ p) ∨ (¬ orec.locked ∧ orec.time ≠ time) then
42:       abort()                                           ▷ Inconsistent snapshot

43: abort()_p:                                              ▷ Replaces same function of Algorithm 3
44:   undolog.rollback()                                    ▷ Undo previous writes in reverse order
45:   ct ← 0
46:   for all ⟨orec, orecval⟩ ∈ w-set do
47:     if incarnation-left(orecval.inc) then               ▷ No incarnation number overflow?
48:       orecs[orec] ←_rel ⟨false, orecval.time, orecval.inc + 1⟩   ▷ Release orec (new incarnation)
49:     else
50:       if ct = 0 then                                    ▷ Acquire new "commit" timestamp
51:         ct ← get-commit-time(rel)
52:       orecs[orec] ←_rel ⟨false, ct, 0⟩                  ▷ Release orec (new timestamp)
```

Algorithm 6 LSA-RT with a shared integer counter as time base (extends Algorithm 5)

```
1: get-time()_p:
2:     t ←_acq clock
3:     return t

4: get-commit-time(mode)_p:
5:     t ← atomic-inc-and-fetch_mode(clock)
6:     return t

7: possibly-more-recent(a, b)_p:
8:     return a > b
```

get−commit−time, which takes a C++11 memory order as parameter (lines 33 and 51).

For completeness, Algorithm 6 shows how the new functions can be implemented using a shared integer counter as time base, leading to basically the same code as in Algorithm 3.

The only subtle and nontrivial change is the fourth one: We always validate on commit (line 34) and cannot skip validation if our snapshot time is right before our commit time (as in Algorithm 3 on line 46).

With real-time clocks, we cannot acquire a new unique commit time; instead, a transaction might *share* a commit timestamp with another transaction. We still require the commit time to be larger than the global time when the transaction had acquired all its write locks (see Section 5.1.1), so all transactions that have a snapshot time equal to our commit time or more recent will still see all our write locks being acquired. Likewise, when we validate on commit, we are guaranteed to be able to detect conflicts with other transactions trying to commit at the same timestamp because all their write locks will be visible to us. Therefore, only nonconflicting transactions will be able to commit at the same timestamp. This will be further discussed in Sections 5.3.2 and 5.3.3.

Note that this approach also enables a potential optimization for the shared integer counters: If the transaction uses a CAS instead of the atomic fetch–and–increment (line 5 in Algorithm 6), then it can share a commit timestamp with another transaction by just using the current global time after a failed CAS attempt (i. e., another transaction's commit time). On architectures that do not support a fast fetch–and–increment, this can be faster than repeated CAS attempts when the counter is contended [123].

5.3.2 Perfectly Synchronized Clocks

Synchronizing real-time clocks in distributed systems is a well studied topic [17, 43]. With the appropriate hardware support, one could achieve perfectly synchronized clocks in the sense that there is no observable semantic difference between accessing some global real-time clock or processor-local replicas of the global real-time clock. There is of course a performance difference because there is no contention when processors access their local clock but there might be quite some contention when instead accessing a single global real-time clock.

In systems that do not have hardware-based clock synchronization, we can synchronize clocks in software. When doing so, we need to expect that there is an observable deviation between these individual, externally synchronized real-time clocks (which I will discuss in Section 5.3.3).

Algorithm 7 LSA-RT with perfectly synchronized clocks as time base (extends Algorithm 5)

```
 1: get-time()_p:
 2:     t ← read-local-clock()                    ▷ Reads the current value of the local clock
 3:     fence_seqcst                              ▷ Memory barrier with sequential-consistency memory order
 4:     return t

 5: get-commit-time(mode)_p:
 6:     fence_seqcst
 7:     t_s ← t ← read-local-clock()
 8:     while t = t_s do                          ▷ Loop until time has advanced
 9:         t ← t ← read-local-clock()
10:     if mode = acqrel then
11:         fence_seqcst
12:     return t

13: possibly-more-recent(a, b)_p:
14:     return a > b
```

Algorithm 7 shows how we can use perfectly synchronized real-time clocks for LSA-RT (Algorithm 5). To understand how this works, we need to take a closer look on the requirements for the time base.

We assume that there is a set of clocks that are all perfectly synchronized to a conceptual real-time clock. Function read−local−clock returns the current time of the local clock associated with the requesting transaction. The timestamps that a clock returns are monotonic (i. e., if a transaction first reads t_1 and then t_2, then we know that t_2 is guaranteed to be equal or larger than t_1). All clocks must return a new, larger timestamp eventually, and they never stop doing this (i. e., after read−local−clock returned a timestamp t_1, it will take a finite number of steps until it will return t_2 with $t_2 > t_1$); this ensures that the TM can make progress.

Perfectly synchronized clocks give the guarantee that if any transaction calls read−local−clock between real-time timestamps t_s and t_e, then the function will return a value t with $t_s \leq t \leq t_e$. In other words, they behave like a single linearizable real-time clock. As a result, possibly−more−recent is very simple: We just have to compare the timestamps (line 14).

Similarly, in get−time, we just read the current time from the local clock (line 2). In get−commit−time, we have to wait for a new timestamp that is larger than the time at which the transaction had acquired all the locks. We do this by reading the current time and then looping until we observe a larger timestamp, which we return as commit time (lines 7–9).

However, the synchronization in LSA-RT is based on the C++11 memory model, in which memory accesses are not guaranteed to be linearizable. Therefore, we have to additionally embed the ordering implied by the linearizable, perfectly synchronized clocks into the *happens–before* relation that the memory model is based on.

We can do that by requiring that reading ordered timestamps of the clock constrains the total order of sequentially consistent memory barriers. In particular, if a and b are such barriers, a is sequenced before a call to read−local−clock that returns t, and b is sequenced after a call to read−local−clock that returns $t + 1$, then a happens before b.

With this constraint, we can express Algorithm 6's acquire and release memory orders on accesses to the shared integer clock in Algorithm 7: acquire mem-

ory orders are transformed into barriers after reading the clock (line 3 and 11), and the release memory order when acquiring a new commit time is transformed into a barrier before waiting for a new timestamp to appear (line 6).

This yields the essential ordering constraint: When a transaction operates at a snapshot time $t + 1$, then all the lock acquisitions of the transactions that committed (or are about to commit) at $t+1$ will happen before the snapshotting transaction's operations. This holds because the update transaction executed a memory barrier still at time t or earlier, and the snapshotting transaction will execute a barrier at $t + 1$ but before executing subsequent operations.

Note that the sequentially consistent barriers are just one possible way to model this and that the optimal way to do it depends on the (hardware) implementation of the perfectly synchronized clocks. The previous approach works well when full memory barriers are already linearizable operations. However, it might also be possible to use barriers with the original acquire and release memory orders if these get ordered by every tick of the clocks.

The resolution of the clocks also affects the algorithm (i. e., how frequently the clocks tick). If reading the clock always takes at least one clock tick, the loop with which we wait for a new timestamp (lines 7–9) might not be necessary. In turn, if the resolution is rather low, then waiting for a new commit timestamp can take a while. Alternatively, we can just skip the waiting on behalf of the committer and mask this misbehavior on the side of reading transactions, similar to how we can deal with externally synchronized clocks.

5.3.3 Externally Synchronized Clocks

Synchronizing clocks perfectly can be difficult, for example because the cost of hardware clocks increases when accuracy needs to be high. In turn, if accuracy is lower, a perfectly synchronized clock's resolution will be lower too, which leads to update transactions having to wait longer on average when obtaining a new commit timestamp (see Section 5.3.2).

Therefore, we want to also be able to use externally synchronized clocks that return imprecise values but for which the deviation *dev* between real-time t and the value of the local clock at time t is bounded. However, with such clocks, it might not be possible to always determine whether one timestamp was read later or earlier than another one.

For a time-based transactional memory, the imprecision essentially means that there is *uncertainty* about when a transaction committed an update and how this is really ordered with respect to another transaction's snapshot.

We can handle that uncertainty in a straight-forward way. If a transaction cannot be sure that a snapshot extended by some committed version is valid at a certain time, it assumes that the snapshot is not valid. Thus, it masks uncertainty errors. Because the deviation is bounded, the uncertainty is bounded too; it can render commits inaccessible for other transactions, but only some time right after the commit.[25] Note that a time-based TM algorithm can always fall back to using validation (i. e., snapshot extensions), so we always have some way to access a data item.

Algorithm 8 shows how we can use such externally synchronized clocks for LSA-RT. First of all, all timestamps are now tuples (lines 6–8). They consist

[25] If keeping multiple versions of each data item, only the lower and upper bounds of a version's validity range are affected [93].

Algorithm 8 LSA-RT with externally synchronized clocks as time base (extends Algorithm 5)

```
 1: Global state:                                   ▷ Replaces global state of Algorithm 5
 2:   orecs: word-sized ownership records, each consisting of:
 3:     locked: bit indicating if orec is locked
 4:     owner: thread owning the orec (if locked)
 5:     time: commit timestamp (if ¬ locked), consisting of:
 6:       t: timestamp
 7:       cid: ID of the clock that t originated from
 8:       dev: maximum deviation of t from real time
 9:     inc: incarnation number (if ¬ locked)

10: get-time()_p:
11:   ⟨t, cid, dev⟩ ← read-local-clock()          ▷ Reads the current value of the local clock
12:   fence_seqcst                                ▷ Fence with sequential-consistency memory order
13:   return ⟨t, cid, dev⟩

14: get-commit-time(mode)_p:
15:   fence_seqcst
16:   ⟨t_s, cid, dev⟩ ← ⟨t, cid, dev⟩ ← read-local-clock()
17:   while t = t_s ∧ clock-is-shared(cid) do   ▷ Loop unnecessary if dev > 0 and clock isn't shared
18:     ⟨t, cid, dev⟩ ← read-local-clock()
19:   if mode = acqrel then
20:     fence_seqcst
21:   return ⟨t, cid, dev⟩

