Université de Neuchâtel
Faculté des Sciences
Institut d'Informatique

Efficient Transactional Memory Runtimes for Unmanaged Environments

par

Patrick Marlier

Thèse

présentée à la Faculté des Sciences
pour l'obtention du grade de Docteur ès Sciences

Acceptée sur proposition du jury:

Prof. Pascal Felber, Directeur de thèse
Université de Neuchâtel, Suisse

Prof. Peter Kropf
Université de Neuchâtel, Suisse

Dr. Gilles Muller
INRIA/LIP6, France

Dr. Osman Unsal
Barcelona Supercomputing Center
Microsoft Research Centre, España

Dr. Etienne Rivière
Université de Neuchâtel, Suisse

Soutenue le 12 août 2011

Université de Neuchâtel

FACULTE DES SCIENCES
Secrétariat-Décanat de la faculté
Rue Emile-Argand 11
CH-2000 Neuchâtel

IMPRIMATUR POUR LA THESE

Efficient Transactional Memory Runtimes for Unmanaged Environments

Patrick Marlier

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel,
sur le rapport des membres du jury

MM. P. Felber (directeur de thèse), P. Kropf, E. Rivière,
G. Muller (INRIA, Paris)
et O. Unsal (Barcelona Supercomputing Center)

autorise l'impression de la présente thèse.

Neuchâtel, le 19 août 2011

Le doyen :
P. Kropf

Téléphone : +41 32 718 21 00 E-mail : secretariat.sciences@unine.ch www.unine.ch/sciences

ACKNOWLEDGMENTS

First of all, I am truly grateful to Gilles Muller for being part of my jury and for interrupting his vacation for my PhD defense.

I would like to thank Osman Unsal for being part of my jury and for coming from Barcelona to my PhD defense.

I would also like to thank my advisor, Pascal Felber, for his help throughout my PhD. I would especially like to thank him for his availability, his knowledge, his motivation, and his enthusiasm. I will keep good memories of some sleepless nights we have been working together to meet conference deadlines.

I would like to thank the doyen of the science university, Peter Kropf, for his dedicated time, his kindness, and for being part of my jury.

I would like to express my gratitude to Etienne Rivière for all his valuable feedbacks for the dissertation, for proofreading it, and for being part of my jury.

The research leading to the results presented in this dissertation has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under the VELOX Project, grant agreement No 216852. I am grateful to the EU for this support. A special thank for the Velox group and particularly to Martin Nowack and Javier Arias for the great team work.

My sincere thanks also go to the Department of Computer Science at University of Neuchâtel for the excellent working environment and for the typical "Sorties de l'institut". A special thought for my office mate, Derin Harmanci and also to Vincent Gramoli, Walther Maldonado and Heiko Sturzrehm for the interesting and helpful discussions about Transactional Memory. I would like to thank all my colleagues for the good times spent and, for the Tuesday volleyball sessions and Friday soccer sessions.

Je tiens aussi à adresser un remerciement particulier à toute ma famille pour leurs encouragements mais aussi à mes parents pour l'éducation qu'ils m'ont donné et pour m'avoir offert la possibilité de continuer mes études jusqu'à ce doctorat. Last but definitely not least, I would thank my wife, Nathalie, for her love and unconditional support. Without her I would not have started this PhD. Thank you so much for following me everywhere, and for your encouragement.

RÉSUMÉ

Pour profiter pleinement de la puissance de calcul des processeurs multi-cœurs, les programmeurs doivent utiliser la programmation concurrente. Cependant, l'utilisation des verrous qui est la méthode de programmation concurrente la plus répandue, est particulièrement difficile à maîtriser. C'est pourquoi il est nécessaire d'utiliser des alternatives aux verrous. Un des paradigmes le plus prometteur est la Mémoire Transactionnelle, qui permet l'exécution optimiste du programme en utilisant le concept des transactions. Dans cette thèse, nous proposons d'améliorer le support et la performance de la mémoire transactionnelle dans des environnements non-supervisés, aussi bien au niveau logiciel qu'au niveau matériel.
D'abord, nous améliorons la performance de la mémoire transactionnelle logicielle en developpant LSA, un algorithme basé sur une horloge virtuelle pour assurer la cohérence des transactions. Nous proposons plusieurs optimisations pour augmenter l'efficacité des transactions et nous développons de nouvelles fonctionnalités dans le but de favoriser l'utilisation de la mémoire transactionnelle par les développeurs d'applications.
Ensuite, nous tirons parti du support matériel pour la mémoire transactionnelle afin d'améliorer les performances d'exécution des transactions. Nous montrons que ce support permet d'obtenir de meilleurs résultats par rapport aux approches purement logicielles. Cependant, les capacités limitées du matériel nous tournent vers une approche hybride. Notre mémoire transactionnelle hybride qui utilise l'algorithme LSA combine les performances de l'approche matérielle avec les capacités de l'approche logicielle pour outrepasser les limitations du matériel.
Finalement, nous intégrons la mémoire transactionnelle dans un ensemble logiciel. Nous décrivons la standardisation de la mémoire transactionnelle dans les langages C et C++ ainsi que l'interface binaire pour les bibliothèques transactionnelles. Nous étendons notre bibliothèque transactionnelle pour suivre ces spécifications et la rendre compatible avec les compilateurs qui supportent la mémoire transactionnelle dont le compilateur GCC. Le système qui en résulte fournit aux développeurs une solution facile et efficace pour créer des applications qui tirent avantage des processeurs multi-cœurs.

Mots-clés: Multi-cœur, Programmation Concurrente, Exécution Optimiste, Mémoire Transactionnelle Logicielle, Mémoire Transactionnelle Matérielle, Mémoire Transactionnelle Hybride, Intégration Système.

ABSTRACT

The adoption of multi-core processors requires programmers to use concurrent programming to fully benefit from the available processing power. Efficient concurrent programming using locks is notoriously difficult to master. This makes the case for alternative concurrent programming paradigms. One of the most promising of these paradigms is Transactional Memory, which uses optimistic execution of code via the concept of transactions. In this thesis, we propose to improve the support and performance of transactional memory for unmanaged environment, at all levels of the system software and hardware stack.

First, we improve the performance of software transactional memory by developing LSA, an algorithm based on a virtual clock to ensure transaction consistency. In this context, we propose several optimizations for efficiency and develop features that will favor the uptake and usability of transactional memory for application developers.

Next, we extend our Transactional Memory library to leverage the availability of hardware mechanisms that can support the execution of transactions. We show that Hardware Transactional Memory can deliver a high performance compared to software-only approaches but suffers from several limitations. Our Hybrid Transactional Memory, extending on our LSA algorithm, combines the advantages of hardware and software transactional memory to achieve a performance close to pure hardware transactional memory while overcoming its limitations.

Finally, we describe the integration of transactional memory in a complete system stack. We describe the standardization of the C/C++ language transactional constructs and the binary interface for transactional memory runtimes. We extend our Transactional Memory library to follow these specifications and make it compliant with two transactional compilers, including GCC. The resulting framework provides developers with an easy and efficient way to create applications that can take advantage of multi-core processors.

Keywords: Multi-core, Concurrency, Optimistic Execution, Software Transactional Memory, Hardware Transactional Memory, Hybrid Transactional Memory, System Integration.

Contents

Chapter 1

Introduction

The shift towards multi-core

From the beginning of the computer era, programs have been executing sequentially, i. e., running instructions one by one in order. When it was necessary for a program to execute faster, the deal was simple: wait for the next generation of processors with increased clock speed to process more instructions per second and execute the application faster.

This steady increase in clock speed has continued for decades together with the well-known Moore's law, which states that the number of transistors that can be placed on an integrated circuit doubles approximately every two years. Figure 1.1[1] illustrates that this empirical law has closely matched reality.

Things have changed in 2004, however, after CPU architects got into troubles because of physical problems and were no longer able to increase frequency as we can see in Figure 1.2. Indeed, high clock speeds require a greater amount of power and very small internal component sizes (wire, transistors, etc.), and thus dissipate more heat. Instruction level parallelism has reached a state where all optimizations are now complex, prone to error, and with a high energy cost.

Therefore, major CPU manufacturers followed another strategy: instead of increasing the frequency of a single processor but retaining the ability to add more transistors on a chip, they introduced the use of multiple processors, or cores, on the same chip (multi-core CPUs). 2005 was the year of massive introduction of multi-core processors for Intel, as can be observed in Figure 1.2.

Unfortunately, the availability of multiple cores does not necessarily allow running straightforwardly existing programs faster (e. g., text processor, internet browser, etc.). While it is straightforward to run several different programs in parallel, each on one of the available cores, there is a high complexity associated with the use of multiple cores within the same program.

Multi-core architectures

Multiple cores on the same die is the standard for a few years now and software programmers are expected to invest much effort in parallelizing existing applications. The number of cores

[1] Picture under the creative commons license, available at `http://en.wikipedia.org/wiki/Moore%27s_law`

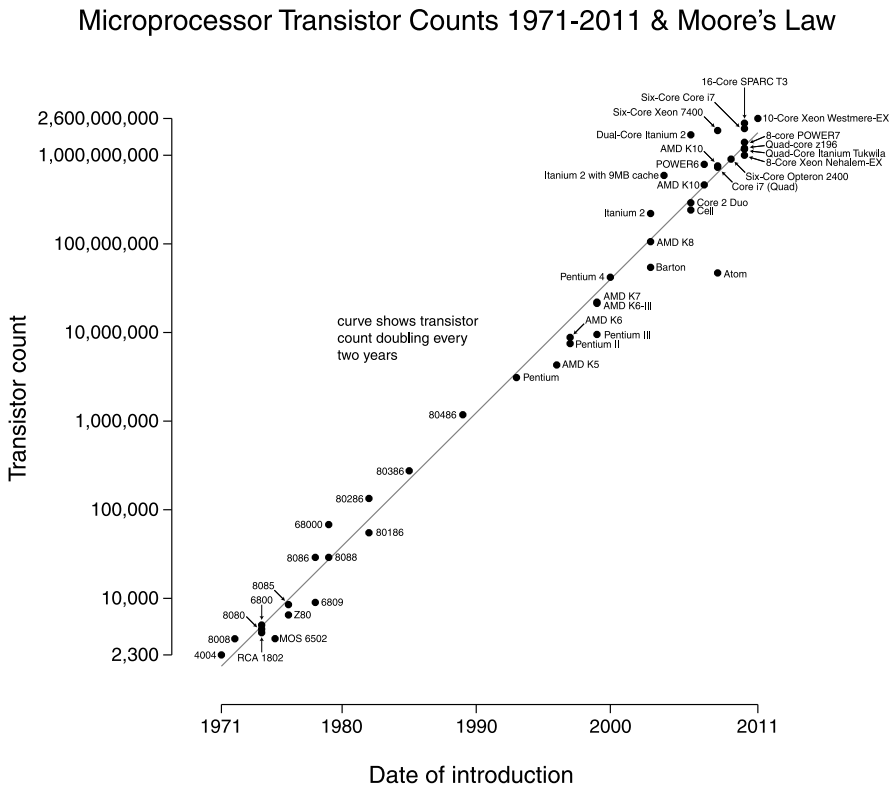


Figure 1.1: Number of transistors in CPU against dates of introduction and Moore's law.

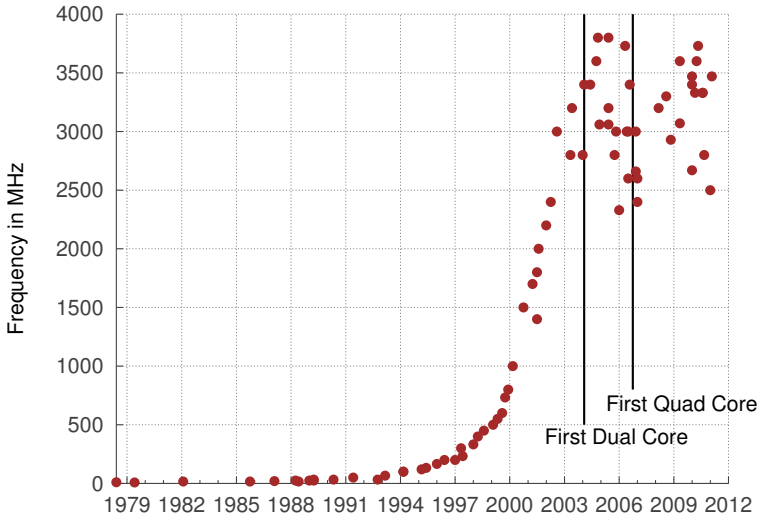


Figure 1.2: CPU frequency of Intel processors against dates of introduction.

per processor is expected to increase with each new processor generation. Using all cores available is now crucial to continue to raise software performance. Unfortunately, the number of parallel applications and parallel algorithms is limited because of the inherent complexity associated with parallelization.

Shared-memory synchronization plays a big role in parallel software, either when synchronizing and merging results of parallel tasks, or when parallelizing programs by speculatively executing tasks concurrently. Indeed, an application running on multiple cores will have several sequences of operations (called *threads*) executing concurrently and accessing shared data. This requires synchronization between the threads running on the different cores and identification of parallel work to assign to threads.

Until now, most concurrent programs have been programmed using lock-based synchronization. Yet, locks are considered difficult to use for the average programmer, especially when locking at a fine granularity to provide scalable performance. Lock-based synchronization is also sensitive to programming mistakes leading difficult to reproduce errors such as deadlocks. This is particularly important when considering that large classes of programs will have to be parallelized by programmers who are not well trained in concurrent programming.

A promising way to increase the parallel part of a program is to use *speculation*. The idea is to execute blocks of code that could conflict (i. e., read or modify the same data) with blocks executed by other cores in such a way that conflicts are detected dynamically and state changes are only committed if it is guaranteed that there was no conflict. Speculation is an optimistic synchronization strategy that is especially helpful for improving the degree of parallelism in the following scenarios: first, when there is a good chance that two code blocks do not conflict because, for example, they access different memory locations; and second, when there is no easy way to predict at compile time if and when two code blocks will conflict, and pessimistic strategies like fine-grained locking unnecessarily limit scalability.

Transactional Memory

A few years ago, Herlihy and Moss [31] introduced the concept of dynamic transactional memory as a scalable alternative to locks. *Transactional memory* (TM) [29] is a shared-memory synchronization mechanism that supports speculation at the level of individual memory accesses. It is one of the most promising approaches for exploiting emerging multi-core architectures, as it provides a simple-to-use, safe, and scalable paradigm for concurrent programming. TM is not a new programming language but it can be proposed as an extension of existing languages to solve problems of concurrency.

It allows programmers to group any number of memory accesses into transactions, which are executed speculatively and take effect atomically only if there have been no conflicts with concurrent transactions executed by other threads. In the case of conflicts, transactions are rolled back and restarted. In programming languages, one can introduce *atomic block* constructs that are directly mapped onto transactions. Atomic blocks are also likely to be easier to use for programmers than other mechanisms such as fine-grained locking because they only specify what is required to be atomic but not how this is implemented.

In Figure 1.1, the atomic block defined by the "atomic" keyword ensures that all operations in it will appear to take effect atomically to external observers. The atomicity

```
void insert(node prev, node newNode) {
  atomic {
    newNode->next = prev->next;
    prev->next = newNode;
  }
}
```

Listing 1.1: Example of transactions, insertion into a Linked List

guarantees that modifications appear to other threads either in their entirety or not at all. Then, the linked list is consistent because all others parallel modifications will not conflict.

1.1 Motivations and objectives

First, applications do not take advantage of the full potential of modern multi-core processors and this trend is likely to worsen as core count increases. Since the shift toward to multi-cores, programs did not change much, i. e., most are still sequential and thus only one core is used by each program. Software is still one step behind the hardware because multi-core programming is difficult for most of developers.

TM promises an easier way to develop applications and to use the full computation capacity of multi-core processors. TM is still in its incubation phase and is moving slowly and steadily to a wider public. The objective of this thesis is to continue in that direction and to provide improvements of Transactional Memory support in unmanaged environments.