22: possibly-more-recent(a, b)_p:
23:   if a.cid = b.cid then                       ▷ Timestamps from same clock?
24:     return a.t > b.t                          ▷ Yes, no deviation between timestamps
25:   return a.t + a.dev > b.t − b.dev           ▷ No, need to take possible deviation into account
```

of a timestamp value t, an ID cid of the local clock that the timestamp was obtained from, and the maximum deviation dev of this timestamp from real time (i. e., if a call to read−local−clock is executed between t_s and t_e in real time and returns t, then $t_s - dev \leq t \leq t_e + dev$). However, we still assume that each clock is monotonic, so two timestamps obtained from the same clock can be compared as when using a perfectly synchronized clock.

The get−time and get−commit−time functions are similar to those for perfectly synchronized clocks shown in Algorithm 7. However, the loop in function get−commit−time is not required if we assume that $dev > 0$ and each thread has its own clock that is shared with no other thread.[26]

The major difference to Algorithm 7 is that we need to take uncertainty into account when reasoning about the validity of a snapshot based on timestamp values. If timestamps are from the same clock, we can compare them directly because they are from the same time base (line 24). If they are from different clocks, then we have to take both clocks' maximum deviation into account: We can simply use the sum of these deviations as uncertainty interval, and make a conservative decision (line 25).

Note that even with large deviations, we will always be able to construct a snapshot becaus we can fall back to full validation after each access (i. e., returning false from possibly−more−recent will trigger a snapshot extension).

Furthermore, the deviations only matter if we read right after an update performed by another thread; in this case, a reading transaction will suffer from a cache miss anyway, which will take additional time. Conceptually, the

[26] If a clock is not shared between several threads and a thread only executes one transaction at a time, then there can never be any conflicts between transactions using the same clock.

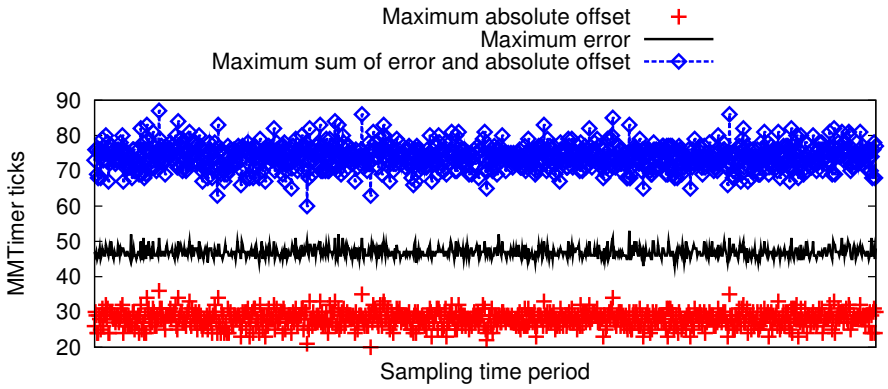

Figure 5.15: MMTimer synchronization errors and offsets.

length of the cache miss reduces the length of the deviation and its effect on performance; the cache miss results in having to wait some time anyway before a transaction can read another transaction's updates.

5.3.4 Evaluation and Discussion

To evaluate how time bases can affect performance, I will next present results of a case study about LSA performance on an SGI Altix 3700, which is a cache-coherent NUMA machine with Itanium II processors. Pairs of two processors are tightly connected and share 4GB of memory; all processor pairs are (indirectly) connected to all other pairs, and I executed my experiments on a 32-processor partition of the whole 192-processor system. On the Altix, shared counters have noticeable overhead because inter-processor communication is slow compared to the compute speed of individual processors. Executables were generated using an older version of the TM runtime library (see Section 3.4), which does not ensure privatization safety. I experimented with two time bases: (1) an ordinary shared integer counter and (2) MMTimer, a hardware clock that is part of the Altix machine.

MMTimer. MMTimer is a real-time clock with an interface similar to the High Precision Event Timer widely available in x86 machines. It ticks at 20 MHz but reading from it always takes an amount of time that is equivalent to 7 to 8 ticks of MMTimer, so the effective granularity is coarser than one would expect from 20 MHz. In particular, MMTimer is therefore strictly monotonic (i. e., both get−time and get−commit−time just return the value of MMTimer).

At first, I had no information about whether MMTimer is a synchronized clock or not. I therefore used a simple test to measure the synchronization error by (1) having threads on different CPUs read from MMTimer and (2) comparing the clock value obtained at each CPU with a reference value published by a thread on another CPU. Figure 5.15 shows the results of a four-hour run with synchronization rounds every ten seconds.

In the figure, offsets represent the estimated difference of local clock values

to the reference clock value and errors denote the largest possible deviation between the estimated offset and the offset that could be achieved by a perfect comparison. Only the maximum values of all CPUs are shown for each round. The results show that there is no drift, so MMTimer behaves like a global clock or a set of synchronized clocks. Second, errors are always larger than offsets, so MMTimer could indeed be a perfectly synchronized clock. Third, the error seems to be bounded and is not too large: 90 ticks seems to be a reasonable estimate for this bound (see Figure 5.15). However, the clock comparison algorithm that I used suffers from the overheads of communicating over shared memory, so MMTimer's actual synchronization error bounds could be much smaller.

After performing these experiments, I got the information that MMTimer is indeed a synchronized clock [61]. Every node in the Altix system has one register for the clock that is accessible via the MMTimer interface. Before system boot, a single node is selected as source for the clock signal, and all other nodes' clocks are synchronized to this node. During runtime, the source clock then advances all other nodes' clocks. Dedicated cables are used for the clock signal. However, I do not know how the synchronization mechanism works in detail (e. g., synchronization errors could arise from a varying latency of the clock signal), but I believe that such potential errors are already masked by the time that it takes to read the MMTimer (7 or 8 ticks of MMTimer), which would mean that MMTimer behaves like a perfectly synchronized clock.

The important observation is that, unsurprisingly, hardware support can ensure a much better clock synchronization than mechanisms that require communication via shared memory (in our case, 8 ticks vs. 90 ticks).

Time base overheads. To investigate the overheads of using shared counters and MMTimer as time bases, I used a simple workload in which transactions update distinct objects (but this fact is not known a priori). This type of workload is likely to exist in many larger systems: programmers rely on the TM to actually enforce atomicity and isolation of concurrent computations, but the program has been optimized to prevent global synchronization bottlenecks. Furthermore, performance in this workload is not affected by other TM properties (e. g., contention management), which makes the overhead of the time base more apparent.

Figure 5.16 shows throughput results for this workload for update transactions of different sizes. For very short transactions, throughput in single-threaded executions is decreased by MMTimer's overhead (i. e., the 7 to 8 ticks required to read from it); for transactions with 10 accesses, using MMTimer results in half the throughput. However, the overhead becomes small when transactions are larger; with 100 accesses, using MMTimer results in a roughly 10% throughput decrease.

More importantly, using a shared counter as time base prevents the STM from scaling well, whereas with MMTimer, performance increases linearly with the number of threads on this workload. For example, with just two threads and transactions with 10 accesses, MMTimer already leads to 60% more throughput. The impact of the shared counter's overheads decreases when transactions get larger because the contention on the counter decreases. However, the influence of the shared counter's overhead would increase again if the STM would perform its operations faster or more CPUs would be involved.

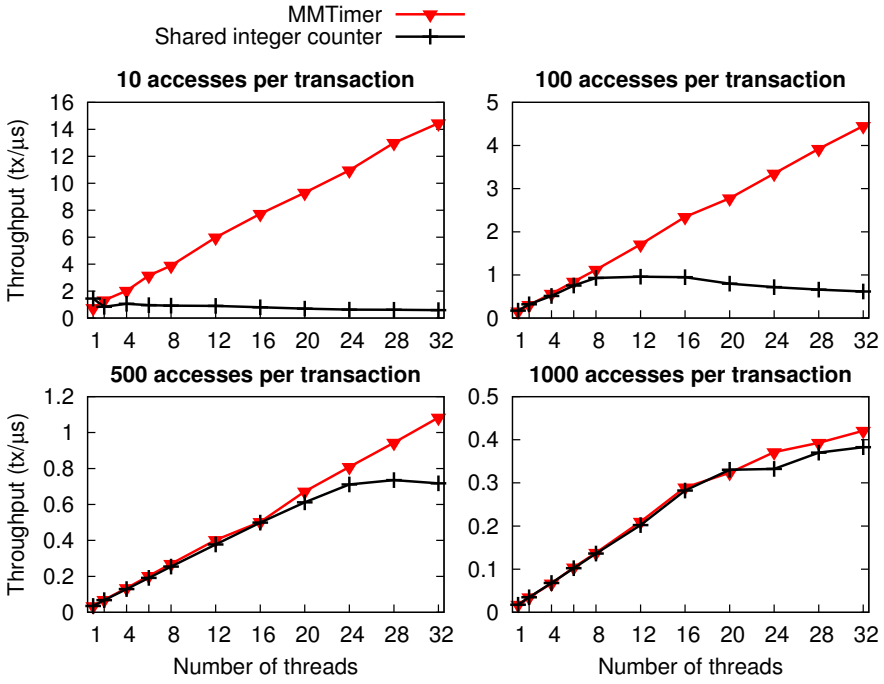

Figure 5.16: Overhead of time bases for update transactions of different size.

Synchronization errors. Larger synchronization errors increase the amount of time in which transactions cannot be certain whether they are allowed to access a memory location or not. To resolve the uncertainty, they need to extend their snapshot time far enough to make it clear that an orec's timestamp is not possibly more recent than the snapshot time.

However, which effect larger synchronization errors actually have on performance depends a lot on the workload (e. g., locality, update frequencies, or the duration of transaction) and on the overheads that the TM and synchronization on the respective architecture introduce. First, a large synchronization error only matters if one transaction needs to read another transaction's updates immediately after the other transaction committed. This is unlikely in workloads with little contention, and will not happen for threads that share the same clock or are close to each other in terms of where on the system they execute (in which case the errors are likely to be smaller). Second, while the synchronization errors imposes a limit on the maximum scalability for transactions that do not use snapshot extensions, transactions can still use those extensions like incremental validation to make progress regardless of any time base synchronization errors. Thus, scalability is only limited by the minimum of the limits imposed by either errors or incremental validation overheads.

To illustrate the effect that synchronization errors might have for contended workloads, Figure 5.17 shows transaction throughput for a linked list that is accessed by all the threads in the Altix' partition. We can observe that even with a synchronization error of 100 MMTimer ticks (which means an effective

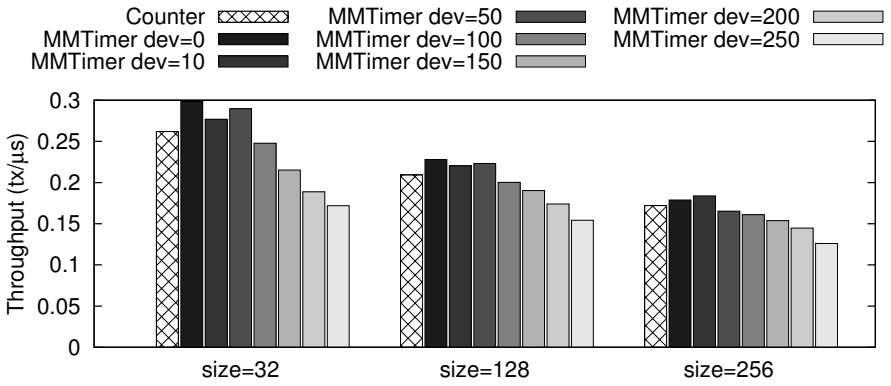

Figure 5.17: Influence of synchronization errors on throughput on LinkedList (see Section 3.4.3) with a 20% update transaction rate, using 32 threads, and lists with an average number of elements of 32, 128, and 256. MMTimer's synchronization error ("dev", see Algorithm 8) is specified in terms of MMTimer ticks.

maximum offset between two clocks of $10\mu s$), using MMTimer still results in roughly the same throughput as if using a shared integer counter as time base. Only larger synchronization errors result in a performance decrease compared to the counter, and this decrease is stronger on the shorter lists where transactions are more likely to hit other threads' updates after a short amount of time. Note that this workload is unlikely to allow for good performance of transactions independently of which time base is used. There is not a lot of parallelism in a system-wide shared linked list, so this would be a bottleneck in the program in any case (e. g., due to frequent cache misses and transaction conflicts). In contrast, if the program would pay attention to locality and threads would access separate lists, then we would see advantages for MMTimer similar to those shown in Figure 5.16.

As I have mentioned previously, the STM implementation used for this evaluation does not guarantee privatization safety. Applying the quiescence-based implementation described in Section 5.2 would be possible but might cause excessive overheads on large systems due to a large number of threads and large cache miss overheads. Nonetheless, we can still ensure privatization safety by waiting until the snapshot times of other active transactions are guaranteed to be more recent that the updating transaction's commit time. Likewise, the TM could still use combining to decrease communication overheads because all snapshot times originate from the same time base.

Overall, this case study and evaluation shows that scalable time bases are practical and allow LSA to scale to large systems despite it relying on a global time base. For synchronized hardware clocks to be useful as a time base, their synchronization errors must be bounded and the bounds must be known a priori (e. g., at program start, but lease-like guarantees would be useful as well).

5.4 Related Work

Dice et al.'s TL2 STM [27] is very similar to LSA, and was proposed only slightly later than the first version of LSA [92]. While TL2 also uses time-based validation and a global time base to ensure that the snapshot taken by a transaction is always consistent, it does not include the possibility to extend the validity of a snapshot to a more recent time (except as a mandatory step during transaction commit); thus, a transaction can only read from an object if the most recent update to the object is before the start time of this transaction. TL2 combines the time-based TM approach with a write-back, word-based, blocking TM design that uses ownership records and does not maintain multiple versions of objects in memory; this combination provides good performance, and later implementations of LSA also moved towards this combination [93, 42]. As time base, TL2 uses a shared integer counter but also proposes optimizations that allows transactions to share a commit time in certain cases, which can be beneficial if the hardware only implements CAS and not atomic fetch–and–increment operations. Dice et al. also suggest to use hardware clocks instead of a shared counter to avoid its overhead but do not investigate this in detail.

Concurrently with TL2, Spear et al. proposed a heuristic [109] aimed at reducing the overheads of incremental validation: A global variable is used to count the number of attempted commits of update transactions in the whole program, which allows transactions to skip incremental validation if the value of the variable did not change since their most recent validation. This approach is significantly less effective than TL2 or LSA because even updates to memory locations not accessed by a transaction will always trigger full incremental validation. Furthermore, with such a heuristic the global variable is a bigger bottleneck than the global time bases used by TL2 or LSA because transactions have to read this variable on every transactional access; thus, even disjoint updates will lead to costly cache misses.

Zhang et al. expand on the earlier commit time sharing schemes of LSA-RT and TL2 by investigating further variations of the commit phase in time-based TM algorithms [123]. They focus on shared counters as time base, and try to either reduce the number of updates to the global time base or avoid unnecessary validations on commit.

Overall, time-based validation is now used by most STMs that use invisible reads and need to provide transactions with consistent snapshots. For example, SwissTM [34] consists of LSA and a variation of TinySTM's [42] encounter-time locking. Likewise, the STM in Intel's TM stack [83, 118] uses LSA with a write-through design but combines it with a transaction execution mode that uses visible reads; transactions that need to make progress can use the latter mode, and can still run concurrently with LSA-based transactions.

Two other STMs proposed more recently than LSA and TL2, NOrec [22] and TML [20], try to reduce STM runtime overheads by using as little metadata and instrumentation as possible. TML is essentially equivalent to a write-through LSA implementation (i. e., Algorithm 3) that uses only a single ownership record, which allows optimizations such as not having to maintain a read set or an undo log for write. However, TML does not scale in the presence of update transactions even if concurrent transactions do not conflict with each other. NOrec aims at similar optimizations but tries to improve scalability in workloads with update transactions by using a write-back design and combining time-based

validation with value-based validation (see Section 7.3.2 for details).

STMs proposed prior to LSA either used visible reads, like in SXM [46], or invisible reads with incremental validation, like in DSTM [54] or ASTM [79].

There are several STMs [28, 98, 51, 1, 88] that do not guarantee the consistency of a transaction during its execution, and are thus not suitable for transactions in a C/C++ environment as explained in Section 4.2.

How to map memory locations to TM synchronization metadata has seen little investigation so far compared to other aspects of STMs. To the best of my knowledge, all major word-based STMs for C/C++ environments use a mapping similar to the Simple hash function in Algorithm 4. For example, Yoo et al. [121] observe that 2^{14} orecs and a conflict detection granularity of 64B result in very frequent false conflicts in Genome and Vacation; as a new approach, they essentially propose to instead use 2^{20} orecs that are packed more densily than in the prior implementation. Multiplicative hashing has not been evaluated so far, which might have been motivated by the larger runtime overhead of multiplication in less recent CPUs. Felber et al. [42] propose hierarchical locking, which adds one additional level of TM metadata whose items each summarize a subset of all orecs; this allows for skipping the validation of certain orecs if no orec in the respective subset has changed. Nonetheless, the Simple hash function is still used for this mapping. Zilles and Rajwar [125] show that in a simplified model of execution under a TM, the probability of false conflicts grows quadratically with both the number of locations accessed by each transaction or the number of transactions executing concurrently. They propose to use a tagged ownership table in which orecs also store details about the memory addresses they have been acquired for, and refer to a chaining hash table as one possible implementation of this; however, they do not evaluate such a scheme in practice but only describe the potential runtime overheads of such an implementation. They do not further consider specific memory–to–orec mappings but rather assume universal hash functions in their model.

Several authors [110, 83, 80, 121] have investigated how to reduce the runtime overhead of ensuring privatization safety. All of these approaches either rely on quiescence schemes using timestamps from a global time base, or use it as a baseline implementation for transactions that use invisible reads. Thus, time-based TM is a good fit for these schemes because it already provides transactions with suitable timestamps. Besides explicit privatization fences based on quiescence, Spear et al. [110] also consider the instrumentation of nontransactional code as an alternative. Yang et al. [83] observe that there is no need to quiesce transactions that have not (yet) performed invisible reads (e. g., because they use visible reads instead). Marathe et al. [80] investigate optimizations based on making read operations visible to a limited extent that is still useful for privatization safety purposes but incurs less runtime overheads than fully visible reads. Yoo et al. [121] propose to provide interfaces that allow programmers to mark transactions that do not privatize data.

Tuning the performance of TM implementations, possibly at runtime, is explicitly considered by a few authors (but is to a lesser extent part of many TMs). Intel's TM runtime library [83] can switch between TM algorithms at runtime by changing a function pointer table that dispatches calls to the library to different internal TM implementations. GCC's TM runtime library [44] also has this ability but currently uses only a very simple policy for the choice of TM algorithm. Spear [108] investigates this in more detail, including switching

between differently optimized variants of the same STM algorithm (e. g., avoiding write-set lookup costs in write-back STMs by running a specialized variant until the first write happens in a transaction). Felber et al. investigate using a hill-climbing strategy to tune memory-location–to–metadata mapping of an STM.

However, validating the effectiveness of any tuning decision is difficult given the current lack of a wider range of realistic TM benchmarks; thus, automatic tuning is not not within the focus of my work (see Section 3.2). This also applies to tuning the temporal aspect of TM implementations. One important part of this is deciding how to react to conflicts between transactions, which is called contention management and can range from simple back-off schemes to more advanced conflict resolution schemes that dynamically compute priorities of transactions (e. g., as proposed by Scherer and Scott [101]).

Finally, time-based validation is also beneficial outside of memory-only transactions: Google's Percolator [85], which is a system built to incrementally process very large data sets with transactional guarantees on data centers with many servers, uses essentially the same algorithm as the snapshot-isolation version of LSA [92]. Application data is kept in the Bigtable distributed storage system, which can provide atomic access to single data items extended by a lock flag and a timestamp. As global time base, Percolator uses a centralized server that offers functions to get the current time and acquire a new commit time.

Chapter 6

Compile-Time TM Optimizations

TM support in compilers is an essential building block of a realistic TM implementation (see Section 3.2). This is not just due to ease of use and hosting the implementation of programming-language integration, but also for performance reasons: Everything in an application with transactions that can be done and optimized at compile time does not lead to overheads at runtime.

Furthermore, compilers have more high-level information than pure runtime-library–based solutions, for example. With a split of responsibilities between TM compilers and runtime libraries as explained in Section 4.2, compilers have all the information available in the source code of the transactional application (e. g., data types, or all nontransactional code). In contrast, runtime libraries typically implement a simpler ABI that requires passing less information to it and focuses just on the execution of transactional code.

There are many possible TM-centric compile-time optimizations. For example, the ABI discussed in Section 4.2 allows compilers to convey whether a certain transactional memory access always follows after a certain other access to the same memory location (e. g., that a certain load is always preceded by a store). DTMC (see Section 3.4.1) does not perform perform this kind of optimizations but instead applies just general-purpose optimizations to transactional code.

Instead of such low-level optimizations, the category that I focus on in what follows are *divide-and-conquer* approaches. My past experience indicates that it is unlikely that a one-size-fits-all TM algorithm exists that is optimal for each workload. This is also backed up by the performance results shown for STMs in Section 5.2.2 and for HyTMs in Section 7.3, and the many different STM and HyTM algorithms that have been proposed and make different trade-offs to optimize for different workloads (see Sections 5.4 and 7.3.3).

For example, consider the difference between visible reads and invisible reads discussed in Section 5.1: While invisible reads are probably the better general-purpose technique, visible reads might perform better for workloads with a high probability of conflicts between transactions because they allow the TM to choose which transaction to abort or stall. Another example is the granularity of conflict detection: data structures that suffer from a high probability of conflicts

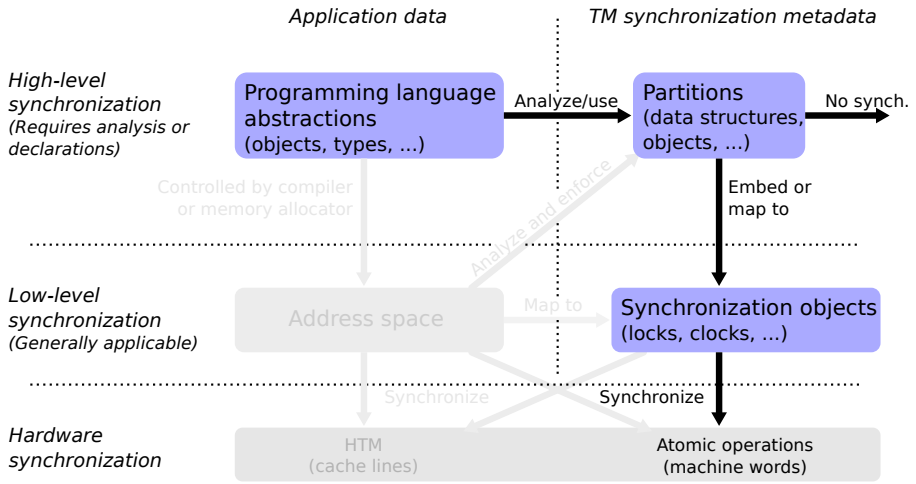

Figure 6.1: TM-based synchronization: Automatic partitioning of application data. Note that even with partitions, a TM runtime library might still map memory locations to low-level TM metadata; however, it can do so differently for each partition.

might benefit from coarse-grained detection (e. g., at the granularity of whole data structures), whereas one would rather use fine-grained detection for data structures like hash tables for which conflicts are typicallly less likely.

Supporting a divide-and-conquer approach does not yield a one-size-fits-all solution to this performance problem, but it allows for combining the strengths of different TM algorithms as required by a workload. This rests on the assumption that a large application is likely to execute transactions with different purposes on different parts of the application state and that workloads are more homogeneous within rather than across those parts. In turn, this can make it beneficial for TM performance to synchronize differently on each part; also, certain optimizations might only be possible on some parts but not for all parts of the application.

Thus, following the analysis of TM-based synchronization in Section 3.1.2, the dimension across which we have to partition is space (i. e., application data accessed by transactions). This partitioning should be automatic in the sense of not requiring programmers to provide any information about partitions. Manual partitioning would put programmers in control but would also bring back the difficulties of maintaining something like a locking scheme; acquisition order would not matter but programmers would have to follow and agree on a program-wide data–to–partition mapping.

But how can we best partition application data? Consider Figure 6.1: Trying to partition on the low level of raw addresses is not helpful because a large amount of high-level information has already been lost, other placement decisions affect the layout of data in the address space (e. g., compilers or memory allocators), and runtime overheads are harder to avoid. These difficulties are discussed in more detail in Section 5.2.2.

Instead, we should try to analyze the program at a higher level and involve

the compiler in finding partitions. Once we have high-level partitions, and know which partition a transactional memory access targets, we can still do address-based mappings within each partition.

With high-level partitioning, we thus get two levels of mapping application data to synchronization metadata, instead of just one. One could envision to use more levels but each level potentially increases the runtime overheads (e. g., by having to navigate through another layer of indirection to find synchronization metadata at runtime), so two levels seems to be a good choice. Also, the high-level partitioning imposes only little runtime overhead as most of the analysis happens at compile time.

It is important that partitions are disjoint (i. e., that each memory location is associated with exactly one partition) because this allows the TM to synchronize per partition, including choosing different synchronization algorithms for different partitions. Also, the partitioning should be stable in the sense that a memory location's associated partition should not change during the lifetime of the programming-language–level object at this location.[1]

Because we want to move most of the overheads of partitioning to compile time, the compiler must infer which partition a certain transactional memory access is associated with. For programming languages like C/C++ that use pointers, this requires the compiler to perform points-to analysis, which tries to provide this information by tracking how pointers are used and passed between parts of a program (e. g., functions).

Fortunately, there already exists a powerful points-to analysis technique, Lattner's Data Structure Analysis (DSA) [70], that has an LLVM-based implementation and can thus be used by DTMC. To make DSA applicable for the TM use case, I had to apply a few changes to it, most importantly adding support for analyzing multi-threaded programs, replacing the runtime component with TM-specific support, and tuning DSA-internal policies and heuristics. I will describe both DSA and the changes in Section 6.1.

Next, in Section 6.2, I will present how to use partitioning to improve overall TM performance by tuning the TM's concurrency control scheme separately for each partition. This employs just a simple tuning mechanism but already results in good performance improvements, and is thus a proof of concept for such divide-and-conquer TM optimizations. For example, the partitioning helps to reduce false conflicts between transactions, and can remove synchronization altogether for thread-local and transaction-local data.

In Section 6.3, I will then show how to colocate TM synchronization metadata with application data, which allows for a conceptually different mapping from memory locations to TM metadata and can improve the cache footprint of transactions and thus decrease runtime overheads. This is only possible for application data that is used like objects, so that it is clear at compile time which object a memory access targets and thus where the metadata resides in memory. We can use the modified DSA to detect which partitions have this property, and an additional compile-time transformation to actually extend each object with

[1] Note that this does not mean that a certain memory address is always associated with the same partition. If a variable is allocated dynamically on the application's heap, then the associated partition must be stable until the variable's memory is deallocated; but another variable which happens to be allocated later at the same memory address could very well be associated with a different partition. This is possible because TM synchronization is defined on the programming-language level, not on the memory address level.

TM metadata in such partitions.

Finally, I will discuss related work in Section 6.4. The implementations of partitioning and both optimizations reside in branches with older versions of DTMC and TinySTM++ (see Section 3.4.2), so the performance results presented in this chapter cannot be directly compared to those in Sections 5.2.2 and 7.4; for example, privatization safety is not yet guaranteed by these implementations.

6.1 Automatic Partitioning of Application Data

In this section, I will describe the compiler analysis and transformations that provide partitioning and detect which properties are common for all objects in a certain partition. It is automatic in the sense that it does not require programmers to provide any special annotations, but instead works on plain C/C++ programs, for example.

The underlying techniques are Lattner's Data Structure Analysis (DSA) and Poolalloc [72, 70]. I changed the implementations of these techniques to be applicable in a multi-threaded setting and tuned towards the TM use case. Also, Poolalloc originally uses partitioning information to improve how memory is allocated for objects within a partition, which I replaced with the TM-specific usages presented in Sections 6.2 and 6.3.

DSA is implemented as an LLVM analysis pass, and Poolalloc is a transformation pass. Both do whole-program analysis or transformation of code expressed in LLVM's Intermediate Representation (LLVM-IR), which is conceptually similar to Java bytecode. This is straight-forward to do with LLVM because source files are by default compiled to object files carrying LLVM-IR and code generation happens in the traditional linking phase on the combination of all object files. In other words, LLVM does link-time optimization by default.

DTMC also schedules the TM transformations to be executed at the linking phase. If partitioning is enabled, DTMC schedules DSA and Poolalloc to run before the TM transformations pass, which then has partitioning information available and can, for example, additionally pass a pointer to partition metadata in each call to the TM runtime library's load or store functions.

Poolalloc also has a runtime component, which allows it to calculate and track more detailed points-to information (i. e., instances of partitions, see below for an example). Doing this at compile time would require cloning functions and specializing those clones, which would result in too much code bloat. The resulting runtime overheads are small, and Poolalloc can decide at compile time and on a per-partition basis whether it tracks this additional information.

In what follows, I will first describe DSA and then Poolalloc, and conclude with a description of my changes to both of them.

Overview of DSA. DSA is a points-to analysis that is inter-procedural, context-sensitive, unification-based, and field-sensitive (see below for explanations). It basically tries to analyze which data structures are used in a program, and which properties they have (e. g., whether they are always of a certain data type).

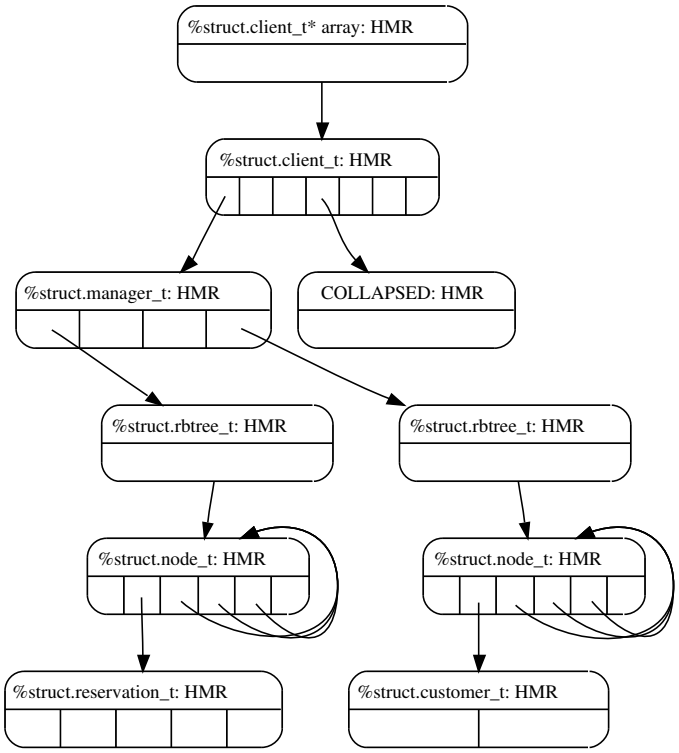

Figure 6.2: A part of the DS graph for function main of Vacation.

DSA models points-to information as a graph of nodes that represent data structures (DS). These nodes are built from and associated with (1) instructions in the program that access memory through pointers, (2) function arguments and return values, and (3) global variables. DSA considers a certain value (e. g., a global variable) as a pointer iff it is used as a pointer.

As an example, Figure 6.2 shows a part of the DS graph for the function main of the Vacation benchmark (see Section 3.4.3); the complete graph is significantly larger, and also contains more edges starting at the nodes shown in the figure. It shows some of the important data structures used in this benchmark. Edges between the nodes represent points-to information. The small boxes in the bottom of each node represent one field in the respective structure types. For example, the first and fourth field of structure manager_t point to two separate red-black tree instances; DSA can detect separate instances of the same data type. The analysis of all shown nodes is complete. All nodes except the one marked as "collapsed" have a type, meaning that all memory locations associated with such a node are accessed in a type-safe way in the program. Only the node shown at the top is an array, meaning that all other nodes originate from non-array allocations and uses.

DSA builds such DS graphs incrementally, starting with an intra-procedural analysis. It analyzes each function and creates an initial graph based on how pointers and data structure instances are used in the function. For example, if

the function contains a load to an address calculated as a pointer plus a field offset derived from a certain structure type, then the pointer is assumed to point to a DS node with this type. Besides type and points-to information for pointers, DS nodes also contain a set of flags stating whether the data structure is on the stack or on the heap (which is derived from where the pointer originated; for example, nodes on the heap are marked with "H"), whether the information about the node is complete (i. e., all of its uses have been analyzed), and a few other things.

This analysis is *field sensitive* in the sense that pointers stored in different fields of a data structure can point to different nodes within the DS graph.

However, a field in an existing node can point to at most one other node. To guarantee this property, whenever it is discovered that pointers stored in the same field point to disjoint nodes, these nodes are *unified*. Likewise, any pointers that are found to potentially alias each other (i. e., potentially point to the same object in memory) have to be unified as well.

To unify two nodes, DSA merges them into a single node in a way that preserves uncertainty. For example, if one instruction uses a pointer of type A but another instruction uses the same pointer with incompatible type B (e. g., after a cast), then DSA will unify the DS nodes for both uses of the pointer and will infer that the type for the pointer is not known in *any* of the uses (i. e., marking the node as "collapsed"). Other properties are handled in a similar way (e. g., nodes pointed to by the two to-be-unified nodes will also be unified).

A DS graph node is marked as *complete* in a function if DSA has analyzed all its uses. Thus, after the intra-procedural phase, DS graphs contain complete information about DS nodes whose associated pointers do not (transitively) escape from the function. For example, a node is not complete if a pointer associated with it escapes to callees due to being used as an argument in function calls.

To obtain whole-program information, DSA then executes a bottom-up pass on the program's call graph and merges the DS graphs of callees into callers, which results in complete information for nodes that do not escape to callers. Like in the intra-procedural phase, DS graphs are merged by unifying aliasing DS nodes into a single node. Thus, DSA is also *context-sensitive* (i. e., data structures are distinguished based on call graphs and not just allocation sites, for example).

Finally, DSA can make a top-down pass on the call graph to propagate information from callers to callees, again by merging the DS graphs. This is an optional step and only required by some clients of the points-to information (see below for further discussion).

DS nodes that escape to non-analyzable functions (e. g., external functions) or through external globals will remain marked as incomplete. This is partially mitigated by DSA having built-in knowledge of the semantics of frequently used standard library functions such as `malloc` or `memcpy`.

The unification of nodes is an important property from the perspective of the partitioning use case for DSA: If two pointer values in a function are associated with different DS nodes marked as complete, then it is guaranteed that they point to non-overlapping memory regions. Otherwise, DSA would have unified the DS nodes to a single node.

Overview of Poolalloc. However, the reverse is not necessarily true: Non-overlapping memory regions can be associated with the same DS node. This is where Poolalloc comes into play, which is a mixed runtime/compile-time technique that can identify and track instances of data structures.

In a nutshell, Poolalloc instruments the program so that whenever a function creates a complete DS node (e. g., before using malloc to create objects associated with this node), another call to Poolalloc's runtime library is inserted that creates a new metadata object for this DS node. These metadata objects are called *pools*, and Poolalloc uses them to optimize memory allocation and locality within such pools. Which nodes pools are created for is driven by heuristics.

Poolalloc then also changes function calls and signatures so that callers pass pointers to pools that are required by callees. For example, if a pointer in a caller escapes to a callee, then the caller would also pass the pool along with the pointer in the call.

Thus, this adds runtime tracking of pools to the program, and allows for pools to be distinguished from each other even deep in the call graph, and without having to clone functions. For example, two separate data structures instances created using the same initialization function can thus be assigned to two different pools.

Furthermore, we can thus detect whether two pointers might address overlapping memory regions simply by comparing the pools associated with the pointers: If the two pointers belong to different pools, then the objects pointed to will be disjoint as well and will not alias. This allows us to use Poolalloc's pools as partitions for TM.

Implementation details. To be applicable for the TM use case, I had to apply a few changes to both DSA and Poolalloc. For example, both were originally implemented for single-threaded programs.

The TM runtime library replaces the original Poolalloc library and does not retain Poolalloc's original per-pool memory optimizations. It allocates the metadata objects that represent pools on the heap rather than on the stack as the original Poolalloc runtime does. This is necessary because threads can share data even if one does not dominate the other in the call graph.[2] Furthermore, because pools can be shared between threads, the runtime uses garbage collection for the metadata objects that represent pools. I also slightly changed Poolalloc's heuristics for when to create pools to be more suitable for TM and partitioning than for Poolalloc's original purpose.

For DSA to support multi-threaded programs, I primarily had to add custom handling of pthread_create (i. e., the POSIX Threads function that creates new threads). Work items for a thread are typically passed through it as an argument, so DSA needs to know that a call to pthread_create is conceptually similar to an indirect call to the thread function. Likewise, Poolalloc performs additional instrumentation of the program so that calls to pthread_create get redirected to a wrapper function in the TM runtime library that additionally communicates all necessary pools to the newly created thread.

I also added support for detecting transaction-local pools (i. e., pools which

[2] In single-threaded programs, two functions can only operate on the same pool if one is an (indirect) callee of the other or if the pool is also associated with a global variable. Pools for global variables are global variables too and not dynamically allocated.

are created and destroyed in the same transaction) and thread-local pools in simple cases.

Finally, I changed the DSA implementation to be more rigorous in some cases because the original clients of DSA's results (e. g., pool allocation) can cope with inconsistencies that would break the TM-related uses. Second, for certain programs, DSA cannot always guarantee consensus on DS node information with just the bottom-up and top-down merge passes. To guarantee this, it would have to continously merge in both directions until the DS graph is stable and DS node information does not change anymore. However, this is just necessary for the optimization in Section 6.3 and it is an implementation issue, not a conceptual limitation. The benchmark programs used to evaluate those optimizations do not have the problematic call graph patterns.

6.2 Partitioning-Aware STM and Dynamic Tuning

As described previously, automatic partitioning of application data enables us to use a divide-and-conquer approach when tuning a TM implementation. DSA and PoolAlloc provide the TM with partitions that are disjoint in memory (see Section 6.1), and determining which partition a transactional memory access targets is possible at very little cost.

This allows the TM to isolate the partitions with respect to concurrency control and to facilitate the tuning of the TM by considering each partition independently [94]. The underlying assumption is that a typical application contains a variety of data structures, each of which being subject to different transactional workloads. Thus, by selecting a concurrency control mechanism suitable for each partition and by tuning each partition individually, we can improve the overall transaction throughput.

In contrast, TMs that are not aware of partitions have to choose one configuration for controlling concurrency in the whole application. For example, an STM like the one presented in Section 5.2 will have to use the same array of orecs for all transactions and all data in the program, which can result in false conflicts and makes tuning more difficult (see Section 5.2.2 for more examples). With partitions, the TM can use a separate array of orecs for each partition, or can use other TM algorithms (e. g., that only use a single orec). Combining different per-partition concurrency control schemes is straight-forward (especially for orec-based STMs), which is necessary to allow programs to access several partitions in the same transaction.

Partitioning implementation. For partitioning, it is sufficient to use just the bottom-up merging of DS graphs in DSA. The current implementation instructs PoolAlloc to create one partition for each complete node of a DS graph, although it would be also possible to group several nodes in the same partition. For example, for a DS graph like Figure 6.2 on page 135, it might make sense to use one partition per red-black tree consisting of both the root and inner nodes of each tree instance. Also, there is no partitioning of data structures for which PoolAlloc uses only a single node in the DS graph.

Partitions are represented by partition descriptors, which are instantiated at runtime when the control flow reaches a partition creation point in the program. These descriptors are small data structures that store the TM metadata for partitions, including the type of concurrency control in this partition (see Section 6.2.1 for further details).

The TM runtime library's functions for transactional loads and stores receive an additional argument that holds a pointer to the descriptor of the partition that is associated with the targeted memory address of the load or store. The compiler supplies the descriptors with the help of PoolAlloc. Functions that access memory and have transactional wrappers (e. g., malloc or memcpy) are treated similarly: The compiler redirects calls to those to transactional wrappers that are also partitioning-aware and accept the required partition descriptors as additional arguments.

As a result, whenever the TM runtime library has to perform a transactional memory access, it is aware of which partition this access targets. On such

an access, it loads the type of the partition from its descriptor and dispatches execution to the TM runtime library code responsible for this type. This constitutes a large part of the runtime overhead of partitioning (see Section 6.2.2) but much of this overhead could be removed by better compiler optimizations (e. g., creating different code paths specialized for different partition types, or switching between types only when necessary).

The TM runtime library also provides a default partition for every transactional memory access that is not associated with a partition. This is the fall-back implementation for accesses that, for example, target incomplete DS nodes (e. g., data that is also accessed in external functions and whose uses are thus not completely analyzed) or accesses that are assumed to be associated with a partition by a callee but not by the caller.

6.2.1 Tuning

Using multiple partitions allows us to perform various kinds of optimizations that would be ineffective with a single partition. For example, it is unreasonable to assume that a single global partition would be read-only, but it is not unlikely that some partitions in an application would be read-only (see Section 6.2.2).

Table 6.1 shows the types of concurrency control for partitions that I implemented to evaluate the potential benefits of partitioning-aware STM. They offer various trade-offs in terms of concurrency and overhead.

Note that even though the STM can use a different algorithm in each partition, it still provides transactions with the same guarantees as the original STM does. The integration of other concurrency control mechanisms than those shown in Table 6.1 should be straight-forward as long as they can participate in some kind of two-phase commit protocol. Note that all mechanisms need to be able to roll back a transaction. For example, even though partitions of type Exclusive Lock will never have to abort due to concurrent accesses by other transaction to the partition, it could be used in a transaction that uses other partitions in which aborts are possible.

Partition descriptors contain the TM metadata required to do per-partition concurreny control: (1) the partition's type, (2) a single orec for the Single Orec and Exclusive Lock types, (3) a pointer to the array of orecs, the number of orecs, and the Shift parameter for the Multiple Orecs type (see below), and (4) a few counters to maintain statistics (e. g., the number of aborts in the partition).

Partitions can be tuned on demand and independently of each other. When a thread wants to tune a partition, it (1) tries to change the partition type to Tuning using a CAS operation, (2) acquires a new timestamp from the global clock used by LSA for time-based validation (see Section 5.2), and (3) waits until every active transaction has a start timestamp (i. e., snapshot time) larger or equal than the acquired timestamp. If a transaction accesses a partition that is being tuned, it aborts and updates its start timestamp. Thus, if step (1) succeeded, then after step (3) every transaction will discover or will already know that the partition is being tuned. The thread that performs the tuning can thus—after step (3)—change the partition's metadata and finally set the new partition type. Note that, although we do have to wait for active transactions to finish, tuning will only delay the transactions that actually access the partition being tuned.

Type	Concurrency control	Performance	Purpose
Multiple Orecs	LSA (see Section 5.2) but with per-partition arrays of orecs.	The indirection via partitions adds some overhead over the original LSA.	General purpose.
Single Orec	Single per-partition orec embedded in partition descriptors. Apart from that, same algorithm as with multiple locks.	Lower overhead than "Multiple Orecs" but supports only a single updating transaction.	Mostly-read and uncontended partitions.
Exclusive Lock	Single per-partition exclusive lock used for both reads and writes.	Does not allow concurrent accesses by transactions. Lower overhead than "Single Orec" because reads do not need to be validated.	Partitions that are rarely accessed concurrently at runtime.
Read-Only	No concurrency control. Does not allow updates.	Very low overhead but the partition type must be changed when a transaction wants to update data.	Read-only partitions.
Tuning	Transactions will abort.	N/A	Tuning.
Thread-local	No concurrency control. Undo-logging for writes.	Low overhead.	Special purpose.
Transaction-local	No concurrency control.	Very low overhead.	Special purpose.

Table 6.1: Partition types.

	First Update	20 aborts	1000 aborts	2000 aborts
Part-1	Multiple Orecs (O=2^{18}, S=6)			
Part-2	Exclusive Lock	Multiple Orecs (O=2^{18}, S=6)		
Part-3	Exclusive Lock	Single Orec	Multiple Orecs (O=2^{18}, S=6)	
Part-4	Exclusive Lock	Single Orec	Multiple Orecs (O=2^{10}, S=8)	Multiple Orecs (O=2^{18}, S=6)

Table 6.2: Partition tuning strategies Part-1 to Part-4. Initially, all partitions are of the Read-Only type. For Multiple Orecs, LSA uses the Simple hash function to map memory locations to orecs (see Algorithm 4 on page 96), with O being the number of orecs and S the value of the Shift parameter.

My prototype implementation uses only simple tuning strategies (see Table 6.2) based on runtime measurements and heuristics. All strategies initially set the type of all partitions to Read-Only. The type is changed if the number of aborts in the partition exceeds a certain threshold (e. g., on reaching 1000 aborts, Part-3 changes the partition type to Multiple Orecs, 2^{18} orecs and 6 as Shift parameter). I chose values for the number of orecs and the Shift parameter that provided good overall performance in the benchmarks (see Section 6.2.2). The type of the default partition is always Multiple Orecs.

Please note that while this approach supports workloads that dynamically change their characteristics at runtime (e. g., initializing a lookup table and later using it just for read-only lookups), simple tuning strategies can effectively limit the ability of the STM to adapt to these changes. For instance, the prototype never tunes a partition back to the Read-Only type.

6.2.2 Evaluation

Let us now evaluate the effectiveness of partitioning and per-partition tuning by looking at results from Vacation, Genome, KMeans, and LinkedList (see Section 3.4.3). The DTMC branch with partitioning support was used to compile those benchmarks, both with and without partition being enabled. Applications were compiled to 32b executables and run on an two-socket x86 machine with 8 CPU cores in total.

The benefits of partitioning depend very much on the quality of the compile-time analyis, so let us look at this first. Table 6.3 shows how many of the transactional loads and stores in an application are associated with partitions.[3] Accesses not associated with a partition are implicitly linked to the default partition by the STM. Partition creation points are calls in the program code that instantiate partitions. Note that the number of partitions actually created and accessed at runtime can be different from the number of creation points.

[3]Note that this shows the number of call sites in the application binary. The number of loads and stores executed at runtime is shown in Table 6.4.

Benchmark	Partition creation points	Partitioned/total transactional loads	stores
Vacation	19	160 / 164	103 / 105
KMeans	11	5 / 8	1 / 4
Genome	23	47 / 47	15 / 16
LinkedList	3	13 / 13	6 / 6

Table 6.3: Partitioning compile-time statistics.

The table shows that most of the accesses can be associated with partitions. KMeans has fewer partitioned accesses because it uses global variables, for which PoolAlloc does not by default create partitions.

Table 6.4 shows runtime statistics for partition accesses and aborts. All partitions were forced to be of the Multiple Orecs type and to use 2^{18} orecs and a Shift parameter value of 6. Partitions without transactional accesses are not shown. The abort statistics were gathered from benchmark runs with 8 threads. Load/store statistics were measured in benchmark runs with a single thread, a smaller 2K input file for KMeans, and only 40K segments for Genome.

We can see that the number of accesses varies a lot between the partitions. There are more reads than writes, but the ratio differs per partition. There are several read-only partitions but the largest partitions are often updated (e. g., in Vacation).

The default partition receives significantly fewer accesses than the other partitions (or none at all in Genome). This number could be further decreased by tuning PoolAlloc's heuristics for when to create partitions, and by improving the thread-local compiler analysis.

Transaction abort counts also have a high variance, which further indicates that partitions are different and subject to different patterns of transactional operations.

Overall, Table 6.4 shows that partitioning is effective for the majority of application state that is accessed transactionally in the benchmarks and creates many opportunities for different kinds of optimizations.

Table 6.5 illustrates the performance of the different partition types (see Table 6.1 for details about each type). For this measurement, benchmarks were run with a single thread only and all partitions were forced to be of a certain type. The LinkedList benchmarks run transactions that look for a specific element in lists with 2000 and 250 elements, respectively.

In my prototype implementation, partitioning adds non-negligible overhead to transactional accesses. The first reason is that the STM dispatches execution based on the partition type during each access (see the difference between the second and the third column). Further compiler optimizations could remove this overhead. For example, the compiler could detect that only one partition is used in a function and create a special version optimized for read-only or thread-local partitions. Note that the runtime tuning mechanism itself does not require transactions to check the partition type on every transactional access.

The second part of the overhead (i. e., the third column's values being less than 1) is due to the extra level of indirection that partitions result in. The partitioning-aware STM has to load the pointer to the array of orecs and the

Benchmark / Partition	Transactional loads	Transactional stores	Read/write aborts	Write/write aborts
Vacation-2M / 1	105M	11M	1.3K	43K
2	125M	20M	2.8K	33K
3	4.3M	450K	9	3K
4	192M	3.5M	107K	10K
5	253M	184K	192	0
6	258M	41K	33	0
7	37M	15M	17K	63
8	263M	41K	24	0
9	6.9M	5	328	0
10	8.2M	0	0	0
11	8.4M	0	0	0
12	8.4M	0	0	0
13	22M	0	0	0
default	21M	19M	5K	894