We are interested to get the best performance for TM so we are focusing on *unmanaged environments*. Indeed, managed environments based on virtual machines add an execution layer and make them less efficient than unmanaged one.

Most TM systems are based on software only approaches because these offer an easy way to test ideas. Unfortunately, current such Software Transactional Memory (STM) systems still have much overhead for monitoring memory accesses. This even led some researchers to claim that STM was only a "research toy" [11]. All STMs also don't offer the same level of completion and features, which makes some TM users frustrated.

Our goal is to propose a scalable STM with reduced overhead but at the same time an STM with all features required by real world applications. Read operations are the more frequent operations so we want to minimize the overhead associated with them. Similarly, we also intend to favor read-only transactions to commit. Our STM must also provide all facilities for an easy use, e. g., irrevocable transactions must be allowed to deal with non-undoable operations such as I/O.

Although STM scales quite well, it requires additional code compared to sequential code, which make some people consider TM as not viable. Indeed, the TM approach may benefit the help from the hardware to reach near to the sequential baseline.

To greatly reduce the overhead associated to accesses monitoring, we propose to leverage hardware mechanisms and reduce the gap with the sequential baseline. Some Hardware Transactional Memory (HTM) proposals start to emerge from industry and we would like to propose an extensive evaluation of the interest of such proposals in the context of transactional memory with different applications.

Unfortunately, HTMs and hardware support for transactions have some limitations inherent to hardware. Since HTM highly improves performance of TM, its usage is recommended in most of situations. In case of impossibility of using HTM, software transactions can be used as a fallback solution. Our goal is then to propose a Hybrid Transactional Memory (HyTM) that can mix hardware and software transactions without the penalty to switch to *serial execution*.

Finally, Transactional Memory has received much attention from the research community during the past, which has led to the proposal of many mechanisms of theoretical and practical nature. The integration of all these mechanisms on different levels was usually ignored by researchers who think it is just an engineering problem. Unfortunately, the integration is not a simple problem due to all complex layers of the software stack.

Transactional Memory has the potential to greatly simplify the development of concurrent software but to make it easy to use, transactions must be integrated into the development environment. The sustainability of transactional code comes with the standardization at the language level but also at the binary level. This binary standardization permits the independence of binary applications from the TM library. Thereafter, existing binary applications can benefit from the performance increase from new TM library implementations. The objective of this thesis is thus to propose a complete integration solution for an unmanaged environment.

1.2 Contributions

The main contributions of this thesis are:

A novel time-based STM implementation. Our first contribution is a new Software Transactional Memory implementation which reduces overhead and exhibits good scalability. Additionally, we propose new additional features at low cost for easing TM usage. Among these improvements, we explain how a lock-based TM can provide progress guarantee thanks to an advanced contention manager. This work was published in a journal paper [24] in *IEEE Transactions on Parallel and Distributed Systems* in March 2010.

Study of a hardware support for synchronization and speculation We study deeply a hardware extension proposal for concurrent programming. The assessment of this hardware support shows that it can be efficiently used in the context of transactional memory and can tackle completely the overhead problem. This work was published in the proceedings [13] *5th European conference on Computer systems* (EuroSys) in April 2010.

Implementation of a hybrid TM that mixes Hardware and Software transactions. Based on our novel STM and on hardware support, we propose a new Hybrid Transactional Memory (HyTM) algorithm that can run transaction of both types in parallel. Indeed, the hardware solution comes with some hard limitations like capacity limits and forbidden instructions in speculative region. We propose that transactional memory be supported by a hybrid combination of hardware and software and we conclude that Hybrid transactional memory can offer the best of both hardware and software in terms of performance and

implementation complexity. This work was initially presented in the brief announcement proceedings [23] of the *24th International Symposium on Distributed Computing* (DISC10) in September 2010 and was later published in the proceedings [55] of *23rd ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA11) in June 2011.

Integration in a complete software stack for application engineers. Finally, to make valuable our work on Transactional Memory, we propose a full integration of our STM, our HTM, and HyTM libraries into a complete software stack. We study the Transactional Memory Programming Interface (TM-API) to provide mechanisms for developers to use TM and to understand the impact on the TM library. We added all mechanisms to our TM library to be fully compliant with the Transactional Memory Binary Interface (TM-ABI). We also propose a code multi-path to obtain the best performance when we integrate hardware support in transactional programs. This work is briefly presented in the *IEEE Micro magazine* [4] in September 2010. The work on the hardware integration is also part of the conference papers cited previously [13, 55].

The Velox Project funded by the European Union under the FP7 programme aimed at proposing the first integrated Transactional Memory Stack. The Velox project established some principles and some standards at different level of the stack. As a member of this project, the work presented in this document was included into the delivered software of the project.

Additional contributions to transactional memory. Some additional contributions of the author to this research field are not presented in this thesis.

Our conference paper [44] published in the proceedings of the 15th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP 10) in January 2010 proposed to schedule transactions based on conflicts. It proposed different approaches to avoid conflicts by informing the kernel of the transaction scheduling. My specific contribution was to extend support for controlling and scheduling transactions in the TM library.

Finally in our conference paper [43] published in the proceedings of the 41st *Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (DSN 2011) in June 2011, we studied the possibility for TM users to associate time constraints to transactions. A body of applications such as reactive applications needs such requirements to provide a good user experience. My contribution was to propose a set of mechanisms to bound execution time of transaction while keeping a good concurrency level in the TM library. We can cite the concurrent irrevocable mode, the visible read mode, and the customizable contention manager.

1.3 Outline and organization of this thesis

The remaining of this thesis is organized as follows.

In Chapter 2 we introduce the multi-core hardware and its programming models. Then we describe in details the Transactional Memory programming model. We explain most important design aspects of TM to understand performance tradeoff of current algorithms.

Chapter 3 is focused on Software Transactional Memory. First, we present the LSA algorithm and its implementation in C. Second, we explore the fine-tuning of our STM for performance and we extend it in order to provide the additional features required for ease of use. Finally, we evaluate it with well-known TM applications.

In Chapter 4 we present first the evaluation of a hardware extension for Transactional Memory. Then, we describe our Hybrid Transactional Memory based on the LSA algorithm and on the evaluated hardware extension.

In Chapter 5 we introduce the integration of Transactional Memory into a software stack. We describe the language extension and the binary interface for TM. Finally, we present our TM library that is fully integrated with different software to build a complete stack.

Chapter 6 concludes the thesis.

Chapter 2

Background

In this chapter, we present the background onto which we build the research presented in this thesis.

2.1 Multi-core processors

Execution unit If an application was known as being too slow, until recently the typical answer has been to rely on the continuous increase of single thread performance allowed by micro-architecture evolution. Increasing speed lied mainly on the increase of CPU clock rates and improvements of the instruction pipeline efficiency and the out-of-order execution.

The clock rate indicates the pace of processing in cycles per second. Internally, the CPU processes instructions that may require different numbers of cycles depending on the operation performed. From a single-thread perspective, the more the clock rate increased, the more the number of instructions per seconds increased.

The fundamental idea of pipelining is to split the processing of instructions into a series of independent steps. A generic pipeline is composed of four stages: Fetch, Decode, Execute, and Write-back. The goal is to avoid idle CPU components and to parallelize the execution of the different steps of several instructions. Unfortunately, it can happen that the pipeline is empty because of a misprediction and thus the CPU stalls, which degrades program execution significantly.

Internally, CPU instructions are composed by micro-instructions that can be executed by different units. An out-of-order CPU reorders these micro-instructions in the pipeline in order to optimize the usage of all units of the CPU. Even if instructions are not executed in the same order internally, the out-of-order mechanism guarantees that the new ordering conforms to the original code order.

Engineers used all these techniques to continue increase the CPU speed. Until recently, the aforementioned techniques and technology improvements have been used to improve on the number of instructions executed per cycle, and on the number of cycles per second.

From single to multiple cores In 2005, Herb Sutter, a prominent C++ expert wrote an article "The Free Lunch Is Over" [64]. He stated that CPU speed is reaching a physical limit and that multi-core hardware and software are the way to go. As a result, programmers will

have to learn and adapt their programs for multi-cores. A new era unfolded. One interesting question programmers is to determine "what is the potential gain for applications".

Amdahl's law Thanks to the Amdahl's law, we can estimate the speedup of a program with the knowledge of its sequential and parallel parts of code and the number of processors. The law establishes the maximum speed improvement of the program.

$$\textbf{Amdahl's Law: } S(p) = \frac{1}{T_1 + \frac{T_p}{p}} \tag{2.1}$$

where:

- p is the number of processors;
- S is the speedup;
- T_1 is the sequential part of code;
- T_p is the parallel part of code.

It instructs that a significant portion of parallel code is important to get a scalable algorithm.

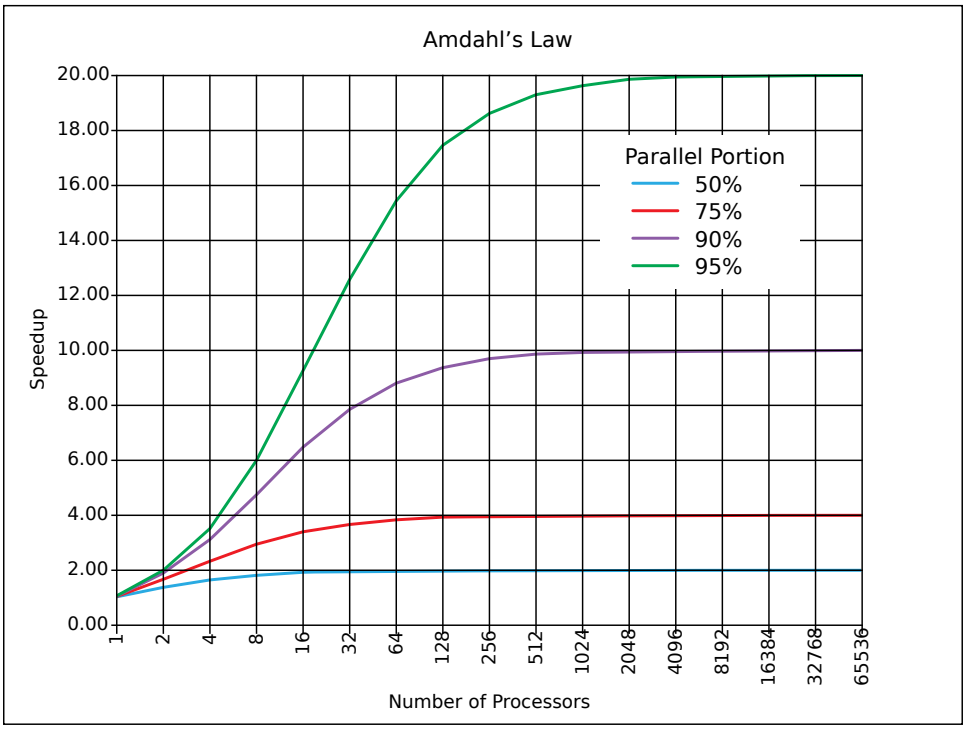


Figure 2.1: The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. (Source wikipedia.org)

Indeed, if we consider that 95% of a program is parallel, the theoretical maximum speedup is 20× as shown in Figure 2.1, no matter how many processors are used. Of course this formula is theoretical and does not take into account practical constraints e. g., the hardware characteristics.

2.1.1 Shared memory architecture

Multi-core processors are based on previous single-core processor generations within the same chip but all cores access to the same global memory. As we can see in Figure 2.2, multi-core processors on single chip share the same execution unit, the same memory, the same memory bus and also almost the same cache mechanisms.

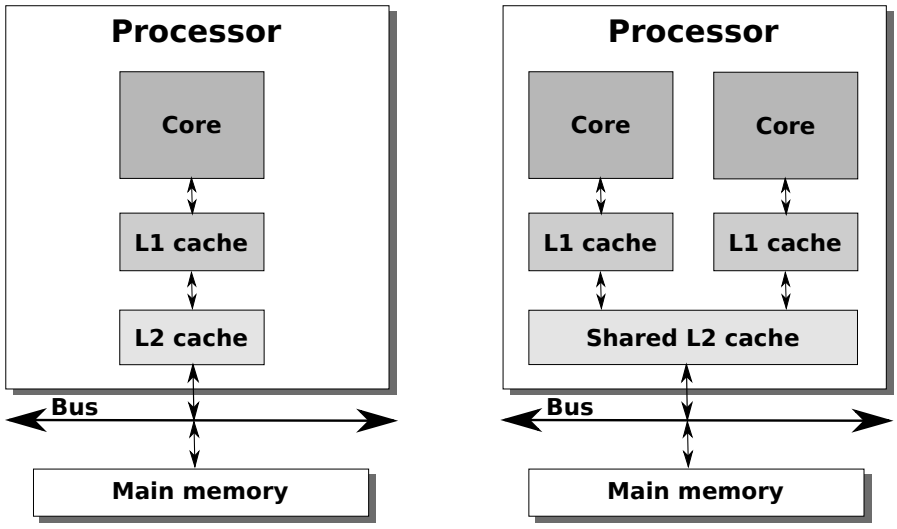


Figure 2.2: Intel CPU designs overview. Single core on the left and dual core on the right with shared L2 cache.

Cache memory

Caching is an important consideration when designing concurrent programs. Indeed, modern processors heavily use memory cache to accelerate execution and reduce "the memory wall"[1].

Cache memory (or CPU cache) is a cache used by the CPU to reduce the time to access memory. The cache is a fast memory which stores copies of frequently used data. Its size is usually small because it is costly to integrate inside the die. Access latencies and bandwidth to a cache is an order of magnitude faster than accesses to main memory.

Current processors have three independent caches: an instruction cache to speed up instructions fetch, a data cache to speed up data load and stores, and a translation lookaside buffer (TLB) to speed up address translation from virtual to physical mapping of pages.

The unit of the cache memory is the **cache line**, which may range in size from 8 to 512 contiguous bytes depending on the architecture.

When the processor needs to load or store data in main memory, it first checks whether the desired memory location lies in the cache. If it finds that the memory location is in the

[1]The problem of memory wall is not specific to multi-core. It arises when the bandwidth necessary for execution is constrained by the memory bandwidth available. With a single die multi-core, this problem arises more frequently because cores shared this memory bandwidth.

cache, it will immediately fetch or store the data in the cache line. We called this a **cache hit**. On the contrary, if the data was not found in the cache, we call it a **cache miss**.

If the processor encounters a cache miss, the time required to fetch data from main memory matters because the CPU will reach the **stall** state i. e., it will run out of instructions to process. This duration is important since current CPUs can execute hundreds of instructions during the time necessary to fetch the data from main memory. Out-of-order CPUs attempt to avoid idleness situations by executing independent instructions while the instructions that have caused the cache miss stall.

Since the size of the cache is limited, a replacement policy decides which entry in the cache should be discarded to accommodate a new entry. If the entry can go in any place in the cache, the cache is called fully associative. If the entry can go in just one place, the cache is called direct mapped. Most of current caches are a compromise of both; the cache is then called **N-way set associative**.

Since each processor can use the same memory location, the cache must manage read and write accesses conflicts. **Cache coherence** is intended to manage such conflicts and maintain consistency between caches and memory. The coherency protocol maintains the consistency between all the caches in a multi-processor or multi-core system and maintains memory coherence according to the consistency model.

False sharing If two processors operate on independent data in the same cache line, the cache coherency mechanisms will force the whole line to be coherent with every write on each processor in the same line (but not necessary in the same position in the line). If a processors needs to modify this line, it will be first invalidated in the other caches. This invalidation is costly because it forces the cache to write the data to memory and it also forces the other processors to re-fetch data from memory on the next access. This performance degradation is called **false sharing**.

Consistency model

In order to allow concurrent programming, multi-core processors propose specific instructions for synchronization and define strict rules on memory accesses consistency.