thread-local	14M	10M	0	0
KMeans-Lo / 1	524K	524K	451K	28K
2	524K	0	0	0
3	524K	0	0	0
4	524K	0	0	0
5	32K	0	0	0
default	44K	44K	16M	157K
Genome-4M / 1	11.8M	106K	316	49
2	91K	30K	186	436
3	118M	4.7M	567K	3K
4	952K	0	0	0
5	65K	75K	0	14
6	15K	0	0	0
7	1.9M	0	0	0
8	952K	0	0	0
9	30K	0	0	0
10	0	15K	0	0
11	337K	75K	1K	516
12	57K	0	0	0
13	141K	0	0	0
14	42K	0	0	0
15	84K	0	0	0
txn-local	73K	0	0	0
thread-local	0	42K	0	0

Table 6.4: Runtime statistics for partitions: Number of transactional accesses and aborts due to read/write or write/write conflicts.

Benchmark	Multipe Orecs	Multiple Orecs, No Dispatch	Single Orec	Exclusive Lock	Read-Only
Vacation	0.71	0.80	1.21	1.32	N/A
KMeans	0.85	0.94	1.07	1.13	N/A
Genome	0.74	0.84	1.17	1.34	n/a
LinkedList-2000 (only lookup)	0.68	0.86	1.36	1.90	2.30
LinkedList-250 (only lookup)	0.61	0.80	1.29	1.73	2.22

Table 6.5: Performance of partition types and overhead of supporting partitions. Shows speedup relative to the non-partitioning-aware STM that uses the Multiple Orecs scheme (single-threaded execution).

number of orecs and Shift parameter from the partition descriptor, whereas these values are constant in the non-partitioning-aware STM.

Nevertheless, the other columns show that despite these overheads, even just using a single orec instead of multiple orecs can increase performance significantly. It seems likely that further compiler optimizations could increase the performance advantage of the partition types that need no or very little per-access concurrency control code (e. g., Read-Only or Exclusive Lock) because no calls into a TM runtime library are necessary and undo-logging can be efficiently inlined in the application code.

After showing the applicability of partitioning and the potential of optimizations that it enables, let us now look at performance results for the STAMP benchmarks (see Table 3.2 for details about the benchmark configurations). Figures 6.3, 6.4, and 6.5 show comparisons of the performance of non-partitioning-aware LSA and of the four simple tuning strategies of the partitioning-aware STM (see Table 6.2). LSA-O-S denotes LSA using the Simple hash function, 2^O orecs and S as Shift parameter. The LSA-O-S configurations shown in the figures are the ones that provided the best performance for eight threads, and the figures show both benchmark execution time (lower is better) and relative speedup to the best-performing configuration of the original STM (higher is better). Note that the data shown here can differ from the data shown in Section 5.2.2 due to the differences in the benchmarks, TM implementations (e. g., 32b versus 64b), and the machines the experiments have been executed on.

With Vacation (see Figure 6.3), the partitioning-aware STM is often slightly slower than the original STM due to the higher per-access overhead discussed previously (e. g., with single-threaded Vacation-2M-Lo, Part-1 is 30% slower than LSA-18-6, even though the latter uses the same STM).

However, the partitiong-aware STM performs significantly better if (1) it uses the Exclusive Lock type in single-threaded runs (available in Part-2, Part-3, and Part-4) or (2) in the high contention variant of the benchmark with a large number of threads. The latter also shows that increasing the number of

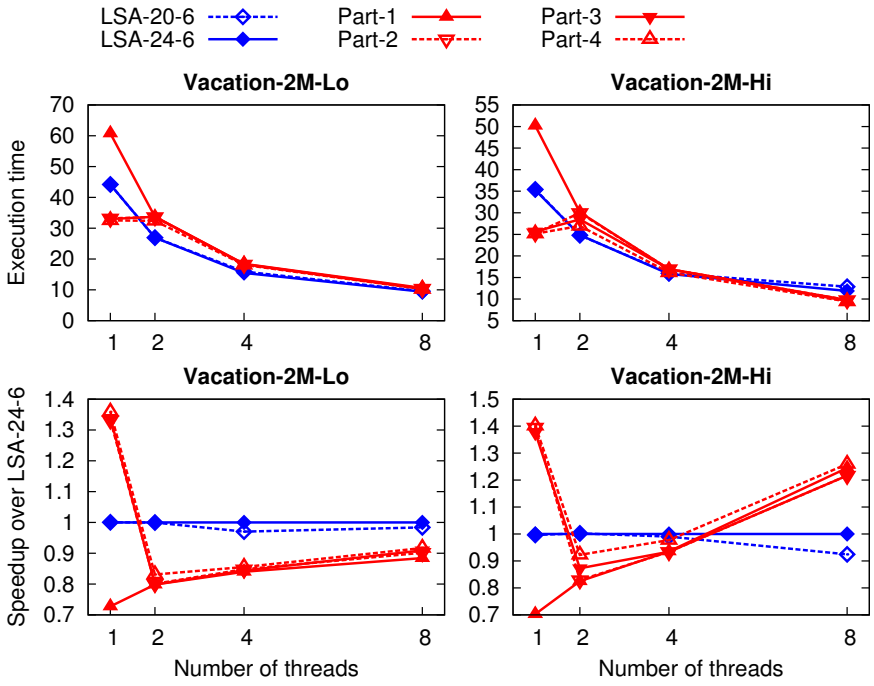

Figure 6.3: Performance of partitioning with Vacation.

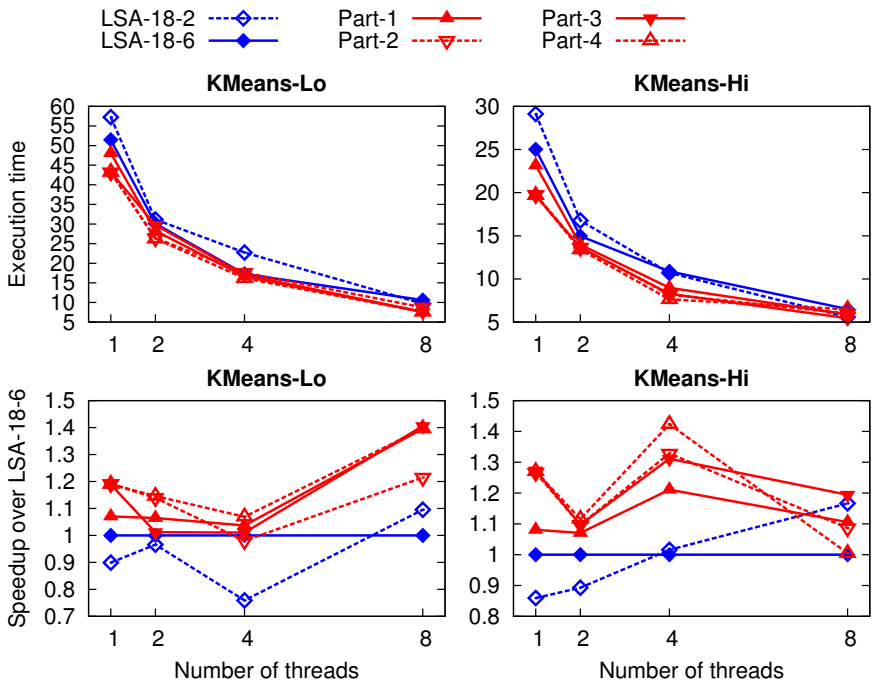

Figure 6.4: Performance of partitioning with KMeans.

orecs as in LSA-24-6 is not sufficient to avoid false conflicts in a global array of orecs; in contrast, the partitioning-aware STM can still scale even though it uses less orecs because the partitioning works like a high-quality mapping function that mitigates the limitations of the simple memory–to–orec mapping used by the STM.

With KMeans (see Figure 6.4), the partitioning-aware STM can take advantage of the three often-accessed read-only partitions (see Table 6.4), which can be exploited by all tuning strategies and thus help overcome the other partitioning overheads. Using a fine-granular memory–to–orec mapping seems to be important in the high-contention variant of the benchmark, which indicates that advanced tuning strategies should also adapt the number of orecs and the memory–to–orec mapping for each partition.

With Genome (Figure 6.5), the partitioning-aware STM performs significantly better than the original STM (with one exception). Although the statistics in Table 6.4 do not show this clearly, I observed that the two most-frequently accessed partitions (1 and 3) are read-only during the first phase of the benchmark, which allows the STM to handle many of the accesses using a Read-Only partition type. More importantly, the results show that partitions can decrease the STM's space overhead required for synchronization metadata significantly when using the Simple hash function, especially in Genome-8M: Part-4 uses five partitions with 256K orecs each and one with 1K orecs, whereas the non-partitioning-aware LSA uses 256MB of memory just for its orecs in the LSA-26-6 configuration. With 2^{24} locks, LSA suffers from frequent aborts due to false con-

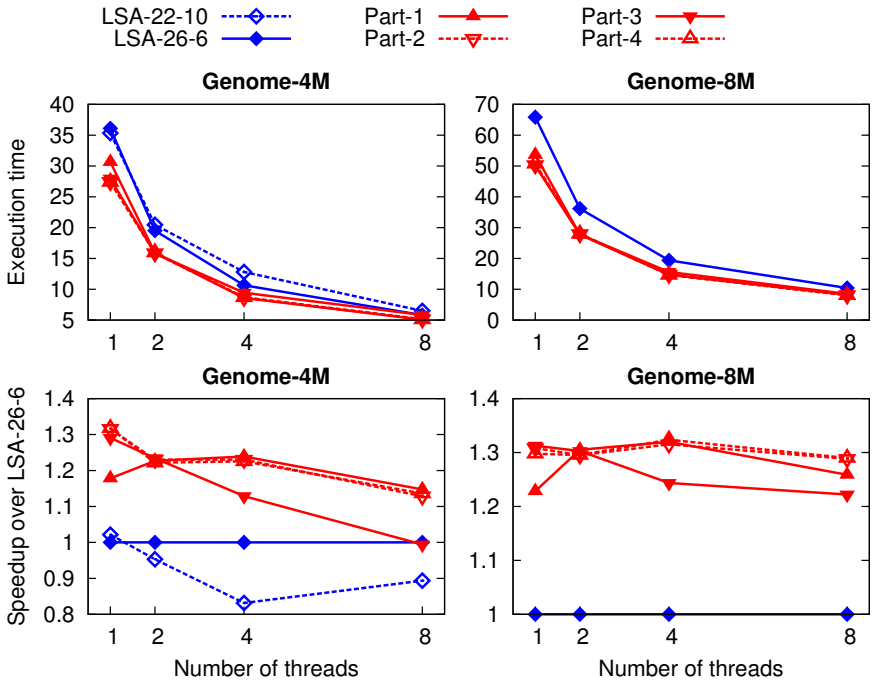

Figure 6.5: Performance of partitioning with Genome.

flicts resulting from the simplicity of the address-based memory–to–orec mapping, which leads to a sharp decrease in performance.[4] While this degradation could probably be partially reduced through proper STM contention management[5], partitioning allows the STM to attack the root cause of this performance problem.

Overall, these performance results show that even a not well-optimized prototype of a partitioning-aware STM and simple tuning strategies can already yield better performance than an STM without partitioning support. Furthermore, partitioning allows TMs to benefit from optimizations targeted at low-contended or read-only workloads, which would be less likely to occur in practice for larger programs whose application state is not partitioned.

It seems likely that more advanced code generation should be able to reduce the runtime overheads of partitioning (e. g., generating code paths specialized for a certain partition type so as to decrease the overheads of runtime dispatching, see Table 6.5). Likewise, better runtime tuning strategies should be able to increase the benefits of employing per-partition synchronization schemes.

[4]This is also the reason why I show only results for LSA-26-6 for Genome-8M.

[5]The LSA implementation evaluated in Section 5.2.2 switches a transaction to execute in serial-irrevocable mode when it has aborted more than 100 times (see Section 3.4.2), which weakens the worst-case effect of false conflicts.

6.3 Colocating Application Data and TM Metadata

TMs designed for C/C++ environments or other unmanaged environments[6] often use a low-level mapping from memory addresses to TM synchronization metadata, which is called *word-based* (see Section 5.2 for details).

Another alternative is to use *object-based* accesses, for which TM conflict detection happens on the granularity of memory objects that represent the objects or data structures used in the source program. This approach is popular and useful in managed environments [1, 51]. However, it cannot be easily used by TMs for C/C++ programs because those can access arbitrary memory locations via pointers, including pointers to fields of an object, for example. Type information is available in the source code but accesses are not guaranteed to be type-safe at runtime unless restricted programming-language dialects are used. This means that TM compilers cannot easily derive which object a certain access targets, and runtime lookups would be much too costly in terms of performance.

Besides restricting programming languages, TMs for C/C++ programs can also provide object-based accesses by requiring programmers to make explicit calls to the TM runtime library [113, 109]. However, both options do not provide the same level of ease of use and programming language integration that I am aiming for.

Nonetheless, object-based TMs have two potential advantages. First, if they keep metadata *external* and map to it from the base address of the object (i. e., the smallest memory address that is within the object) using a hash function, then they can thus use the knowledge about object granularity and base addresses to try to map more beneficially than with the flat-address-space approach that word-based STMs typically use (e. g., to avoid false sharing or an excessive amount of metadata).

Second, if object-based STMs use *in-place* metadata by embedding it into every transactionally accessed object, they can potentially benefit from the improved locality compared to when using external metadata. If objects are not too large, the object data and metadata will likely reside in the same cache line, which reduces the cache footprint of transactions. It can also result in fewer cache misses when reading data modified by other transactions because there is only one cache miss for both object data and metadata and not two. Also, no indirection is necessary to access the metadata.

However, the very same properties can also decrease performance, depending on the workload. For example, if there is very little contention, using only a few orecs could yield the highest STM performance but this kind of tuning is not possible with the fixed object–to–metadata mapping in the case of in-place metadata. Similarly, in-place metadata increases the size of objects and can thus increase the cache footprint.

So, how can we enable object-based accesses for C/C++ programs? We do not want programmers to provide extra information nor restrict the programming language, and we cannot look up object boundaries at runtime. Thus, we have to solve this at compile time, and we will not be able to always use

[6] Informally, in unmanaged environments, application code directly accesses the resources provided to its process. In managed environments, applications access resources through an intermediate layer such as a Java Virtual Machine.

object-based accesses. We need to partition memory into objects that we know and memory locations for which we are not certain which object they belong to.

Given that this is a partitioning problem, we can indeed use automatic partitioning of application data to solve it [89]. In particular, we can use DSA (see Section 6.1) to find partitions that contain only objects of the same type. For such partitions, DTMC can then generate transactional load and store calls to the TM runtime library that provide the base address and size of the object as additional parameters. The runtime library can use this information to map from object base addresses to external metadata. We can also go one step further and support in-place metadata by letting the compiler modify all allocations of objects in such partitions such that there is additional space for the metadata (i. e., an orec), which then can be used by the TM runtime library. Note that this compile-time approach is not bound to certain data types in an all-or-nothing fashion but can differentiate between instances of a type.

Similar to the default partitions used in partitioning-aware tuning (see Section 6.2), we can use a word-based default TM algorithm for all other partitions that are not known to contain objects. Because partitions are guaranteed to be disjoint, all accesses to a certain memory location will be either word-based or object-based, and combining both in a transaction is safe.

Next, in Section 6.3.1, I will describe this solution in more detail. The evaluation in Section 6.3.2 shows that an object-based STM can be significantly faster than a word-based STM with an otherwise identical design and implementation, even if the parameters of the latter have been tuned.

6.3.1 Enabling Object-Based STM

For our purposes, let us consider an object to be a continuous chunk of memory that can be defined and identified by its size (most likely derived from its type) and its base address in memory (i. e., the start of the memory chunk). We also want to only consider objects that could correspond to user-defined data structures in the program (e. g., ignore primitive types because they seem too small to justify object-based accesses).

Accesses to objects are identified by relying on the DS graphs produced by top-down DSA, which produces points-to information that covers all accesses to a memory location in the whole program. Note that as pointed out in Section 6.1, in general we would need to merge DS graphs up and down the call graph until no changes to DS graphs occur anymore, but the two merging phases of top-down DSA are sufficient for our benchmarks.

The compile-time support for object-based accesses is implemented as a variant of an older version of DTMC. When it is about to transform a memory access in transactional code, it checks the DS node associated with the access' target address and transforms the access into an object-based or word-based STM call. A node can be accessed using object-based STM functions iff (1) it is of a known structure type and accessed in a type-safe way, (2) has been completely analyzed, (3) is not external, and (4) is not an array. Otherwise, the access is transformed into a word-based STM call. I will discuss these constraints again below.

The compiler determines the size of the object based on the type of the DS node associated with the target of the access. The base address is computed

by either reusing base pointers available in the program[7] or based on the offset information contained in the points-to edges in DS graphs (pointers can target specific fields of objects).

In-place metadata. To be able to support STMs that use in-place metadata (i. e., embed transactional metadata such as orecs in each object), DTMC has to make sure that there is enough space for the metadata. The current implementation appends metadata to the end of the object (i. e., at base address plus object size), but prefixing the object with metadata would also be possible. To reserve sufficient space, DTMC enlarges all allocations (on stack and on heap) that could be used in an object-based access by the size of the metadata and adds calls to the TM runtime library that initialize the metadata after the allocation.

To counter potential memory bloat issues, DTMC could additionally track—via DSA—which nodes are actually accessed from within transactions, and only enlarge the allocations of those nodes.

Safety. The basis for the safety of these transformations are the guarantees given by DSA and the properties that are required for DS nodes to be eligible for object-based accesses: Typed data structures with stable DS node information that all parts of the program agree on.

DTMC ensures that there is no unknown behavior, constraints or uses of each object-based DS node by requiring complete node information. This also ensures that memory for objects was allocated by a function whose semantics are known by DTMC and which it can transform. For example, external functions would result in external and incomplete DS node information, and programs that forge pointers in nonanalyzable ways would either lead to nodes being part of arrays or no type information being available.

Data structure instances that are part of an array are not considered as potentially object-based accesses, so allocations reserve memory for only one object and metadata does not need to be explicitly added to data type definitions. DTMC only treats data structures with structure types as object-based to filter out primitive types.

DSA does not have custom handling of inheritance of C++ objects. If there is conflicting information when two DS nodes are merged (i. e., two uses of potentially the same pointer), node unification will lead to the respective parts of DS node information being marked as unknown. This can prevent object-based accesses but also ensures that in-place metadata can be safely appended to all fully analyzable objects.

Object-based STM implementations. Because data partitioning guarantees that object-based and word-based memory accesses are separated at compile time, it is easy to support both in an STM. The STM is essentially LSA as described in Section 5.2 (for word-based accesses), but with additional object-based load and store functions. The latter use the same algorithm as word-based

[7] Address computations for pointers to structure fields are explicit instructions in LLVM's intermediate representation for code. They start at a pointer and navigate through the data structure definitions contained in the program (and DS node information is associated with such instructions).

loads and stores but manage and map to orecs differently. For word-based accesses, the STM implementation uses the Simple hash function to associate memory locations with orecs (see Algorithm 4 and Section 5.2.2).

With external metadata (called *LSA–ObjE* in what follows), the address of the orec that is associated with an object is computed by mapping from the object's base address to an orec in the external array of orecs also used by word-based accesses, using the Simple hash function as well. Note that using the same set of orecs for word-based and object-based acccesses is possible because object-based load and store functions reuse most of the functionality in the word-based STM implementation.

With in-place metadata (*LSA–Obj*), each object has a single orec that is located right after the object in memory. Such orecs have the same layout as the orecs in the external array of orecs. Remember that we can obtain the address of each object's orec because the compiler supplies the accessed object's base address and size as arguments on each object-based STM load or store call. Privatization safety (see Section 5.2) ensures that the memory used for in-place metadata is never released and reused for other memory allocations until other transactions will not access it anymore. However, LSA–Obj's implementation does not provide general privatization safety but guarantees this just with respect to released memory; it keeps lists with to-be-freed memory chunks per thread and only releases memory after all other transactions that could still access the data have aborted or committed, using the same time-based approach that is the base for the implementation of general privatization safety.

Using the same STM algorithm for object-based accesses as for the purely word-based STM variants allows for a meaningful comparison between both; all that differs is how orecs are chosen and where those orecs are located in memory. Nevertheless, the underlying approach should be applicable to other word-based and object-based STM algorithms and implementations because of the compile-time separation between object-based and word-based accesses.

6.3.2 Evaluation

I will focus on two aspects in this section: First, I will show to which extent TM benchmarks contain object-based accesses, and whether DTMC can detect and transform these accesses. Second, I will evaluate the performance of the benchmarks when using the new object-based STM functionality in comparison to the word-based implementation. As benchmarks, I will use Genome, KMeans, Vacation, LinkedList, and RBTree as described in Section 3.4.3, but with different workload parameters and also a slightly different implementation in case of LinkedList and RBTree (see below).

Compile-Time Transformation to Object-Based Accesses

Table 6.6 shows compile-time (static) and run-time transformation results for the different benchmarks. Static loads and stores are instructions in the program that access memory from within a transaction and are replaced by DTMC with calls to the TM runtime library. The number of loads and stores at runtime were measured by instrumenting LSA-Obj and running the benchmark with a single thread in the default configuration for at least 10 seconds. Thus, the first columns give an indication of how many object-based accesses are in

Benchmark	Number of static WB loads/stores	Number of static OB loads/stores	Percentage of OB loads/stores (static)	Percentage of OB loads/stores (at runtime)
RBTree	0 / 0	84 / 77	100% / 100%	100% / 100%
LinkedList	0 / 0	16 / 6	100% / 100%	100% / 100%
Genome	47 / 19	32 / 18	40% / 48%	12% / 98%
KMeans	9 / 7	0 / 0	0% / 0%	0% / 0%
Vacation	25 / 9	243 / 188	90% / 95%	97% / 84%

Table 6.6: Applicability of object-based compiler transformations (OB is object-based, WB is word-based, single-threaded benchmark runs, counting only transactional loads/stores).

these programs, and the last column shows whether these are important for the benchmark's performance due to being executed often.

Both RBTree and LinkedList access a single logical[8] data structure transactionally, and DTMC detects this and uses just object-based accesses.

Genome's transactions access a mix of linked data structures and a large number of character strings. Word-based accesses are used for the strings, which leads to the low percentage of object-based accesses at runtime compared to the percentage of static object-based accesses in the program. However, while just half of the transactional stores in the program can be transformed to object-based accesses at compile time, at runtime they comprise the vast majority of stores that get executed.

KMeans mostly operates on arrays of primitive types. DSA detects these arrays but they are not considered for object-based transformations.

In Vacation, DSA detects the tree and linked-list data structures used by the benchmark with just a few omissions (e. g., the collapsed node in Figure 6.2). As a result, the majority of transactional accesses at runtime are object-based. Note that this depends on the portability and clean up changes to STAMP described in Section 3.4.3: The original source code of STAMP used containers holding 32b integer elements also for elements of pointer types, which makes portability difficult. DSA could have infered that the values are not integers but pointers by, for example, proving that the values are not modified by integer arithmetic, but this is not implemented.

Overall, DTMC is able to detect and transform the majority of object-based accesses. Better tuning of DSA could perhaps allow for an even higher ratio of object-based accesses (e. g., tuning how frequently it collapses DS nodes even though it could try harder to analyze the code). Likewise, adding special support for arrays could enable further compile-time optimizations in KMeans and

[8]DSA usually detects more data structures because it distinguishes, for example, between the head of the red-black tree and its nodes (see Figure 6.2).

Benchmark	Update transactions	Initial number of elements
LinkedList-1	20%	512
LinkedList-2	20%	16384
LinkedList-3	80%	512
LinkedList-4	80%	16384
RBTree-1	20%	512
RBTree-2	20%	65536
RBTree-3	80%	512
RBTree-4	80%	65536

Table 6.7: Configurations of IntegerSets benchmarks used to evaluate object-based accesses.

Genome.

Performance

To evaluate the performance of object-based versus word-based accesses at runtime, I will next show which transaction throughput LSA, LSA–Obj, and LSA–ObjE provide on a two-socket x86 machine with eight CPU cores in total. All benchmarks were compiled to 32b executables using DTMC, with the object-based transformations only enabled when compiling for LSA–Obj and LSA–ObjE. Every benchmark uses eight threads to execute transactions.

The IntegerSet benchmarks used in what follows differ to some extent from those described in Section 3.4.3. First, while they execute the same kind of operations, they strictly alternate insert with remove operations: If an update transaction is to be run, a thread will either remove the element it inserted previously into the set, or insert a new element with a random value. This keeps the ratio of actual update transactions closer to the targeted ratio, and the total number of elements in the set is likely to remain close to the initial number of elements. Second, integer values put into the set are always selected randomly from the $[0, 65536)$ range, and the set is initially populated with up to a certain number of random elements. Table 6.7 shows the configurations that I consider. The RBTree configurations with 64K elements are populated with 256K random elements to increase the number of elements that are actually in the set initially.

For STAMP, I will use the Genome-4M and Vacation-1M benchmark configurations listed in Table 3.2. I will not show any measurements for KMeans because it contains no object-based accesses.

The performance of word-based accesses (LSA) and object-based accesses that use external metadata (LSA–ObjE) is influenced by how the TM maps memory locations to orecs (see Section 5.2.2). Both use the Simple hash function, which is parametrized by the number of orecs and the Shift parameter (i. e., the number of least-significant bits that are discarded from input addresses). Therefore, I will show performance measurements for different settings of Shift (3–8 for IntegerSet, 4–10 for Genome and Vacation) and different numbers of orecs (2^{16}–2^{24}); parameters outside of these ranges are probably not feasible in practice. Note that the data shown here can differ from the data shown in

Section 5.2.2 due to the differences in the benchmarks, TM implementations (e. g., 32b versus 64b), and the machines the experiments have been executed on.

Let us first look at the performance of the IntegerSet benchmarks, which do not use any word-based accesses. Figure 6.6 shows transaction throughput for RBTree, omitting RBTree-3 results because they form a shape similar to the RBTree-1 results.

The throughput results for LSA–Obj form a plane because it only uses the orecs embedded into the objects and is thus not affected by the parameters to the hash function. LSA–ObjE is usually better than or equal to LSA but is still vulnerable to disadvantageous hash function parameters. With small trees (RBTree-1 and RBTree-3), LSA–ObjE can reach the performance of LSA–Obj with selected hash function parameters but not in general; LSA provides at least roughly 10% less throughput, and in some cases up to 30% less. For large trees and a 20% update rate (RBTree-2), LSA–Obj performs better than the other STMs except for a few hash function parameters. Only with frequent updates and large trees (RBTree-4) does LSA–Obj performs worse than the other STMs—but LSA–ObjE performs best in this case, despite also relying on hash functions to map memory locations to external metadata. This is interesting because it indicates that even associating each object with exactly one orec can be beneficial. Another interesting observation is that the hash function parameters that provide the best performance for LSA differ between small and large trees.

Figure 6.7 shows transaction throughput for LinkedList using LinkedList-1 as example; all four configurations show similar performance characteristics. LSA–Obj always performs best, and both LSA–ObjE and LSA provide 5–40% less throughput; only a few hash function parameters result in maximum throughput, and performance degrades considerably as soon as parameters do not match those sweet spots.

Overall, both the RBTree and the LinkedList measurements support what I already showed in Section 5.2.2: The performance of STMs that use arrays of orecs as synchronization metadata is significantly influenced by the hash function and the choice of hash function parameters, especially with the Simple hash function; furthermore, the workload also influences which hash function parameters yield the best throughput. In contrast, object-based accesses that use in-place metadata do not need to use a hash function and can still perform better even when the hash function parameters are well-tuned.

Let us now switch to the STAMP benchmarks, which execute both word-based and object-based accesses. Figure 6.8 shows transaction throughput for Genome. LSA–Obj's performance now is affected by the hash function parameters, but to a much lesser extent than both LSA–ObjE and LSA. Although most of the transactional loads are not object-based accesses (see Table 6.6), 98% of all transactional stores are. Thus, if the latter use in-place metadata as in case of LSA–Obj, the likelihood of false conflicts with other transactions is decreased substantially, and the the choice of hash function parameters also becomes less important. LSA–ObjE and LSA perform similarly but cannot provide the same maximum throughput as LSA–Obj.

Figure 6.9 shows transaction throughput for Vacation. The performance of all three TMs suffers heavily with a few hash function parameters (i. e., 2^{18} orecs and a Shift value of 4 or 6, and 2^{20} orecs and a Shift value of 4), which

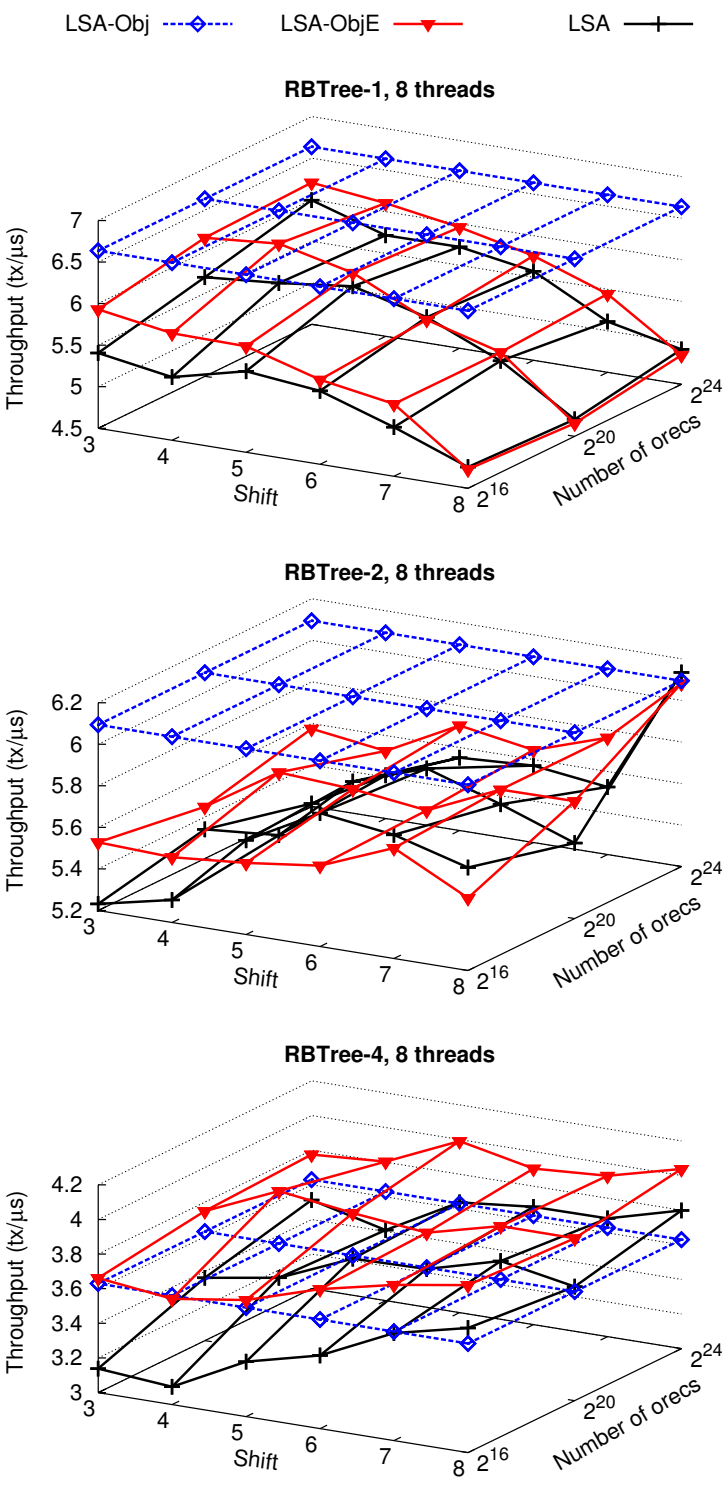

Figure 6.6: Performance of object-based accesses with RBTree.

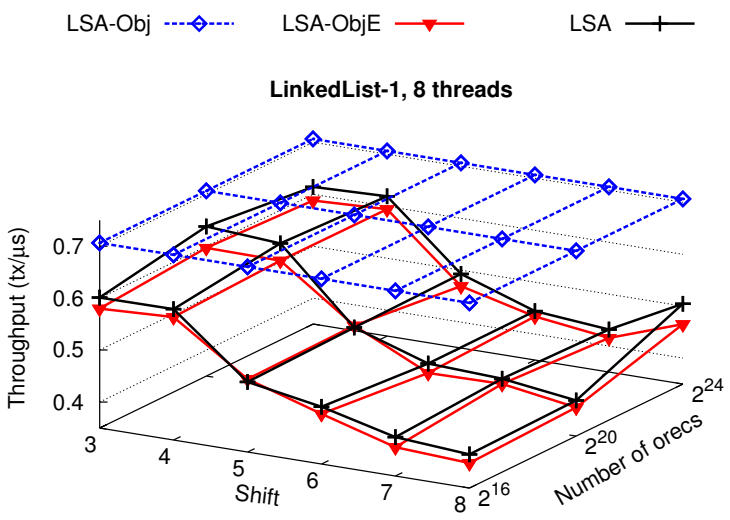

Figure 6.7: Performance of object-based accesses with LinkedList.

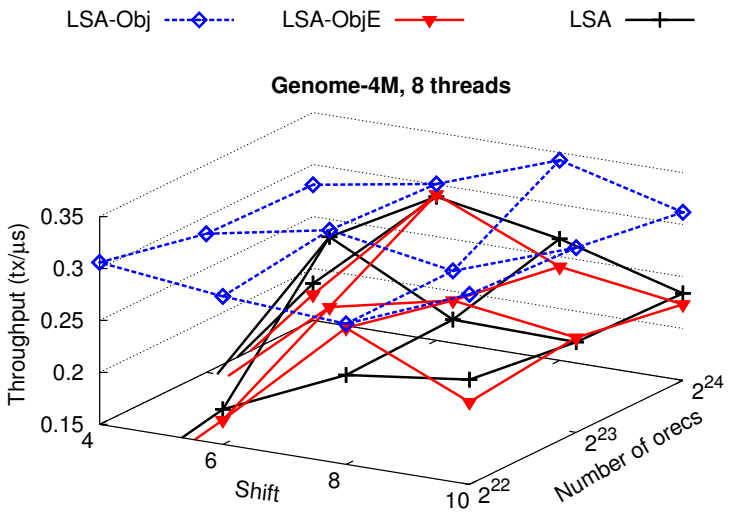

Figure 6.8: Performance of object-based accesses with Genome.

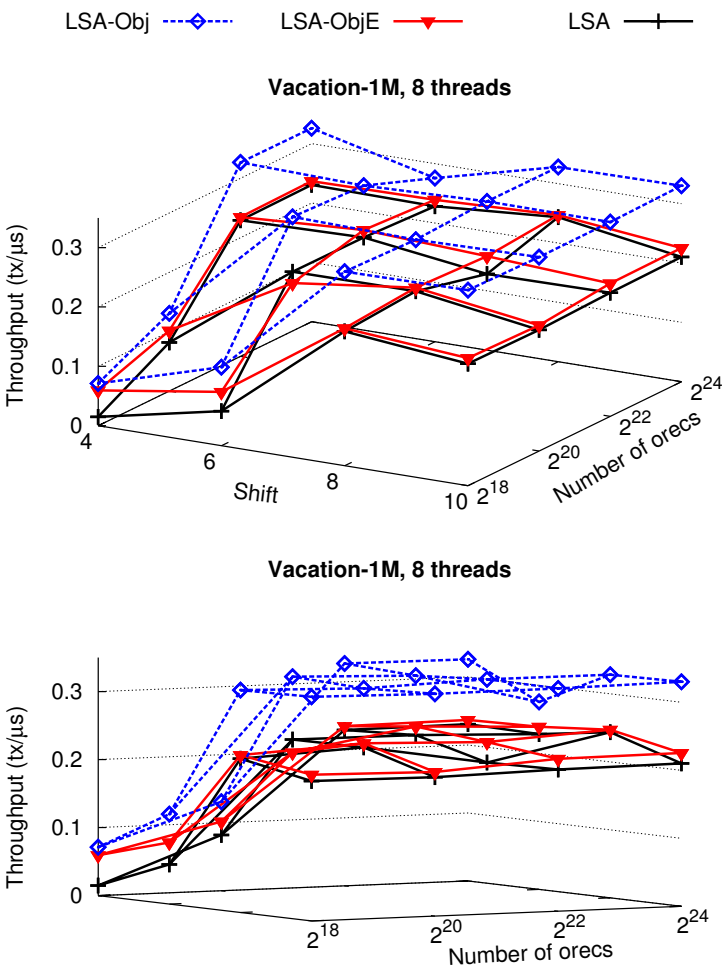

Figure 6.9: Performance of object-based accesses with Vacation. Both plots show the same data but from different angles.

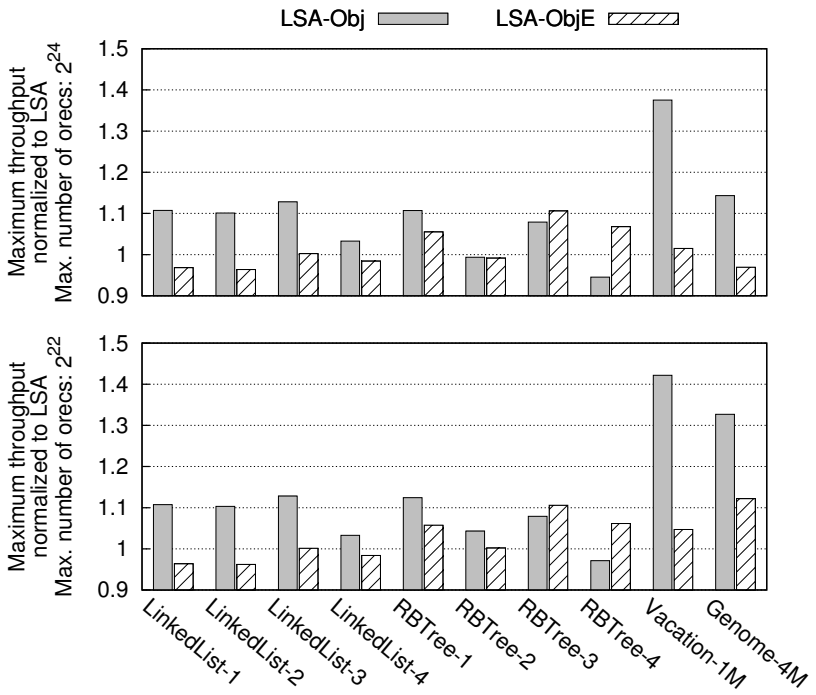

Figure 6.10: Peak performance of LSA–Obj and LSA–ObjE compared to LSA (i. e., with the best-performing hash function parameters selected separately for each benchmark and each STM), when the maximum number of orecs is 2^{24} or 2^{22}, respectively.

suggests that even the few word-based accesses in Vacation are prone to frequent false conflicts. However, when avoiding the problematic settings, hash function parameters do not affect performance significantly anymore. Interestingly, LSA–Obj then performs significantly better than both LSA–ObjE and LSA, providing usually about 30% more throughput. This indicates that this performance advantage is not just caused by fewer false conflicts but probably also by the increased locality that in-place metadata can provide. With smaller objects such as nodes of a red–black tree, the in-place orec is likely to reside on the same cache line as the object, which leads to potentially fewer cache misses and a smaller cache footprint of transactions.

Figure 6.10 summarizes the effect of object-based accesses on performance. It compares the maximum throughput that each of the STMs was able to reach when using the most favorable hash function setting for the respective benchmark configuration and STM (i. e., with perfect tuning). The benefits of object-based accesses with in-place metadata result in LSA–Obj always performing better or equal to LSA except for RBTree-4; LSA–Obj's throughput advantage is modest with the IntegerSet microbenchmarks but significant in the larger STAMP applications. Furthermore, LSA–Obj is much less dependent on a well-tuned hash function if object-based accesses are frequent. This also means that fewer orecs can be sufficient to provide equal performance; for example, in the

lower part of Figure 6.10 the number of orecs is limited to 2^{22}, resulting in a higher relative advantage of LSA–Obj and LSA-ObjE in several benchmarks.

6.4 Discussion and Related Work

Both optimizations that I have presented, dynamic tuning in a partitioning-aware STM [94] and colocating application data with TM metadata [89], rely on the divide-and-conquer approach of automatically partitioning application data. Related work can be categorized as either automatic lock allocation at compile time, TM metadata colocation, or other compile-time optimizations. Prior work primarily falls either in the first or last of these categories; related work published more recently than my work also investigates optimizations similar to my work but then also uses very similar techniques to do so.

Automatic lock allocation at compile time. Lock allocation refers to techniques that try to let compilers automatically infer a locking scheme for the atomic regions (e. g., transactions) in a program, often without any additional lock-related information by programmers except the demarcation of those atomic regions. They are implemented as compiler analyses that look for partitions in the application data that can be associated with locks, and compiler transformations that generate code that automatically acquires and releases those locks around or in the atomic regions.

While all the approaches described next perform lock allocation for concurrency control, almost none of them investigate other divide-and-conquer–based optimizations such as the per-partition tuning in my work. Because deadlock must be avoided with automatic lock allocations, transactions often have to allocate locks early during a transaction (e. g., when the transaction is started), and need rather conservative compiler analyses to be safe. In exchange for these limitations, lock allocation can avoid STM overheads like the instrumentation of memory accesses in transactions.

Note that a partitioning-aware STM can synchronize in a similar way as with lock allocation by forcing partitions to always have the Exclusive Lock partition type (see Table 6.1 on page 141). The only difference is that this would be a dynamic locking scheme that uses deadlock prevention instead of deadlock avoidance and thus needs to be able to roll back transactions; static schemes could be implemented by (1) tuning the compiler's heuristics for when to create partitions and (2) making outermost transactions also aware of all the global partitions that are used in them. Nonetheless, the techniques proposed by the related work described next could often be used to optimize the analysis quality and heuristics of DSA and PoolAlloc towards the concurrency control use case.

In prior work, McCloskey et al. present a compile-time tool [81] that can transform transactions in a C program into critical sections iff the programmer provides source code annotations that declare which data is protected by which lock. However, the tool can only detect potential deadlocks but cannot avoid them automatically; programmers have to resolve such issues by, for example, providing locks at different levels of granularity regarding the data that is protected. Hicks et al. present a lock inference algorithm [59] based on points-to analysis for atomic regions in a simple source language. Locking annotations are not required and the infered locking scheme is guaranteed to be deadlock-free, but each atomic region always acquires a fixed set of locks when started. Emmi et al. formulate lock allocation [36] as an optimization problem and express it using integer linear programming (ILP): The compiler needs to find a data–to–lock mapping that minimizes the number of locks (i. e.,

to reduce lock acquisition runtime overheads) and minimizes the conflict costs between atomic regions (i. e., to decrease the runtime overheads due to mutually exclusive execution). Zhang et al. present a similar ILP formulation and also present a heuristic solution [124]. Halpert et al. investigate lock allocation at the granularity of whole atomic regions [48]: Instead of looking for a minimal lock allocation based on a data–to–lock mapping, their analysis starts with the conservative assumption that all atomic regions interfere with each other and then tries to refine this interference graph; the result is an assignment of a static or dynamic lock for each set of atomic regions that can interfere with each other.

In concurrent work, Cherem et al. present a lock inference analysis that relies on unification-based points-to analysis [13] (as DSA does). They combine this with a multi-granularity locking scheme that forms a tree structure, which matches the output of a unification-based analysis nicely: The top-level lock protects all memory locations, it's children each protect one of the points-to sets, and within each such set a limited number of locks provide protection of finer granularity. Their locking runtime library uses intention locking to achieve fine-granular locking despite the existance of coarse-grain locks in the hierarchy (see the Weikum and Vossen textbook [119] for details).

In subsequent work, Upadhyaya et al. propose to use knowledge about the semantics and internals of data structures—specifically, containers—to make alias analysis and lock allocation more precise [114]. This requires data structure programmers to provide additional information, for example whether it is safe to follow a locking scheme that is only based on the hashes of keys in associative containers such as hash maps. The description of the requirements for such specially supported containers is rather vague, but one assumption seems to be that they are already thread-safe. Upadhyaya et al. compare the performance of their compiler-generated locking to TL2 (see Section 5.4) on the STAMP benchmarks, but this evaluation is flawed. First, they use the original STAMP for TL2 but for the locking, they use a modified version of STAMP in which the red-black trees used in Vacation seem to have been replaced by hash maps; this changes the synchronization characteristics significantly. Second, they do not consider making the knowledge about the containers available to the STM or a TM compiler, even though it would enable conceptually similar optimizations as with lock allocation (e. g., as proposed by Herlihy and Koskinen [53]).

Sreeram and Pande describe a lock allocation scheme that uses DSA as points-to analysis to build must-alias and may-alias sets for all memory accesses in a transaction [111]. If a transaction has an empty may-alias set, then all its accesses are known and it can be run as an irrevocable transaction using the allocated locks; otherwise, the transaction will execute using an STM instead. However, the authors' description of the synchronization between such irrevocable and the default revocable transactions seems to be flawed; they state in Section VII.C of the paper that irrevocable transactions acquire commit locks when started, which would serialize all transactions that use the lock allocation.[9]

Mannarswamy et al. also propose to use lock allocation in combination with

[9] If irrevocable transactions would only acquire the commit lock during commit, only the commits would be serialized. However, in that case, irrevocable transactions could read inconsistent data, which would violate the atomicity requirements for transactions. This may be the behavior intended by the authors because they highlight in Section VII.C that transactional reads of irrecovable transactions must be non-faulting; nontheless, this would not be sufficient to guarantee correctness.

an STM [77], but they decide between both on the granularity of memory accesses and not whole transactions. In particular, to avoid false conflicts, they propose to let the compiler infer locks only for memory accesses for which it can perform a better-than-conservative analysis (e. g., when an access has an empty may-alias set). However, to ensure a consistent data–to–lock mapping for all transaction, the STM has to perform additional runtime lookups: For transactional accesses that may target a memory location with a compiler-allocated lock, the STM has to consult a table with the compiler-allocated lock mappings to determine whether it needs to use the compiler's lock or the TM metadata managed by the STM. In essence, this has thus many similarities with my earlier work, except that their work (1) moves the performance-relevant tuning decisions to compile-time (i. e., whether to use an allocated lock for a certain access) and (2) uses a different points-to analysis.

Colocating application data and TM metadata. The lock allocation tool built by Mannarswamy et al. can also colocate a compiler-allocated lock with a structure by adding it as a new field of the structure, provided all fields of this structure are protected by this lock. This is performed by a whole-program structure layout transformation in the compiler they use, and is conceptually very similar to my prior work.

In further subsequent work, Mannarswamy and Govindarajan focus specifically on improving the cache footprint and cache miss costs of STMs [78] and investigate three optimizations. The first, lock–data colocation, is like my prior work except that they add a heuristics-based profitability analysis that tries to take into account the overall cache footprint of a program. The performance gain that they report for Vacation ($< 2\%$) is much lower than what I observed with my implementation ($> 30\%$, see Section 6.3.2), but their results are based on TL2 and manually instrumented code and they do not report how many of the transactional accesses this optimization could be applied to. The second optimization removes redundant lock accesses by having the compiler allocate a single lock for several shared locations that are always accessed together in all transactions; thus, this is a special kind of lock allocation. The third optimization, using per-partition time bases instead of a global time base, is just another implementation option for time-based STMs that is enabled by making the STM partitioning-aware (see Section 6.2); the authors even reimplemented DSA and PoolAlloc in their compiler. I had considered this approach during my prior work but rejected it because of (1) the runtime overheads this results in (e. g., snapshot extensions are necessary whenever a new partition is accessed, transactions need to maintain several snapshot times, etc.) and (2) the possibility to use other scalable time bases such as synchronized clocks (see Section 5.3).

McRT-STM [98] supports TM metadata colocation for small objects by using a custom memory allocator that allocates small memory chunks in a special memory region. Chunks are grouped in memory blocks together with other chunks of the same size. The STM has to check at runtime whether a transactional access targets a memory region reserved for small objects. If so, it computes the base address of the object by loading the size of objects in this memory block from the block's header. This approach avoids the need for compile-time analysis, but results in higher runtime overheads.

The TM support presented by Adl-Tabatabai et al. for a managed runtime

environment [1] can choose between TM metadata colocation and a mapping to external metadata on a per-data-type granularity.

Other compile-time optimizations. Shpeisman et al. present a compile-time analysis for Java programs [105] that can detect memory locations that are never accessed in transactions, which is used to reduce the runtime overhead required to ensure strong isolation. The TM-aware compiler presented by Yang et al. can emit calls to specialized TM runtime library functions when several transactional accesses target the same memory location [83], which probably uses an intra-procedural analysis. More information about TM support in other compilers is provided in Sections 2.4 and 3.3.

Chapter 7

Exploiting Hardware Support for TM

Unfortunately, current STMs, including the implementations described previously, still have a relatively large performance overhead compared to the execution of sequential code. While there is likely room left for further software-only optimizations, hardware support for TM can decrease these overheads substantially.

My focus is on first-generation hardware support that has been integrated into real products or has been proposed by industry for inclusion in high-volume microprocessors. Different to many other HTM proposals, these HTMs are simple designs that provide best-effort HTM in the sense that only a subset of all reasonable transactions are expected to be supported by hardware. They have several limitations (e. g., the number of cache lines that can be accessed in a transaction can be as low as four) and thus have to be complemented with software fallback solutions. In turn, they are easier to implement in hardware and to integrate into current microprocessors, which makes them more likely to be available as TM building blocks in the near future.

My work uses *AMD's Advanced Synchronization Facility* (ASF) [2] as the base TM hardware support. It is a proposal for shared-memory synchronization extensions to the AMD64 architecture. ASF is a good basis because it is designed to be practical and have reasonable costs when implemented in hardware, is publicly available in the form of a near-cycle-accurate simulator, and provides a more useful set of features than other HTMs. In Section 7.1, I will provide more information about first-generation HTMs and about ASF in particular.

First-generation best-effort HTMs cannot execute every possible sequential code as a transaction. First, even if a transaction uses the HTM for concurrency control (called a *hardware transaction* in what follows), it typically needs additional instrumentation of the code or additional software runtime support. For example, custom hardware instructions might have to be used for transactional memory accesses, transaction demarcation instructions have to be added to the code, or parts of a transaction's rollback might have to be implemented in software (e. g., if the HTM does not restore all CPU registers on abort).

Second, some transactions will not be able to use the HTM for concurrency control and will instead use an STM implementation (*software transactions*)

because, for example, the HTM does not offer enough capacity for all the transactional loads and stores in those transactions. A simple fallback strategy is to execute software transactions serially (i. e., not concurrent with any other hardware or software transaction) and hardware transactions concurrently.

This integration is the first part of exploiting hardware support for TM and will be covered in Section 7.2. I investigated this and built the required runtime support in the context of a study of ASF's suitability for TM [14]. Overall, ASF provides useful hardware support for TM and aligns well with other TM building blocks. However, some of the design decisions for ASF make it harder to use in a TM context than what seems necessary. Many of the findings discussed in this section will also apply to other proposed hardware mechanisms that share some features with ASF (e. g., execution of nonspeculative code in Alert-On-Update [110]).

However, using serialized execution as fallback limits performance when software transactions are not infrequent because the whole TM then frequently switches to serial execution without any parallelism. It is therefore desirable to build a *hybrid TM* (HyTM), in which multiple hardware and software transactions can run concurrently.[1]

Thus, as the second part of exploiting TM hardware support, I will present novel HyTM algorithms [96, 95] in Section 7.3. They are based on ASF and combine with either LSA for C/C++ environments (Section 5.2) or NOrec [22] on the STM side. Most previous HyTM proposals have assumed HTMs in which every memory access inside a transaction is speculative (i. e., transactional), whereas ASF supports nonspeculative memory accesses (including nonspeculative atomic instructions). This allows for the construction of efficient HyTM algorithms that improve on previous HyTMs. In particular, they decrease the runtime overhead, abort rates, and HTM capacity requirements of hardware transactions, while at the same time allowing hardware and software transactions to run and commit concurrently. LSA and NOrec are optimized for different workloads (i. e., a higher level of concurrency vs. lower single-thread overheads), so the HyTM algorithms complement each other.

I will evaluate the performance of the HyTMs in Section 7.4 on simulations of several reasonable implementations of ASF that differ notably in their capacity limits (i. e., how they keep track of transactional loads and stores and how many such accesses they can track per transaction). AMD implemented the ASF extensions in their simulator using ASF cycle costs and pipeline interactions that they would expect from a real hardware implementation. Together with the realistic implementation of the building blocks in the rest of the TM stack, this yields a reasonably accurate outlook on the TM-performance benefits of having ASF implemented in real microprocessors.

7.1 First-Generation TM Hardware Support

Current commercial microprocessors are very complex and costly to design and verify. This typically results in incremental evolution instead of large changes to a processor's design between consecutive generations. This constraint applies

[1] There is also the option of running either software or hardware transactions concurrently (like in PhasedTM [74], see Section 7.3.3) but this will force all concurrent transactions into software transactions even if just one of them cannot be run as a hardware transaction.

especially to new features that still need to show that they present a large-enough benefit to customers.

HTM is such a new feature. Many academic hardware-extension proposals mandated modifying critical components such as the cache-coherence protocol, which would be too costly in the context of a commercial microprocessor and first-generation support for a feature. For example, the first HTM design, published by Herlihy and Moss [55], proposed to use a separate transactional data cache that is accessed in parallel to the conventional data cache. This would require many intrusive modifications to a current typical microprocessor (e. g., to the main load–store path) and require space for the additional cache, making such an implementation impractical to add. Similary, Shriraman et al. [106] propose a mechanism for STM acceleration that would extend the standard MESI cache-coherence protocol to twice the number of states and transitions.

Instead, HTM proposals by industry such as AMD's Advanced Synchronization Facility, Intel's TSX [64], or the TM support in Sun Microsystems' Rock processor [26], are *best-effort* designs that provide a limited feature but at an affordable implementation cost. Typical limitations are a bounded capacity for transactional loads and stores, or disallowing certain instructions in a transaction. Their implementations try to leverage existing components and features (e. g., Rock's TM reuses support for single-thread speculative execution, and ASF does not change the cache-coherence protocol). These HTMs are also unobtrusive with respect to other parts of the system stack (e. g., they do not require custom OS support and can be used from kernel and userspace contexts).

To be useful despite the limitations, each of these hardware approaches relies on accompanying software to provide a complete TM support with features like large transaction sizes, less restrictions on transactional code, or advanced contention management strategies. Likewise, transaction virtualization is considered to be best handled in software (e. g., in contrast to the virtualized transactional memory proposed by Rajwar et al. [87], which increases the complexity of the hardware implementation).

HASTM's hardware extensions can be used to accelerate STM concurreny control for transactional reads by providing a mechanism to monitor read sets in hardware. Writes are not accelerated and must rely on pure software implementations. It allows for a reasonable, low-cost hardware implementation. I will discuss its HyTM algorithms in more detail in Section 7.3.3.

The Rock processor, which was actually implemented in hardware but was never publicly available, contained an HTM that showed encouraging performance results but also had limitations that made the HTM difficult to employ effectively. For example, hardware transactions abort on TLB misses and often on certain code sequences commonly used for function calls. The former is cumbersome to deal with because some TLB misses will have been resolved after a restart, whereas others will have to be replayed in software by the TM runtime (e. g., using dummy accesses to the respective page) to be resolved. This limits the applicability of the HTM to simpler transactions that might have to be heavily tweaked to be able to use the HTM regularly. These use cases definitely exist (e. g., in custom concurrent code in an OS kernel, or in a garbage collector), but the hardware support is less useful for general-purpose transactions.

Azul Systems has developed multicore processors that contain an HTM [16] which seems to be primarily used for lock elision [86] in Java. I will discuss lock

elision in Section 7.3.3.

7.1.1 Advanced Synchronization Facility (ASF)

AMD's Advanced Synchronization Facility is a proposal [2] of HTM extensions for x86-64 CPUs. It essentially provides hardware support for the speculative execution of regions of code. These *speculative regions* are similar to transactions in that they take effect atomically and can access memory transactionally using speculative loads and stores. ASF provides *selective annotation*, which means that nonspeculative memory accesses are supported within transactions (including nonspeculative atomic instructions) and speculative memory accesses have to be explicitly marked as such.

ASF is a best-effort design that would be feasible to implement in high-volume microprocessors. It is more advanced than the designs described previously. For example, unlike with Rock's HTM, TLB misses do not abort transactions. It comes with a number of limitations, of course. The number of disjoint locations that can be accessed in a speculative region is limited—depending on ASF's implementation variant—either by the size of speculation buffers (which are expensive and thus have been designed with small capacity) or by the size and associativity of caches (when tracking speculative state in caches). It follows that speculative accesses and concurrency control have cache line granularity. ASF transactions are not virtualized and therefore, abort on events such as context switches or page faults. However, page faults triggered in speculative regions will be visible to the operating system after the abort, so a custom replay of the fault by the TM runtime is not necessary to be able to retry the hardware transaction.

The original aim behind ASF was to make concurrent nonblocking programming easier and faster by providing an atomic read–modify–write operation for more than a single memory location. To that end, ASF ensures eventual forward-progress in the absence of contention and exceptions if a speculative region does not access more than four distinct cache lines.[2] This guarantee prevents programmers from having to always provide a software fallback path that does not use ASF even if the speculative region is small. Note that in a general-purpose TM, we need to always provide the software fallback because we cannot make assumptions about the transactions.

In what follows, I will summarize ASF's properties [14]. More information can be found there, in ASF's specification [2], and in two papers about its internals and the background of the design [31, 15].

ISA extensions. The new instructions that ASF provides allow for entering and leaving speculative regions (SPECULATE, COMMIT, and ABORT) and accessing protected memory locations (i. e., memory locations that can be read and written speculatively and which abort the speculative region if accessed concurrently by another thread: LOCK MOV, WATCHR, WATCHW, and RELEASE). All of these instructions are available in all system modes (user, kernel; virtual-machine guest, host).

[2] *Eventual* means that there may be transient conditions that lead to spurious aborts, but eventually the speculative region will succeed when retried continuously. The expectation is that spurious aborts almost never occur and speculative regions succeed the first time in the vast majority of cases.

```
1  // DCAS Operation:
2  // if ((mem1 = rax) && (mem2 = rbx)) {
3  //   mem1 = rdi; mem2 = rsi;
4  //   rcx = 0;
5  // } else {
6  //   rax = mem1; rbx = mem2;
7  //   rcx = 1;
8  // }
9  DCAS:
10   mov     %rax, %r8
11   mov     %rbx, %r9
12 retry:
13   SPECULATE       // SR begins
14   jnz     retry   // Restart SR after aborts
15   mov     $1, %rcx  // Default result, overwritten on success
16   lock mov (mem1), %r10
17   lock mov (mem2), %rbx
18   cmp     %r8, %r10
19   jnz     cmpfail
20   cmp     %r9, %rbx
21   jnz     cmpfail
22   lock mov %rdi, (mem1)
23   lock mov %rsi, (mem2)
24   xor     %rcx, %rcx // Success indication
25 cmpfail:
26   COMMIT
27   mov     %r10, %rax
```

Figure 7.1: An example implementation [14] of a DCAS operation using ASF.

Speculative regions are started using the SPECULATE instruction. When a speculative region is aborted, execution resumes at the instruction following the SPECULATE instruction (with a matching error code in the rAX register, which allows clients to handle aborts in a custom way). COMMIT and ABORT both finish the execution of a speculative region: COMMIT makes all speculative modifications instantly visible to all other CPUs, whereas ABORT discards these modifications. Flat nesting is used for nested speculative regions.

In a speculative region, speculative/protected memory accesses can be expressed in the form of ASF-specific LOCK MOV CPU instructions, and can be mixed with ordinary nonspeculative/unprotected accesses (MOV). This selective annotation allows the TM or the programmer to use speculative accesses sparingly and thus preserve precious ASF capacity. Second, the availability of nonspeculative atomic instructions allows us to use common concurrent programming techniques during a transaction, which enables novel HyTMs (see Section 7.3) and can reduce the number of transaction aborts due to benign contention (e. g., when updating a TM-internal, shared counter). In a speculative region, nonspeculative loads are allowed to read state that is speculatively updated in the same speculative region, but nonspeculative stores must not overlap with previous speculative accesses.

ASF also provides CPU instructions for just monitoring a cache line for concurrent stores (LOCK PREFETCH) or loads and stores (LOCK PREFETCHW), and for stopping monitoring a cache line (RELEASE).

Figure 7.1 shows a simplified example of a double CAS (DCAS) operation implemented using ASF.

Speculative region aborts. As explained by Christie et al. [14], there are several conditions that can lead to the abort of a speculative region, besides

CPU A		CPU B cache line state	
Mode	**Operation**	**Protected Shared**	**Protected Owned**
Speculative region	`LOCK MOV` (load)	OK	*B aborts*
Speculative region	`LOCK MOV` (store)	*B aborts*	*B aborts*
Speculative region	`LOCK PREFETCH`	OK	*B aborts*
Speculative region	`LOCK PREFETCHW`	*B aborts*	*B aborts*
Speculative region	`COMMIT`	OK	OK
Any	Read operation	OK	*B aborts*
Any	Write operation	*B aborts*	*B aborts*
Any	Prefetch operation	OK	*B aborts*
Any	`PREFETCHW`	*B aborts*	*B aborts*

Table 7.1: Conflict matrix for ASF operations (from [2], §6.2.1).

the ABORT instruction: (1) contention for protected memory, (2) system calls, exceptions, and interrupts, (3) the use of certain disallowed instructions, and (4) implementation-specific transient conditions. In case of an abort, all modifications to protected memory locations are undone, and execution flow is rolled back to the beginning of the speculative region by resetting the instruction and stack pointers to the values they had directly after the `SPECULATE` instruction. No other register is rolled back; software is responsible for saving and restoring any context that is needed in the abort handler (see Section 7.2). Additionally, the reason for the abort is passed in the rAX register. Because all privilege-level switches (including interrupts) abort speculative regions and no ASF state is preserved across such a context switch, all system components (user programs, OS kernel, hypervisor) can make use of ASF without interfering with one another.

Conflict detection for speculative accesses is handled at the granularity of a cache line. Conflict resolution in ASF follows the "requester wins" policy (i. e., existing speculative regions will be aborted by incoming conflicting memory accesses) with cache line granularity. Table 7.1 summarizes how ASF handles contention when CPU A performs an operation while CPU B is in a speculative region with the cache line protected by ASF. These conflict resolution rules are important for understanding how the HyTM algorithms presented in Section 7.3 work.

Isolation and ordering guarantees. The isolation and ordering guarantees that ASF provides for mixed speculative and nonspeculative accesses are important for the correctness of the HyTM algorithms because they access shared data nonspeculatively. Also, a speculative region can trigger externally visible side effects such as page faults. It is important to know whether these effects were caused by misspeculation (i. e., were caused by a memory access that would cause an abort) or by a consistent (yet potentially incomplete) speculative region or transaction. The guarantees described next complement the rules layed out in Table 7.1. They are not yet part of the ASF specification but reflect the intended design [29].

Aborts of a speculative region are designed to be instantaneous with respect to the program order of instructions in a speculative region. For example, aborts are supposed to happen before externally visible effects such as page faults or

$$Store_{spec}(A) \rightarrow_{hb} Store_{nonspec}(B) \rightarrow_{hb} Commit$$
$$\Rightarrow Monitor(A) \rightarrow_{hb} Retire(A) \rightarrow_{hb} Retire(B) \rightarrow_{hb} Visible(B)$$

$$Load_{spec}(A) \rightarrow_{hb} Load(B)$$
$$\Rightarrow Monitor(A) \rightarrow_{hb} DataBind(A) \rightarrow_{hb} DataBind(B)$$

Figure 7.2: Ordering guarantees provided by ASF. $\rightarrow_{hb}$ expresses happens-before relationships conceptually similar to happens-before in the C++11 memory model (see Section 4.1). A and B are memory locations.

non-speculative stores appear. This behavior also illustrates why speculative accesses can also be referred to as "protected" accesses. A consequence of this is that speculatively accessed cache lines are monitored early for conflicting accesses (i. e., once the respective instructions are issued in the CPU, which is always before they retire). Together with the standard memory model of x86 architectures, this leads to two rules that are relevant for the HyTM algorithms. Figure 7.2 shows the order of CPU effects that is implied by certain program or execution orders. The first rule essentially states that if a speculative store to A happens before a nonspeculative store, then A's cacheline will be monitored before the nonspeculative store is visible to other threads. Similarly, the second rule states that if a speculative load happens before a nonspeculative or speculative load, then the monitoring of the former will happen before the second load actually retrieves a value from memory.

Furthermore, atomic instructions such as CAS or an atomic fetch-and-increment retain their ordering guarantees. For example, a CAS ordered before a COMMIT in a program will become visible before the transaction's commit, and a CAS will be a full memory barrier with respect to memory accesses and monitoring.

ASF implementation variants. ASF could be implemented in different ways in hardware. One major implementation choice that affects ASF's clients is how uncommitted speculative reads and writes are tracked.

First, one can introduce a new CPU data structure called the *locked-line buffer* (LLB), which holds the addresses of protected memory locations accessed in the current speculative region and is fully associative. It also holds the prior values of speculatively modified memory lines. Finally, it monitors remote memory requests and aborts a current speculative regions on probe requests that represent conflicting memory accesses by other CPUs.

Second, the L1 cache of each CPU core can be extended with an additional speculative–read bit per cache line and the regular cache-coherence protocol can be used to monitor protected reads and abort a current speculative region if required. Similarly, the L1 cache could be extended with another bit for speculative stores.

One can also combine these options, using the LLB only for speculative stores and tracking speculative reads in the L1 cache. From the perspective of ASF clients, the trade-off is mostly in terms of *capacity* for speculative state

Name	State stored in	HTM capacity
LLB8	8-line LLB	8 distinct lines (loads and stores)
LLB256	256-line LLB	256 distinct lines (loads and stores)
LLB8L1	Stores: 8-line LLB Loads: 1K-line L1	Stores: 8 distinct lines Loads: Minimum of 1K lines or 2-way set associativity, shared with nonspeculative accesses

Table 7.2: ASF implementation variants.

(i. e., how many distinct memory lines can be accessed by a speculative region before it will exceed the capacity and will have to abort). The L1 cache is relatively large but its effective capacity can be limited by its associativity, and nonspeculative accesses will potentially compete with speculative accesses for cache space. The LLB does not suffer from these problems but will likely be smaller in size because fully associative structures are quite costly.

Table 7.2 shows the implementations that I will consider in the evaluation. LLB8 represents a minimal implementation that can only be used for small transactions. LLB256 has a large LLB whose capacity is unlikely to be exceeded based on current TM benchmarks, but thus is also costly to implement. LLB8L1 is a middle ground offering a capacity that is often sufficient.

ASF simulator. ASF is not yet implemented in hardware, so one has to rely on simulation to evaluate it. AMD has extended PTLsim [122] with support for ASF and a more detailed model of the interactions between multiple separated processor cores and memory hierarchies (PTLsim-ASF [30]). PTLsim can simulate a full AMD64 system, which is important to be able to evaluate a realistic TM stack with applications, libraries, and an operating system kernel that would also run on a real machine.

AMD configured [14] the simulator to match the general characteristics of a system based on AMD Opteron processors (family 10h), with a three-wide clustered core, out-of-order instruction issuing, and instruction latencies modeled after the AMD Opteron microprocessor. The cache and memory configuration is:

- L1D: 64 KB, virtually indexed, 2-way set associative, 3 cycles load-to-use latency.
- L2: 512 KB, physically indexed, 16-way set associative, 15 cycles load-to-use latency.
- L3: 2 MB, physically indexed, 16-way set associative, 50 cycles load-to-use latency.
- RAM: 210 cycles load-to-use latency.
- D-TLB: 48 L1 entries, fully associative; 512 L2 entries, 4-way set associative.

The simulated machine used for the evaluation has 16 CPU cores, each having a clock speed of 2.2 GHz. PTLsim-ASF does not yet model topology

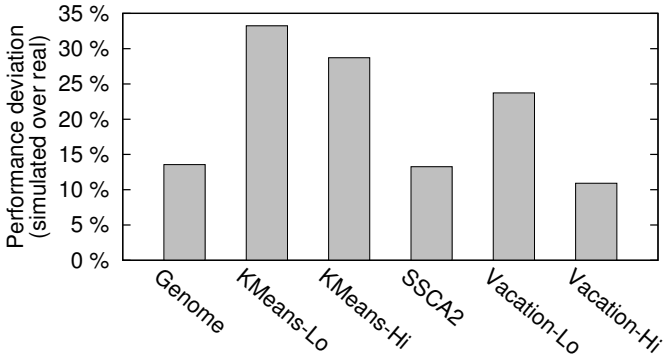

Figure 7.3: PTLSim accuracy for the execution time of the STAMP benchmarks (no TM, no ASF, one thread) for simulated execution time normalized to native execution time.

information such as placement of cores on chips or sockets, and thus also does not model limited cross-socket bandwidths. Therefore, these cores behave as if they were located on the same socket, resembling future processors with higher levels of core integration. The cache-coherence model is simplified but captures first-order effects caused by cache coherence [30]. Additional ordering constraints and fencing semantics for the ASF primitives are modeled as well. However, the version of the simulator that was available did not yet correctly model the ordering guarantees between speculative and nonspeculative loads, so the implementations of HyTMs affected by this have to use additional read–read memory barriers.

Simulator accuracy. To illustrate the accuracy of the simulator, Figure 7.3 shows the difference in execution times on a real machine[3] compared to a simulated execution within PTLsim-ASF[4]. A close match between the performance of simulated and real executions is desirable because this increases the confidence in the results of the evaluation. All experiments used for the evaluation (Section 7.4), including baseline STM runs, have been conducted inside the simulator to make sure that the results are comparable.

For many of the STAMP benchmarks (see Section 3.4.3), PTLsim-ASF stays within 10–15% of native execution speed, which is in line with earlier results for smaller benchmarks [30]. However, the results for Vacation-Lo and KMeans show that not all mechanisms in the microarchitecture are simulated precisely by PTLsim-ASF.

[3]AMD Opteron processor family 10h, 2.2 GHz.

[4]These results are from an experiment on an earlier version of PTLsim-ASF, which simulated an 8-core machine.

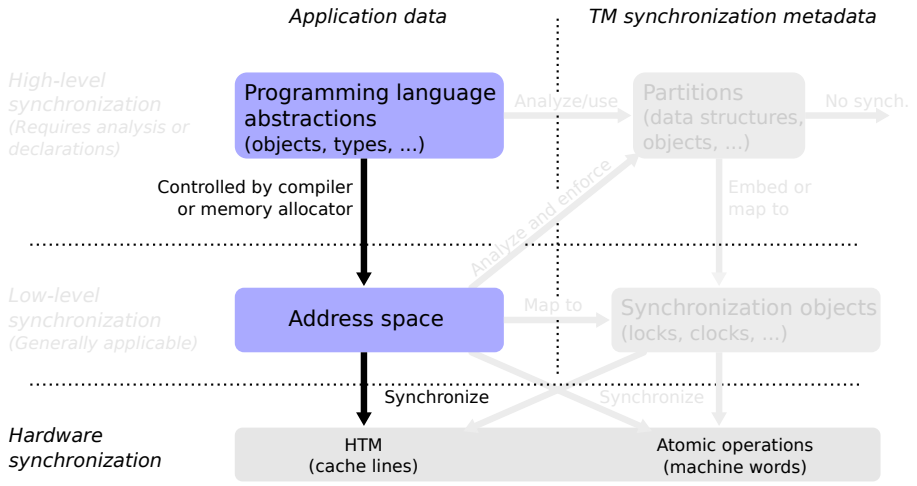

Figure 7.4: TM-based synchronization: HTM with serial execution of transactions as fallback mode. Note that the fallback is implemented using standard atomic operations, which has been omitted to increase clarity.

7.2 An ASF-Based TM Runtime Library

First-generation HTMs such as ASF are not yet widely available in hardware and can only be added to new hardware designs but not to existing machines. To give an incentive to the hardware manufacturers to include HTM support in new designs, the transition from STM to HTM should be easy for future users.

The best way to achieve this would be to encapsulate the HTM support in a TM runtime library that provides the same ABI as STM-only implementations. This allows to keep other TM building blocks generic (e. g., the compiler). If the TM runtime library is linked dynamically to an application, these applications can use STM or HTM without recompilation.

I have built *ASF-TM*, which is a TM runtime library that can use ASF to execute transactions. It provides the ABI described in Section 4.2, with a few minor differences. In what follows, I will first describe an HTM-like implementation that falls back to serial execution for each transaction that cannot be executed in hardware (see Figure 7.4). Hybrid TM algorithms can be implemented in the similar way, and I will describe those in Section 7.3.

Building ASF-TM showed that ASF's design has only a few issues that make ASF hard to support in a TM library, and that features such as selective annotation are indeed useful from a TM perspective. The findings also apply to other HTM designs and present a software-side validation of HTM design decisions.

7.2.1 Implementation Overview

ASF-TM's implementation can be roughly split into the groups of functionality required by the ABI: (1) starting and committing transactions, (2) data transfers that load from or store to shared data, (3) other functions such as queries or

infrequently used functions, and (4) executing nontransactional code or custom wrappers.

The first two will be discussed in Sections 7.2.2 and 7.2.3. The third category is straight-forward to handle, either because these functions do not modify any state and only read transaction-private or immutable state (e. g., getting the current transaction's ID) or because ASF-TM can fall back to serial execution for functions that are only infrequently called.

Nontransactional code (e. g., accessing nonshared data on the stack or TM-internal functions) and code with custom instrumentation (e. g., functions declared with transaction_pure or tm_wrapper attributes) are more difficult to handle. First, ASF disallows some combinations of speculative and nonspeculative accesses to data. While the compiler can consistently separate nonshared and potentially shared data, it does so on the granularity of individual bytes and not on the cache line granularity that ASF accesses have. The problems resulting from this potential false sharing are discussed in Section 7.2.3.

Second, nontransactional code can contain CPU instructions such as RDTSC that are disallowed in ASF speculative regions ([2], §6.3) but allowed in sequential code or if executing the code with a TM runtime library other than ASF-TM. Note that even though such instructions might not seem to be very useful for application code in transactions, they can be reasonable code or even required in cases like transaction_pure functions. ASF makes this difficult to handle for the software side because it will not only abort the speculative region but also raise a general protection fault that is visible to the operating system and signal handlers installed by the application. ASF-TM cannot reliably and efficiently prevent these side effects.[5] To not add additional dependencies on operating-system support and to ease the transition from STM to HTM, ASF should not raise general protection faults in speculative regions unless these would be raised for sequentially executed code as well. Instead, it should only abort the transaction and perhaps signal the execution of disallowed code using a special abort reason code ([2], §6.1.1). In fact, PTLsim-ASF implements this already and only aborts the speculative regions in most cases.

Third, ASF speculative regions can be aborted at any time without consideration of what kind of code is currently executed in the speculative region. Thus, there can be asynchronous aborts when executing nontransactional code, which is difficult to handle and discussed in Section 7.2.4.

There are two generic features in my TM stack that are particularly useful to increase ASF-TM's performance. First, the compiler can generate several different code paths for each transaction (see Section 3.4.1). Each code path will use different sets of data transfer functions (or no instrumentation at all). This allows ASF-TM to provide distinct functions for execution of hardware and software transactions, which avoids having to switch between both dynamically at runtime during each transactional memory access. In what follows, functions for hardware transactions will be prefixed with htm− and those for software transactions with stm−.

[5] For example, ASF-TM has no control over how the exceptions are handled in the kernel internally and whether, for example, they would appear in statistics visible to users. Application-side signal handlers can be set but then ASF-TM has to ensure that its handler is never unset or overwritten by the application. Furthermore, switching to ASF-TM's software-only fallback mode would take longer because the operating system would be involved in delivering the signal.