Atomic operations Multi-core processors have some specific instructions to deal with concurrency and synchronization. Indeed, to implement locking, lock-free and wait-free algorithms the processor must guarantee that some instructions execute *atomically*. Among all different instructions proposed by different industry companies, we can cite the most common:

- Atomic swap: a value in memory is exchanged atomically with the value inside a register.
- Test-and-set/Compare-and-swap: the value in memory is compared to an expected value and if equal, replaced by a new one. The comparison and test form a single atomic step.
- Fetch-and-add: a value in memory is increment atomically and the previous value is returned.

Memory ordering As explain above in Section 2.1, modern processors use out-of-order execution to get better *instruction-level parallelism* (ILP). This reordering also impacts the memory reordering that can be used to fully utilize different cache and memory banks. So on a multi-core CPU, the properties of memory ordering must be strictly defined to characterize their abilitiy to reorder memory operations. If such properties are not defined, the consistency of concurrent algorithms cannot be ensured.

2.1.2 The x86 architecture

The x86 architecture is the most popular CPU architecture nowadays. It is available for server, desktop computer but also for laptop and even mobile phones. The x86 instructions set was introduced in 1978. It is a CISC (Complex Instruction Set CPU) so unlike RISC (Reduced Instruction Set CPU) ISA, instructions can mix computation and memory assignment. The complete Intel x86 ISA is available in a manual from Intel [36]. In the following, we present some details about multi-core and x86 architectures.

The LOCK prefix The `LOCK` prefix ensures that the processor has exclusive use of any shared memory. So it ensures that all instructions with this prefix will be atomic.

Atomic operations As mentioned in the previous paragraph, atomic operations are important for synchronization guarantees. The following details these instructions:

- Atomic swap: `LOCK XCHG` atomically exchanges the value of a register with a value in memory.
- Test-and-set: `LOCK BTS` atomically compares a memory bit with 0 and sets it to 1 if the comparison succeeds.
- Atomic bit-wise operation: `LOCK OR`, `LOCK AND`, `LOCK XOR` atomically do the bit-wise operation in memory.
- Fetch-and-add: `LOCK ADD`, `LOCK SUB`, `LOCK INC`, `LOCK DEC` atomically compute an arithmetic operation and set the result in memory.
- Compare-and-swap: `LOCK CMPXCHG`, `LOCK CMPXCHG16B` atomically compare the memory with an expected value and if the comparison succeeds, set a new value in memory.

Note that the x86 architecture does not propose a Load-Link/Store-Conditional atomic operation.

Memory Consistency Model The memory consistency model defines what values a read operation can return. The simplest memory model is sequential consistency, in which the execution behaves as if there were a single global interleaving of memory operations and the operations of a given thread appear in the same order as they appear in the program.

In the example of Figure 2.3, the sequential consistency memory model ensures there will be an assignment in one processor (A = 1 or B = 1) prior to a read (local1 = B or local2 = A) in the other processor. Unfortunately, the problem with sequential consistency

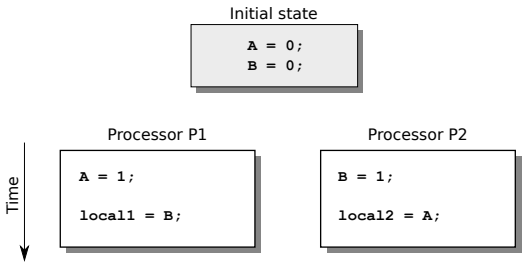


Figure 2.3: Example for consistency memory model. At the end of the execution, local1 and local2 can be any value of 1 or 0 depending on the consistency memory model.

is that the out-of-order execution cannot efficiently reorder instructions so as to hide long latency operations and consequently improve performance. So in order to extract the best performance, modern out-of-order processors can reorder instructions more efficiently if consistency is relaxed. As a result, modern ISAs and x86 in particular introduce relaxed consistency models. In the example, under the x86 consistency model, the final state can be local1 == 0 and local2 == 0.

Section 8.2.2 of the Intel x86 manual [37] states the following:

- Reads are not reordered with respect to reads.
- Writes are not reordered with respect to previous reads.
- Writes to memory are not reordered with other writes (but some exceptions).
- Reads may be reordered with respect to previous writes but not with previous writes to the same location.
- Reads are not reordered with respect to I/O instructions, locked instructions and other serializing instructions.

Memory accesses The **word size**, i. e., the natural register size and the natural address size, is dependent of the type of x86 which can be 32 bits and 64 bits nowadays. The x86-64 ISA provides instructions to read from memory to register with a data size from 1 to 8 bytes but the consistency memory model is always enforced at the level of a cache line.

2.2 Programming for multi-core processors

Multi-core programming is new for many developers whose education was not focused on such architectures. The primary entity for programming multi-core processors is the execution *thread*. Threads have been used for a long time on single-core machines. The interleaved execution of all threads via scheduling creates the illusion of having parallel tasks on a single core machine. With multi-cores, multiple threads can be executed at the same time.

Parallel programming The obvious way to use a multi-core is to distribute the work among available cores. For example, to encode an image, we can split it in equal parts and each core is responsible for one of the parts. This parallel program, or multi-threaded

program, exploits task parallelism, also known as thread-level parallelism (TLP). It only requires synchronization to notify the end of the encoding process. The interaction between cores is limited and there is no data dependency between tasks. This situation is that of *data parallelism*. This is an ideal scenario for multi-cores because the gain can be almost linear with the number of cores.

Speedup The typical gain metrics is the speedup. The speedup measures the parallel execution gain over a sequential execution as described by Equation 2.2. The measurement of the speedup for different number of threads gives the scalability graph. It is often used to show how scalable an algorithm or a program is.

$$\textbf{Speedup: } S(p) = \frac{T(1)}{T(p)} \tag{2.2}$$

where:

- p is the number of processors
- $T(1)$ is the execution time of the sequential algorithm
- $T(p)$ is the execution time of the parallel algorithm with p processors

The ideal speedup is obtained when S(p) is equal to p. It means if the number of processors is 10, the program will execute 10× faster than a single processor.

From parallel programming to concurrent programming Even with the simple example described above, the work distribution may not be perfect because some parts of the image may need more computation than others and thus the overall execution time increases. One solution is to split the images into very small pieces and each core chooses one part of the image to encode. When the processing is finished, the core chooses another piece to encode and so on. Unfortunately, we do not want cores to encode the same part of the image so the program must synchronize to distribute work among available cores. This work distribution requires concurrent programming and synchronization mechanisms.

Concurrent programming Sequential programs tend to be easier for developers to reason about than their equivalent parallel and concurrent programs. The correct synchronization of concurrent objects is often more complex than developers may initially think due to underlying system stack, e. g., instructions reordering of the compilers, consistency memory models of the CPU, etc. The main hurdle for concurrent programming to be efficient and correct is that traditional synchronization mechanisms such as locks are error-prone and difficult to master by most programmers.

2.2.1 Lock-based synchronization

Lock-based synchronization is the most popular technique to deal with concurrency. The properties of a lock are (1) the lock can be acquired only one time; (2) the lock cannot be acquired if it is already held; (3) after the lock is released, it can be reacquired.

A lock allows constructing a *critical section*, i. e., a block of code which can be executed by only one thread. This property is called *mutual exclusion*.

An application can use different locking *granularity*, i. e., one or several locks in its implementation to deal with concurrency. If locks are used for large pieces of code in the application, we call the implementation coarse-grained locking. If many locks are used for small pieces of code, we call it fine-grained locking. Lock granularity affects performance and it is a trade-off between lock overhead and lock contention.

Lock overhead: locks require some memory space and time for being acquired and released.

Lock contention: contention appears when one thread wants to acquire a lock already held by another thread. The contention reduces the parallel part of the program by blocking.

The problem with coarse-grained locking is that it doesn't allow the maximum amount of concurrency. Lock handling adds overhead even when the chances for collision are rare. Fine-grained locking allows better concurrency but it adds complexity to the code, e. g., dealing with deadlocks, and overhead for acquiring and releasing locks.

A deadlock is a situation where a process has the lock associated with a shared resource A and wants the lock associated to resource B. If another process has the lock on resource B and wants the lock on resource A as illustrated in Figure 2.4, both processes will wait indefinitely to acquire lock on resource A, and respectively B.

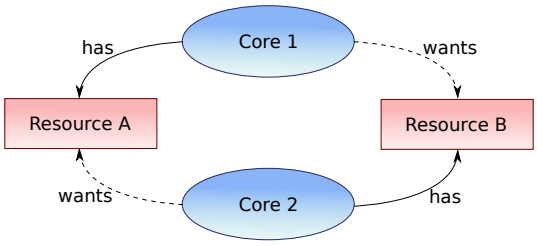


Figure 2.4: Example of a simple deadlock.

The conclusion of years of usage is that locks are hard to manage effectively, especially in large software.

2.2.2 Non-blocking synchronization

Lock-based programming is based on blocking when contention happens. This approach is not optimal because it reduces the parallel part of the programs and Amdahl's law (see Section 2.1) shows that each percent of sequential code drastically reduces the potential overall speedup. A non-blocking implementation ensures that threads will execute within a finite amount of time even with contention.

Non-blocking implementations can be classified with different properties depending on the progress. The *Wait-free* property is the strongest non-blocking guarantee of progress because it guarantees that at each step threads are making progress. Wait-free algorithms are usually complex and therefore rare, both in research and in practice. The *Lock-free* property guarantees that at least one thread is making progress. So it ensures the global progress of the application even if a thread is interrupted. The *Obstruction-free* property is

the weakest non-blocking guarantee of progress because it only guarantees that a thread is making progress if it does not encounter contention. Obstruction-free implementations can lead to a *livelock* situation i. e., two threads prevent each other's progress.

Globally, non-blocking synchronization is hard to program and error-prone for non-experts. In some cases, gains compared to blocking synchronization are not always as good as expected due to the complexity of the implementation.

2.3 Transactional Memory

Transactional Memory (TM) is an alternative concurrency control paradigm that has been the focus of intense research for more than a decade. The TM paradigm hides the complexity of concurrent programming compared to traditional synchronization mechanisms. Rossback has shown in [56] that TM is less error-prone than locks and that it simplifies the writing of parallel programs.

2.3.1 Basics

The principle of transactions comes from databases. Indeed, researchers and engineers designed transactional databases to allow a maximum of parallel users. A transaction is a sequence of actions delimited by *start* (or also called *begin*) and *commit* operations that appear indivisible and instantaneous to an external observer of the database state. This sequence will execute optimistically presuming that no other execution will collide. The transaction appears to be *atomic*, i. e., an indivisible operation from the user view. This makes transactions convenient to use because the atomicity solves the coordination of concurrent reads and writes on shared data.

A transaction is a code region which can be composed of reads and writes to shared data and computations. If no conflict is detected, i. e., no other transaction used the same piece of memory, the transaction *commits*. If two transactions use the same piece of memory, a *conflict* happens and has to be solved. The *Contention Manager* is in charge of solving the conflict. The typical strategy is to *abort* and *roll back* one of the transactions.

The ACID criteria are used in databases to ensure the safety of the transactional system. In Transactional Memory, these criteria differ slightly from database system:

Atomicity: all operations of a transaction appear instantaneously when the transaction commits.

Consistency: the property of consistency may differ depending on the applications. In the case of TM, we consider serializability as the consistency model.

Isolation: when an active transaction has not yet committed, all its operations are not visible to any other transaction.

Durability: when a transaction commits, its modifications are permanent i. e., written to durable storage such as disk.

2.3.2 Software, Hardware and Hybrid Transaction Memory

Transactional Memory systems can be classified according to their implementation levels: entirely in software (STM: Software Transactional Memory), entirely in hardware (HTM: Hardware Transactional Memory) or using a combination of both (HyTM: Hybrid Transactional Memory).

The first STM was proposed by Shavit *et al.* in [61]. STM is flexible and allows implementing easily sophisticated algorithms. However, the overheads associated with transactional reads and writes is the main limitation of STM. The research continues in designing new algorithms to reach practical overhead without hardware support.

Herlihy and Moss were the first to propose HTM in [31]. Hardware transactions execute entirely in processor hardware, which usually imposes less overhead than software-supported transactions. Most HTMs leverage the cache coherency protocol to detect and manage conflicts between hardware transactions. Unfortunately, HTMs are usually limited to transactions with small read and write sets due to hardware limitations compared to STM.

Most HTM designs are evaluated by simulation. The two exceptions to this rule are the Rock Processor from Sun Microsystems and the Vega 2 processor from Azul Systems. The first commercial release with Hardware Transaction Memory was done by the Azul Company but as an internal mechanism, thus it doesn't expose any instructions for creating transactions.

Note that there is a distinction between HTM and hardware support. HTM executes all the code between the "begin" and the "commit" instruction within a hardware transaction. The hardware support can be from different forms, e. g., specific instructions to help the validation, specific instructions for transactional loads and stores. Whereas HTM usually requires important modifications to the CPU, the hardware support can be only an extension of an existing instructions set.

In Hybrid TM, HTM is used to provide low overhead transactions, while STM transactions serve as a fallback solution to handle situation where hardware transactions cannot be executed. Indeed, some features like I/O and context switching may not be supported by HTMs and require software transactions to be executed. HyTM require hardware and software transaction to coexist correctly. Usually hardware transactions come with additional code to ensure that they do not commit if there is a conflict with a concurrent software transaction.

2.3.3 TM design choices

Many alternative implementations are possible for building a TM system. We introduce some of the main design choices in the following paragraphs.

Concurrency control In a TM system, there are two approaches for concurrency control.

With *pessimistic concurrency control*, TM detects and resolves conflicts when a transaction is about to access a location. In this case, the transaction acquires the ownership of the data before accessing it to prevent others from accessing it.

With *optimistic concurrency control*, TM can detect and resolve conflicts later after the conflict occurs. In this case, multiple transactions can access the data concurrently and conflicts can be detected later in the execution of the transaction.

Version management Version management handles how new data and old data are managed in transaction.

With *eager version management* (or also called *write-through*), new data is put directly in place. The transaction maintains an *undo-log* with the previous versions to revert changes in case of an abort. It requires a pessimistic concurrency control upon writes to forbid concurrent access to data that is not already valid.

With *lazy version management* (or also called *write-back*), new data is put aside in a *redo-log* and will be written only when the transaction commits. If a read happens after a write, the transaction must read the data from the redo-log. If the transaction aborts, modifications do not need to be undone and the redo-log is only reset.

The version management affects the latency of transaction commit or abort. Eager version management makes transaction commits straightforward, while lazy version management makes aborts straightforward.

Conflict detection A conflict occurs when the write set of one transaction overlaps the read set or the write set of another concurrent transaction. The detection of such a conflict is called *eager conflict detection* if the transaction detects offending reads or writes immediately or *lazy conflict detection* if the transaction can defer the detection of the conflict until its commit phase. Hybrid approaches are often used in TM where write/write and read/write conflicts are managed differently.

The granularity defines the level of conflict detection. Conflicts can be detected at the level of a memory word, a cache-line or an object. In an *object-based STM*, each object has an associated metadata. This metadata can be a locator that points to the object or it can be integrated to the object itself, e. g., expanding the object's header. All fields of the object are associated to the same metadata. In a *word-based STM*, metadata is associated to memory locations. Typically, it uses a fixed-size set of metadata and a hash function to map any memory addresses into the set.

Lock-based and non-blocking STM STM systems can provide different progress guarantees and particularly non-blocking property such as the property of obstruction-free (see Section 2.2.2). The first STM [29] provided a non-blocking implementation as its goal was to help implementing non-blocking data structures. Unfortunately, obstruction-free implementations come with a complex implementation compared to lock-based implementations, which make them usually slower.

Lock-based STMs use locking to protect accesses to data. The implementation of such STM is usually simple in the un-contended case. To ensure progress of transactions, lock-based STMs must use the support of contention manager to avoid situations such as deadlocks.