```
1   leaq    8(%rsp), %rax // calculate SP of calling function
2   movq    (%rsp), %r8   // get saved IP of calling function
3   subq    $72, %rsp     // allocate space on stack
4   movq    %rax, (%rsp) // save the registers on the stack
5   // ... save r8 (caller IP), rbx, rbp, r12-r15 on stack here ...
6
7   movq    %rsp, %rsi    // provide address of saved registers as 2nd argument
8   call    ASFTMBegin
9   // Use ASF unless we are about to run uninstrumented code or the
10  // code path ID is even.
11  test    $0x22, %eax
12  jnz 1f
13  movq    %rax, %rdi
14  SPECULATE
15  jnz     .Labort_ITM_beginTransaction
16  call    ASFTMPostSpeculate // must return first argument(%rdi) in %rax
17 1:
18  addq    $72, %rsp
19  ret
20 .Labort_ITM_beginTransaction:
21  // Supply ASF abort reason code as first argument
22  movq    %rax, %rdi
23  call    ASFTMRestart
24  test    $0x22, %eax
25  jnz 1f
26  movq    %rax, %rdi
27  SPECULATE
28  jnz     .Labort_ITM_beginTransaction
29 1:
30  jmp     ASFTMRestartLongjmp
```

Figure 7.5: ASF-TM's code to start a transaction.

Second, it is possible to use link-time optimization when statically linking ASF-TM to an application with transactions. Together with a separate hardware transaction code path, this allows for very efficient code when speculative accesses get inlined (see Section 7.2.3 for an example). Note that ASF-TM still provides the same ABI, so this can be used on demand and does not prevent the portability offered by dynamic linking of the TM runtime library.

7.2.2 Begin and Commit

Starting a transaction in ASF-TM requires more care than in an STM and has several implications for the implementation of other TM functions. Therefore, I will discuss this in more detail. The ABI requires the TM to provide a single function to start a transaction (_ITM_beginTransaction), which will return more than once when transactions are restarted. The SPECULATE instruction of ASF provides similar functionality but we cannot rely on the compiler to insert a SPECULATE instruction into the application code because this is not possible with the current ABI. Furthermore, the TM needs to execute additional code (e. g., synchronization code related to serial–irrevocable mode) before starting a transaction. This code might not be able to run in a speculative region, so it needs to be executed before the speculative region is started with SPECULATE. Also, keeping as much code as possible out of the speculative region eases the implementation because code executed from within a speculative region needs to be robust to asynchronous aborts (see Section 7.2.4).

Figure 7.5 shows the implementation of _ITM_beginTransaction in ASF-TM. Its first part up to line 5 is setjmp-like functionality and saves callee-saved regis-

ters on the stack. This is necessary because ASF only restores the stack pointer when restarting a speculative region.

Next, we call ASFTMBegin on line 8, supplying as arguments (1) the first argument of _ITM_beginTransaction and (2) the address of the the saved registers on the stack.[6] This function then has to (1) copy the registers on the stack to a per-transaction buffer so that they can later be restored when restarting the transaction, (2) decide between executing as a hardware or as a software transaction or in serial–irrevocable mode, and (3) start a software transaction or preprare for the execution of a hardware transaction, respectively.

ASF only supports flat nesting (i. e., transactions are always completely rolled back and the outermost transaction is restarted). Therefore, ASFTMBegin basically has to just increment a nesting counter in the case of nested transactions. In case of outermost transactions, ASFTMBegin will be executed before we start a speculative region, so it does not have to deal with asynchronous aborts.

ASFTMBegin must return the value that would otherwise be returned from _ITM_beginTransaction. Unless the former decided to run uninstrumented code or software-only code paths (line 11), we start the speculative region by executing the SPECULATE instruction (line 14) and then run TM code that has to be executed from within the speculative region (ASFTMPostSpeculate, see Section 7.3.2 for examples). Finally, we return to the transaction (line 19), which will execute the proper code path.

If the speculative region aborts, ASF will roll back all speculatively modified cache lines and resume execution at line 15. However, the TM still has to run the software-only implementation parts of the transaction abort, so we detect this and branch to the restart handling starting at line 22. We call ASFTMRestart, which finishes the software-side abort of the current transaction (e. g., rolls back memory allocations) and calls ASFTMBegin to start a new transaction. Because of flat nesting, it is not executed inside a speculative region and thus does not need to be robust to asynchronous aborts. Next, unless we have to restart as a software transaction (line 11), we execute a second SPECULATE instruction. Finally, because this is a restarted transaction, we jump to a function (line 30) that will (1) call ASFTMPostSpeculate if in a hardware transaction and (2) use the saved registers to return directly from the application's call to _ITM_beginTransaction, similar to what longjmp does.[7]

ASF-TM uses a simple policy to decide whether to execute hardware or software transactions. Each invoked transaction is first executed as a hardware transaction. On hardware-transaction aborts due to far calls, disallowed operations, exceeded ASF capacity[8], or serial–irrevocable mode requests, ASF-TM will restart the transaction as a software transaction. When software transactions abort, they will never be switched to hardware transactions. On aborts caused by contention between speculative regions and other accesses, ASF-TM uses a simple back-off strategy and only switches to a software transaction if the transaction encountered a large number[9] of aborts caused by contention.

[6] _ITM_beginTransaction is declared to take a variable number of arguments, but ASF-TM currently considers only the first argument.

[7] To work correctly, _ITM_beginTransaction must not be inlined into the calling function.

[8] If only serial–irrevocable mode is available as software fallback, only the second capacity-related abort will cause a switch to a software transaction

[9] This number is arbitrarily set to 100.