The lock-based approach can use a specific lock design to determine the owner of an acquired lock. In this case, we call the lock an *ownership record* (orec).

Lock-based STMs can be implemented with different strategies of lock acquisition. With *encounter-time locking* (ETL), the lock is acquired when the transaction first accesses a location. With *commit-time locking* (CTL), the lock is only acquired when the transaction reaches the commit phase.

2.3.4 Contention management

The TM system can detect when the execution of a transaction is causing a synchronization conflict. When two transactions conflict, the *contention manager* (CM) decides which one can proceed and eventually commit and which one has to abort and roll back. The first role of contention management is to avoid conflicts that can lead to deadlock situations. Additionally, contention management allows avoiding some aborts but also can solve complex situation like livelock.

The naive implementation is the *suicide* contention manager. The transaction that detects the conflict aborts, rolls back, and retries immediately. This contention manager has a simple implementation, which makes it really efficient when conflicts are rare and when the two transactions are unlikely to conflict again.

But the suicide contention manager is usually not enough if TM has to give some guarantee such as time constraints [43]. Another strategy is *backoff* (or also called *polite*). This CM behaves like suicide except that the aborted transaction waits a certain amount of time before retrying. The duration is random and can increase exponentially as the number of aborts increases for this transaction.

Many other contention managers are defined in the literature (e. g., [28, 59]).

2.3.5 Benchmarks and applications

Since transactional memory is still in the research phase and steadily becomes more popular, there is no commercial software that uses STM. The number of programs using transactional memory is limited to synthetic benchmarks and only few existing applications have been modified to the form of benchmarks that uses transactions for synchronization. Synthetic micro-benchmarks like integer sets are often used to show how scalable a TM is.

Integer sets benchmarks

The skip list (SL), red-black (RB) tree, hash set (HS), and linked list (LL) benchmarks all manipulate a set of integers. An execution consists of both read transactions, which determine whether an element is in the set, and update transactions, which either add or remove an element (reads are also necessary to find the position of the element to add or remove). Operations are chosen randomly and on random elements. The set is initially populated with half of the size of the key range from which elements are drawn. Its size is maintained constant by alternating insertions and removals.

Linked list A Linked list is a data structure that consists of a set of nodes, each of which contains a reference to the next node. Our implementation has integer as node and it is a singly linked list, i. e., it has only one link to the next node. The set is in ascending order.

With TM, this benchmark has a small potential for parallelism. Indeed, accessing an element requires traversing all previous elements, implying that any write to a previous element that occurs before a transaction completes causes a conflict. It has long-length transactions compared to other integer sets.

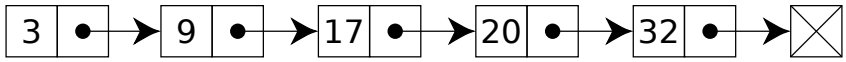


Figure 2.5: Linked list.

Red-black tree A red-black tree is a binary tree that is self-balancing, i. e., the depth on all branch of any sub-tree differs by at most 1. It has complex insertion and removal operations to ensure that the tree is kept balanced. The search, insertion and removal operation are in O(log n) time where n is the total number of elements.

Red-black trees use data structures designed to make it possible to access any element by traversing only a few other elements, and thus exhibit a high potential parallelism. An operation of rebalancing can modify a lot of elements and causes conflicts with many concurrent operations. It has short transactions compared to other sets.

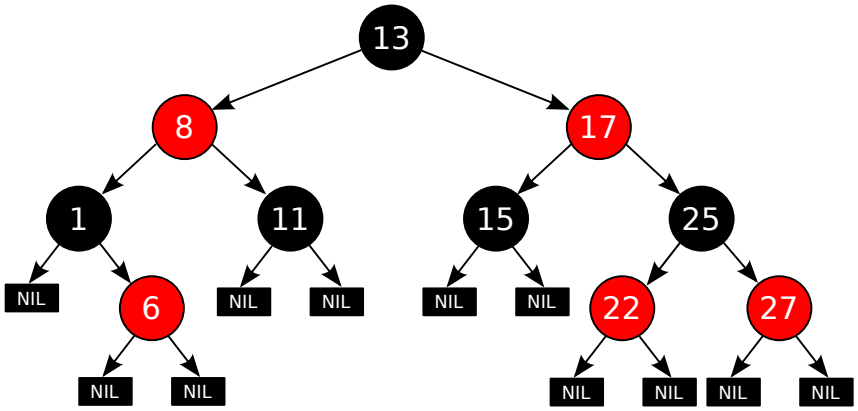


Figure 2.6: Red-black tree. (Source wikipedia.org).

Skip list A skip list is a data structure with sorted elements that uses different levels of linked lists. These intermediate lists allow item lookup with efficiency comparable to balanced binary trees.

Like the red-black tree, the skip list can traverse the data structure via only few elements, and thus also has high potential parallelism. It has short transactions, similarly to the red-black tree.

Hash set A hash set is a data structure that uses a hash function to map items to a position in an array. These elements are thus accessed by their keys. If an item maps to the same position as an existing element, the new item is inserted into a linked list.

The bigger the array, the more the data structure shows a potential parallelism. It has very short transactions compared to other sets.

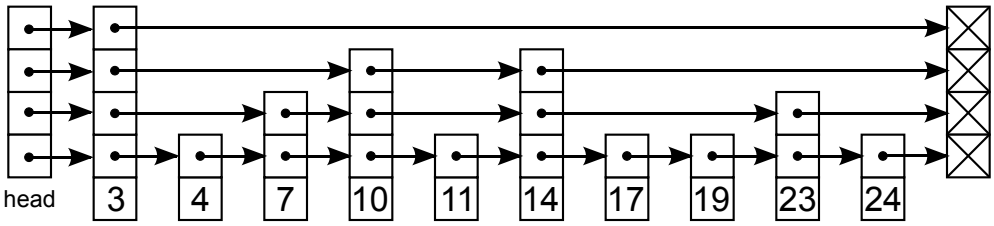


Figure 2.7: Skip list.

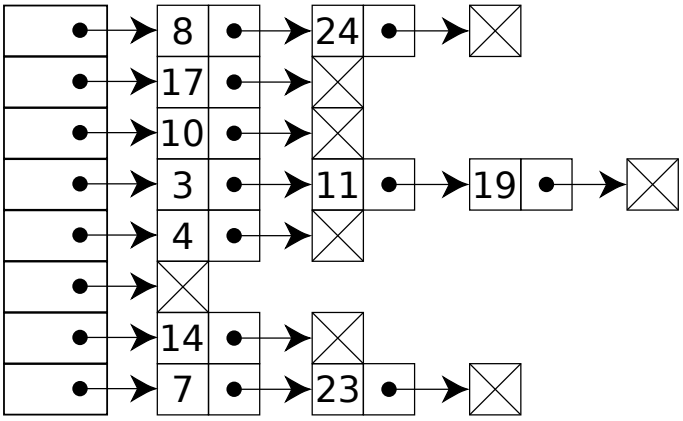


Figure 2.8: Hast set.

Bank benchmark

The `bank` micro-benchmark models a simple bank application performing various operations on accounts (transfers, aggregate balance, etc.). In this benchmark, transactions access a constant number of objects: transfers access 2 objects, balance operations read all objects. The pattern of transaction length is a mix of very short transactions and very long read-only transactions.

STAMP benchmarks suite

The Stanford Transactional Applications for Multi-Processing benchmark suite (STAMP [9]) is a set of realistic benchmarks. It is the first suite to propose software that uses efficiently transactional memory. It is composed of 8 different applications:

- `bayes` learns the structure of Bayesian networks from observed data.
- `genome` takes a large number of DNA segments and matches them to reconstruct the original source genome.
- `intruder` emulates a signature-based network intrusion detection system.
- `kmeans` is an application that partitions objects in a multi-dimensional space into a given number of clusters.
- `labyrinth` executes a parallel routing algorithm in a three-dimensional grid.
- `ssca2` constructs a graph data structure using adjacency arrays and auxiliary arrays.

- `vacation` implements an online travel reservation system.
- `yada` refines a Delaunay mesh using the Ruppert's algorithm.

Two sets of parameters are recommended by the developers of STAMP for `vacation` and `kmeans`, for producing executions with low and high contention.

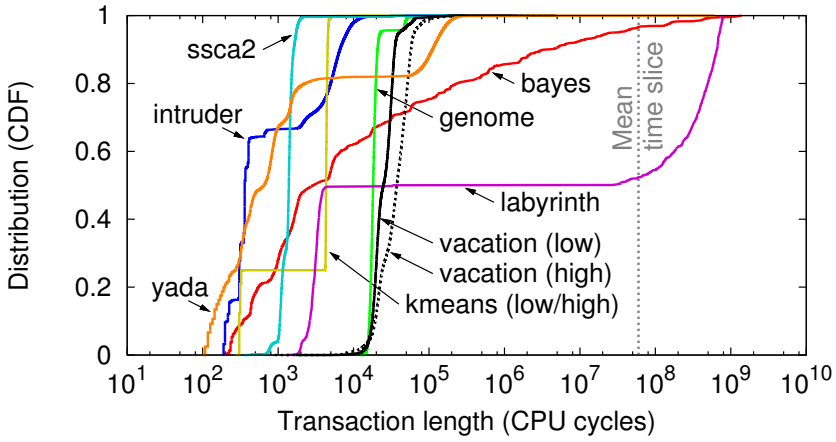


Figure 2.9: Transaction lengths for the STAMP benchmarks.

Additionally to the analysis done in the original paper [9], we have studied the characteristics and the length of transactions of different applications to understand the requirements of transactional applications. Figure 2.9 presents the transaction lengths, and Table 2.1 summarizes the characteristics of the transactional workloads produced by these applications.

Application	**Tx length (cycles)**		**Reads**	**Writes**	**Contention (%)**		
	μ	σ	μ	μ	@2	@8	@16
`ssca2`	1,475	2.3e3	1.0	2.0	3.6e-4	2.6e-3	6.6e-3
`genome`	19,803	9.4e3	30.1	0.03	0.08	0.27	0.41
`vacation-l`	27,039	1.6e4	283.0	5.4	0.02	0.17	0.38
`vacation-h`	39,197	2.6e4	386.7	7.8	0.05	0.35	0.72
`bayes`	14,587,146	9.6e7	28.6	3.2	0.45	2.63	3.95
`yada`	25,664	6.7e5	60.8	18.8	3.31	6.72	6.60
`labyrinth`	207,825,190	2.6e8	180.1	177.0	1.85	6.06	10.56
`intruder`	2,197	3.9e3	23.6	2.7	1.89	23.72	33.93
`kmeans-l`	3,387	2.1e3	25.0	25.0	25.6	31.34	32.40
`kmeans-h`	3,293	1.9e3	25.0	25.0	28.5	45.79	41.33

Table 2.1: Workload characteristics for the STAMP benchmarks.

The single-threaded execution time of STAMP applications takes from a few seconds to several minutes depending on the benchmark and parameters.

Additionally, two kinds of parameters are proposed for different execution platform: real and simulated. Indeed, the simulation requires a lot of computation time and thus parameters for simulation reduce the data size of STAMP applications.

Chapter 3

Design of an efficient Transactional Memory

Our design of a Transactional Memory library is driven by a set of goals. In the context of this dissertation, we propose a new efficient system for an unmanaged environment. We focus on some important aspects in the context of transactional memory: low overhead, scalability, usability. While previous research covered individually those aspects, we integrate all in one to provide a complete transactional library that can be integrated into a software stack.

First, we discuss the design of a Software Transactional Memory Library with the respect of the aforementioned goals. Second, we introduce the chosen algorithm: the Lazy Snapshot Algorithm (LSA) and its implementation. Finally, we describe the new features and the new optimizations that our Software Transaction Memory library provides and we evaluate it with real applications.

3.1 Design choices

The design of a Software Transactional Memory is conducted by many constraints, e. g., targeted platforms. In the context of this dissertation, we focus our work on unmanaged environment, which doesn't rely on any virtualization layer and directly runs on hardware.

Isolation The isolation problem is when transactional and non-transactional accesses can happen concurrently and thus conflict. Indeed, when the isolation is guaranteed all transactional accesses have to appear atomic not only regarding transactions but also with non-transactional accesses.

In the example of Figure 3.1, accesses in or out of transaction can lead to an inconsistent state. One possibility is to provide *strong atomicity*. It guarantees consistency between transactional and non-transactional memory accesses. Strong atomicity simplifies the work of developers by taking care of mixing accesses. Since strong atomicity requires detecting all memory accesses, including those outside of transactions, it require an appropriate hardware support or to be executed within a managed environment. Page access protection of current hardware can be used as a workaround as proposed by [1, 46]. Unfortunately, the overhead induced by page faults is too high to be interesting. Another approach is to recompile code

```
Starting with:      T1                  T2
x = 0;              transaction{
                      x++;
                      if (x==1)         printf(x);
                        x++;
                    }
```

Figure 3.1: Isolation problem: T2 can read the intermediate value (1) of x written by T1 when the strong isolation is not guaranteed.

and enclose all memory accesses in unit transactions (transaction with a single access). Again the inherent overhead makes it a non-viable solution.

Accordingly to performance issues and our assumption of no hardware support, we base our transaction memory on the *weak atomicity* guarantee. It guarantees consistency only *between transactions*. All modifications appear as if they were atomic with regards to transactions and if developers access the same data outside of transaction then consistency is not guaranteed. This requirement is required to keep the transactional system safe.

Consistency models In concurrent programming, we consider linearizability as a proof of correctness because it allows the developer to reason with concurrent code in the same way as sequential code.

Serializibility is another consistency model but its implementation is complex and costly as described in [27], which does not make a viable solution.

While some algorithms rely on value based validation [48, 16], we choose the timestamp based one because it provides an effective solution to avoid extensive validation.

Memory granularity The granularity of the memory accesses is an important consideration in the STM implementation. A word-based or cache-line-based STM detects conflicts for a specific range of memory locations. As a contrary, an object-based STM operates at the granularity of the object (abstraction of memory), which can vary widely. We concentrate our ideas on the word-based approach. Indeed, a word-based STM can indifferently work with object-oriented language like C++ or non-object oriented ones such as C. Moreover, the ability to manipulate pointers makes a word-based approach suitable for the C language.

Such word-based STM usually requires its metadata in a place separate from the data itself. The algorithm does not require maintaining old versions but can take advantage of them if they are available.

Synchronization mechanisms Most of the initial STM implementations were non--blocking; they did not use locks and obviously avoided deadlocks. Later implementations have moved to lock-based algorithms, i. e., a blocking approach due to performance reasons.

The advantages of a lock-based TM lie in its simplicity and lightweight implementation. Lock-based TMs have a simpler fast path and more streamlined implementations of the read/write operations without extra indirection. Non-blocking implementations suffer from costly indirections necessary for meeting their obstruction-free progress guarantee [22, 45].

The drawback of lock-based TM is of course possibility of deadlock but the burden of managing this situation is not left to the application developer but to the STM designer. Moreover, those deadlocks can be eliminated by the use of a smart contention manager.

Finally, in all obstruction-free or lock-free designs, it is difficult to detect and to solve conflicts while ensuring progress.

In this dissertation, the reason for basing our work on a blocking approach instead of an obstruction-free one is mainly driven by performance considerations.

With regards to all these requirements, we choose to focus on the Lazy Snapshot Algorithm [52], which gives a good opportunity to make STM efficient.

3.2 The Lazy Snapshot Algorithm

We first informally explain the general principle of the algorithm and the way snapshots are constructed incrementally. We then give a formal definition of the algorithm and prove its correctness. The description of the algorithm is mainly based on our journal paper [24]. Finally, we will discuss one implementation of the algorithm in C.

3.2.1 Principle of the Algorithm

The Lazy Snapshot Algorithm (LSA) [52] handles transactional accesses to shared *objects*, which can designate either a complex data structure (as in object-oriented programming), or a single memory location, or a range of memory location (e. g., a cache-line).