```
1 Load64: // arguments: address in %rsi
2   lock mov (%rsi), %rax
3   retq
4
5 Store64: // arguments: address in %rsi, value in %rdx
6   lock mov %rdx, (%rsi)
7   retq
```

Figure 7.6: 64-bit load and store functions in an ASF-based HTM.

```
1   lock mov (%r15),%r15 // load address of first node from list head
2 loop:
3   mov      %r15,%rax
4   lock mov 0x8(%rax),%r15 // load address of next node
5   lock mov (%r15),%rcx // load node value
6   cmp      %r14,%rcx
7   jl       loop
```

Figure 7.7: A transactional traversal of a linked list using an ASF-based HTM and link-time optimization.

This policy guides the execution of a single transaction including all its restarts, but does not try to take the results of the execution of other transactions into account. A proper tuning of this policy at runtime could increase performance.

Commiting a hardware transaction is straight-forward because COMMIT can be called anywhere in the call stack. A COMMIT that causes the transaction to abort is not different to an asynchronous abort, which can also happen anytime (see Section 7.2.4 for further discussion of asynchronous aborts).

7.2.3 Loads and Stores

For brevity, let us consider a simple HTM-like implementation of hardware transactions. The compiler will transform all accesses to shared memory in a transaction into calls to the data transfer functions as specified in the ABI (i. e., loads, stores, and copying, moving or setting the values of blocks of memory). ASF uses selective annotation, so we can use LOCK MOV instructions to instruct ASF to make only these accesses transactional. Other accesses do not have to be instrumented, thus we only need LOCK MOV instructions in the implementations of the ABI's transactional data transfer functions.

Figure 7.6 shows 64-bit load and store functions as an example.[10] Static linking of ASF-TM and link-time optimization can reduce the overhead further by inlining the LOCK MOV instructions into the application code (see Section 3.4.1 for details). As an example, Figure 7.7 shows the code generated by the compiler for a transactional traversal of a linked list. The code for the sequential version of this traversal is essentially the same but without the LOCK prefixes.

Even simple HTMs such as the one I am considering here have to provide the transaction guarantees outlined in Section 4.2. ASF obviously handles concurrency control between transactions but ASF-TM has to also ensure publication and privatization safety, which depend on the interaction of nontransactional

[10] The functions in the example take the address of the transaction descriptor as first argument but we do not need to access it in an HTM-like algorithm.

memory accesses and publishing or privatizing transactions.

Privatization safety is implicitely guaranteed by ASF because hardware transactions will abort instantaneously on conflicting accesses to the cache lines that they have accessed (see Section 7.1.1). Informally, the snapshot of a active hardware transaction is always up-to-date. This also means that transactions will never operate on inconsistent data (e. g., they never observe only parts of the updates committed by another concurrent transaction). Furthermore, COMMIT instructions are full memory barriers, so later nonspeculative accesses to privatized data are never ordered before the commit.

Publication safety is a bit more involved because it must be ensured by both the compiler and the TM runtime library. Informally, it requires that transactional loads and stores are not reordered before other transactional loads that appear earlier in program order.[11] Speculative loads are ordered in program order by ASF in terms of both monitoring the respective cache lines and retrieving the data (see Figure 7.2). Speculative stores will not become visible before the speculative region is committed and thus earlier loads will always have started monitoring and loading the data before that. Thus, transactions will always observe the nonspeculative updates that happened before the publishing transaction signaled the availability of these updates. Similar to privatization safety, a publisher's COMMIT instructions represents a full memory barrier. However, the compiler also must take part in ensuring publication safety by not reordering transaction loads or stores before other, earlier loads (see Section 4.2.3).

The final issues that we have to consider is false sharing between speculative and nonspeculative accesses. The compiler has a consistent view of which memory locations are potentially shared with other threads or not. It also provides this view to the TM by using TM load and store functions iff the accessed location is shared. However, this information is at the granularity of bytes, whereas speculative accesses in ASF always operate on full cache lines, leading to a potential false sharing between speculative and nonspeculative accesses from ASF's perspective.

ASF can handle some combinations of such false sharing but will raise a general protection fault if a nonspeculative store targets a cache line that has been accesses speculatively before. This is difficult to avoid by the compiler and TM because this kind of sharing can happen in several situations that can be caused by not just the TM building blocks but also by other parts of the tool chain or the application (e. g., by the linker).

For example, consider a thread's stack. Variables on the stack are typically thread-private but can be shared as well. ASF's selective annotation allows the compiler to avoid wasting ASF's capacity for accesses to nonshared parts of the stack. But if there are shared variables on the stack frames of functions that execute transactions, then these transactions are prone to triggering general protection faults. To avoid this, the compiler's code generator would have to ensure that these potentially shared variables are on a separate cache line than all other stack slots modified in transactions. This seperation would also have to hold with respect to stack frames of calling functions and other future calls' stack frames that would end up on the same cache lines. Using a shadow stack for all on-stack allocations of potentially shared data, starting in the function that

[11] This also applies to externally visible side effects caused by these accesses (e. g., page faults).

starts the outermost transaction, could be a reasonable approach to implement this.

Another example are global variables. Putting every global variable on separate cache lines is not practical because it could bloat applications' memory requirements. In turn, it can be hard to let the compiler infer which global variables are accessed or not accessed in transactions, because this requires whole-program points-to analysis.[12]

In summary, it is hard for the TM compiler and runtime library to always avoid the false sharing between speculative and nonspeculative accesses. Always falling back to using software transactions when such a problem could potentially occur will likely require a very conservative decision, wasting much of the performance benefit that ASF and selective annotation offer.

Note that the problem is not that ASF cannot deal with the false sharing but instead that it raises the general protection fault. If ASF would just abort the speculative region and signal the false sharing with a special abort reason code, the TM could easily fall back to a software transaction. Advanced compiler analysis and trying to avoid the false sharing would still be possible.

Thus, judging from a software perspective, ASF should abort speculative regions when they encounter unsupported false sharing instead of raising a general protection fault. Note that this is similar to the case of disallowed instructions discussed in Section 7.2.1.

7.2.4 Dealing with Asynchronous Aborts

ASF can abort speculative regions asynchronously at any time during their execution. This can make it difficult to execute uninstrumented code that modifies state nonspeculatively. We can roughly categorize this code into (1) modifications to a thread's stack, (2) TM-internal code, and (3) functions declared with the transaction_pure or tm_wrapper attributes (called external code from now on).

Stack modifications are straight-forward to handle because the compiler generates code that can restore the stack slots potentially modified by a transaction. The nonspeculative code that ASF-TM executes after an asynchronous abort uses a new stack frame and is therefore not affected. It is not run from within a speculative region, and will eventually return to the compiler-generated code that restores the values of the stack slots potentially modified.

However, ASF-TM has to execute other TM-internal but nonspeculative code. If it is trivial code such as decrementing the nesting level when a nested transaction commits, then it is usually not much affected by asynchronous aborts (e. g., because the nesting level can be easily reset to zero on transaction restarts). However, if updates of TM metadata consist of modifiying more than one memory location, then asynchronous aborts can cause incomplete updates and a violation of invariants. Thus, such code has to be made robust to asynchronous aborts, which complicates the implementation. For example, compiler or memory barriers might have to be added, and large parts of the TM have to be built in a way similar to reentrant signal handlers.

A good example for this issue is dynamic memory allocation from within a transaction. In an STM, this is straight-forward to do because we can just

[12] The analysis would also have to consider variables that are accessed by functions declared with the transaction_pure attribute, or functions which have a custom transactional wrapper.

call malloc. This is safe because malloc only operates on memory managed by itself, implements its own synchronization, and its callers will wait for it to have finished the operation. This is not safe anymore within a speculative region because it can abort during the operation due to malloc calling into the operating system kernel or a conflicting memory access by another thread (e. g., the abort could happen right after malloc acquired a lock, which in turn could block every thread that subsequently tries to allocate memory).

To handle memory allocation in ASF-TMs implementation, we therefore have to resort to (1) logging the allocation request, (2) aborting the transaction, (3) performing the allocation, and (4) restarting the transaction. This works well for allocations used by ASF-TM itself (e. g., undo-log buffers). For allocations triggered by the application, ASF-TM can just hope that the same allocation will happen in the restarted transaction, so it might have to revert to a software transaction after a small number of mispredictions.

The fact that STMs do not consider asynchronous aborts is also the key point in the case of external code, the last category that we need to deal with. From an STM perspective, this makes a lot of sense because it allows for simple wrapper implementations and simple reuse of a lot of library code, as long as these functions synchronize on their own and access data that is separate from transactionally accessed data. Asynchronous aborts conflict with this assumption, and I will next discuss possible work-arounds for this problem.

Software-side solutions. The simplest option would be to not call external code from within hardware transactions. The ABI would have to be extended so that it requires the compiler to notify the TM if external code is about to be executed by a transaction.[13] TMs could then decide to switch to software transactions before executing such code. However, this is not what we really want because there could be many bits of external code (e. g., built-in functions used by the compiler) making it less likely to be able to use hardware transactions.

Another option would be to classify external code as asynchronous-abort–safe or unsafe. This would require a second group of transaction_pure and tm_wrapper attributes, increasing the complexity on the software side. ASF's particular safety requirements are different than those of STMs and other HTMs. Thus, it is not clear that expecting programmers to maintain this classification is beneficial in the long term.

One could also try to suspend speculative regions around external code, similar to the split hardware transactions proposed by Lev and Maessen [73]. This suspension has to be implemented entirely in software and results in significant performance overheads for hardware transactions because it requires read logging and write buffering, wasting much of the performance benefits of HTM.

Finally, we could try to automatically instrument external code in a way that makes it robust to asynchronous aborts, for example by transforming memory accesses to ASF's speculative accesses or by redirecting them to a simple STM that just provides this robustness but not concurrency control. However, this defeats the purpose of the transaction_pure and tm_wrapper annotations. Also, we cannot generically roll back custom synchronization code using software only.[14]

[13] This could either happen when starting a transaction, or using a new TM callback function.

[14] We cannot expect to know about the semantics of the synchronization operations, and we

Furthermore, not all external code is available as source code or at compile time, so dynamic binary instrumentation would have be used at runtime as well. Overall, relying on instrumentation seems to be too intrusive and fragile.

Hardware-side solutions. It might also be possible to change ASF in a way that avoids expensive hardware-based virtualization (i. e., by continuing to abort transactions on far control transfers) and still makes it easier for software to deal with asynchronous aborts.

ASF could offer speculative regions to run in a mode where consistency of the region is just checked at commit time and on demand during its execution. ASF would have to support a new CPU instruction that aborts a speculative region if it is inconsistent, similar to a validate function in an STM. ASF-TM could then use this instruction after each speculative load to check that the value to be returned to the user is indeed part of a consistent snapshot.

However, this STM-like operation also causes a typical STM problem to appear in hardware transactions: There can be pending speculative loads and stores that get executed even when a speculative region's snapshot is stale. Because aborts would not be instantaneous anymore in the new ASF execution mode, this would create a race condition with other privatizing transactions that could change the protection levels of the memory accessed by the speculative region with the stale snapshot. The software-only solution to this—ensuring privatization safety between hardware transactions—would decrease performance significantly. A hardware-based solution could be to make speculative accesses nonfaulting, and to forward some information from loads and stores to ASF's validating instruction (i. e., so that page faults and TLB misses can be made visible if the speculative region had indeed a consistent snapshot).

This shows that asynchronous aborts are beneficial in code that is robust to them. So, what we would really want is to suspend them when running external code and resume aborting when switching back to instrumented code. We do not need to ensure privatization safety while executing external code because such code must not access shared data.[15]

Thus, ASF could offer new CPU instructions to suspend and resume aborts in an speculative region, which ASF-TM would call before and after the execution of external code. Let us call them SUSPEND and RESUME. ASF would roll back speculative updates instantaneously with the abort as before, but defer the jump back to SPECULATE until RESUME is executed. The abort reason could be carried forward to RESUME (requiring minimal virtualization, similar to what would be needed for the on-demand validation scheme discussed previously). Alternatively, ASF could maintain a single bit indicating whether a speculative region has aborted. This bit would be set on aborts and after context switches, and cleared when new outermost speculative regions start.

Other speculative regions nested in a region where aborts are suspended are more difficult because they need to run with open-nested[16] semantics to provide

have no control over other threads potentially participating in the synchronization protocol.

[15] We can support accessing shared data through calls to the TM. These special transactional data transfer functions would just resume and suspend the speculative region before and after performing the necessary speculative accesses. Obviously, the caller would have to ensure that this is robust to asynchronous aborts at the time at which it calls these functions.

[16] Under open nesting, nested transactions essentially commit independently of parent transactions.

us with the composability that we aim for. Even if this is the case, it will be difficult to provide ASF's minimal-progress guarantees. Thus, such special nested speculative regions could instead of depending on open nesting just abort parent speculative regions or optionally use a software fallback if a parent speculative region exists. However, this only provides partial composability, in that nesting is safe but only one of the speculative regions can actually execute.

7.2.5 Discussion and Related Work

I investigated how to build an ASF-based TM runtime library that can be integrated into a general-purpose TM system. The study that this was a part of used a near-cycle-accurate full-system simulator. Other previous or more recent studies about realistic first-generation HTM either focused on different hardware support and use cases or have not been evaluated publically.

Intel's TSX is an HTM feature that has been announced recently for an upcoming CPU. It provides an interface roughly similar to ASF but does not support nonspeculative accesses; all memory accesses in a speculative region are automatically considered as speculative accesses without the need for any special annotations. Aborted speculative regions do not make any of their side effects visible, which simplifies their use but can also require a slightly more frequent execution of software transactions. GCC's TM runtime library [44] has recently been extended with a simple HTM execution mode that can use Intel's TSX; this uses the same ABI as considered here and employs serially executed software transactions as fallback mode. No performance measurements have been published so far.

The recent IBM BlueGene/Q processor also contains an HTM feature [117]. Using this HTM is significantly more complex than in the case of ASF or Intel's TSX. There are two separate execution modes aimed at short-running and long-running transactions, both of which track speculative state in different hardware resources and require different handling by software. The ABI of the TM runtime library for this HTM is not described in detail but it has to rely on the operating system kernel to handle events like exceeding the HTM's capacity or executing disallowed code; the kernel also executes TM conflict resolution policies in some modes.

In the primary study about Rock TM [26, 12], the authors were able to use a real hardware implementation. However, in comparison to ASF, Rock TM puts more restrictions on the code that it can run as a hardware transaction, so the focus of these studies has rather been on using HTM support for concurrent data structures in operating system kernels or virtual machines than on using it as part of a general-purpose TM (e. g., there are no results for the STAMP benchmarks). There seems to be some level of compiler support for Rock TM but it is not discussed whether it is reasonably close to what would be useful for a generic STM. Because ASF's design is different than Rock TM (e. g., selective annotation or handling TLB misses) and supports a wider spectrum of code in transactions, studying ASF in the context of general-purpose transactions revealed the implications of different hardware design choices in further system areas.

In TxLinux and MetaTM [97, 60], an academic HTM proposal is used as basis for evaluating hardware transactions as replacement for lock-based synchronization in the Linux kernel (using a technique similar to speculative lock

elision [86]). The authors also consider userspace applications (STAMP with manual instrumentation), but in both cases the HTM is used directly and it is not investigated how the HTM would integrate with general-purpose TM support (e. g., there is no compiler support or integration with programming languages). The HTM itself does not use selective annotation and has to implement virtualization for hardware transactions. Only a simple in-order simulation of the x86 architecture is used for the evaluation.

HASTM [99] is evaluated as part of general-purpose TM for userspace applications, including compiler support. It only provides hardware support for concurrency control but does not support transactional updates. It therefore can only accelerate STM algorithms and does not face issues like asynchronous aborts or false sharing of speculative and nonspeculative memory accesses. The simulator used for the study is described as an accurate IA32 simulation.

In comparison, ASF is well aligned with general-purpose, non-ASF-specific TM building blocks. Design decisions such as the visibility of page faults triggered by speculative code make building a TM based on ASF easier than it would be on Rock TM, for example.

Selective annotation is a very valuable feature of ASF because it allows to use costly ASF capacity only for memory accesses that actually need to be protected or speculative. Furthermore, it enables new HyTM algorithms (see Section 7.3). However, even though it makes sense to not support some kinds of false sharing between speculative and nonspeculative memory accesses, ASF should be more forgiving towards its clients and handle exceptional situations by aborting the speculative region instead of raising general protection faults. For customly built synchronization based on ASF, the fatal errors might be useful because they reduce the error cases that need to be handled. In contrast, for generic synchronization mechanisms like TM, it is much easier to just prevent such situations most of the time instead of having to choose a conservative implementation. Unlike the faults, speculative region aborts can be handled locally in the TM. This also applies to faults raised by the execution of instructions not supported in speculative regions.

ASF's instantaneous aborts and the guarantee of snapshot consistency that this implies are very useful for HyTMs and also make other behavior practical (e. g., page faults triggered by speculative code being visible). However, this also implies asynchronous aborts, which complicate executing nonspeculative code. Because none of the software-only solutions to work around this issue are really practical, extending ASF with support for suspending and resuming asynchronous aborts seems to be the best overall solution. It seems to require only a few changes to ASF but allows for more encapsulation of ASF's peculiarities inside the TM runtime library.

Overall, ASF is together with these two changes (aborts instead of faults and suspendable asynchronous aborts) ready for being used inside a general-purpose TM.[17] Support for ASF can be confined to a TM runtime library (ASF-TM) that provides the same ABI as STM runtime libraries. This is important for a new hardware feature like ASF as well because it eases the transition from STM to an ASF-based TM.

[17] PTLsim-ASF implements the first change, aborting speculative regions instead of raising faults. Suspendable asynchronous aborts are not implemented by it but the benchmarks used for the evaluation in Section 7.4 work without this change as they do not require the execution of unsafe external code.

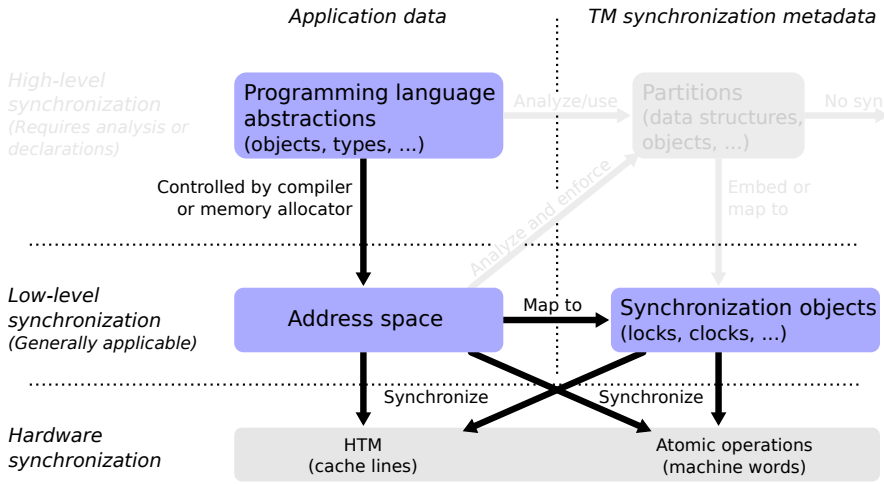

Figure 7.8: TM-based synchronization: HyTM.

7.3 Hybrid TM Algorithms

Previously, I explained why first-generation best-effort HTMs are unlikely to be able to execute all transactions as hardware transactions. The three main causes for falling back to software transactions are (1) that a transaction accesses more memory locations than the HTM has capacity for, (2) that an HTM cannot execute parts of the code in a transaction, or (3) that a different transaction conflict resolution policy than the one implemented in hardware is required (e. g., ASF implements a strict requester-wins policy, which can lead to starvation or live-locks). The fallback modes that I considered so far (e. g., serial–irrecovable mode) required all threads to agree on whether to execute hardware or software transactions. In this section, I will focus on hybrid transactional memory algorithms that can execute hardware and software transactions concurrently.

The overall aim of HyTMs is to execute as many transactions as possible as hardware transactions, while at the same time avoiding global overheads due to having to execute software transactions as well. In HyTMs, hardware transactions have to execute more code than in pure HTMs because they need to synchronize with software transactions as well using TM synchronization metadata such as orecs (see Figure 7.8). HyTM performance can therefore be characterized along the following points:

Hardware transaction performance. The overheads on transactional loads and stores are most important for this as they are the most frequently performed operations within a transaction. HTM code can be very similar to sequential code (see Figure 7.7 on page 178), so additional synchronization-related code required on the hardware transaction code path can quickly decrease performance.

Level of concurrency. Allowing a higher level of concurrency when running hardware and software transactions concurrently is important because

it will decrease the slow-down that the execution of software transactions might cause (i. e., slow software transactions should not slow down hardware transactions as well). Furthermore, the additional synchronization with software transactions should not cause hardware transactions to abort each other more often. Transactions with disjoint accesses also should not abort each other.

Hardware capacity requirements. It is essential for HyTM performance to keep the capacity requirements small for hardware transactions because exceeding the capacity is the main reason why a HyTM has to fall back to software transactions. Even though a HyTM cannot cut down the memory accesses of a transaction nor change the HTM's conflict resolution policy, it is up to the HyTM to decide how and when to use speculative and nonspeculative accesses.

Software transaction performance. Depending on the implementation of software transactions, they can still stall or disturb hardware transactions. Therefore, even if they are infrequent, their performance can have a significant influence on the overall TM performance. For example, if the software transaction implementation does not provide privatization safety, commits by hardware and software transactions will be slowed down because both have to commit in a privatization-safe manner. Likewise, using invisible reads can improve throughput among software transactions (see Section 5) but also helps in the case of HyTMs (e. g., if visible reads modify orecs, then this will abort hardware transactions that synchronize by monitoring those orecs for changes, even if there is no real conflict.

The novel HyTM algorithms that I will present next, HyLSA and HyNOrec-2, rely on *nonspeculative accesses* in transactions—including nonspeculative atomic read–modify–write instructions—to improve HyTM performance in comparison to previous HyTM proposals. In particular, they decrease the runtime overhead, abort rates, and HTM capacity requirements of hardware transactions, while at the same time allowing hardware and software transactions to run and commit concurrently (this is further discussed in Section 7.3.3 and Table 7.4). HyLSA and HyNOrec-2 complement each other in that they optimize different HyTM performance factors. Their STM base algorithms, LSA (Lazy Snapshot Algorithm, Section 5.2) and NOrec [22], focus on different workloads in their optimizations (i. e., a higher level of concurrency vs. lower single-thread overheads). All HyTMs fulfill the TM runtime library specifications discussed in Section 4.2.

For HyLSA, the focus is on decreasing HTM capacity requirements. Hardware transactions use nonspeculative data loads together with the indirection provided by ownership records and an STM-like algorithm to allow the HyTM to control the granularity of conflict detection (e. g., in contrast to the fixed cache-line granularity provided by ASF). This effectively allows them to read more data in transactions than supported by the HTM capacity if using a pure HTM algorithm. In turn, this comes with a higher runtime overhead for transactional loads and stores because of the ownership record indirection. Software transactions just need to execute LSA.

The aim of HyNOrec-2 is to keep runtime overheads of hardware transactions small. It uses speculative accesses only for all transactional loads and stores but

	Technique	Explained in
1.	Monitor metadata but read data nonspeculatively.	Section 7.3.1
2.	Use nonspeculative atomic read-modify-write operations to send synchronization messages.	Section 7.3.1 & 7.3.2
3.	Validate hardware transactions against software synchronization messages.	Section 7.3.2

Table 7.3: General-purpose synchronization techniques enabled by the availability of nonspeculative operations in hardware transactions.

Algorithm 9 Common transaction start code for all HyTMs.

```
1:  hytm-start()_p:
2:    if hytm-disabled()_p then
3:      goto line 7
4:    s ← SPECULATE                          ▷ Start hardware transaction
5:    if s ≠ 0 then                          ▷ Did we jump back here after an abort?
6:      if fallback-to-stm(s) then           ▷ Retry in software?
7:        stm-start()_p                      ▷ We are in a software transaction
8:        return false                       ▷ Execute STM codepath
9:      goto line 4                          ▷ Restore registers, stack, etc. and retry
10:   htm-start()_p                          ▷ We are in a hardware transaction
11:   return true                            ▷ Execute HTM codepath
```

not for accessing any TM metadata, thus requiring basically the same capacity as a pure HTM algorithm. Transactional accesses require very little additional code. Commits of software transactions, which run a modified NOrec algorithm, stall hardware transactions for the duration of the commit. However, in contrast to previous NOrec-based HyTMs, transactions only abort other transactions if they indeed are in conflict. The performance of hardware transactions thus is very close to a pure HTM algorithm.

Nonspeculative operations are useful beyond HyTM optimizations. Table 7.3 shows three *general-purpose synchronization techniques*, which are all combinations of both transaction-based synchronization and classic nontransactional synchronization using standard atomic instructions. The first technique can reduce HTM capacity requirements and has similarities to lock elision [86], whereas the other two are about composability with nontransactional synchronization. I will explain the techniques further in Sections 7.3.1 and 7.3.2. To make them applicable, the HTM does not only have to allow nonspeculative operations but it must also provide certain ordering guarantees (see Section 7.1.1).

I will present the HyTM algorithms using the same notation as in Section 5.2. Nonspeculative accesses are the default in these algorithms; all speculative accesses are prefixed with LOCK MOV. SPECULATE, ABORT, COMMIT, and LOCK PREFETCHW refer to the respective ASF instructions.

Because those algorithms have been built for ASF, they also rely on the AMD64's memory model as well as the ordering guarantees described in Section 7.1.1 (see Figure 7.2 on page 171). To make the algorithms easier to understand, I have annotated the speculative memory accesses with memory orders too (primarily in cases where pairs of speculative and nonspeculative memory accesses need to be ordered with respect to each other).

As I mentioned previously, DTMC generates separate STM and HTM code

paths for each transaction. A common transaction start function (see Algorithm 9) takes care of selecting STM or HTM code at runtime. A transaction first tries to run as a hardware transaction (line 4). SPECULATE returns a non-zero value when jumping back after an abort, similarly to setjmp in the standard C library. If the transaction aborts and a retry is unlikely to succeed (as determined on line 6, for example, because of capacity limitations or after multiple aborts due to contention), it switches to software mode. After this decision for either a software or hardware transaction, only STM or HTM code will be executed (functions starting with stm− or htm−, respectively) during this attempt to execute the transaction.

In what follows, I will present the HyLSA and HyNOrec-2 algorithms in Sections 7.3.1 and 7.3.2, discuss them in comparison to related work in Section 7.3.3, and evaluate them in Section 7.4.

7.3.1 The Hybrid Lazy Snapshot Algorithms

Next, I will discuss two variants of a HyTM based on LSA (see Section 5.2). These algorithms are extensions to Algorithm 3 on page 87 and use the same synchronization metadata as the STM: ownership records (orecs) and a global clock implemented as a shared integer counter. Hardware transactions also synchronize similarly to software transactions, but use ASF to modify and monitor orecs instead of locking orecs nonspeculatively and performing time-based validation.

The Eager Hybrid LSA algorithm. The first variant of HyLSA, shown in Algorithm 10, uses eager conflict detection. Like software transactions in Algorithm 3, hardware transactions speculatively acquire orecs before they write to memory locations associated with those orecs. As a result, data conflicts with other transactions will be detected as early as possible.

Transactional loads first perform an ASF-protected load of the associated orec (line 6). This operation monitors the orec for changes and will lead to an abort if the orec is updated by another thread. If the orec is not locked, the transaction uses a nonspeculative load operation (line 9) to read the target value. Note that ASF will start monitoring the orec before loading from the target address (see the ordering guarantees in Figure 7.2). If the transaction is not aborted before returning a value, this means that the orecs associated with this address and all previously read addresses have not changed and are not locked, thus creating an atomic snapshot.

Loading the target values nonspeculatively allows the HyTM to use HTM capacity just for orecs and data updates. The HyTM chooses the mapping from memory locations to orecs (i. e., the hash function called on line 6), so it can use this mapping to influence how many orecs a certain set of memory locations are associated with. The latter allows the HyTM to run hardware transactions that load more data than the HTM would actually provide capacity for, which can be valuable given that HTM capacity is limited.

Transactional stores proceed as loads, first monitoring the orec and verifying that it is not locked (lines 12–14). The transaction then watches the orec for reads and writes by other transactions (`PREFETCHW` on line 15). This ensures *eager* detection of conflicts with concurrent transactions. Finally, the updated

Algorithm 10 HyLSA (Eager variant, extends Algorithm 3)

```
 1: State of thread p:                                          ▷ Extends state of Algorithm 3
 2:   o-set: set of orecs updated by transaction

 3: htm-start()_p:
 4:   o-set ← ∅

 5: htm-load(addr)_p:
 6:   LOCK MOV : orec ←_acq orecs[hash(addr)]                   ▷ Protected load
 7:   if orec.locked then
 8:     ABORT                                                  ▷ Orec owned by (other) software transaction
 9:   val ← *addr
10:   return val

11: htm-store(addr,val)_p:
12:   LOCK MOV : orec ←_acq orecs[hash(addr)]                   ▷ Protected load
13:   if orec.locked then
14:     ABORT                                                  ▷ Orec owned by (other) software transaction
15:   LOCK PREFETCHW_acq orec                                  ▷ Watch for concurrent loads/stores
16:   LOCK MOV : *addr ← val                                   ▷ Speculative write
17:   o-set ← o-set ∪ {hash(addr)}

18: htm-commit()_p:
19:   if o ≠ ∅ then                                            ▷ Is transaction read-only?
20:     ct ← atomic-inc-and-fetch_acqrel(clock)                ▷ Commit timestamp
21:     for all o ∈ o-set do
22:       LOCK MOV : orecs[o] ←_rel ⟨false, ct⟩
23:   COMMIT                                                   ▷ Commit hardware transaction
```

memory location is speculatively written (line 16). Note that hardware transactions can safely read cache lines nonspeculatively that they have updated speculatively, so read–after–write situations will not abort transactions and the reads will read the transaction's own updates.

Upon commit, an update transaction first acquires a unique commit timestamp from the global time base (line 20). This operation will become visible before the transaction's commit but will happen after monitoring of the orecs has been started. Second, it speculatively writes all updated orecs (lines 21–22), and finally tries to commit the transaction (line 23). Note that these steps are thus ordered in the same way as the equivalent steps in a software transaction (i. e., acquiring orecs or recording orec version numbers before incrementing *clock*, and validating orec version numbers or releasing orecs afterwards). Read-only transactions do not need further actions because loads have ensured an atomic snapshot already by monitoring the associated orecs. If the transaction commits successfully, then we know that no other transaction performed conflicting accesses to the orecs (representing data conflicts). Thus, the hardware transaction could have equally been a software transaction that acquired write locks for its orecs or validated that their version numbers were not changed. If the hardware transaction aborts, then it only might have incremented *clock*, which is harmless because other transactions cannot distinguish this from a software update transaction that did not update any values that have they have read.

By nonspeculatively incrementing *clock* (line 20), a hardware update transaction sends a synchronization message to software transactions, notifying them that they might have to validate due to pending hardware transaction commits. It is thus an application of the second general-purpose technique in Table 7.3. Because ASF provides nonspeculative atomic read–modify–write operations, hardware transactions can very efficiently send such messages. In contrast,

using speculative stores would lead to frequent aborts caused by consumers of those messages. If using just nonspeculative stores instead of read–modify–write operations, concurrent transactions would have to write to separate locations to avoid lost updates, which in turn would require observers to check many different locations. In the case of HyLSA, this would also prevent the efficiency that is gained by using a single global time basis. The ordering guarantees that ASF provides for nonspeculative atomic read–modify–write operations are essential because it allows hardware transactions to send messages after monitoring data and before commit or monitoring further data.

The STM ensures privatization safety using a typical STM quiescence-based protocol (see Section 5.2); update transactions essentially wait until concurrent transactions have extended their snapshot far enough into the future so that they will have observed the updates. Because hardware transactions will be aborted immediately by conflicting updates, their snapshot is always most recent and we do not need to wait for them.

Algorithm 11 HyLSA (Lazy variant, extends Algorithm 3)

```
1:  State of thread p:                                          ▷ Extends state of Algorithm 3
2:    o-set: set of orecs updated by transaction

3:  htm-start()_p:
4:    o-set ← ∅

5:  htm-load(addr)_p:
6:    LOCK MOV : orec ←_acq orecs[hash(addr)]                   ▷ Protected load
7:    if orec.locked then
8:      ABORT                                                   ▷ Orec owned by (other) software transaction
9:    val ← *addr
10:   return val

11: htm-store(addr,val)_p:
12:   LOCK MOV : *addr ←_acq val                                ▷ Speculative write
13:   o-set ← o-set ∪ {hash(addr)}

14: htm-commit()_p:
15:   if o ≠ ∅ then                                             ▷ Is transaction read-only?
16:     ct ←_acq clock + 1                                      ▷ Optimistic commit timestamp
17:     for all o ∈ o-set do
18:       LOCK MOV : orec ←_acq orecs[o]                        ▷ Protected load
19:       if orec.locked then
20:         ABORT                                               ▷ Orec owned by (other) software transaction
21:       LOCK MOV : orecs[o] ←_acq ⟨false, ct⟩                 ▷ Speculative write
22:     t ←_acq clock
23:     if ct ≤ t then                                          ▷ Was optimistic timestamp valid?
24:       ct ← t + 1                                            ▷ Use conservative timestamp
25:       for all o ∈ o-set do
26:         LOCK MOV : orecs[o] ←_rel ⟨false, ct⟩               ▷ Speculative write
27:       t ←_acq clock
28:     if ct > t then
29:       atomic-inc_acqrel(clock)
30:   COMMIT                                                    ▷ Commit hardware transaction
```

The Lazy Hybrid LSA algorithm. Software transactions use eager conflict detection, but we can also combine those with hardware transactions that use lazy conflict detection, as shown in Algorithm 11: Upon stores, we do not read nor watch the orec associated with the accessed memory location, but instead we just speculatively write to the target location (line 12) and detect conflicts for those stores later during commit.

For an LSA update transaction to commit correctly, its commit timestamp must be strictly larger than the value of *clock* at the time when the transaction had acquired—or, for a hardware transaction, started monitoring—all of the orecs associated with updated locations. Therefore, we start the commit phase of update transactions by speculatively writing to all orecs (lines 17–21), which will also start monitoring.

We can use several optimizations to speed up commit in certain cases. Similarly to the commit-phase optimizations by Zhang et al. [123], we first install an optimistic commit timestamp (lines 16 and 21) and next load *clock* again (line 22) to check the invariant. Because the orec "acquisition" with speculative stores is not bound to a value like for a typical software lock, we can optimize more aggressively than in an STM by updating orecs to the assumed final value (*orec.locked* is false, line 21), which can then potentially allow us to skip the second orec update loop (lines 25–26). Nevertheless, the bigger effect on performance might be that we can potentially skip incrementing *clock* and share the same commit timestamp between nonconflicting transactions.

7.3.2 The Hybrid NOrec Algorithms

In this section, I will discuss how the NOrec [22] STM algorithm can be turned into a scalable HyTM for ASF. Roughly speaking, NOrec uses a single orec (a global versioned lock) and relies on value-based validation in addition to time-based validation. The reason for creating a hybrid extension to NOrec is that this algorithm can potentially provide better performance for low thread counts because it does not have to pay the runtime overheads associated with accessing multiple orecs. On the other hand, LSA is expected to provide better scalability with large thread counts or frequent but disjoint commits of software transactions. Therefore, both algorithms are of practical interest depending on the target architecture and workload.

The NOrec algorithm. NOrec, shown in Algorithm 12, is quite similar—when ignoring VBV—to time-based TMs like LSA; the main difference is that NOrec uses a single orec (*gsl*, line 2) and does not acquire the lock before attempting to commit a transaction. As a consequence, it yields a very simple implementation and allows for a few optimizations. In particular, it is not necessary to track which locks are covering loads or stores, and the lock itself can serve as time base (lines 28/32, 11, and 36). However, such a design would not scale well when update transactions commit frequently because timestamp-based validation would also fail frequently (e. g., in the checks on lines 19 and 28). Therefore, NOrec attempts value-based validation (VBV, lines 38–41) whenever timestamp-based validation is not successful (lines 20 and 29).

With VBV, the consistency of a transaction's read set is verified on the basis of the values that have been loaded instead of the versions of the orecs. The disadvantage of using values is that one has to potentially track more data in the read set because several addresses often map to the same orec. VBV is typically paired with serialized commit phases. In NOrec, this is enforced on lines 12 and 37.

My implementation of NOrec differs in a few points from the original implementation [22]. Notably, in my implementation, when writing back buffered updates upon commit, transactions only write to precisely those bytes that were

Algorithm 12 NOrec STM algorithm [22]

```
 1: Global state:
 2:   gsl: word-sized global sequence lock, consisting of:
 3:     locked: most significant bit, true iff locked
 4:     clock: clock (remaining bits)

 5: State of thread p:
 6:   sl: thread-local sequence lock
 7:   r-set: read set of tuples ⟨addr, val⟩
 8:   w-set: write set of tuples ⟨addr, val⟩

 9: stm-start()_p:
10:   repeat
11:     sl ←_acq gsl                          ▷ Get the transaction's start time
12:   until ¬ sl.locked                       ▷ Wait until concurrent commits have finished
13:   r-set ← w-set ← ∅

14: stm-load(addr)_p:
15:   if ⟨addr, new-val⟩ ∈ w-set then         ▷ Read after write?
16:     val ← new-val                         ▷ Return buffered value
17:   else
18:     val ←_acq *addr
19:     while sl ≠ gsl do                     ▷ Timestamp-based validation
20:       sl ← validate()                     ▷ Trigger value-based validation
21:       val ←_acq *addr
22:   r-set ← r-set ∪ {⟨addr, val⟩}
23:   return val

24: stm-store(addr,val)_p:
25:   w-set ← w-set ∪ {⟨addr, val⟩}           ▷ Updates are buffered

26: stm-commit()_p:
27:   if w-set ≠ ∅ then                       ▷ Is transaction read-only?
28:     while ¬ cas_acqrel(gsl : sl → ⟨true, sl.clock⟩) do   ▷ Acquire lock
29:       sl ← validate()                     ▷ Trigger value-based validation
30:     for all ⟨addr, val⟩ ∈ w-set do        ▷ Write updates to memory
31:       *addr ← val
32:     gsl ←_rel ⟨false, sl.clock + 1⟩       ▷ Release lock and increment clock

33: validate()_p:
34:   repeat
35:     repeat
36:       c ←_acq gsl                         ▷ Get current time
37:     until ¬ c.locked                      ▷ Wait until concurrent commits have finished
38:     for all ⟨addr, val⟩ ∈ r-set do
39:       v ←_acq *addr
40:       if v ≠ val then                     ▷ Value-based validation
41:         abort()                           ▷ Inconsistent snapshot
42:   until c = gsl
43:   return c
```

modified by the application, whereas the original implementation always performs updates at the granularity of aligned machine words. This more complex bookkeeping introduces higher runtime overheads but is required for the STM to operate correctly according to the C/C++ TM specification (see Section 4).

Furthermore, Dalessandro et al. assume that NOrec provides privatization safety, but this is not quite true when considering privatization safety as required by the C/C++ TM specification. It is true that software transactions will never operate on an inconsistent snapshot, but the STM can still internally access privatized data (lines 18 and 39). These loads can result in protection faults if the privatizing thread changed the memory protection on the privatized data in the meantime. However, to ease comparisons with related work in the evaluation, I have not added an explicit enforcement of full privatization safety in my implementation. Nonetheless, I will discuss this further when describing my HyTM algorithms.

The Hybrid NOrec algorithm by Dalessandro et al. HyNOred-DSS (Algorithm 13) is based on the informal description by Dalessandro et al. [22], adapted to use ASF as HTM. The main approach of HyNOrec-DSS is to use two global sequence locks, *gsl* and *esl*. The purpose of the additional lock *esl* is for software transactions to be able to interrupt and stop hardware transactions. Software transactions acquire both locks on commit (lines 5–8) and increment their version numbers after committing (lines 17–18).

Hardware transactions monitor *esl* (line 25), so that any store to this location will abort them. Also, they will not proceed if the lock is acquired (line 27). Because software transactions only update data when they have acquired *esl*, hardware transactions never see inconsistent state (e. g., partial software commits). Note that *esl* is only modified by software transactions because hardware transactions would otherwise abort each other, which is not necessary because they access all data speculatively (lines 20 and 23) and thus ASF will resolve any conflict. To avoid false sharing, *gsl* and *esl* are located in separate cache lines in my implementation.

In turn, hardware transactions increase the version number of the global sequence lock *gsl* (line 30), which will trigger validation in active software transactions and thus notify them about the updates. From the perspective of software transactions, committed hardware transactions are thus equivalent to software transactions that committed atomically.

A scalable Hybrid NOrec algorithm. The major problem of HyNOrec-DSS is that it does not scale well in practice (see Section 7.4). For example, Dalessandro et al. assume [22] that the update of the contended *gsl* by every hardware transaction (line 30 in Algorithm 13) is not a performance problem because it would happen close to the end of a transaction. However, experimental evaluation of this algorithm shows a high rate of aborts and poor overall performance.

In what follows, I will construct a new algorithm, HyNOrec-2, that performs much better while being no more complex. *gsl* and *esl* are used by software and hardware transactions to synchronize with each other, so my key approach is to apply the last two techniques from Table 7.3 and use nonspeculative operations to let hardware transactions synchronize more efficiently via these variables. To

Algorithm 13 HyNOrec-DSS: HyTM by Dalessandro et al. [22] (extends Algorithm 12)

```
 1: Global state:                                          ▷ Extends state of Algorithm 12
 2:   esl: extra sequence lock

 3: stm-acquire-locks()_p:
 4:   SPECULATE                                           ▷ Start hardware transaction (retry code omitted)
 5:   LOCK MOV : l ← gsl
 6:   if l = sl then
 7:     LOCK MOV : gsl ← ⟨true, sl.clock⟩                 ▷ Try to acquire commit lock
 8:     LOCK MOV : esl ← ⟨true, sl.clock⟩                 ▷ Also acquire extra lock
 9:   COMMIT
10:   return l = sl                                       ▷ True ⇔ locks were acquired atomically

11: stm-commit()_p:                                       ▷ Replaces function of Algorithm 12
12:   if w-set ≠ ∅ then                                   ▷ Is transaction read-only?
13:     while ¬ stm-acquire-locks() do                    ▷ Acquire gsl and esl atomically
14:       sl ← validate()
15:     for all ⟨addr, val⟩ ∈ w-set do                    ▷ Write updates to memory
16:       *addr ← val
17:     esl ← ⟨false, sl.clock + 1⟩                       ▷ May abort hardware transaction
18:     gsl ← ⟨false, sl.clock + 1⟩                       ▷ Release lock and increment clock

19: htm-load(addr)_p:
20:   LOCK MOV : val ← *addr                             ▷ Protected load
21:   return val

22: htm-store(addr,val)_p:
23:   LOCK MOV : *addr ← val                             ▷ Speculative write

24: htm-start()_p:
25:   LOCK MOV : sl ← esl                                ▷ Protected load (monitor extra lock)
26:   if sl.locked then                                  ▷ Extra lock available?
27:     ABORT                                            ▷ No: spin by explicit self-abort [22]

28: htm-commit()_p:
29:   LOCK MOV : l ← gsl
30:   LOCK MOV : gsl ← ⟨false, l.clock + 1⟩             ▷ Release lock and increment clock
31:   COMMIT                                             ▷ Commit hardware transaction
```

better explain and evaluate the different optimizations involved, I will additionally show two intermediate algorithms.

Algorithm 14 shows the first (intermediate) NOrec-based HyTM, which will serve as the basis for the other two variants. As a first straightforward optimization, a hardware transaction has to update *gsl* only if it will actually update shared state on commit (line 26).

Second, we do not need to use a small hardware transaction to update both *gsl* and *esl* in stm−commit. This is not necessary because *esl* is purely used to notify hardware transactions about software commits[18] and can only be modified by a software transaction that previously acquired *gsl* (line 7). In contrast to Algorithm 13, this allows hardware transactions to try to commit at a time where *gsl* has been acquired but *esl* has not yet been updated (which would have aborted the hardware transaction). However, this case can be handled by just letting the hardware transaction abort if *gsl* has been locked (line 29).

This second change is not about performance but it allows us to have a software fallback path in the HyTM that does not depend on HTM progress guarantees (e. g., no spurious aborts), which are surprisingly difficult to imple-

[18] As a matter of fact, *esl.clock* can contain any value as long as the lock bit is updated properly because such an update will abort hardware transactions monitoring *esl*.

Algorithm 14 HyNOrec-0: STM acquires locks separately (extends Algorithm 12)

```
 1: Global state:                                          ▷ Extends state of Algorithm 12
 2:   esl: extra sequence lock

 3: State of thread p:                                     ▷ Extends state of Algorithm 12
 4:   update: are we in an update transaction?

 5: stm-commit()_p:                                        ▷ Replaces function of Algorithm 12
 6:   if w-set ≠ ∅ then                                    ▷ Is transaction read-only?
 7:     while ¬ cas_acqrel(gsl : sl → ⟨true, sl.clock⟩) do  ▷ Acquire commit lock
 8:       sl ← validate()
 9:     esl ← ⟨true, sl.clock⟩                            ▷ Also acquire extra lock (no need for cas)
10:     for all ⟨addr, val⟩ ∈ w-set do                     ▷ Write updates to memory
11:       *addr ← val
12:     esl ←_rel ⟨false, sl.clock + 1⟩                   ▷ Release locks and increment clock
13:     gsl ←_rel ⟨false, sl.clock + 1⟩

14: htm-load(addr)_p:
15:   LOCK MOV : val ← *addr                              ▷ Protected load
16:   return val

17: htm-store(addr,val)_p:
18:   LOCK MOV : *addr ← val                              ▷ Speculative write
19:   update ← true                                       ▷ We are in an update transaction

20: htm-start()_p:
21:   LOCK MOV : l ← esl                                  ▷ Protected load (monitor extra lock)
22:   if l.locked then                                    ▷ Extra lock available?
23:     ABORT                                             ▷ No: spin by explicit self-abort
24:   update ← false                                      ▷ Initially not an update transaction

25: htm-commit()_p:
26:   if update then
27:     LOCK MOV : l ← gsl
28:     if l.locked then                                  ▷ Main lock available?
29:       ABORT                                           ▷ No: we will be aborted anyway
30:     LOCK MOV : gsl ← ⟨false, l.clock + 1⟩             ▷ Release lock and increment clock
31:   COMMIT                                              ▷ Commit hardware transaction
```

ment [31]. Also, programs can use the software path in the HyTM as is on hardware that does not support ASF.

Algorithm 14 can still suffer from conflicts on *gsl* if updating hardware transactions commit frequently. Algorithm 15 shows that we can replace the speculative update of *gsl* with a nonspeculative atomic fetch–and–increment instruction (line 9), which allows hardware transactions that access disjoint data to not abort each other anymore and makes the algorithm scale better. Note that the fetch–and–increment operation will be ordered before the commit of the transaction. Also, using a typical CAS loop instead of the fetch-and-increment yields lower performance.

To understand why this is possible, consider possible orderings of the hardware transaction's fetch-and-increment and a software transaction's CAS on *gsl*. If the increment comes first, the CAS will fail and will cause a software transaction validation. If the software transaction accesses during validation any updates of the hardware transaction before the former can commit, it will abort the hardware transaction, making the situation look like if some transaction committed without updating anything. If in contrast the CAS comes first, the hardware transaction will notice that *gsl* was locked before it incremented *gsl* and will abort. The hardware transaction's update to *gsl* is harmless in this

Algorithm 15 HyNOrec-1: HTM writes *gsl* nonspeculatively (extends Algorithm 14)

```
1:  htm-start()_p:                                  ▷ Replaces function of Algorithm 14
2:    wait until ¬ esl.locked                       ▷ Spin while extra lock unavailable
3:    LOCK MOV : l ← esl                            ▷ Protected load
4:    if l.locked then                              ▷ Extra lock available?
5:      ABORT                                       ▷ No: explicit self-abort
6:    update ← false                                ▷ Initially not an update transaction

7:  htm-commit()_p:                                 ▷ Replaces function of Algorithm 14
8:    if update then
9:      l ← atomic-fetch-and-inc_acqrel(gsl)        ▷ Increment gsl.clock (gsl.locked is MSB)
10:     if l.locked then
11:       ABORT                                     ▷ Main lock unavailable, we will be aborted anyway
12:   COMMIT                                        ▷ Commit hardware transaction
```

Algorithm 16 HyNOrec-2: HTM does not monitor *esl* (extends Algorithm 14)

```
1:  htm-start()_p:                                  ▷ Replaces function of Algorithm 14
2:    update ← false                                ▷ Initially not an update transaction

3:  htm-load(addr)_p:                               ▷ Replaces function of Algorithm 14
4:    LOCK MOV : val ←_acq *addr                    ▷ Protected load
5:    wait until ¬ esl.locked                       ▷ Spin while extra lock unavailable
6:    return val