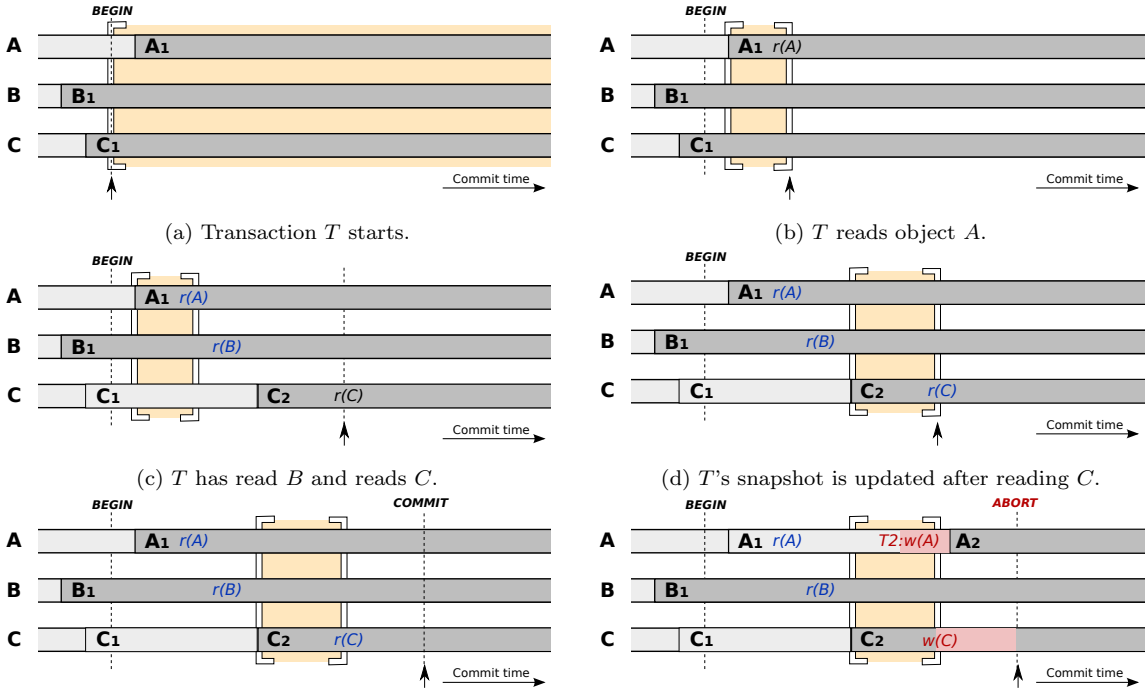
Our transactional memory uses a discrete logical global clock, designated by ***clock***[1]. When an update transaction commits, it acquires a unique timestamp from ***clock*** (informally, this represents progress by advancing the global time) and associates it with the objects it has written. That is, every shared object in the system has a timestamp that indicates the time from which its current version is valid, as well as an optional set of older versions with associated timestamps. The latest version of an object remains valid until it is overwritten by a committed transaction.

Every transaction maintains a *snapshot* that corresponds to a range of valid linearization points. The transaction can only commit if its snapshot is non-empty at completion time. Initially, the snapshot of a transaction is $[start, \infty]$, where *start* is the value of ***clock*** at the time the transaction starts (see Figure 3.2(a)).

When a transaction reads an object, it must pick a version whose "validity range" (i. e., the period during which it is valid, see Section 3.2.2) intersects with the transaction's snapshot. The bounds of the snapshot are adjusted to the intersection. When reading the latest version of an object—the usual case—the upper bound is capped by the current value of the clock (see Figure 3.2(b) with version 1 of object A being read).

If the latest version of an object read by a transaction has a validity range that starts after the upper bound of the transaction's snapshot (see Figure 3.2(c) with object C being read), the transaction can either read an old version with a validity range that overlaps the snapshot, or attempt to *extend the snapshot*. An extension consists of trying to move the

[1]Note that the global clock can be replaced by more scalable alternatives, like approximately synchronized clocks as discussed in [54].



(a) Transaction T starts.

(b) T reads object A.

(c) T has read B and reads C.

(d) T's snapshot is updated after reading C.

(e) T can is read-only, it can commit immediately.

(f) T writes C, it aborts as A has been updated.

Figure 3.2: Principle of the LSA-STM algorithm illustrated on a transaction T accessing three objects A, B, and C. Object versions are delimited by vertical lines and denoted respectively by A_i, B_i, and C_i $(i = 1, 2, \ldots)$. We represent the last committed version with a darker shade of grey. The thick arrow below the figures indicates the current time and the shaded region between large square brackets represents the transaction snapshot.

upper bound to some later point in time no higher than—but typically equal to—the current value of the clock.

To that end, the transaction must verify that the versions of all the objects previously accessed by the transaction are still valid. If the extension succeeds, the transaction can read the latest version of the object and adjust the snapshot accordingly (see Figure 3.2(d) with version 2 of object C being read). Otherwise, if the transaction cannot read a valid version of the object while maintaining a non-empty snapshot (more precisely, a snapshot with a non-empty validity range), it aborts.

A transaction can only commit if it has a non-empty snapshot and a commit time that falls within the bounds of that snapshot. For a read-only transaction, as long as the snapshot is not empty, any point within the snapshot is a possible linearization point and, hence, a valid commit time. Therefore, such transactions can commit immediately (see Figure 3.2(e)).

Committing update transactions is slightly more complicated. In LSA, writes are visible, i. e., a transaction can determine whether an object is being written by another transaction. When an update transaction commits, it writes new versions of each updated object timestamped by the commit time of the transaction. Consider the example in Figure 3.2(f) where transaction T reads objects A and B before writing C. At commit time, transaction T must acquire a new, unique timestamp from the global clock that will be associated with the new version of C being written. Then, it must *validate* that all objects previously accessed are still valid at commit time, which corresponds to the linearization point of the transaction. In our example, another transaction has written a new version of object A, i. e., the version read by T is not valid anymore. Therefore, the transaction must abort.

We now describe the algorithm more precisely in the rest of this section.

3.2.2 Notations

A transactional memory consists of a set of shared objects $\mathcal{O}$. Transactions are either *read-only*, i. e., they do not write any object, or are *update* transactions, i. e., they write one or more objects.

We designate the discrete logical global time base of LSA by ***clock***. It can be implemented using a simple shared integer counter that is incremented *atomically* by update transactions to acquire a unique commit timestamp[2].

A transaction T accesses a finite set of objects $\mathcal{O}_T \subseteq \mathcal{O}$. Each object o traverses a series of versions $o_1, o_2, \ldots, o_i$. The transactional memory may—but does not need to—keep multiple versions of an object at a given time; only the latest version is necessary.

We 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***. We denote by $\lfloor o_i \rfloor$ 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. We 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

[2]Atomic increment is achieved by hardware instructions like "increment-and-fetch" or "compare-and-swap" available on most modern processors.

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, we denote by $o_\star$ 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., $\forall o_i \neq o_\star : \lfloor o_i \rfloor < \lfloor o_{i+1} \rfloor$.

Each transaction T maintains a read set $T.\mathcal{R}$ and a write set $T.\mathcal{W}$ that keep track of the object versions read and written by the transaction, respectively. To simplify the presentation, we assume in the pseudo-code that an object is accessed only once by a transaction (it is either read or written). We will explain in the description of the algorithm how multiple accesses by the same transaction are dealt with.

A transaction T incrementally constructs a snapshot of objects versions and keeps track of the validity ranges of these objects. To that end, T maintains the known bounds on the validity range $T.\mathcal{S}$ of the snapshot. These bounds, denoted by $\lfloor T.\mathcal{S} \rfloor$ and $\lceil T.\mathcal{S} \rceil$, are computed as the intersection of the validity ranges of the objects accessed by the transaction. We say that the snapshot is *consistent* if its bounds correspond to a non-empty range. Note that, by construction, the object versions contained in a consistent snapshot are always the most recent versions at any time $t \in T.\mathcal{S}$.

3.2.3 Snapshot construction

The lazy snapshot algorithm is presented in Algorithm 1. A transaction completes successfully if it executes the algorithm until commit without encountering a call to Abort (in which case it immediately terminates). Note that the pseudo-code does neither show how mutual exclusion is achieved nor how objects are atomically updated in memory. This will be discussed in Section 3.2.7 where we present a lock-based implementation.

The main idea of the algorithm is to construct consistent snapshots on the fly during the execution of a transaction and to extend the validity range on demand (lazily). By this, we can reach two goals. First, transactions working on a consistent snapshot always read consistent data. Second, verifying that there is an overlap between the snapshot's validity range and the commit time of a transaction can ensure linearizability. We first describe the basic algorithm and then prove its correctness in Section 3.2.6.

The objects accessed by a transaction T are only discovered during its execution, i. e., the snapshot cannot be constructed beforehand. The final value of $T.\mathcal{S}$ might not even be known at the commit time of the transaction. We therefore maintain a *preliminary validity range* in $T.\mathcal{S}$ that represents the known bounds. When the transaction is started, we set $T.\mathcal{S}$ to $[\textbf{\textit{clock}}, \infty]$ (line 4). Note that $T.\mathcal{S}$ will never hold values smaller than the start time of T.

When accessing (i. e., reading or writing) the most recent version $o_\star$ of object o, it is not yet known when this version will be replaced by a new version. We therefore conservatively approximate the upper bound of its validity range by the current time t and we set the new snapshot range to $T.\mathcal{S} \cap [\lfloor o_\star \rfloor, t]$ (lines 11 and 22). During the execution of a transaction, time will advance and thus the preliminary validity ranges might get longer. We can try to "extend" $T.\mathcal{S}$ by re-computing its upper bound (lines 9, 20, 26–29). Note that this is not required for correctness—it only increases the chance that a suitable object version is available.

Algorithm 1 Lazy Snapshot Algorithm (LSA) for transaction T

```
1:  Global state:
2:      clock ← 0

3:  start(T):                                              ▷ Start transaction
4:      T.S ← [clock, ∞]                                   ▷ Snapshot bounds
5:      T.R ← ∅                                            ▷ Read set
6:      T.W ← ∅                                            ▷ Write set

7:  read(T, o):                                            ▷ Read a shared object
8:      if ⌊o⋆⌋ > ⌈T.S⌉ then                               ▷ Is latest version too recent?
9:          extend(T, clock)                               ▷ Try to extend
10:     if ⌊o⋆⌋ ≤ ⌈T.S⌉ then                               ▷ Can use latest version?
11:         T.S ← [max(⌊T.S⌋, ⌊o⋆⌋), min(⌈T.S⌉, clock)]     ▷ Yes: use latest
12:         T.R ← T.R ∪ {o⋆}
13:     else if T.W = ∅ ∧ (∃oi : ⌊oi⌋ ≤ ⌈T.S⌉ ∧ ⌈oi⌉ ≥ ⌊T.S⌋) then
14:         T.S ← [max(⌊T.S⌋, ⌊oi⌋), min(⌈T.S⌉, ⌈oi⌉)]     ▷ No: use older
15:         T.R ← T.R ∪ {oi}
16:     else
17:         abort(T)                                       ▷ Cannot find valid version: abort

18: write(T, o):                                           ▷ Write a shared object
19:     if ⌊o⋆⌋ > ⌈T.S⌉ then                               ▷ Is latest version too recent?
20:         extend(T, clock)                               ▷ Try to extend
21:     if ⌊o⋆⌋ ≤ ⌈T.S⌉ then                               ▷ Can use latest version?
22:         T.S ← [max(⌊T.S⌋, ⌊o⋆⌋), min(⌈T.S⌉, clock)]     ▷ Yes
23:         T.W ← T.W ∪ {o⋆}
24:     else
25:         abort(T)                                       ▷ Cannot find valid version: abort

26: extend(T, t):                                          ▷ Try to extend the snapshot
27:     ⌈T.S⌉ ← t
28:     for all oi ∈ T.R ∪ T.W do
29:         ⌈T.S⌉ ← min(⌈T.S⌉, ⌈oi⌉)

30: commit(T):                                             ▷ Try to commit the transaction
31:     if T.W ≠ ∅ then
32:         tc ← (clock ← clock + 1)                       ▷ Unique timestamp (atomic increment)
33:         if ⌈T.S⌉ < tc − 1 then
34:             extend(T, tc − 1)                          ▷ Try to extend
35:             if ⌈T.S⌉ < tc − 1 then
36:                 abort(T)                               ▷ Inconsistent snapshot: abort
37:         for all oi ∈ T.W do                            ▷ Atomically commit updates
38:             oi⋆ ← oi                                   ▷ Write new version of shared object
39:             ⌊oi⋆⌋ ← tc                                 ▷ Validity starts at commit time
```

3.2.4 Read accesses and read-only transactions

Read accesses in LSA are optimistic and invisible to other transactions. The algorithm assumes that the underlying STM always keeps the most recent version of an object. In addition, we might also have access to some older versions (e. g., objects that have not yet been garbage collected) that can be used to increase the probability of obtaining a consistent snapshot. When a transaction reads object o at time t, it first tries to select the most recent object version $o_\star$ (lines 10–12). If that version cannot be used because it was created after $T.\mathcal{S}$, we might still read some older version $o_i \in \mathcal{H}(o)$ whose validity range overlaps $T.\mathcal{S}$ and, hence, keeps the snapshot consistent (lines 13–15). In that case, we simply set the new range to $T.\mathcal{S} \cap [o_i]$. As a simple optimization (not shown in the code), we can mark the transaction as "closed" to indicate that it cannot be extended anymore. If there are multiple versions to choose from, we select the most recent one. If no such version exists, the transaction needs to be aborted (line 17).

If an object previously accessed by the current transaction is read, the same version must be returned to preserve consistence even if a new version has been committed in the meantime; otherwise the snapshot would contain multiple versions of the same object with non-overlapping validity ranges and the transaction would obviously have no linearization point.

By construction of $T.\mathcal{S}$, LSA guarantees that a transaction started at time t has a snapshot that is valid at or after the transaction started, i. e., $\lfloor T.\mathcal{S} \rfloor \geq t$. Hence, a read-only transaction can commit if and only if it has used a consistent snapshot for its whole lifetime (i. e., $T.\mathcal{S}$ remains non-empty). The global clock does not need to be increased when committing a read-only transaction because no object has been written. This optimization improves the memory cache hit rate if the clock is implemented as a counter in shared memory. Note that, as a consequence, multiple read-only transactions (even in the same thread) may share the same commit time.

3.2.5 Write accesses and update transactions

Write accesses are very similar to reads except that one *must* always access the latest version $o_\star$ of an object o (lines 21–23) because a new version will be written at commit time. If the validity range of the latest version does not intersect with the snapshot even after extension, the transaction aborts (line 25).

When writing an object that has already been accessed by the current transaction, the version previously read or written must still be the most recent one. If a new version has been committed in the meantime, the transaction should abort because snapshot validation cannot succeed at commit time.

Informally, an update transaction T performs the following steps when committing: (1) it acquires a unique commit time t_c from the global time base ***clock***, which is atomically incremented (line 32), (2) it validates T (lines 33–36), and (3) it writes new versions of updated objects with timestamp t_c if validation was successful (lines 37–39), or aborts otherwise (line 36).

Update transactions can only commit if their validity range and their unique commit time (i. e., the global version that they are going to produce) overlap, which guarantees

that the transaction is atomic. This is checked during the validation step: $(t_c - 1) \in T.\mathcal{S}$. Therefore, accessed object versions must always be the most recent versions during the transaction.

The way conflicts are detected and new versions are atomically updated will be discussed in Section 3.2.7. One should note at this point that, if a new version of an object accessed by T has been written by another transaction with an earlier commit time $t < t_c$, validation will fail because $T.\mathcal{S}$ will have an upper bound strictly smaller than t and, hence, will not contain $t_c - 1$.

3.2.6 Proof of linearizability

We now sketch proofs that transactions executed by an STM using LSA are linearizable. To that end, we need to show that T takes effect atomically between its start and its commit time. After introducing two lemmas, we demonstrate that this is the case for read-only and update transactions.

Lemma 3.2.1 *For any transaction T that started at time t_s, we have at any time $\lfloor T.\mathcal{S} \rfloor \geq t_s$.*

Proof This property directly follows from the algorithm. $\lfloor T.\mathcal{S} \rfloor$ is initialized with the start time of the transaction and it never decreases (it is always set to the maximum of its current value and another value).

Lemma 3.2.2 *For any transaction T that has accessed at least one object, at any time t we have $\lceil T.\mathcal{S} \rceil \leq t$.*

Proof This property also follows from the algorithm. Each time an object is accessed, $\lceil T.\mathcal{S} \rceil$ is set to the minimum of the current time and another value. Upon extension, it never exceeds the current value of the clock.

With the help of these lemmas, we can now prove that transactions executed with LSA are linearizable.

Theorem 3.2.3 *LSA guarantees that every read-only transaction T that started at time t_s and that successfully commits between $t_c \geq t_s$ and $t_c + 1$ is linearizable.*

Proof T can only commit if its preliminary validity range $T.\mathcal{S}$ is non-empty when it commits. We know from lemmas 3.2.1 and 3.2.2 that $T.\mathcal{S}$ is contained in $[t_s, t_c]$. As $T.\mathcal{S}$ defines by construction a range during which all accessed objects are valid and not updated, T takes effect *atomically* at some time during $T.\mathcal{S}$, which happens *between the start and the end* of the transaction.

Theorem 3.2.4 *Each update transaction T that started at time t_s, that commits at time $t_c \geq t_s$, and that satisfies $\lceil T.\mathcal{S} \rceil \geq t_c - 1$, is linearizable.*

Proof On commit, LSA checks that $(t_c - 1) \in T.\mathcal{S}$ (lines 33–36) and, hence, that all object versions that T has accessed are still valid up to the time t_c when T commits its changes. Since each update transaction has a unique commit time, no other transaction can commit at t_c. This means that, logically, T reads all objects and commits all its updates *atomically* at time t_c, which happens *between the start and the end* of the transaction.

3.2.7 An efficient C implementation

We have developed a C implementation of LSA with several variants and adaptations.

The LSA C implementation is *word-based*, i. e., conflict detection is achieved at the level of memory addresses, and it uses revocable *locks* to protect shared data from concurrent accesses. While LSA allows multiple versions, it uses a *single-version* variant, i. e., transactions can only read the latest committed versions of an object. The goal for this is to keep the implementation as light as possible and avoid extra operations to maintain older versions of objects. We call our implementation TinySTM because of the simplicity of its implementation.

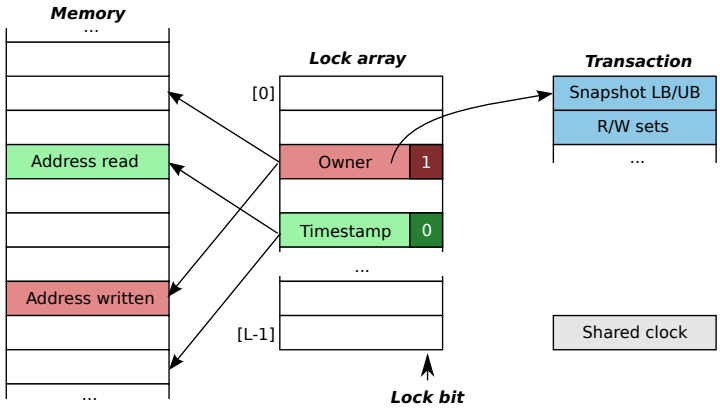


Figure 3.3: Data structures for the lock-based design.

As several other word-based STM designs, TinySTM relies upon a shared array of *locks* to protect memory from concurrent accesses (see Figure 3.3). Each lock covers a portion of the address space.

In our implementation, we use a per-stripe mapping where addresses are mapped to locks based on a hash function. The choice of the hash function is a trade-off between quality and speed. Among all possibilities, the simplest seems to give the best results. The hash function used is only a right binary shift of the address to be written on which we apply the AND binary operation to match the size of the locks array. Since the implementation uses word-based accesses, which mean all addresses are word-aligned, the 2 lowest bits on 32 bits architecture (3 on 64 bits architecture) are unused, so we use this value (2 or 3) as right shift. Additionally, we show that using an extra shift of 2 to the address gives better results because in most applications addresses closed in memory are used by the same thread and are in the same cache line.

Each lock is the size of an address on the target architecture. Its least significant bit is used to indicate whether the lock has been acquired by some transaction. If it is free, we store in the remaining bits a version number that corresponds to the commit timestamp of the transaction that last wrote to one of the memory locations covered by the lock.

If the lock is taken, we store in the remaining bits an address to the owner transaction. Because the lock can be owned by only one transaction, we call it also "ownership record" (ORec). To be accurate, we store a pointer to an entry in the write set of the owner transaction for faster lookup after a write operation. Note that addresses point to structures that are

word-aligned (same for lock hash calculation) and their least significant bit is always zero; hence one of these bits can safely be used as lock bit.

When writing to a memory location, a transaction first identifies the lock entry that covers the memory address and atomically reads its value. If the lock bit is set, the transaction checks if it owns the lock using the address stored in the remaining bits of the entry. In that case, it simply writes the new value in place or in transaction-private write set depending on the write strategy (see Section 3.3.5) and returns. Otherwise, the transaction can try to wait for some time or abort immediately depending on the contention management strategy. By default, we use the latter option in our implementation. Note that the transaction must not wait indefinitely as this might lead to deadlocks.

If the lock bit is not set, the transaction tries to acquire the lock by writing a new value—a pointer to itself and the lock bit—in the entry using a CAS operation. Failure indicates that another transaction has acquired the lock in the meantime and the whole procedure is restarted. If the CAS succeeds, the transaction becomes the owner of the lock.

Our basic design thus implements two lock acquisition approaches: visible writes with objects being acquired when they are first encountered (this approach is usually called "encounter-time locking" or "eager acquire semantics"); lock acquisition is delayed until the end of the transaction ("commit-time locking" or "lazy acquire semantics"), as will be discussed in Section 3.3.4.

When reading a memory location, a transaction must verify that the lock is not owned nor updated concurrently. To that end, the transaction reads the lock, then the memory location, and finally the lock again (obviously, appropriate memory barriers are used to ensure correct ordering of accesses). If the lock is not owned and its value (i. e., version number) did not change between both reads, then the value read is consistent. If the lock is owned by the transaction itself, the transaction returns the value from its write set.

Once a value has been read, LSA checks if it can be used to construct a consistent snapshot. If that is not the case and the snapshot cannot be extended, the transaction aborts.

Upon commit, an update transaction that has a valid snapshot acquires a unique commit timestamp from the shared clock, writes its changes to memory and releases the locks (by storing its commit timestamp as version number and clearing the lock bit). Upon abort, it simply releases any lock it has previously acquired.

3.3 Features and challenges for Transactional Memory

In this section, we discuss some aspects in the design space of the LSA algorithm and implementation related to the performance and to the features provided by the software transactional memory library.

3.3.1 Snapshot extensions

Validation is typically the performance bottleneck of STMs that use invisible reads. LSA only performs validation at commit time (for update transactions), or upon extension when accessing object versions that are more recent than the snapshot's upper bound. One might expect that LSA needs to perform extensions frequently when there are concurrent updates.

However, it turns out that LSA is quite independent of the speed in which concurrent transactions increase time.

If there are no concurrent updates to the objects that a transaction T accesses, the most recent object versions do not change and no extension is required for obtaining a consistent read snapshot. This is the case, in particular, if the value of clock has not changed since the start of T. If clock has been increased concurrently and T is an update transaction that commits at time t_c, one extension to $t_c - 1$ is needed. LSA requires at most one extension per accessed object. However, this worst case is extremely rare in practice because it requires very specific update patterns. In addition, once a concurrent update to an object previously accessed by T is detected, the validity range snapshot becomes closed and no further extension is attempted. Experimental results also suggest that extensions are seldom required.

Figure 3.4 shows the transaction throughput using or not using the snapshot extension with two integer set micro-benchmarks. The benefit of the snapshot extension in these benchmarks is limited. The obvious reason is that TinySTM does not keep multiple versions. Nevertheless, with high update rates, a non-negligible number of extensions lead to a commit especially in the linked list benchmark.

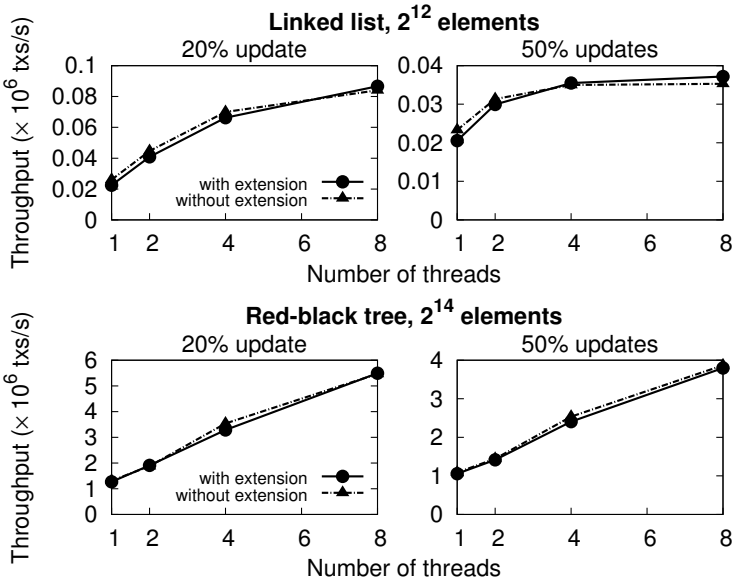


Figure 3.4: Performance of snapshot extensions with the integer set benchmarks.

3.3.2 Global time

Accesses to the global commit time might become a bottleneck when many transactions execute concurrently. In practice, however, the number of accesses to the 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 the algorithm). Time information gathered from the accessed objects can thus be used instead of reading the global commit time.

Note again that the global clock can also be replaced by more scalable alternatives, such as approximately synchronized clocks [54], and various optimizations can be applied to improve performance of the commit phase [68].

3.3.3 Linearizability vs. snapshot isolation

Most STM implementations, including LSA, guarantee linearizability; i. e., each transaction appears to take effect atomically at a point between its start and its commit time. Some STMs guarantee serializability (e. g., [5, 51]) in an attempt to increase the commit rate of the transactions, but they require more complex algorithms [3] and are not competitive in terms of performance.

LSA can be configured to provide *snapshot isolation* [7] semantics. The idea of snapshot isolation is to take a consistent snapshot of the data at the time when a transaction starts, and have all its read and write operations performed on that snapshot. When an update transaction tries to commit, it must acquire a unique timestamp that is larger than any existing start or commit timestamp. Snapshot isolation does not guarantee serializability but avoids common isolation anomalies like dirty reads, dirty writes, lost updates, and fuzzy reads. Snapshot isolation is an optimistic approach that is expected to perform well for workloads with short update transactions that conflict minimally and long read-only transactions. This matches many important application domains and slight variations of snapshot isolation are used in common databases.

When configured for snapshot isolation, only three minor modifications are necessary to the algorithm of Figure 1. First, no extensions are performed upon read or write (lines 9 and 20). Second, all read accesses are directed to the object versions that were valid at the start time of the transaction (lines 10–15). Third, validation is omitted upon commit (lines 33–36). It naturally follows that, when keeping sufficiently many versions, transactions can always commit except in case of write/write conflicts when executing under snapshot isolation.

Algorithms typically need to be adapted for snapshot isolation. Unlike linearizability, snapshot isolation permits read/write conflicts. In our experience, this makes algorithms more difficult to design because a programmer needs to identify which read/write conflicts need to be detected and convert them into write/write conflicts. For example, when removing an element from a linked list, one would need to add an extra write to the node that is removed. This prevents a concurrent transaction to insert a new element right after the removed one. Such a conversion is not always easy, e. g., trying to modify a red/black tree to support snapshot isolation proved to be more difficult than expected. Since the performance

[3] Unlike linearizability, serializability is not a local property. Serializable STM algorithms must typically maintain (partial) transaction dependency graphs at runtime.

improvement of using snapshot isolation instead of linearizability appeared to be minimal [53], we only support linearizability.

3.3.4 Encounter time locking vs. commit time locking

Conflict detection can be done in an optimistic manner or in a pessimistic manner by acquiring the lock associated to the memory accessed. We are giving some advantages and drawbacks for both.

The pessimistic conflict detection is based on early locking or Encounter Time Locking (ETL). Locks are acquired at write access and all conflicts are checked on all transactional accesses. This early detection reduces the time passed in the commit phase. In contrast, late locking or Commit Time Locking (CTL) acquires locks at commit which make the time passed in commit phase dependent of the number of writes.

CTL allows more concurrency in some cases because it allows multiple writers so it can result in different performance benefits. For instance, two transactions that are writing the same memory location but not reading it can commit without any conflict. While with ETL a conflict will be detected, CTL permits such interleaving.

CTL suffers from slower read accesses after speculative write because to detect such access pattern, it must check if the address read has not been previously written. To that end, it relies on a bloom filter to reduce the number of traversal into the write set buffer for each speculative read access. Moreover, a lot of work can be wasted by letting a transaction process until the commit when a conflict is inevitable whereas ETL detects early the conflict, which avoids this amount of work and may improve overall performance.

However, the contention management with CTL is less precise because of conflicts detected at commit time. Conflicts cannot be solved with a clever strategy because the conflicting transaction is likely to have already committed.

In Figure 3.5, we can observe that in some cases CTL can improve the performance but ETL is generally better.

3.3.5 Eager vs. lazy versioning

Encounter time locking permits two types of version management: Eager or Write-through (WT) and Lazy or Write-back (WB).

STM must record transactional writes to be able to abort and to undo the changes. The write-through (WT) version stores the modifications in place and records its old versions (undo-log) elsewhere in case the transaction has to abort and roll back changes. The write-back (WB) version stores the modifications in a backup area and only applies them once a transaction commits successfully.

WT and WB offer a tradeoff between a fast commit and a fast abort. WB has cheap aborts because there are no speculative updates inside the transaction but transaction commits are expensive due to the need of writing data to shared memory. Conversely, WT pays an overhead on transaction abort as transactions must process their undo-logs but the commit is fast. So in a highly contended application, it is beneficial to use WB instead of WT.

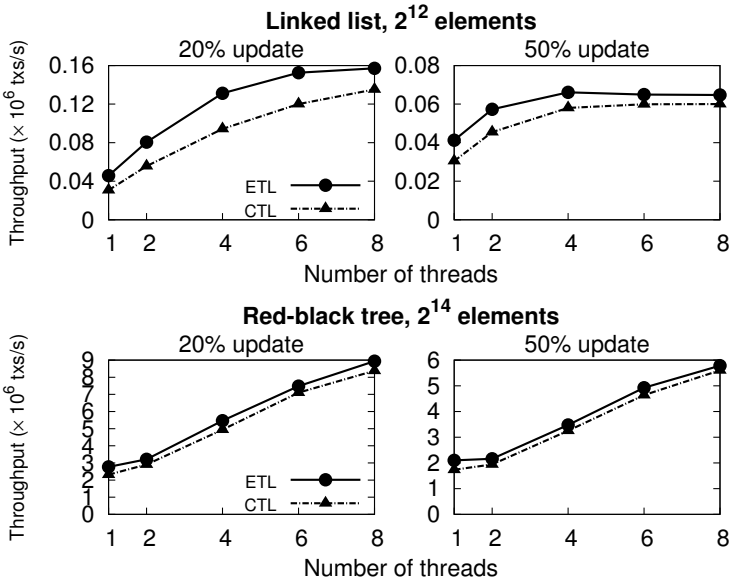


Figure 3.5: Throughput for the linked list and red-black tree micro-benchmarks.