7:  hytm-commit()_p:                                ▷ Replaces function of Algorithm 14
8:    if update then
9:      atomic-inc_acqrel(gsl)                      ▷ Increment gsl.clock (gsl.locked is MSB)
10:     wait until ¬ gsl.locked
11:   COMMIT                                        ▷ Commit hardware transaction
```

case because no transaction interprets *gsl.clock* if *gsl* is locked.

Additionally, hardware transactions spin nonspeculatively if *esl* is locked before accessing it speculatively to avoid unnecessary aborts (line 2).

The remaining problem of Algorithm 13 (and Algorithm 15) is that committing a software transaction aborts *all* hardware transactions that execute concurrently. One might see this as a minor issue assuming that, typically, software transactions are much longer than hardware transactions, but this is not necessarily the case. There are several reasons why a transaction cannot use ASF, for example because it contains instructions that are not allowed in ASF speculative regions (e. g., rdtsc), or because its access pattern quickly exceeds the associativity of the cache used to track the speculative loads, hence leading to capacity aborts after only few accesses.

Fortunately, software transactions can commit without having to abort nonconflicting hardware transactions. The key insight to understand this second extension is that the monitoring in hardware transactions is like an over-cautious form of continuous value-based validation (any conflicting access to a speculatively accessed cache line will abort a transaction). In NOrec, software transactions tolerate concurrent commits of other transactions by performing value-based validation when necessary.

This leads to my final optimization shown in Algorithm 16. Hardware transactions do not monitor *esl* using speculative accesses anymore. The purpose of *esl* is to prevent hardware transactions from reading inconsistent state such as partial updates by software transactions. To detect such cases and thus still obtain a consistent snapshot, hardware transactions first read the data specu-

latively (line 4) and then wait until they observe with *nonspeculative* loads that *esl* is not locked (line 5). If this succeeds and the transaction reaches line 6 without being aborted, it is guaranteed that it had a consistent snapshot valid at line 5 at a time when there were no concurrent commits by software transactions. Again, note that ASF will have started monitoring the data before performing the subsequent nonspeculative loads.

The reasoning for waiting until *gsl* is not locked on line 10 is similar and just applied to the commit optimization in HyNOrec-1. Waiting for *gsl* is as good as waiting for *esl* because *esl* will be locked iff *gsl* is locked (see Algorithm 14).

Thus, hardware transactions essentially *validate against commit messages by software transactions* (the third general-purpose technique in Table 7.3). This consists of the nonspeculative spinning on *esl* (reading commit messages by software transactions) combined with the implicit value-based validation performed by ASF monitoring the data accessed by the hardware transaction. The nonspeculative accesses allow hardware transactions to observe and tolerate software commits that create no data conflicts (i. e., pass value-based validation).

Note that *esl* could be removed and replaced by just *gsl*. A downside of this approach is that it would increase the number of cache misses on line 5 because both hardware and software commits would update the same lock. In contrast, if there are no frequent commits by software transactions, checking *esl* on line 5 is likely to hit in the cache. Therefore, it is better to keep *gsl* and *esl* separate.

Hardware transactions in the HyNOrec algorithms are privatization-safe. Unconditional memory accesses will never target privatized data in a data-race-free program. Conditional memory accesses depend on the snapshot formed by the prior loads in the transaction. This snapshot will always be up-to-date because ASF monitors all the accessed data and transactions will be aborted immediately if the data changes. However, NOrec software transactions can still have pending reads so we need to ensure full privatization safety for them (see the prior discussion of this for details).

7.3.3 Discussion and Related Work

Table 7.4 shows a comparison of my HyTM algorithms (second and third row) with previous HyTM designs. The columns list HyTM properties that have a major influence on performance.

First, at least first-generation HTM will not be able to run all transactions in hardware. Thus there likely will be software transactions, which should be able to run concurrently with hardware transactions (see column two[19]). Second, HyTMs should not introduce additional runtime overheads for hardware transactions, which would decrease HTM's performance advantage compared to STM. Third, HTM capacity for transactional memory accesses is scarce, so HyTM should require as little capacity as possible[20]. Furthermore, HyTM algorithms that do not guarantee privatization safety for software transactions have to ensure this using additional implementation methods (see Section 5.2), resulting in additional runtime overhead. Visible reads are often more costly for STMs than invisible reads and can introduce artificial conflicts with trans-

[19] "Yes" means that non-conflicting pairs of software/hardware transactions can run concurrently.

[20] "Data" refers to the application data accessed in a transaction.

HyTM	HW/SW concurrency	HW transaction load/store runtime overheads	HW capacity required for	Privatization safety (SW)	Invisible reads (SW)	Remarks
HyNOrec-2	Yes, SW commits stall other HW/SW ops	Very small	Data	Yes[a]	Yes	See Algorithm 16
HyLSA (eager)	Yes	Small (load orec)	Orecs and data updates	No	Yes	See Algorithm 10
Phased TM [74]	No	None	Data	N/A	N/A	Can use any STM
Hoffman et al. [60]	Little	None	Data	Yes	No	Dirty reads not prevented
Kumar et al. [69]	Yes	High (indirection)	Data	Yes	No	
Damron et al. [23]	Yes	Small (load orec)	Data and orecs	Yes	No	
HASTM [99] cautious	Yes	Medium (load+log orec)	Read data	No	Yes	Stores in SW only
HASTM aggressive	Yes	Small (load orec)	Read data and orecs	No	Yes	Stores in SW only
HyNOrec-DSS [22]	Partial, SW commits abort HW txns	None, but concurrent commits abort each other	Data and 2 locks	Yes[a]	Yes	
HyNOrec-DSS-2 [19]	Yes, SW commits stall other HW/SW ops	Very small, concurrent commits can still abort each other (but less likely)	Data and 3 locks/counters	Yes[a]	Yes	Applies to their best-performing algorithms

Table 7.4: Overview of HyTM designs.

[a]The NOrec-based algorithms do not guarantee full privatization safety as required by the C/C++ TM specification (see Section 7.3.2 for details).

actional HTM reads (e. g., if the STM updates an orec even though it just reads the associated data).

In phased TM [74], the implementation mode for transactions is switched globally (i. e., only software or hardware transactions are running at a time). The serial irrevocable mode that is present in most current STMs is a special case of the phased approach, as it can be used as a very simple software fallback for HTMs. This leads to no HyTM overhead when in hardware mode, but even a single transaction that has to run in software reduces overall performance to the level of STM. The phased TM approach is orthogonal to hybrid TM.

Similarly, the HyTM [60] presented by Hofmann et al. uses a simple global lock as software fallback mechanism instead of an STM that can run several software transactions concurrently. Hardware transactions wait for a software transaction to finish before committing, but are not protected from reading uncommitted and thus potentially inconsistent updates of software transactions ("dirty reads"). Remember that with ASF, for example, hardware transactions are not completely sandboxed; page faults due to inconsistent snapshots will abort speculative regions but will also be visible to the operating system.

Kumar et al. describe a HyTM [69] based on an object-based STM design with indirection via locator objects, which uses visible reads and requires small hardware transactions even for software transactions. Prior research has shown that STM algorithms with invisible reads and no indirection have significantly lower overhead (e.g., [27, 42]).

Damron et al. present a HyTM [23] that combines a best-effort HTM with a word-based STM algorithm that uses visible reads and performs conflict detection based on ownership records. The HTM does not use selective annotation and thus hardware transactions have to monitor application data and TM metadata (i. e., ownership records) for each access, which significantly increases the HTM capacity required to successfully run transactions in hardware. Likewise, visible reads result in significant overheads for STMs. This HyTM is also used in a study about the HTM support in Rock [26].

The hardware-accelerated STM algorithms (HASTM) by Saha et al. [99] are based on multiple ownership records[21], like LSA but unlike NOrec. HASTM in cautious mode monitors application data and does read logging, whereas HyLSA monitors ownership records and does not log reads. HASTM in aggressive mode monitors both application data and ownership records, thus suffering from higher HTM capacity requirements (evaluated in Section 7.4). Thus, only HyLSA can change the memory–to–orec mapping to achieve a larger effective read capacity. Transactional stores in HASTM are not accelerated but executed in software only. Furthermore, HASTM in cautious mode as presented in the paper does not prevent dirty reads[22], which can crash transactions in unmanaged environments such as C/C++.

Spear et al. propose to use Alert-On-Update (AOU) [110] to accelerate snapshots by reducing the number of necessary software snapshot validations in STMs based on ownership records. However, LSA already has efficient time-based snapshots due to its use of a global time base, whereas AOU uses a commit counter heuristic, which can suffer from false positives that lead to costly revalidations. The details of the AOU algorithm are not presented, thus it is

[21] I am considering HASTM's cacheline-based variants here.

[22] It first checks the version in an ownership record and then loads data speculatively. Executing these steps in reverse order fixes this problem.

difficult to assess the remaining HyTM aspects and overheads (and I do not include it in Table 7.4). Similar to HASTM, AOU does not use speculative writes for transactional stores.

Dalessandro et al. informally describe a HyTM [22] based on the NOrec STM ("HyNOrec-DSS"). It features low runtime overheads and capacity requirements but it shows less scalability because (1) commits of software transactions abort hardware transactions and (2) concurrent commit phases of hardware transactions can abort each other as well. See Sections 7.3.2 and 7.4 for a detailed discussion and evaluation.

In concurrent work [19] that has been published after the first publication [39] of my HyTM algorithms, Dalessandro et al. describe optimizations of HyNOrec-DSS ("HyNOrec-DSS-2", last row in Table 7.4) and evaluate them on Rock [26] and on ASF. They try to reduce conflicts on metadata (NOrec's global lock, see Section 7.3.2) by distributing commit notifications using speculative stores over several counters, which leads to additional runtime overhead for software transactions because they then have to validate all these counters (and at least two) after each transactional load. In contrast, my HyTMs use nonspeculative read–modify–write operations for such notifications (the second technique in Table 7.3), which enables software transactions to validate using only a single counter. Their algorithms also use nonspeculative loads to validate during a hardware transaction's runtime ("lazy subscription", the third technique in Table 7.3) but still use speculative reads for validation during commit, and thus require more HTM capacity than HyNOrec-2. Furthermore, they propose an optimization similar in spirit to phased TM, but embedded into the HyTM algorithms ("SWExists"), which avoids commit-time synchronization with software transactions if none is running. However, this requires speculative accesses to one further location (thus increasing HTM capacity requirements), and only helps in workloads in which software transactions are rare. SWExists could be applied to our algorithms as well and could increase scalability if mostly hardware transactions execute. Their evaluation results on Rock cannot be easily compared with ours because Rock is fairly limited compared to ASF. On ASF, they only show results for one ASF implementation, LLB256 (see Section 7.1.1), which has sufficient capacity to run almost all transactions in hardware and represents the best case in terms of HTM capacity. Other ASF implementations with reduced capacity (e. g., because of cache associativity or a smaller LLB) might be more likely to appear in real hardware but in turn make extra speculative accesses for HyTM metadata much more costly. Furthermore, the choice of LLB256 makes it more difficult to compare their optimizations in detail to my implementations because, as I show in Section 7.4, the interesting behavior of HyTMs (and arguably, the target workload for best-effort HTM) appears with workloads in which software transactions are *not* rare.

As shown in Table 7.4, the new HyTM algorithms that I have presented improve on previous designs. In the class without orecs, HyNOrec-2 provides a high level of concurrency and good scalability while not wasting HTM capacity and requiring only a very small runtime overhead. For HyTMs with orecs, HyLSA features either lower HTM capacity requirements or a smaller runtime overhead than other algorithms.

7.4 Evaluation

In this section, I will evaluate TMs that use ASF, focusing on the performance of HyTM algorithms and how they compare against using ASF as a pure HTM with serial-irrevocable mode as fallback. I will not evaluate ASF itself or its design decisions because this is out of the scope of my work; instead, I will assume that ASF is a realistic first-generation best-effort HTM.

The experiments were executed on the CPU simulator discussed in Section 7.1.1, which has reasonable simulation accuracy (see Figure 7.3 on page 173). All measurements were performed on the simulator, including those for STMs.

I will show results for LLB8, LLB8L1, and LLB256, the ASF implementation variants explained in Table 7.2 on page 172 because this highlights the impact that differences in HTM capacity can have on the performance of HTMs and HyTMs.

Because of the huge runtime overhead of simulation, shorter benchmark runs had to be used to make it practically feasible to perform a large number of experiments. For the STAMP benchmarks (see Section 3.4.3), this can be achieved by using the standard configurations recommended for simulated runs (annotated with "Sim" in Table 3.2 on page 40). For the IntegerSet benchmarks (see Table 3.1), a fixed number of operations is executed instead of running each benchmark for several seconds.

All benchmarks have been compiled with DTMC, which creates a separate code path in each transaction for each TM algorithm that can be used (e. g., one path each for hardware and software transactions when a HyTM is used). The HyTM algorithms are implemented as part of ASF-TM (see Section 7.2), which in turn is part of the prototype TM runtime library that I built (see Section 3.4.2).

7.4.1 ASF-TM performance

The focus of my work on ASF-TM was primarily on functional aspects of using ASF in a typical TM software stack (e. g., how to deal with asynchronous aborts). To synchronize between transactions, ASF-TM uses ASF like a pure HTM and simply falls back to running transactions in serial-irrevocable mode if it seems unlikely that they can finish execution as hardware transactions. While ASF-TM and the quality of compiler support affect the single-thread overheads of hardware transactions to some extent, scalability and how often transactions can execute completely as hardware transactions is essentially determined by just the performance properties of ASF. Therefore, I will evaluate ASF-TM performance by summarizing the results of a study of ASF performance that I coauthored [14]. Note that these results have been obtained on earlier versions of both the simulator and the TM software prototypes than those used to obtain the results presented in Section 7.4.2.

Figure 7.9 shows the performance of single-threaded executions of the STAMP benchmarks in comparison to executions of sequential versions of those benchmark that do not employ any synchronization code. These measurements are for the LLB256 implementation variant of ASF, which can execute essentially all transactions in the benchmarks as hardware transactions. We can see that ASF-TM is about 20–40% slower than sequential execution, whereas STM (LSA, see

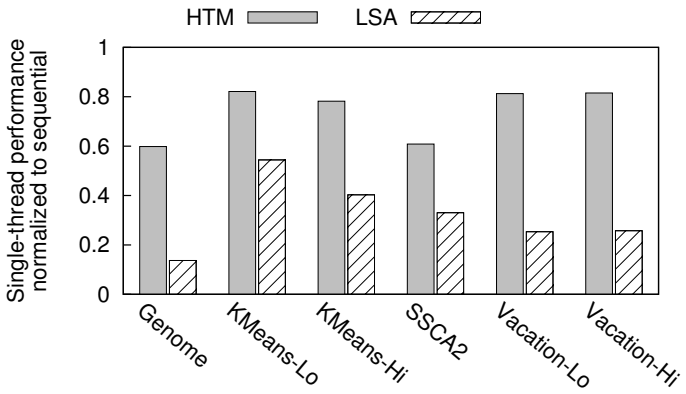

Figure 7.9: Single-thread performance of the STAMP applications normalized to the execution time of sequential, nonsynchronizing code (larger is better).

Algorithm 3) only has one fifth to half of the the execution speed of sequential runs.

These runtime overheads of ASF-TM are caused by several factors. First, all transactions have to adhere to the TM runtime library ABI and thus need to call the respective begin and commit functions (see Section 7.2.2); this includes an explicit save or restore of CPU registers because ASF restores just a few of them. Second, transactions in ASF-TM have to take part in serial-irrevocable mode synchronization (see Section 5.2) before they can execute as hardware transactions, which does not require hardware transactions to update shared state but does require memory barriers. Third, the execution of speculative regions using ASF also results in some runtime overhead compared to sequential execution. Finally, without any synchronization in the sequential code, the compiler is free to optimize code across what would otherwise be separate transactions. On the positive side, the results show that DTMC's inlining of transactional load and store calls can keep overheads relatively low; this works even for HyTMs because of DTMC's ability to generate separate code paths for each transaction.

Besides these single-thread overheads, ASF-TM's performance and scalability depends primarily on (1) how many transactions can run as hardware transactions instead of having to fall back to serial-irrevocable mode and (2) how likely hardware transactions conflict with other concurrent transactions.

Figure 7.10 shows a break-down of the reasons why hardware transactions abort for selected STAMP applications. We can see that exceeding ASF capacity is the primary abort reason on LLB8 and LLB8L1, whereas LLB256 has enough capacity to execute even the larger transactions. Note that even though LLB8L1 can use the L1 cache to track transactional reads, the cache's associativity (two-way set associative) can still limit the effective capacity; furthermore, the cache line displacement logic in the simulator has not been optimized to try to minimize the impact of capacity limitations. The "malloc" aborts are caused by how ASF-TM handles memory allocation inside of hardware transactions (see Section 7.2.4), which only infrequently requires transactions to fall back to serial-irrevocable mode. Page faults and system calls are very unlikely to occur

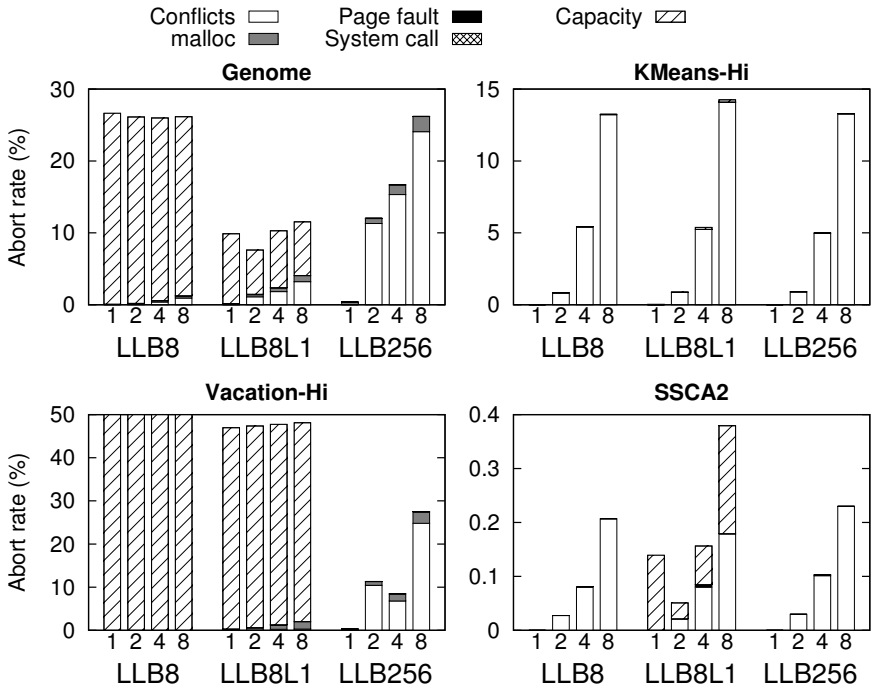

Figure 7.10: Hardware transaction abort rates of selected STAMP applications for 1, 2, 4, and 8 threads for HTM. The different patterns identify the cause of aborts. Note that a capacity abort can occur only once per executed transaction because after such an abort, the transaction will fall back to serial-irrevocable mode.

in transactions in these benchmarks, so capacity limitations are the most likely cause when a transaction falls back to serial-irrevocable mode. This is further illustrated by the HTM results in Tables 7.5 and 7.6, which show which ratio of transactions can commit as hardware transactions.[23]

Overall, LLB8 is often insufficient to execute many transactions using ASF except those with very small transactions such as HashTable. LLB8L1 can typically execute many hardware transactions except in STAMP applications like Vacation that traverse several data structures. LLB256 has enough capacity to execute all transactions in our benchmarks.

Scalability results for using hardware transactions with serial-irrevocable mode as fallback can be found in several figures in Section 7.4.2 (e. g., results labeled "HTM" in Figure 7.17). When transactions cannot execute as hardware transaction frequently (e. g., on LLB8), contention on the lock that serial-irrevocable mode is implemented with can cause a slowdown when the number of threads increases. This lock's implementation is optimized with the assumption that serial-irrevocable mode would be infrequent, as is the case with typical

[23]Note that as I mentioned previously, different versions of both the simulator and the TM software prototypes were used to obtain the results in Section 7.4.2 and the results in Figure 7.10.

Benchmark	Percentage of hardware transaction commits		
	LLB8	**LLB8L1**	**LLB256**
SkipList-Large	< 1%	Figure 7.11	100%
SkipList-Small	< 1%	HTM: 98–100% HyNOrec: 95–100% HyLSA-eager: 90–95%	100%
RBTree-Large	0–2%	Figure 7.11	100%
RBTree-Small	2–10%	HTM: 99–100% HyNOrec: 99–100% HyLSA-eager: 90–95%	100%
HashTable	100%	HTM: 100% HyNOrec: 99–100% HyLSA-eager: 95–96%	100%
LinkedList-Large	1–3%	Figure 7.11	100%
LinkedList-Small	Figure 7.11	100%	100%

Table 7.5: IntegerSet microbenchmarks and approximate ratio of hardware transaction commits to total number of commits. "HyNOrec" represents all of HyNorec-0, HyNOrec-1, and HyNOrec-2.

STM scenarios and workloads; a tuning of the lock implementation that is more appropriate for such HTM scenarios could avoid these slowdowns.

7.4.2 HyTM Performance

In what follows, I will evaluate the HyTM algorithms presented in Section 7.3 by comparing them with pure HTM (see Sections 7.2 and 7.4.1) and the matching STM algorithms, LSA (see Algorithm 3) and NOrec (see Algorithm 12). The HyTM implementations have the same names as the respective algorithms (e. g., Algorithm 16 is denoted "HyNOrec-2"), are based on ASF-TM, and use the STM implementations for their software transaction code paths. HyLSA and LSA use the Simple hash function (see Algorithm 4 on page 96), 2^{20} orecs, and a Shift parameter value of 3.

Because hardware transactions have lower runtime overheads than software transactions in all our HyTM algorithms, it is important for HyTMs to run as many transactions as hardware transactions as possible. Table 7.5 shows for the IntegerSet benchmarks which percentage of transaction commits happen on the hardware transaction code path in comparison to the total number of commits. Remember that in ASF-TM, only ASF aborts due to nontransient conditions like exceeding ASF's capacity make the HyTM switch to the software transaction code path. Aborts due to conflicts between transactions will not result in such a switch unless a transaction suffers from a high number of retries (100 in our experiments). Therefore, the ratio of HTM's commits that I show is essentially independent of the level of contention in a workload.

On LLB8L1 and LLB256, HTM and the HyNOrec algorithms (or at least HyNOrec-2) can execute almost all transactions in the IntegerSet benchmarks

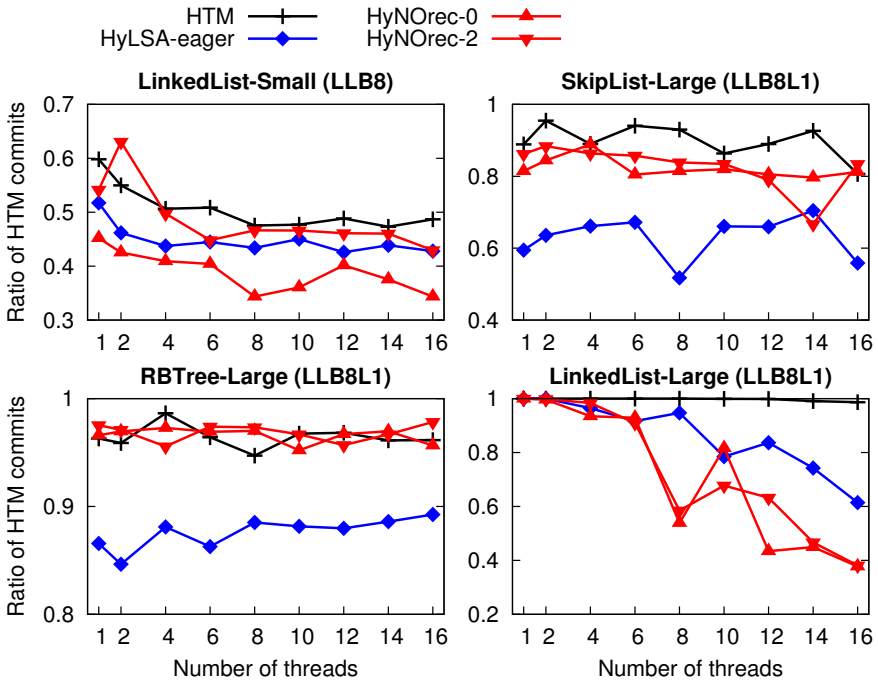

Figure 7.11: Ratio of hardware transaction commits to total number of commits.

using ASF; HyLSA is close but has to fall back to STM more often due to its higher capacity requirements. LLB8 is typically not sufficient to execute hardware transactions, except for really short transactions such as in HashTable. Note that even though HyNorec-2 does not access more data speculatively than an HTM, it can effectively reach capacity limits earlier. It has to always check *esl*, which keeps *esl* in the cache and can thus reduce the capacity limit by one, which can matter if the effective limit is the cache associativity.

HyNOrec algorithms. Let us now focus on how the different NOrec-based HyTMs perform. Figure 7.12 shows a comparison between those algorithms for the same SkipList benchmarks but with two different ASF implementations. With LLB8 (left side), all transactions have to fall back to software executions (see Table 7.5), but interestingly HyNOrec-2 is able to scale better than the other algorithms. The abort rate due to transaction conflicts shows that this is because hardware transactions in the other HyNOrec variants suffer from conflict aborts before they notice a capacity abort (which would make them switch to executing as a software transaction). Because HyNOrec-2 does not monitor *esl*, it will not be aborted by commits of nonconflicting software transactions, and will find out quickly that it should switch to software, then taking advantage of STM scalability.

When using LLB8L1 (right side), many transactions can execute in hardware (see Table 7.5 and Figure 7.11). HyNOrec-2 also scales much better in this case, showing that its ability to survive commits of nonconflicting software

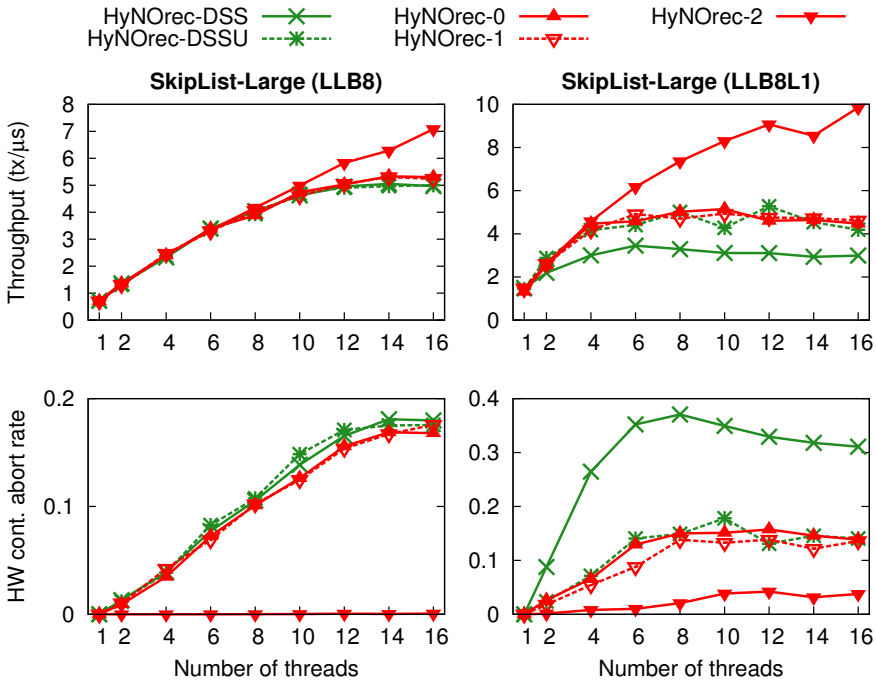

Figure 7.12: Comparison of the performance of the HyNOrec algorithms. "HW contention abort rate" is the number of aborts due to inter-transaction data conflicts per transaction that commits in hardware or switches to the software codepath.

transactions (e. g., in contrast to HyNOrec-1) is beneficial even when the majority of transactions execute in hardware. Furthermore, HyNOrec-DSS suffers from many more aborts than the other TMs. To explain this, I also show results for HyNOrec-DSSU, which is like HyNOrec-DSS but only updates *gsl* when update transactions commit, thus reducing the number of speculative updates to *gsl* (SkipList has 20% update transactions). HyNOrec-DSSU performs similar to HyNOrec-0, indicating that this part of the HyNOrec-0 optimizations is crucial. HyNOrec-DSS never performed better than HyNOrec-0 in any of the benchmarks and often performed significantly worse. It does not scale beyond 4 to 6 threads in IntegerSet unless transactions execute the software code path most of the time. Therefore, I will not consider HyNOrec-DSS any further in what follows.

Figure 7.13 shows HashTable, which runs short and mostly update transactions. HyNOrec-1 performs and scales much better than HyNOrec-0 and suffers from very few aborts, whereas the rate of aborts due to contention is still significant for HyNOrec-0. This shows that updating *gsl* nonspeculatively is an important optimization, especially if commits of update transactions are frequent. Second, it highlights that updating *gsl* speculatively can indeed lead to contention. Thus, both the optimizations HyNOrec-2 has compared to HyNOrec-0 (or HyNOrec-DSS) are effective in increasing performance significantly.

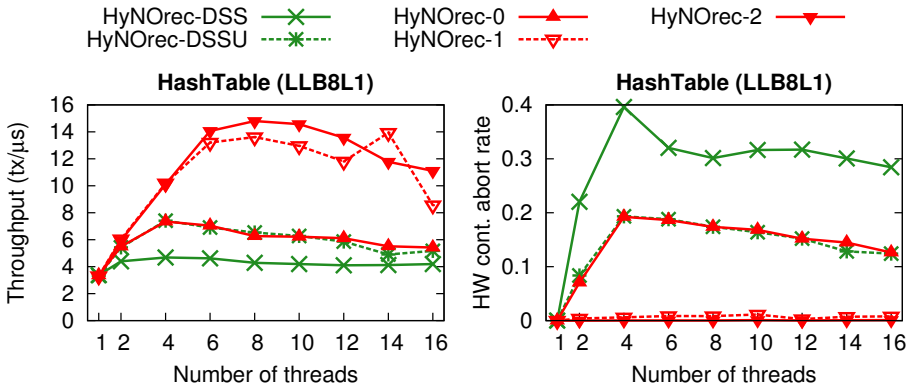

Figure 7.13: Comparison of the performance of the HyNOrec algorithms (HashTable). "HW contention abort rate" is the number of aborts due to inter-transaction data conflicts per transaction that commits in hardware or switches to the software codepath.

HyLSA algorithms. After looking at the HyNOrec algorithms, let us now focus on HyLSA. Figure 7.14 shows the performance of three HyLSA variants. Unfortunately, the current version of the ASF simulator does not always provide the ASF ordering guarantees for nonspeculative accesses (see Figure 7.2). To be able to run the same HyLSA TMs in all benchmarks, I had to add memory barriers (i. e., an lfence instruction) between the speculative load of an orec and the nonspeculative load of data (e. g., lines 6 and 9 in Algorithm 10). HyLSA-lazy-noMB is Algorithm 11 without such barriers, which shows their runtime overhead. However, scalability remains similar in the benchmarks for all variants. HyLSA-lazy can scale slightly better than HyLSA-eager, which is likely due to lazy conflict detection. Because both perform similar in many situations, I will focus on HyLSA-eager in what follows.

HyLSA's capacity requirements are different than those of HyNOrec. HyLSA buffers updates speculatively and thus, for stores, needs ASF capacity for both data and orecs. However, for loads, only orecs are accessed speculatively, and the hash function that maps data to orecs influences capacity requirements. The HyLSA implementations map memory at the granularity of machine words to word-sized orecs (i. e., the three least-significant bits of addresses are discarded and the remaining bits select a slot in an array with 2^{20} orecs). Orecs are not cache-line padded because padding would likely increase capacity requirements for HyLSA unless more than one adjacent cache line of memory maps to the same orec. Without padding, hardware transactions detect conflicts on cache line granularity, whereas STM transactions can detect conflicts on word-size granularity and can thus potentially scale better in high-contention workloads.

Table 7.5 and Figure 7.11 show that HyLSA is already more likely to hit capacity limitations than HyNOrec just because it needs twice the capacity for stores, so it is important for HyLSA to read data nonspeculatively. Figure 7.15 illustrates this point further, showing that when HyLSA is changed so that it accesses data speculatively (HyLSA-eager-SDL), less transactions can execute

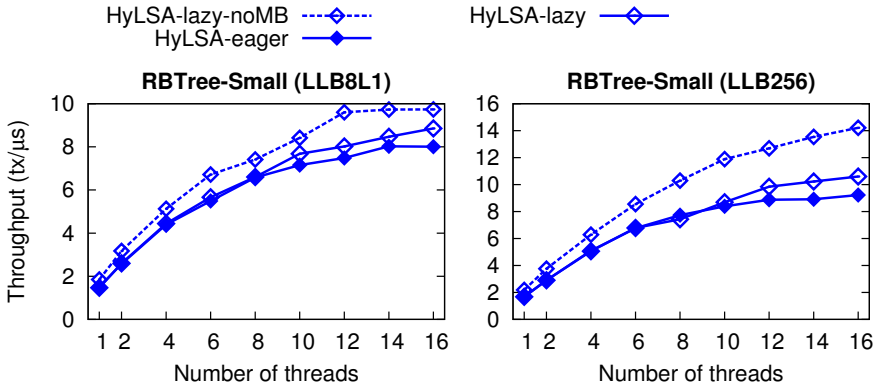

Figure 7.14: Comparison of the performance of HyLSA algorithm variants.

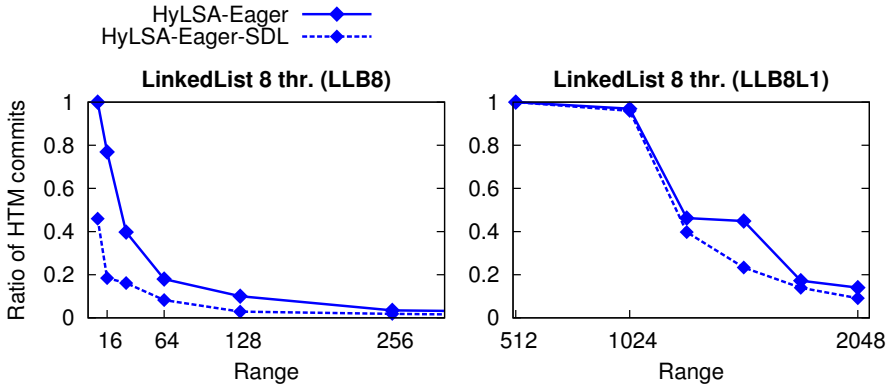

Figure 7.15: Ratio of hardware transaction commits to total number of commits for 8 threads and read-only LinkedList of various sizes ("Range"), when accessing data speculatively ("SDL") or not.

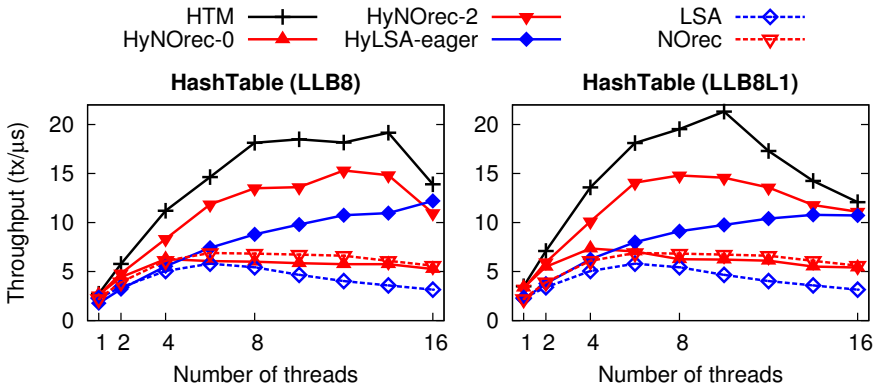

Figure 7.16: Comparison of the performance of HTM, STMs, and HyTMs with HashTable. LLB256 is not shown because it yields results very similar to LLB8L1.

as hardware transactions.

IntegerSet benchmarks. Figures 7.16, 7.17, 7.18, and 7.19 show a comparison of the performance of HTM, the HyTM algorithms, and both STMs with the IntegerSet benchmarks.

HashTable (see Figure 7.16) performs roughly similar on all ASF implementations and scales very well, but ultimately suffers from external bottlenecks when the level of concurrency increases beyond a certain point (e. g., in the memory allocator). HyNOrec 2 performs significantly better than HyNOrec-0 because it updates *gsl* nonspeculatively. It also performs better than HyLSA due to less runtime overhead per transactional data access, yet HyLSA scales well. However, HyNOrec-2 cannot reach the performance of HTM because it has to update *gsl*, which leads to contention overheads.

On all other benchmarks, LLB8 is not sufficient to run many transactions as hardware transactions, and STMs perform slightly better than HyTMs because the latter try to first execute in hardware, unsuccessfully. Better runtime tuning of when the HyTMs attempt to execute hardware transactions could help avoid this misspeculation.

SkipList-Large on LLB8L1 (see Figure 7.17) shows that HyNOrec-2 can significantly outperform HTM even if just 5–10% of all transactions cannot execute as hardware transactions (see Table 7.5). HTM has the lowest runtime overheads per transactional access but its simple fallback (serial-irrevocable mode) can quickly limit scalability. In contrast, HyNOrec-2 can run nonconflicting software and hardware transactions concurrently without introducing articial aborts, and yet does not impose large overheads on hardware transactions.

RBTree-Large and RBTree-Small (see Figure 7.18) show the same win for HyNOrec-2 on LLB8L1, but for this benchmark HyNOrec-2 is even as fast as HTM on LLB256, where all transactions can execute as hardware transactions.

In LinkedList-Small (see Figure 7.19) on LLB256, all transactions can execute in hardware but HyTMs and HTM do not scale. The reason for this behav-

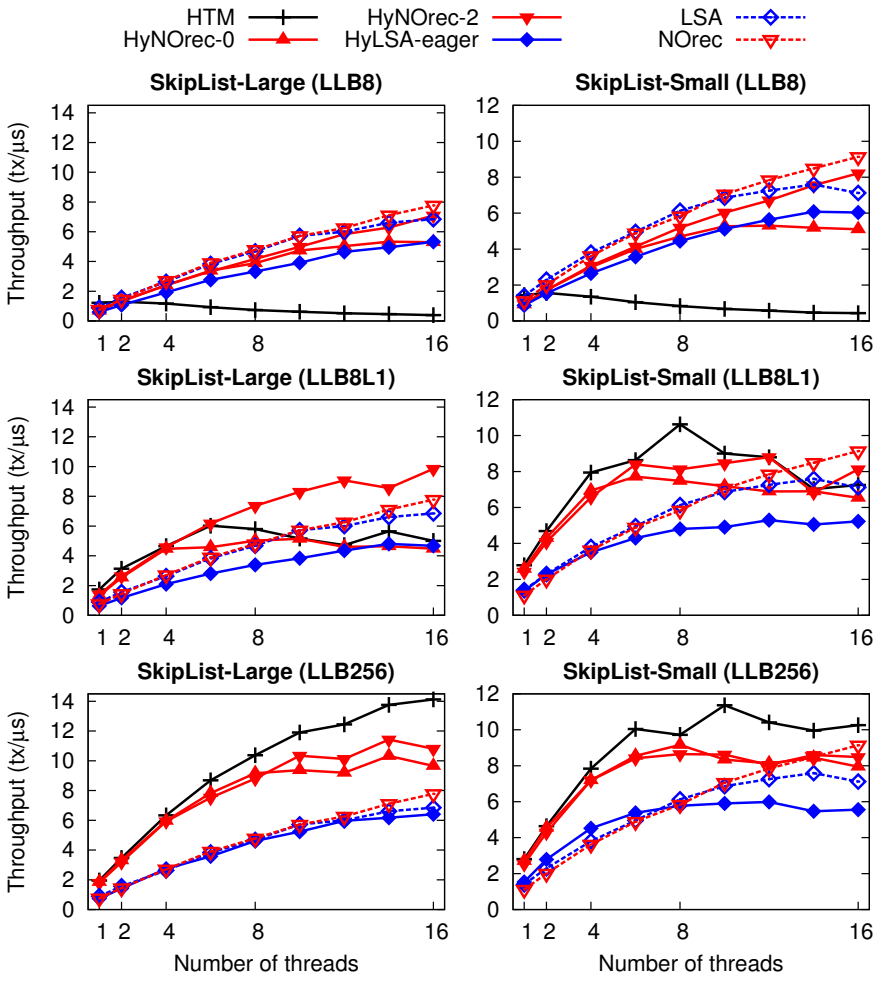

Figure 7.17: Comparison of the performance of HTM, STMs, and HyTMs with SkipList.

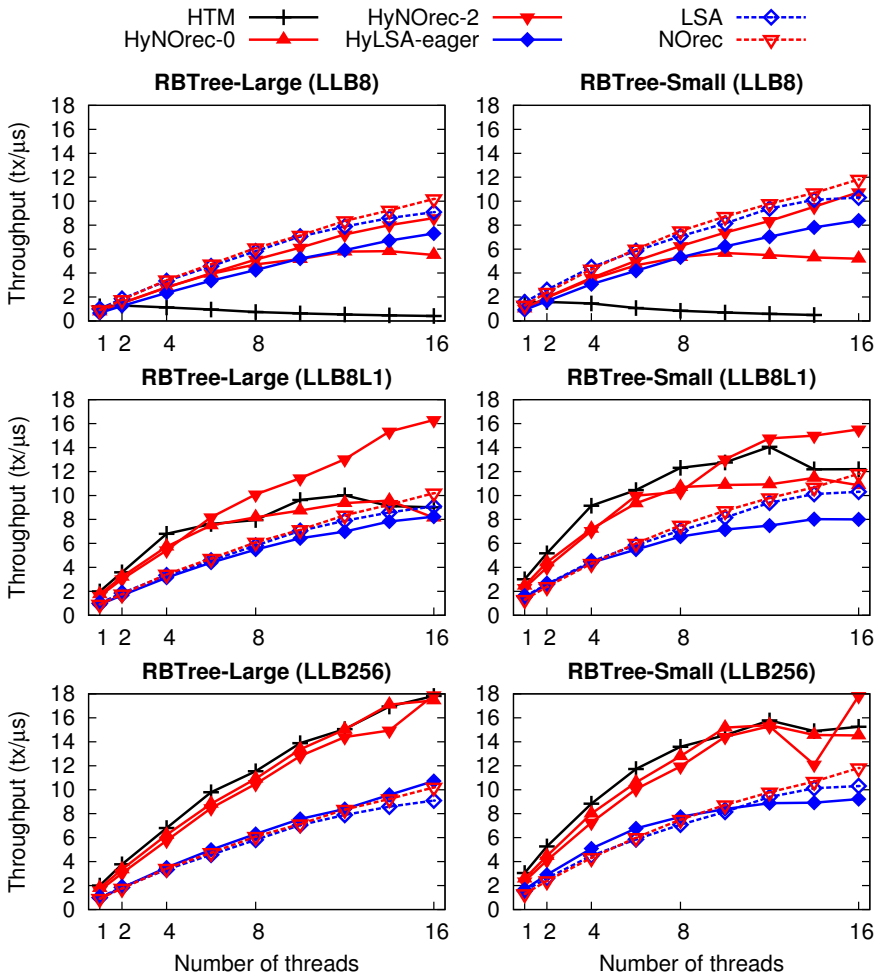

Figure 7.18: Comparison of the performance of HTM, STMs, and HyTMs with RBTree.

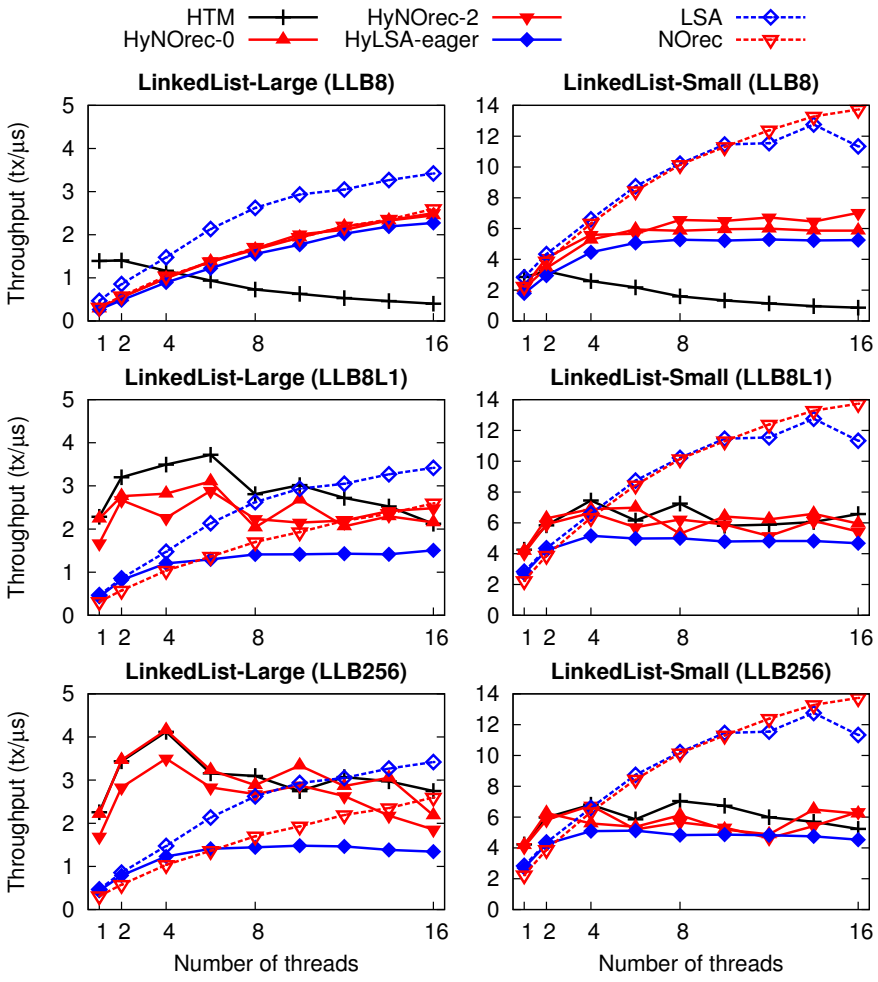

Figure 7.19: Comparison of the performance of the HTM, STM, and HyTMs with LinkedList.

Benchmark	LLB8	LLB8L1	LLB256
Genome	HTM, HyNOrec-2: 65% HyNOrec-1: 60% HyNOrec-0: 50% HyLSA: 38–42%	HTM: 90–95% HyNOrec: 85–90% HyLSA: 75%	100%
KMeans-{Hi,Lo}	HTM, HyNOrec-{1,2}: 100% HyLSA, HyNOrec-0: 25%	95–100%	100%
Vacation-Hi	0%	HTM: 13–14% HyNOrec: 9–12% HyLSA: 3–5%	100%
Vacation-Lo	0%	HTM: 8–11% HyNOrec: 6–9% HyLSA: 1–2%	100%
SSCA2	99–100%	99–100%	99–100%

Table 7.6: Approximate ratio of hardware transaction commits to total number of commits in STAMP.

ior is that ASF's conflict detection is on the granularity of cache lines, whereas STMs can use smaller granularities (word-sized in LSA, value-based validation in NOrec), which can be beneficial in high-contention workloads with a high level of false sharing. As explained before, HyLSA could use the indirection of the orecs and the memory-to-orec mapping to emulate a smaller granularity for conflict detection. However, this would waste ASF capacity and thus does not seem to be a generally useful strategy. Instead, a HyTM should perhaps switch proactively to software to try to employ a more contention-resistant STM algorithm. The HyNOrec algorithms cannot change the conflict-detection granularity of hardware transactions, but at least HyNOrec-2 performs similar to HTM on LLB8L1 and LLB256 with both LinkedList-Large and LinkedList-Small.

STAMP benchmarks. To conclude the evaluation, I will next show performance results for selected applications from STAMP, starting with the hardware transaction ratio (see Table 7.6). LLB256 is again sufficient to execute all transactions in hardware. SSCA2 and KMeans have small transactions and thus can execute almost completely with hardware transactions even on LLB8 and LLB8L1. However, HyNOrec-0 seems to require just a little too much capacity for LLB8 to be sufficient with KMeans-Hi; in contrast to HyNOrec-2, it accesses *esl* and *gsl* speculatively. Genome on LLB8 also shows this difference in the capacity requirements of the HyNOrec algorithms. Transactions run by Vacation are too large for LLB8's capacity and often too demanding for LLB8L1 as well.

HyLSA's larger capacity requirements for stores decrease the hardware transaction ratio; except when capacity is clearly sufficient, HyLSA can run significantly fewer transactions as hardware transactions than HyNOrec-2.

Figures 7.20, 7.21, and 7.22 show scalability results for the STAMP applications. With KMeans, which has small enough transactions to allow them always to be executed as hardware transactions by an HTM, HyNOrec-2 performs best

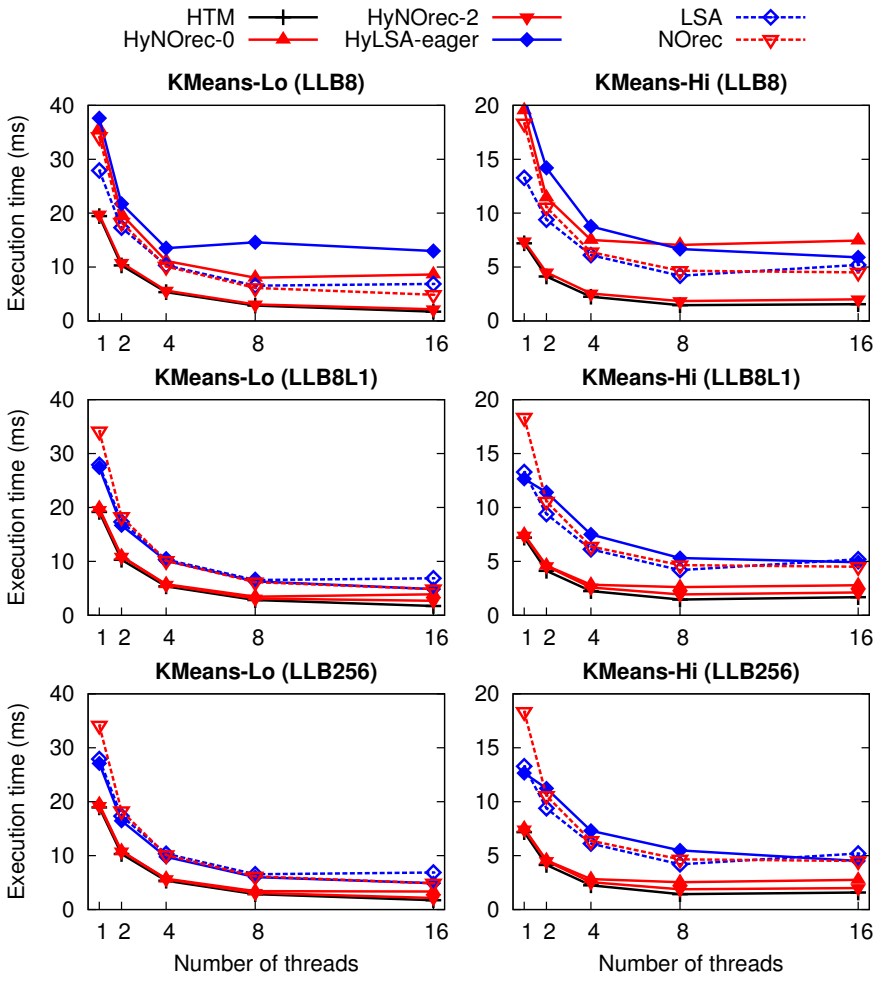

Figure 7.20: Comparison of the performance of the HTM, STM, and HyTMs with KMeans.

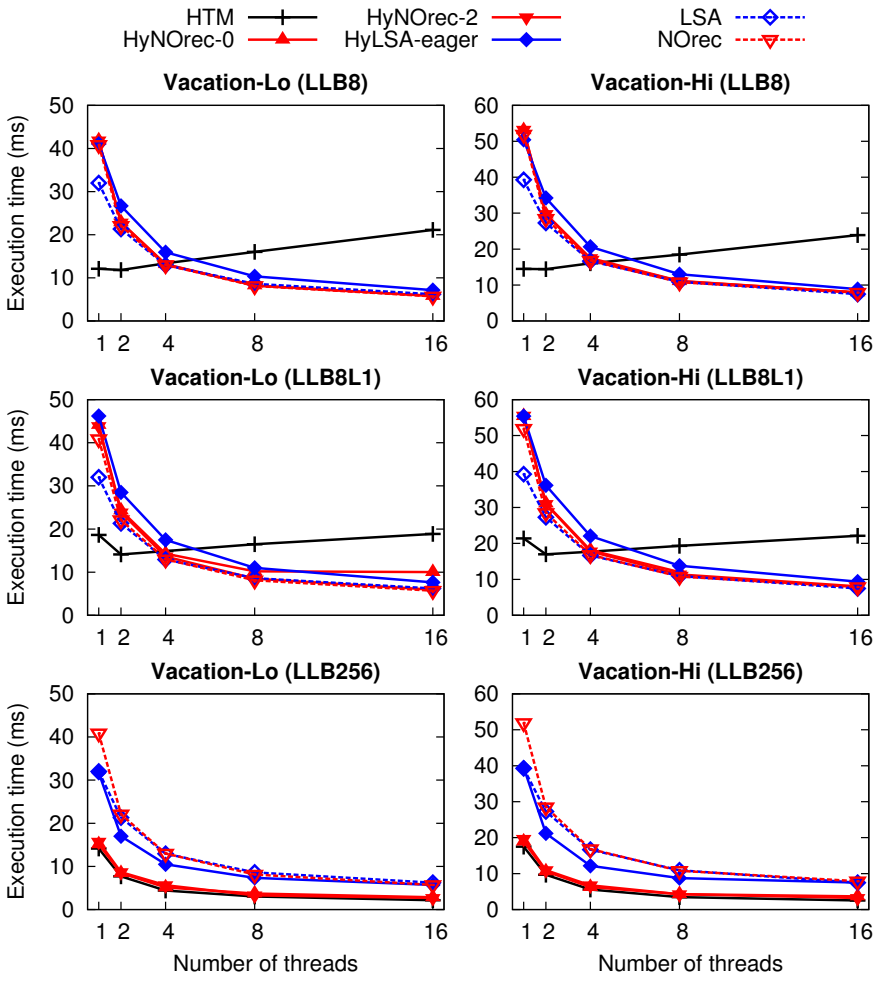

Figure 7.21: Comparison of the performance of the HTM, STM, and HyTMs with Vacation.

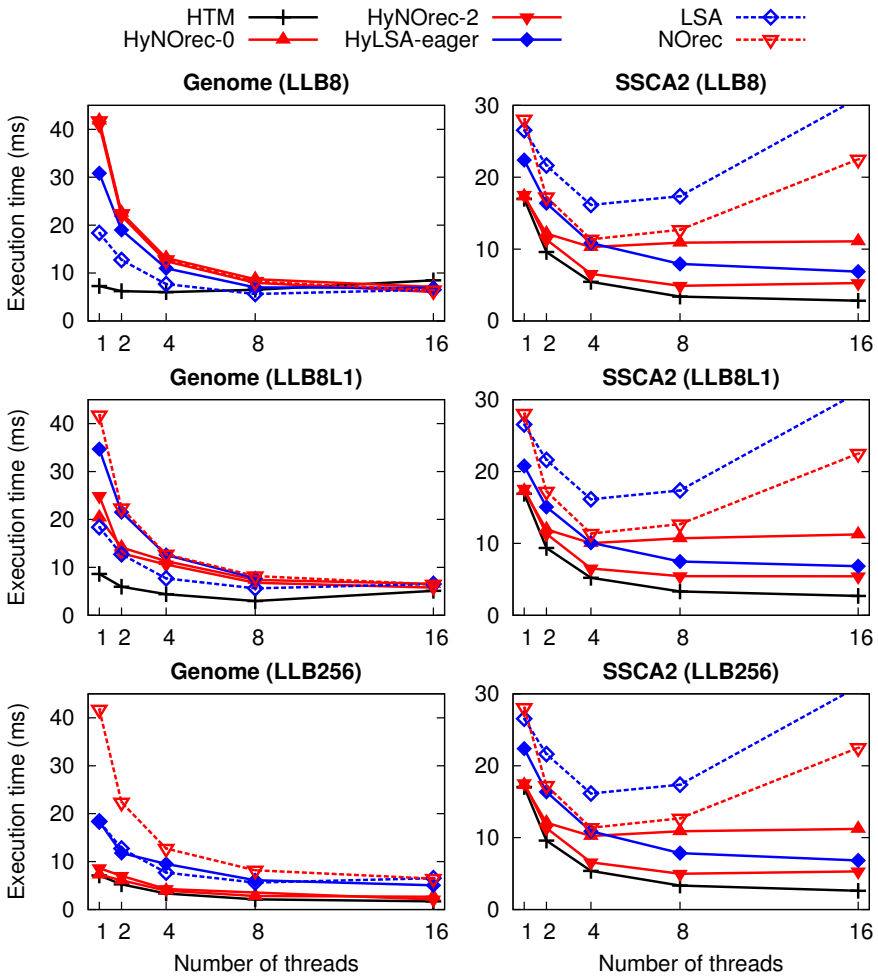

Figure 7.22: Comparison of the performance of the HTM, STM, and HyTMs with Genome and SSCA2.

among the HyTMs and STMs and almost as good as HTM. It also scales better than HyNOrec-0. Vacation and Genome on LLB256 as well as SSCA2 show similar performance patterns. Vacation on LLB8 and LLB8L1 suffers from a low hardware transaction ratio, which limits HyTMs to the performance of the respective STMs and prevents scalability for HTM. With Genome on LLB8 and LLB8L1, HyNOrec is slowed down compared to HyLSA because NOrec is slower than LSA for this benchmark. In all other configurations, HyLSA causes a higher runtime overhead than HyNOrec for hardware transactions, but typically scales well; its performance can be worse than LSA's in some cases due to unsuccessful attempts to run hardware transactions or due to larger conflict-detection granularity.

Overall, TM performance with the STAMP applications thus follows the same general trends as with the IntegerSet benchmarks. HyNOrec-2 performs best among the HyTMs most of the time and typically close to or better than HTM. The results show that my HyTM algorithms indeed improve upon previous HyTM algorithms as shown in Table 7.4 on page 198; either by allowing for a larger level of concurrency between hardware and software transactions, by reducing runtime overhead of hardware transactions, or by requiring less HTM capacity and thus allowing more transactions to run with hardware acceleration.

While previous HyTM designs have used nonspeculative memory accesses inside of hardware transactions, my results show that this has a much larger potential and importance if algorithms also make use of nonspeculative atomic read-modify-write instructions. I believe that the general-purpose techniques that I used in my algorithms (Table 7.3) apply not just to HyTM but can be useful in general for concurrent algorithms based on new synchronization hardware like ASF.

Chapter 8

Conclusion

I have investigated, implemented, and evaluated major building blocks that are required for a high-performance, practical, and realistic TM: Efficient STM algorithms, compiler support and compile-time optimizations, integration of first-generation HTMs, efficient HyTM algorithms, and precise requirements for TM implementations.

I focused on TM for C/C++ userspace applications, which allows programmers to synchronize using transactional programming-language constructs that extend the C++11 memory model. I also made no assumptions about the kind of parallelization used in the applications. This puts my work into the context of mainstream programming, and allows for a larger applicability of the results of my work. C/C++ are also low-level languages in terms of allowing programmers to work close to the hardware, which in turn results in stronger constraints for the implementations of such languages compared to, for example, managed environments such as Java; for TM implementations, this means that they, for example, have to deal with pointers and cannot expect to be able to control or transform most of the code that is executed. Thus, this makes it more likely that results for C/C++ TM implementations could also be applicable in other languages, even though the latter might allow for different optimizations.[1]

Furthermore, I have focused on standard hardware and realistic first-generation HTM, and did not expect custom support for TM by other parts of the execution environment such as the operating system. This is important to make TM practical for programmers, in particular for the early adopters that TM needs to overcome the chicken-and-egg situation it faces. While the software prototypes that I have built are not directly used by people outside of the TM research community (but were valuable for research), my work has influenced GCC's TM implementation, for example: GCC's TM runtime library uses LSA and follows its implementation discussed in Section 5.2, and the compiler adheres to the TM implementation requirements laid out in Section 4.2.3. This shows that the TM implementation stack comprised by the building blocks that I have presented is realistic in the sense that it is feasible to integrate it with current commercial systems software.

[1] One notable aspect that might differ in other languages is the behavior of programs with data races: C/C++ simply allows undefined behavior for such programs, whereas languages such as Java require stronger guarantees, which might require changes to the TM implementation that I have presented.

Likewise, the transactional language constructs that I used in my work follow a specification that is the result of a joint effort of a large part of the TM community and also aims at providing practical and efficient TM programming abstractions.

Regarding performance, I focused on general-purpose optimizations without relying on programmer-supplied optimization hints. This choice was also motivated by the aim at practical TM building blocks. The STM and HyTM algorithms that I have presented provide state-of-the-art performance; they significantly improve upon prior work by showing ways to avoid major sources of runtime overhead and scalability limitations (e. g., via time-based validation or by exploiting nonspeculative operations in hardware transactions). However, while especially the HyTMs provide good performance, the STMs still face significant performance challenges, primarily caused by single-thread runtime overheads and quiescence-based privatization safety implementations. Nonetheless, further optimizations seem to be possible, in particular those based on compile-time analyses and specialization.

In summary, the building blocks thus comprise a good first-generation TM implementation that is sufficient for early adopters to provide feedback and to start using transactions in C/C++, potentially even in noncritical production deployments. Its high level of integration with the programming language and its compatibility with existing systems provide for a large amount of the level of usability that TM promises. However, whether the current level of performance is sufficient for TM to provide a good usability–performance trade-off in practice, is something that depends on how programmers will use TM and how the performance of HTMs will develop over time. I will discuss this issue further at the end of this chapter.

In what follows, I will assess my work's individual contributions to the state of the art, discuss open questions and future work, and provide an outlook for TM.

STM algorithms. The time-based validation technique presented in Section 5.1 influenced the design of STMs significantly because it showed how to implement always-consistent atomic snapshots very efficiently with invisible reads by relying on a global time base. Combined with pessimistic concurrency control for write operations and the use of multiple ownership records, this yields one of the most important STM implementation variants for C/C++ (see Section 5.2), and is used in commercial TM implementations.

The performance of such an STM can scale well to larger numbers of threads, but also suffers from single-thread runtime overheads and scalability deficiencies in current privatization safety implementations. It is also heavily influenced by the quality of the hash function used to map memory locations to ownership records; as I have shown in Section 5.2.2, multiplicative hashing can provide a better mapping with lower space overhead compared to the simple hash functions used in prior work.

Even though the global time base used by such STMs is a central shared component, the benefits of time-based validation often outweigh this potential bottleneck. Furthermore, it can also be implemented using imperfectly synchronized hardware clocks (see Section 5.3), which are much easier to parallelize.[2]

[2] Whether this will become the primary way to implement the STM's time base will depend

Finally, systems such as Google's Percolator (see Section 5.4) show that time-based validation based on centralized global time bases are useful even when implementing transactions on large distributed systems.

Requirements for TM implementations. The requirements that I present in Section 4.2 provide no benefits to programmers using transactions but instead provide guidance to TM implementers. They are split into responsibilities of compilers and TM runtime libraries at the boundary of a TM runtime library ABI. Thus, this also defines the previously unspecified semantics of this ABI, which is important in practice.[3]

Different to most of the prior work on correctness criteria for TM implementations, these TM implementation requirements are based on the semantics and correctness concepts of the programming language hosting the transactional language constructs (e. g., on the C++ as–if rule). This is useful to TM implementers because it expresses the trade-offs in the design of these requirements more clearly and in terms meaningful in the rest of the language's implementation.

Compile-time optimizations based on partitioning. These divide–and–conquer optimizations have a lot of potential and result in performance benefits even with simple prototypes, but they also face significant implementation obstacles in practice. While they are transparent to programmers due to relying on automatic partitioning of application data, the compiler needs to be able to analyze the whole program.

Partitioning-aware STMs incur additional runtime overheads (see Section 6.2) but allow for tuning the performance of individual partitions separately; however, this also means that they need high-quality automatic tuning and heterogeneous workloads to overcome the overheads. The tuning policies that I evaluated are too simple to be of general use in practice. Likewise, I have not investigated compiler optimizations aimed specifically at reducing the runtime overheads of per-partition tuning.[4] TM metadata colocation (see Section 6.3) provides some speedup over simple memory–to–orec mappings but requires a rather complex transformation of programs (e. g., the change of memory allocation).

Thus, the prototypes show the potential of partitioning, but only further investigation with more advanced compilers and non-benchmark workloads will show how much of the potential performance benefits can be realized in practice. Open questions are, for example, whether link-time optimizations and

on the communication overheads in future hardware and the availability of reliably synchronized hardware clocks. The current trends in hardware are that communication becomes more costly, but hardware clocks in mainstream CPUs are also getting more tightly synchronized. Furthermore, this of course also depends on characteristics of future TM workloads and on whether these workloads can be partitioned into smaller pieces with higher locality (e. g., using the techniques presented in Chapter 6).

[3]Having precisely defined ABIs is necessary for cross-vendor compatibility between compilers and between runtime libraries; otherwise, deploying executable code that uses dynamic linking would only work correctly if all pieces would have been generated by the same compiler version.

[4]For example, my prototypes check the type of the associated partition on every access; it would likely be more efficient to select a TM variant for groups of accesses, which should be possible in many cases (e. g., for operations such as those in the IntegerSet benchmarks that access just a single partition).

whole-program analysis will become more common, how to deal with library code, and whether the implementation complexity required by partitioning is feasible. Nonetheless, I believe that partitioning will remain important, even if just to break larger applications down into smaller parts to increase locality and scalability. Finally, other TM building blocks such as transaction schedulers or contention managers could also benefit from the additional information provided by partitioning.

Hybrid TM algorithms on HTMs with nonspeculative operations. The first-generation HTM proposal that my work is based on, AMD's Advanced Synchronization Facility (ASF), differs from many other HTMs in that it allows nonspeculative memory accesses to be executed as part of hardware transactions. This adds complexity to TM stacks that want to use ASF, but the implementation can be confined to a TM runtime library that uses the standard ABI as long as the HTM avoids certain shortcomings in its design (see Section 7.2).

Nonspeculative operations in hardware transactions are tremendously useful for hybrid software/hardware TMs (HyTMs) because they allow hardware transactions to communicate with other threads without having to abort. ASF also supports atomic read–modify–write operations such as CAS to execute nonspeculatively, which provides additional benefit. Furthermore, it is also important that the HTM monitors speculatively accessed locations eagerly for conflicting accesses by other threads because this can be used to ensure consistency of nonspeculative operations (e. g., see Algorithm 10).

I have constructed two HyTM algorithms, HyLSA and HyNOrec-2, that use nonspeculative operations to improve significantly over prior work (see Section 7.3). HyLSA is similar to prior HyTMs that use multiple ownership records but read application data nonspeculatively, which—in combination with a coarse-granular memory–to–orec mapping—allows it to increase the effective capacity for transactional reads beyond the HTM's capacity.

HyNOrec-2 executes hardware transactions in such a way that they are not aborted by commits of concurrent nonconflicting software transactions. At the same time, these hardware transactions execute with very low overhead compared to pure HTMs that cannot execute software transactions concurrently; in particular, hardware transactions essentially require the same HTM capacity as with a pure HTM, and the additional runtime overhead for transactional memory accesses is very small. This allows HyNOrec-2 to often perform similar to pure HTMs when all transactions can execute as hardware transactions, and to provide much better scalability than pure HTM or prior HyTM algorithms otherwise.

Even though the HTM facilities currently announced for mainstream processors do not yet allow nonspeculative operations in transactions, my HyTM algorithms show the benefit that this would provide to HyTMs and to transaction-based synchronization algorithms in general.

Open questions and future work. Besides the follow-up work suggested previously, there are several open questions that future work should try to answer and other TM building blocks that future TM stacks should include.

First of all, the temporal aspect of TM-based synchronization (see Section 3.1.2) needs to be further investigated, in particular transaction scheduling,

contention management, conflict resolution policies, and when to use pessimistic or optimistic concurrency control.

However, it is difficult to, for example, know which transactions should be prioritized in a parallel program when the program's performance utility function is unknown. Integrating TM and programming abstractions for parallelization could provide the TM with better insight into what the program actually wants to execute in parallel (e. g., with task-based parallelism, transactions on the critical path in the task graph could be executed with a higher priority).

Second, the quality of automatic performance tuning of TM needs to improve considerably. Currently, most tuning strategies are more or less rules of thumb or heuristics that have not been validated on many benchmarks. Thus, better and more benchmarks are needed so that tuning strategies can be tested on a hopefully more representative sample of all workloads that might appear in practice.

Furthermore, we also need a better understanding and classification of workloads to be able to predict which performance effect a certain tuning decision would have. This needs to include both building TM performance models offline and figuring out how TM implementations can detect at runtime which kind of workload they execute. Valuable inputs to any automatic classification could be, for example, the parallelization approach in a program, compile-time analyses on transactions (e. g., whether they access many or few memory locations), or programmer-supplied hints.

Once TM workloads are better understood, including which kinds are important in practice, it would also be beneficial to investigate specializations of TM aimed at certain workloads.

Finally, there is a need for more studies about what remains of the promises of TM once it is put into the hands of real programmers. The few existing studies such as the one by Pankratius and Adl-Tabatabai [84] show that TM can provide advantages compared to other programming abstractions, but they have not evaluated TM in the context of commercial applications.

Outlook for TM. TM has not yet overcome the chicken-and-egg situation that many new programming abstractions face, but it is still making continuous progress towards being widely available. Major hardware vendors have announced hardware support for transactions for upcoming mainstream CPUs, which can indicate that these vendors think that TM will become useful. Even if these HTMs are meant to rather enable lock elision than transactional constructs in mainstream programming languages, it will still benefit the latter due to the better performance that HTM can result in.

The existence of a study group on TM in the ISO C++ committee as well as the participation of several large companies in the C++ TM specification drafting group show that there is also significant interest for transactional programming abstractions on the software side.

The future importance of TM will also be determined by how synchronization in general will develop over time. This, in turn, depends a lot on which parallelization paradigms will be most important, and which role synchronization will take in those. Likewise, the shape of future hardware architectures will of course also affect at least how synchronization is implemented; nonetheless, transactions could be still useful programming abstractions in larger or

more loosely coupled systems (e. g., with only partially cache-coherent memory) because of how conveniently they allow to express atomicity requirements.

To summarize, while TM can definitely be implemented and continues to be a promising approach, it is still too early for a final judgement on the usefulness and importance of TM. We need to first let more programmers get in touch with optimized TM implementations and first-generation hardware support for TM, so that they can evaluate TM in real applications and give feedback to TM implementers. This feedback cycle needs more time, both for early adoption and evaluation and for further refinement of TM implementations and the programming-language constructs. Only after this we will be able to judge how often TM is a useful tool for programmers in the sense of providing them with the trade-off between performance and ease of use that they are looking for. Thus, while my work does not answer whether the TM idea will be widely useful, it has contributed to make it possible to answer this question in the future.

Bibliography

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press.

[2] Advanced Micro Devices, Inc. *Advanced Synchronization Facility - Proposed Architectural Specification*, 2.1 edition, Mar. 2009.

[3] Y. Afek, U. Drepper, P. Felber, C. Fetzer, V. Gramoli, M. Hohmuth, E. Riviere, P. Stenstrom, O. Unsal, W. M. Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Pohlack, A. Cristal, I. Hur, A. Dragojevic, R. Guerraoui, M. Kapalka, S. Tomic, G. Korland, N. Shavit, M. Nowack, and T. Riegel. The Velox Transactional Memory Stack. *Micro, IEEE*, 30(5):76 –87, 2010.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Readings in computer architecture*, 2000.

[5] W. Baek, C. Cao Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM Transactional Application Programming Interface. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ Concurrency: The Post-Rapperswil Model. Technical Report N3132, ISO IEC JTC1/SC22/WG21, Aug. 2010. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3132.pdf`.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 117–128, New York, NY, USA, 2000. ACM.

[9] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing Transactions: The Subtleties of Atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, Jun 2005.

[10] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[11] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[12] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, Mar. 2009.

[13] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 304–315, New York, NY, USA, 2008. ACM.

[14] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 27–40, New York, NY, USA, 2010. ACM.

[15] J. Chung, D. Christie, M. Pohlack, S. Diestelhorst, M. Hohmuth, and L. Yen. Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support. In *TRANSACT*, 2010.

[16] C. Click. Azul's Experiences with Hardware Transactional Memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, Jan. 2009.

[17] F. Cristian. A probabilistic approach to distributed clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[18] L. Crowl, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Integrating Transactional Memory into C++. In *TRANSACT 2007*, 2007.

[19] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2011.

[20] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional mutex locks. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 2–13, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] L. Dalessandro and M. L. Scott. Sandboxing transactional memory. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 171–180, New York, NY, USA, 2012. ACM.

[22] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 67–78, New York, NY, USA, 2010. ACM.

[23] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[24] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *Computer*, 21:57–69, 1988.

[25] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 325–334, New York, NY, USA, 2010. ACM.

[26] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168, New York, NY, USA, 2009. ACM.

[27] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006.

[28] D. Dice and N. Shavit. What really makes transactions fast? In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT '06)*, June 2006.

[29] S. Diestelhorst. Personal communication, 2010.

[30] S. Diestelhorst and M. Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *Proceedings of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM)*, Boston, MA, USA, Apr. 2008.

[31] S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, J.-W. Chung, and L. Yen. Implementing AMD's Advanced Synchronization Facility in an out-of-order x86 core. In *TRANSACT*, 2010.

[32] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19 – 51, 1997.

[33] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele, Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 216–224, New York, NY, USA, 2004. ACM.

[34] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[35] U. Drepper. What Every Programmer Should Know About Memory, 2007.

[36] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–296, New York, NY, USA, 2007. ACM.

[37] R. Ennals. Software Transactional Memory Should Not Be Obstruction-Free.

[38] J. L. Eppinger and L. B. Mummert, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann Publishers, 1991.

[39] P. Felber, C. Fetzer, P. Marlier, M. Nowack, and T. Riegel. Brief Announcement: Hybrid Time-Based Transactional Memory. In N. Lynch and A. Shvartsman, editors, *Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 124–126. Springer Berlin / Heidelberg, 2010. The full version is available as technical report TUD-FI10-06-Nov.2010.

[40] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based Software Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.*, 21:1793–1807, December 2010.

[41] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *TRANSACT*, August 2007.

[42] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[43] C. Fetzer and F. Cristian. Integrating External and Internal Clock Synchronization. *Real-Time Syst.*, 12(2):123–171, Feb. 1997.

[44] *TM support in the GNU Compiler Collection*. `http://gcc.gnu.org/wiki/TransactionalMemory`.

[45] R. Guerraoui, T. A. Henzinger, M. Kapalka, and V. Singh. Transactions in the jungle. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 263–272, New York, NY, USA, 2010. ACM.

[46] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, pages 303–323, Berlin, Heidelberg, 2005. Springer-Verlag.

[47] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[48] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-Based Lock Allocation. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 353–364, Washington, DC, USA, 2007. IEEE Computer Society.

[49] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of OOPSLA*, pages 388–402, Oct 2003.

[50] T. Harris, J. Larus, and R. Rajwar, editors. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[51] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *PLDI '06: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 14–25, New York, NY, USA, June 2006. ACM Press.

[52] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP '90, pages 197–206, New York, NY, USA, 1990. ACM.

[53] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[54] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[55] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[56] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[57] M. Herlihy and N. Shavit. On the Nature of Progress. In A. Fernandez Anta, G. Lipari, and M. Roy, editors, *Principles of Distributed Systems*, volume 7109 of *Lecture Notes in Computer Science*, pages 313–328. Springer Berlin / Heidelberg, 2011.

[58] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[59] M. Hicks, J. S. Foster, and P. Pratikakis. Lock Inference for Atomic Sections. In *TRANSACT*, 2006.

[60] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 145–156, New York, NY, USA, 2009. ACM.

[61] R. Holt (SGI). Personal communication, 2007.

[62] Intel. *Intel Transactional Memory Compiler and Runtime Application Binary Interface*. Intel, 1.0.1 edition, Nov. 2008.

[63] Intel. *Draft Specification of Transactional Language Constructs for C++*. Intel, IBM, Sun, 1.1 edition, Feb. 2012.

[64] Intel. *Intel Architecture Instruction Set Extensions Programming Reference*. Intel, 319433-012a edition, Feb. 2012.

[65] ISO JTC 1/SC 22. *Programming Languages — C++*, 14882:2011 edition, 2011.

[66] ISO JTC 1/SC 22/WG21 SG5. `http://isocpp.org`.

[67] ISO JTC 1/SC22. *Programming Languages — C*, WG14 N 1539 edition, Nov. 2010.

[68] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[69] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220, New York, NY, USA, 2006. ACM Press.

[70] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[71] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[72] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chigago, Illinois, June 2005.

[73] Y. Lev and J.-W. Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 197–206, New York, NY, USA, 2008. ACM.

[74] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[75] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, Mar. 1988.

[76] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM.

[77] S. Mannarswamy, D. R. Chakrabarti, K. Rajan, and S. Saraswati. Compiler aided selective lock assignment for improving the performance of software transactional memory. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 37–46, New York, NY, USA, 2010. ACM.

[78] S. Mannarswamy and R. Govindarajan. Making STMs Cache Friendly with Compiler Transformations. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 232–242, Washington, DC, USA, 2011. IEEE Computer Society.

[79] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, pages 354–368, Berlin, Heidelberg, 2005. Springer-Verlag.

[80] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 67–74, Washington, DC, USA, 2008. IEEE Computer Society.

[81] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–358, New York, NY, USA, 2006. ACM Press.

[82] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 314–325, New York, NY, USA, 2008. ACM.

[83] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *OOPSLA '08: Proceedings of the 23rd annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, 2008.

[84] V. Pankratius and A.-R. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 43–52, New York, NY, USA, 2011. ACM.

[85] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[86] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[87] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[88] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-Aware Transactions for Increased Concurrency. In *Proceedings of the 41st International Symposium on Microarchitecture (MICRO)*, 2008.

[89] T. Riegel and D. Becker de Brum. Making Object-Based STM Practical in Unmanaged Environments. In *TRANSACT 2008*, 2008.

[90] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[91] T. Riegel, P. Felber, and C. Fetzer. Composable Error Recovery with Transactional Memory. *Bulletin of the European Association for Theoretical Computer Science*, 2009.

[92] T. Riegel, C. Fetzer, and P. Felber. Snapshot Isolation for Software Transactional Memory. In *TRANSACT*, June 2006.

[93] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[94] T. Riegel, C. Fetzer, and P. Felber. Automatic Data Partitioning in Software Transactional Memories. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 152–159, New York, NY, USA, 2008. ACM.

[95] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. Technical Report TUD-FI10-06-Nov.2010, Technische Universität Dresden, November 2010. Full version of the DISC 2010 brief announcement.

[96] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

[97] C. J. Rossbach, O. S. Hoffman, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in the Operating System. In *21st ACM Symposium on Operating Systems Principles*, 2007.

[98] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[99] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *International Symposium on Microarchitecture (MICRO'06)*, 2006.

[100] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[101] W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM Press.

[102] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53:89–97, July 2010.

[103] N. Shavit and D. Touitou. Software Transactional Memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, Aug. 1995. ACM.

[104] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards transactional memory semantics for C++. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 49–58, New York, NY, USA, 2009. ACM.

[105] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, NY, USA, 2007. ACM Press.

[106] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 104–115, New York, NY, USA, 2007. ACM.

[107] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, 8(1):66–73, Jan. 1991.

[108] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 273–283, New York, NY, USA, 2010. ACM.

[109] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *20th Intl. Symp. on Distributed Computing (DISC)*, 2006.

[110] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2007.

[111] J. Sreeram and S. Pande. Hybrid Transactions: Lock Allocation and Assignment for Irrevocability. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 1192–1203, Washington, DC, USA, 2012. IEEE Computer Society.

[112] STM.NET. `http://blogs.msdn.com/b/stmteam/`.

[113] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking Zero-Indirection Transactional Memory. In *TRANSACT*, 2007.

[114] G. Upadhyaya, S. P. Midkiff, and V. S. Pai. Using data structure knowledge for efficient lock generation and strong atomicity. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 281–292, New York, NY, USA, 2010. ACM.

[115] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: safe I/O in memory transactions. In *EuroSys '09: Proceedings of the fourth ACM european conference on Computer systems*, pages 247–260, New York, NY, USA, 2009. ACM.

[116] J.-T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber. RobuSTM: A Robust Software Transactional Memory. In S. Dolev, J. Cobb, M. Fischer, and

M. Yung, editors, *Stabilization, Safety, and Security of Distributed Systems: 12th International Symposium (SSS 2010)*, volume 6366 of *Lecture Notes in Computer Science*, pages 388–404. Springer Berlin / Heidelberg, September 2010.

[117] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[118] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *International Symposium on Code Generation and Optimization (CGO)*, 2007.

[119] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[120] V. Ying, C. Wang, Y. Wu, and X. Jiang. Dynamic Binary Translation and Optimization of Legacy Library Code in an STM Compilation Environment. In *WBIA*, 2006.

[121] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 265–274, New York, NY, USA, 2008. ACM.

[122] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007.

[123] R. Zhang, Z. Budimlić, and W. N. Scherer III. Commit Phase in Timestamp-Based STM. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 326–335, New York, NY, USA, 2008. ACM.

[124] Y. Zhang, V. C. Sreedhar, W. Zhu, V. Sarkar, and G. R. Gao. Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections. In *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pages 141–155. Springer-Verlag, Berlin, Heidelberg, 2008.

[125] C. Zilles and R. Rajwar. Implications of False Conflict Rate Trends for Robust Software Transactional Memory. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 15–24, 2007.

Index