WT would be the best choice since transactional memory is optimistic and it should give an interesting performance gain over locks when aborts are unlikely. Moreover, WB adds more pressure on the cache coherency protocol by writing all shared data at the commit phase. We have implemented WB and WT in a way that they can be mixed in order to provide the best approach depending on the benchmark. This adaptation could be done automatically like in [49], where the runtime measures the abort rate and decides to switch from one to another regarding a threshold.

Figure 3.6 shows that in the case of micro-benchmarks no major performance difference is visible. There is no real winner so we implemented both versions such that they can be changed depending on the application.

3.3.6 Garbage collection support

Garbage collection is a mechanism by which memory allocated by the application is automatically freed when it is no longer referenced. This relieves the need for the programmer to explicitly call a "free" method to deallocate memory.

In the context of TM, the garbage collection translates to a delayed free: allocated areas are freed only when no other threads can access the corresponding memory. Internally, STM is using metadata objects such as read and write sets, which may be accessed by other transactions. The object reference is enqueued into a thread-local queue of objects to be freed, along with an associated timestamp. This timestamp indicates when the freeing request was issued. When all active transactions have started later than this timestamp the object's memory can be safely freed. In order to avoid huge overheads, the queue, which is naturally ordered, is periodically checked.

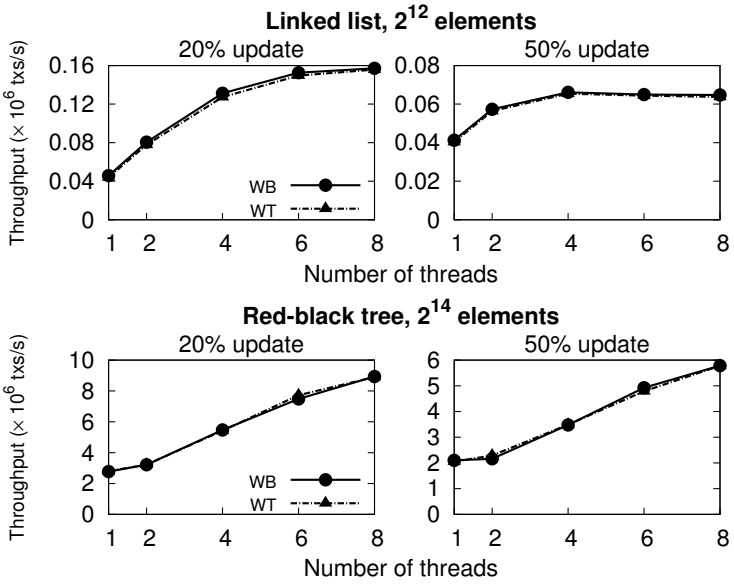


Figure 3.6: Throughput for different versioning for the linked list and red-black tree micro-benchmarks.

This garbage collection is required for designing advanced contention managers because a transaction may need to read the state of another transaction and we must guarantee its consistency.

3.3.7 Advanced contention managers

When a conflict occurs between two transactions, one of them has to wait or to roll back and retry. The restart can lead potentially to a significant waste of computation cycles. The goal of a contention manager is to reduce contention in order to solve difficult conflicts such as deadlocks or livelocks and also to improve performance by avoiding conflicts.

The straightforward *suicide* contention manager aborts the transaction that detects the conflict. But even if this CM usually gives good performance results, it may waste a lot of processor cycles. Indeed, if a transaction detects a conflict and then rolls back and retries, the same conflict is likely to happen again if the other conflicting transaction did not commit yet. A trivial solution for this problem is to back off for a bounded duration. Unfortunately, it is not enough precise and usually leads to slightly lower performance.

A better solution could be to wait until the conflicting transaction commits but then this requires reading the status of other transaction from its descriptor. While it seems straightforward to do, we must ensure that the status that is read is still associated to the same transaction and that the descriptor has not been freed. In our implementation, the transaction descriptor is allocated only once when the thread is created. It is then re-used for all transaction in the same thread. This allows avoiding extra overhead from the memory allocator upon transaction abort.

We use the garbage collector for all transaction descriptors to allow reading a descriptor of another transaction. We also implement another feature to *kill* a transaction (abort an enemy transaction) and to steal the locks it currently holds. This feature is required to be able to implement most of the smart contention manager, as described in [60].

Lock stealing We considered the transaction status as a shared resource that can be modified by any other transaction. To ensure coherency, all modifications from an active state to non-active state are done using the atomic operation Compare And Swap (CAS). An active transaction checks its status at each transactional operation to ensure it was not killed by a concurrent transaction. When a transaction observes that it was killed, it releases the acquired locks using Compare And Swap (CAS) in order to allow lock stealing by the other transactions.

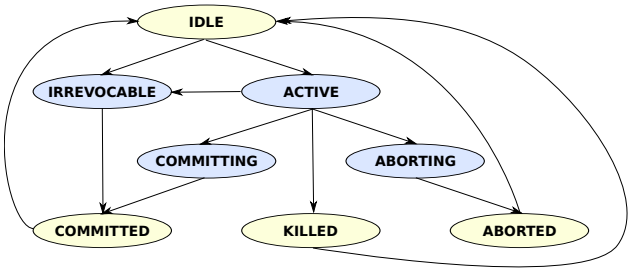


Figure 3.7: State diagram of transaction status with Advanced Contention Manager.

Lock stealing works thanks to the thread-safe change of transaction mode, which ensures that the transaction is in safe state. Figure 3.7 shows how a transaction changes from active status to an inactive status. A regular transaction starts by changing its status from IDLE to ACTIVE. When the transaction needs to commit or abort, its status is changed from ACTIVE to COMMITTING (respectively ABORTING) using a CAS operation. This CAS operation ensures that no other transaction modifies the status concurrently. The change from COMMITTING to COMMITTED (respectively ABORTING to ABORTED) is done without using an atomic operation by the local thread, as these are inactive states.

In the case of lock stealing, the offender transaction first changes the status of the conflicting transaction from ACTIVE to KILLED (if this one was not already killed, which is verified as part of the CAS operation). Then, the offender transaction tries to steal the lock using CAS. If the stealing fails it will retry by reading the lock again. The killed transaction releases its acquired lock using CAS, which will fail if the lock was stolen.

We also use a versioned status which is incremented when a transaction becomes active to avoid killing a wrong transaction (avoid ABA issues). Indeed, without the use of this incarnation counter, the thread may have committed the conflicting transaction and started a new active transaction since the transaction state is reused for all transactions of the same thread. The incarnation counter enables to distinguish those two transactions.

Figure 3.8 shows that the contention manager has not a big impact on performance but the progress guarantee is different. In this case, the "suicide" strategy gives better results than other strategies. Finally, it is not trivial to decide which of the contention manager is

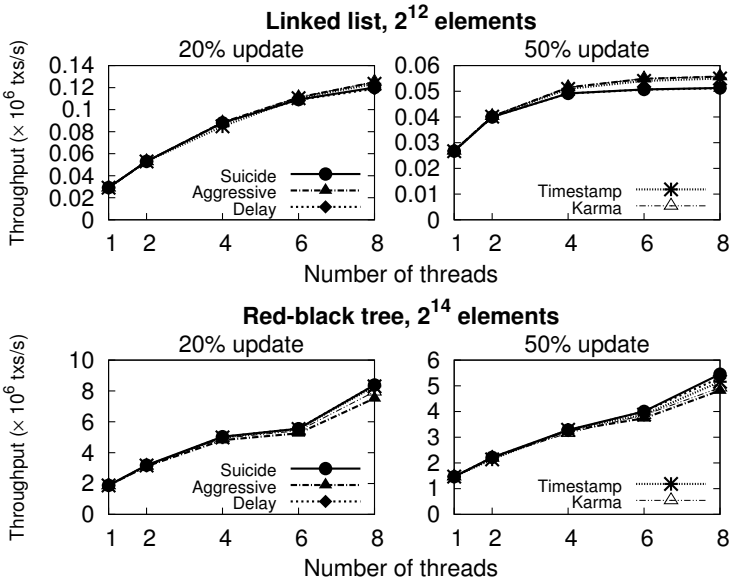


Figure 3.8: Throughput for different contention managers for the linked list and red-black tree micro-benchmarks.

best, as performance seems to be workload dependent. However, some of them work better than others across a wide range of workloads [60].

3.3.8 Visible read barriers

If other transactions are not able to detect that a transaction has read a piece of memory, it is called an invisible read. As a contrary, if other transactions have a way to detect this read before the commit of the other transaction, the read is called a visible read (VR). Our implementation, like several others [19, 21], relies on an execution mode with invisible reads to be very efficient in situations that induce few conflicts. However, it does not provide strong guarantees of progress.

As transactions use invisible reads, read/write (R/W) conflicts are not detected when they happen (conflict with the read happening before the write). A transaction might thus have to abort when discovering upon validation that it has read a memory location that has since been overwritten by another committed transaction. Even without considering invisible reads, a transaction may abort an unbounded number of times because its writes conflict with those of other update transactions.

Both issues are problematic because, as transactions may repeatedly abort, one cannot easily bind their execution time and ensure a progression. Priority-based contention managers [60] would not solve the problem because, with invisible reads, read/write conflicts are not detected as they occur.

Visible read mode (VR) allows an update transaction to detect read-write conflicts with a reader transaction. The motivation is to detect R/W conflicts as they happen, and thus to favor the reader in VR mode over other transactions executing in the optimistic

mode. Indeed, some applications use long read-only transactions which are prone to never commit since writers are always winner in invisible read mode. This VR mode allows any conflicting writer to back off and let the reader complete its execution. It enables a read-mostly transaction to make progress while reducing the probability of its abort.

Implementation To implement visible reads, one may consider simulating a visible read by a write. However, this solution would trigger R/R conflicts with regular transactions, and a transaction using VR would thus prevent others transactions from committing even when there is no real W/R or R/W conflict and vice versa. Some STMs (e. g., SXM [30]) implement visible reads by maintaining a list of readers for each shared object. With such an approach, one can keep track at each point in time of the number and identity of the readers, and allow multiple readers or a single writer to access the object. Writer starvation can be prevented by letting readers "drain" as soon as a writer requests ownership of the lock. The main drawback of this approach is that it imposes a significant overhead for the management of the reader list and creates additional contention. To address these problems, SkySTM [41] implements "semi-visible" reads by just keeping a counter of readers for each memory location.

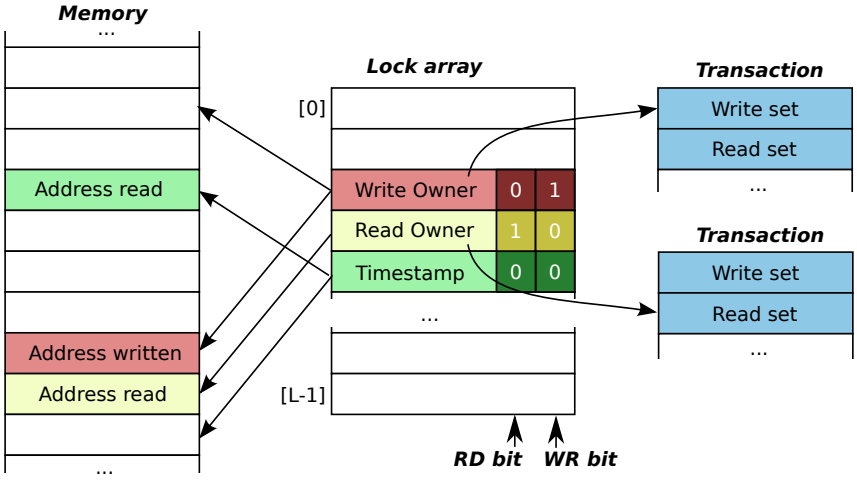


Figure 3.9: Description of locks with the visible read bit.

We propose an even more extreme approach relying on a single additional bit in the orec to indicate that associated memory locations are being read by some transactions. The bit is atomically set using a CAS operation when reading the associated memory location for the first time. A single visible reader is allowed at a given time and only if there is no writer—unless the writer is the same transaction that performs the visible read. Therefore, a visible reader behaves almost identically to a writer, with one major difference: *there is no conflict between a visible reader and a transaction accessing the same memory location optimistically*, i. e., with transactions that use invisible reads.

The rationale behind this design choice is that transactions will seldom use visible reads. In fact, the VR mode will be carefully used only if transaction fails to commit in the

optimistic mode. An additional bit is added for indicating a read-lock to all the locks which is possible thanks to natural alignment of bytes (see Section 3.2.7).

This bit prevents writers to acquire the lock while letting readers proceed. We allow only one visible reader per orec to avoid extra overheads for the maintenance of the number of visible readers.

To that end, in addition to the WR bit used for writers, we use an additional RD bit in the lock metadata to indicate that a transaction is reading the associated data (see Figure 3.9). An invisible reader can read data that is locked in read mode. To obtain the associated timestamp, it must peek into the read-set of the thread that locked the data. Conflicts with visible readers are handled as for writers, i. e., only one transaction is allowed to proceed. The use of visible reads makes all conflicts detectable at the time data is accessed: a well-behaved transaction that wins all conflicts is guaranteed not to abort.

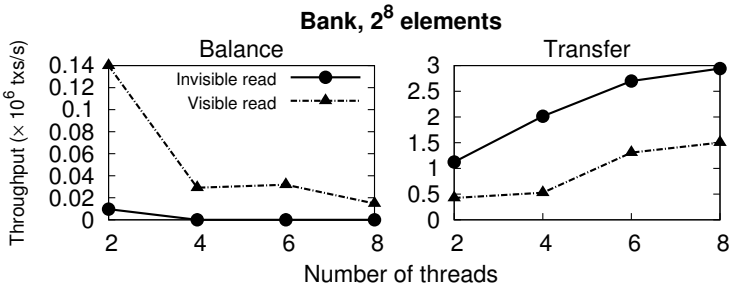


Figure 3.10: Throughput of balance and transfer transactions for the bank micro-benchmarks with and without visible read.

Figure 3.10 presents a result with the bank benchmark (see Chapter2) with one thread doing balance operations only and others transfers only. It shows that the visible read mode is fairer with long transactions. Moreover, the invisible read mode with this specific workload suffers from lack of progress when the number of threads increased. Of course, the throughput of transfer operations is lower with visible reads because long read-only transactions prevent transfers to proceed.

3.3.9 Read locked data

One problem which can occur with early lock acquisition (see Section 3.3.4) is that when a lock is acquired for a write, no reader transaction is allowed anymore to access this memory element. However, we can provide even more concurrency by letting readers read the previous value from memory, before the write. Indeed, the read can read into the undo log of the concurrent writer transaction with write-through design (or from the memory with write-back design).

Unfortunately, as we can see in Figure 3.11, this improvement doesn't give the expected improvement because the transaction is then likely to abort due to the conflict. Moreover such mechanism increases the code complexity and disturbs the cache coherency protocol because the reader transaction will fetch data from writer transaction.

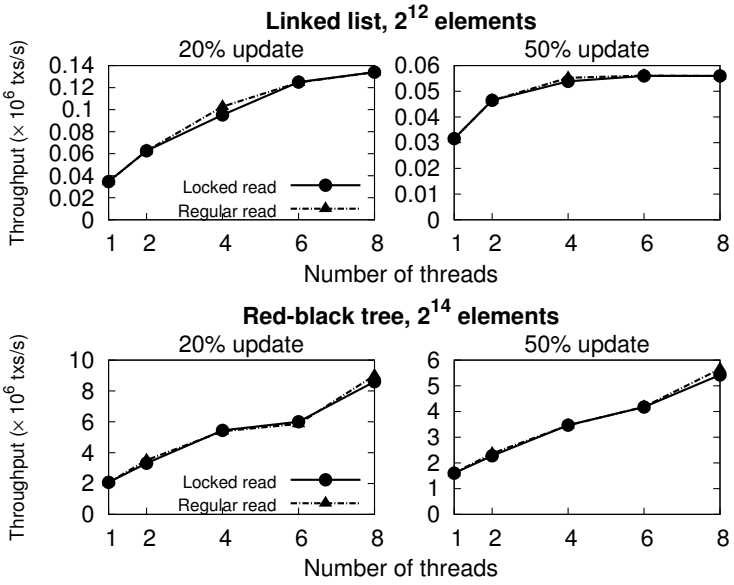


Figure 3.11: Throughput for the linked list and red-black tree micro-benchmarks with and without read locked data barrier.

3.3.10 Local memory barriers

In a transactional program, some pieces of memory like stack allocated variables are thread-local. If these are modified in a transaction, they have to be backed up and restored to their original value when the transaction aborts. While transactional store can be used to buffer the value, it is prone to create false sharing conflicts. To that end, we develop specific functions to only record previous values in case of transaction abort.

These functions were implemented as a transaction aware module (see Section 3.3.12).

3.3.11 Irrevocability

Transactions typically execute optimistically and in case of conflict roll back and retry their execution. Unfortunately, some operations are irreversible, which makes then impossible to use in a transaction. For example, I/O (e. g., keyboard input) or system calls whose behavior cannot be compensated are not compliant with undoable transactions.

Therefore, we implemented a specific transaction mode, the *irrevocable* mode that the transaction can switch to, to deal with such operations. An irrevocable (also called inevitable [63]) transaction is guaranteed not to abort and will eventually commit. Interestingly, an irrevocable transaction can avoid keeping track of all operations done in memory. This allows running faster than an optimistic transaction.

Serial irrevocable

The serial irrevocable transaction is the only manner to fully support I/O and libraries or system calls with unpredictable write sets within transactions.

A simple implementation of the irrevocable mode is to execute an irrevocable transaction alone once no other transaction is in progress (serial mode). To become irrevocable, a transaction T first gains exclusive permission to perform inevitable operations by acquiring a global token.

While this approach is safe, it does not provide any concurrency and should only be used as a fallback mechanism for special situations such as I/O.

Concurrent irrevocable

Because serial irrevocable mode is a performance killer, we propose a more promising approach to allow concurrency between an irrevocable transaction and other non-irrevocable transactions. Several such algorithms have previously been discussed and evaluated [63, 65].

For our new variant, we assume that the irrevocable mode is seldom used and, hence, should work jointly with optimistic transactions. We limit the system to allow only one irrevocable transaction at a given time because it is the only way to ensure that there will be no abort when the read and write-set are not known in advance. Otherwise, one can trivially construct an interleaving with just two transactions that leads to a deadlock.

Our implementation follows the general design of previous approaches [63, 65], by using a global token that a transaction must acquire before it becomes irrevocable. Once the global token has been acquired, no other update transaction can commit.

A transaction can request to enter irrevocable mode at any point in its execution. If the transaction has already accessed some shared object, it must validate its read set before irrevocability can be granted. Failed validation triggers an abort and the transaction directly restarts in irrevocable mode. Since an irrevocable transaction is guaranteed to never abort, in case of a conflict the other conflicting optimistic transactions will systematically abort. Interestingly, we allow a read-only optimistic transaction to commit while an irrevocable transaction is in progress, but delay the committing of optimistic transaction with updates until the commit of the irrevocable transaction. This optimistic approach permits non-conflicting transactions to execute concurrently while allowing for interesting optimizations in irrevocable transactions: they do not need to use visible reads or to validate the timestamp of read values, or even to maintain a read set, resulting in a reduced overhead.

In Figure 3.12, we show that the serial irrevocable mode reduces largely the throughput of the benchmarks even if only 5% of transactions are irrevocable. As a contrary, the concurrent irrevocable mode permits to achieve a better scalability compared to the serial irrevocable mode and it should be used instead of serial irrevocable mode when possible.

3.3.12 Extensibility

The extensibility of a transactional memory library is essential for TM developers to propose new features and for users to deal with their own specific problems. We develop a callback mechanism inside the library to notify external code of internal events. These events are raised on transaction start, commit, abort, restart, and conflict. These events are only notifications, except for the conflict callback. This latter callback also allows taking decisions on conflict management.

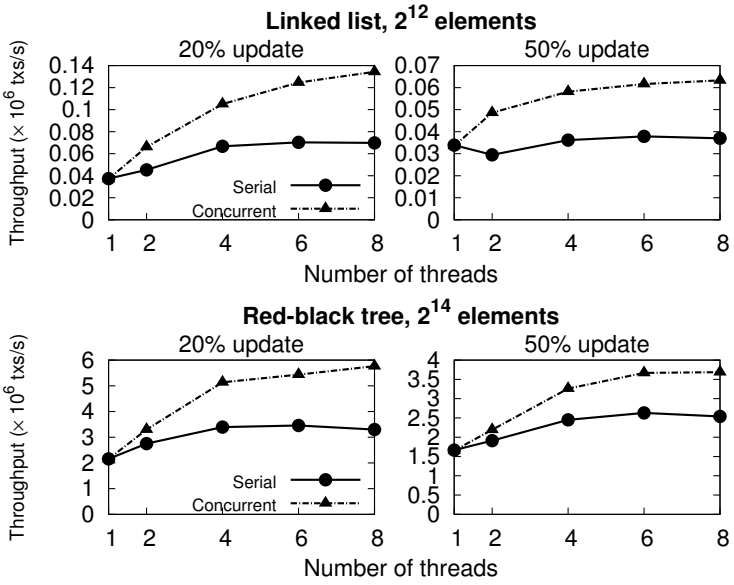


Figure 3.12: Throughput for the linked list and red-black tree micro-benchmarks with 5% of irrevocable transaction.

The modules can be registered dynamically at run-time in order to provide a streamlined and fast version by default that can be enriched with additional features afterwards. For example, the application developer may need to know when a transaction rolls back because he has acquired a lock in an external library. Such mechanism also provides support for external actions.

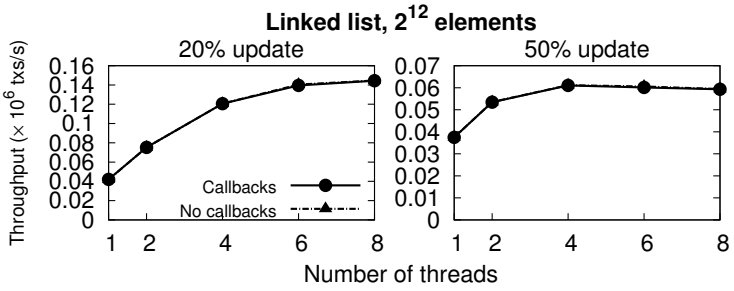


Figure 3.13: Throughput for the linked list micro-benchmark with and without callback enabled.

In Figure 3.13, we show that the overhead due to the additional code for callbacks is minimal and this mechanism can be added at low cost.

3.3.13 Memory allocation

Applications need to allocate memory dynamically, and this includes allocation and deallocation within a transaction. It is thus necessary to support memory allocation inside transaction. Unfortunately, it is not trivial to address this challenge in an unmanaged environment.

Consider the case of a transaction that inserts an element in a dynamic data structure such as a linked list. If memory is allocated but the transaction fails, it might not be properly reclaimed, which results in memory leaks. Similarly, one cannot free memory in a transaction unless one can guarantee that it will not abort.

Two solutions can be provided:

- Using the system memory allocator and provide *compensation action* (See Chapter 5) on abort for allocation and do a delayed action on commit event for deallocation.
- Propose a new memory allocator which is aware of underlying transactions as proposed by Hudson in [33].

We choose to provide memory-management functions that allow transactional code to use the system dynamic memory allocator. As for the garbage collection (see Section 3.3.6), transactions keep track of memory allocated or deallocated: Allocated memory is automatically disposed of upon abort, and freed memory is not disposed of until commit (unless it was allocated in the same transaction). Further, a transaction can only free memory after it has acquired all the locks covering it as a free is semantically equivalent to an update.

Padding and alignment of allocated memory is essential in the context of transactional memory because otherwise it could lead to false sharing problem (see Section 2.1.1). Luckily, most of modern memory allocators like Doug Lea memory allocator or Hoard [8] memory allocator take care of those problems. Additionally, the change of the default memory allocator to Hoard enables some applications to obtain better performance thanks to its efficient design for multi-core.

A particular case is the memory allocated for the TM metadata, e. g., ownership records (locks) array, read and write sets. Indeed, we could imagine that the memory allocator can use transactions in its implementation but transactions require dynamic memory to work. So the TM library should use system calls to allocate its metadata to avoid this problem.

3.3.14 Transaction descriptor

The transaction descriptor is the principal metadata of a TM. In our implementation, it contains information about the transaction status, the validity range for the LSA algorithm, the read and write sets, and others attributes and statistics. A careful optimization of it is required because it is accessed often, i. e., in all transactional operations. The alignment, the size and the padding of this structure can make the performance of the algorithm degrade so all unnecessary data like statistics can be removed for a released version.

Yet, the transaction descriptor has to be recorded for each thread in order to gather all information of the same transaction. To that end, we propose two alternative mechanisms:

- Application intrusive: the application code is modified to include the transaction descriptor, which will be explicitly used for transactional operations. It uses one register

in the user program state to keep a pointer to descriptor. This may degrade performance in particularly in case of CPUs with few general purpose registers like x86/32bits.

- Library implicit: the transaction descriptor is implicit for the current thread and the transactional memory library manages it by itself. The library uses thread-local mechanism proposed by the system to save the descriptor pointer.

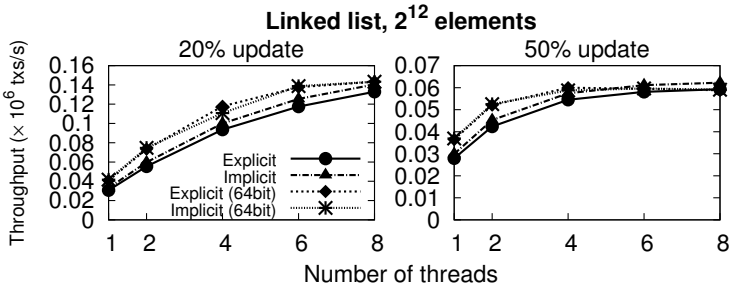


Figure 3.14: Throughput for the linked list micro-benchmark with explicit and implicit transaction descriptor.

In Figure 3.14, we see that the implicit transaction descriptor gives better results than explicit transaction descriptor with 32 bits architecture. Indeed, as explain before, the low number of register available in x86/32bits penalize the execution. In x86/64bits, we observe that the implicit or explicit have the same performance.

3.3.15 Fast path

Transactional memory uses an optimistic execution of code and thus we can assume that in most of the case the TM library executes only the "without conflict" part of code. So we can use this assumption for designing a fast path for the code that is the most probable, the path without conflict.

Code locality STM operations are costly operations because each individual load or store corresponds to a large number of instructions. By keeping the implementation lightweight, the number of instructions to process will be lower. Moreover processors have a limited instructions cache so limiting the size of code will exploit code locality to improve general performance of the program.

Inlining Inlining a function means that the function will be expanded directly where the function is called. The good effect of inlining is to avoid the overhead of calling a function. Indeed, calling a function requires preparing function arguments, saving return address, changing instruction pointer and clobbering some registers. Unfortunately in some cases of long and hotspot functions, the generated code becomes too big and all the benefit will fade out due to the bad code locality. Globally, there is a tradeoff between code locality and calls overheads.

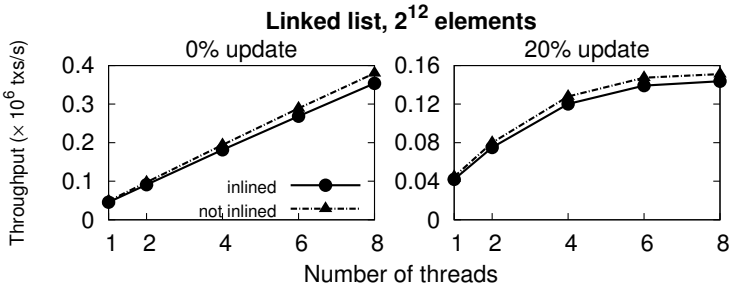


Figure 3.15: Throughput for the linked list micro-benchmark with and without inlined rollback.

In Figure 3.15, the impact of inlining the rollback is reducing the code locality, particularly when the rollback is unlikely. The improvement is up to around 7% compared to the inlined rollback function.

Branch Predictions Additionally, all branches in the STM library code can be instrumented to improve code locality but also the branch prediction for the CPU.

```
void f(){
  if(condition) {
    // 1 cache line used here
  }
  // 1 cache line
  return;
}
```

Listing 3.1: Example of optimization using branch predictions

In Listing 3.1, the compiler will generate the code naively and the "if" block will be expanded directly at the beginning of the function. One instruction cache line will be used for the condition but this cache line could be wasted if the condition is unlikely. To avoid such situation, the STM library source code is tuned to inform the compiler of condition likelihood.

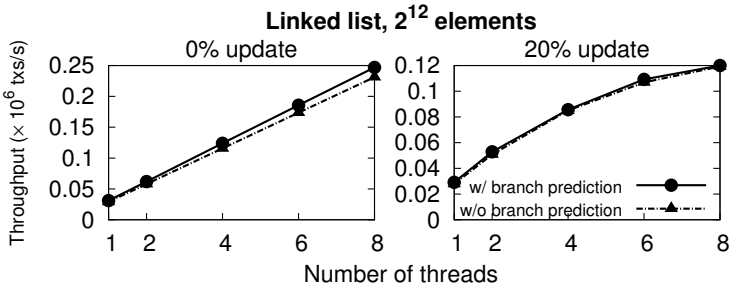


Figure 3.16: Throughput for the linked list micro-benchmark with and without branch prediction for visible read.

In Figure 3.16, we show that the tuning of branch prediction for a condition can improve performance up to around 7% with no update workload independently of the number of threads.

Padding and alignment As in all multi-threaded programs that use multi-core CPU, the padding and the alignment of data are important for performance because they avoid cache-line false sharing effects. All metadata uses the exact size required to avoid wasting memory cache and tries to be as simple as possible. Global variables of the STM library are glued together if they are used for the same purpose and then aligned and padded to fit a cache-line which avoids false-sharing. Transaction descriptors aggregate all data used in the same context together and also frequently used data (hotspot data).

We apply all these optimizations to have a STM designed for multi-core and we implements all these features to be easily integrated within a transactional program. We will show in the next section the performance evaluation of the constructed TM library.

3.4 Evaluation of a LSA implementation

We now evaluate the performance of our LSA implementation in C, called TinySTM.

We compare the ETL-WB variants of TinySTM that use *encounter-time locking* (i. e., locks are acquired at the time data is written) and a *write-back* update strategy (i. e., writes are buffered until commit time) with the x86 port of TL2 [19].

3.4.1 Micro-benchmarks

Figures 3.17, 3.18 and 3.19 evaluate the throughput of TinySTM with the integer set micro-benchmarks. We first observe that TinySTM systematically outperforms TL2 by a small margin. Part of this difference can be explained by the extension mechanism of LSA, which helps improve throughput over TL2 especially with high update rates. Scalability is good for all workloads, except write-dominated linked list where the cost of aborts is high due to the large number of transactional accesses. Remarkably, all STMs scale well with the skip list and red-black tree benchmarks even with 100% updates.

3.4.2 Realistic applications

STAMP benchmarks

We now evaluate our STM implementation on STAMP [9], a set of realistic benchmarks described in Chapter 2.

We ran tests using all applications but `bayes` and `yada`. We have observed non-reproducible behavior for Bayes with several TM implementations and Yada has extremely long transactions and does not show any scalability with any of the TMs we analyzed.

Performance results of Figure 3.20 represent the scaling factor compared with a sequential execution without STM. While not all applications benefit as much from using STM, one can observe that both TinySTM and TL2 exhibit good scalability up to 8 cores.

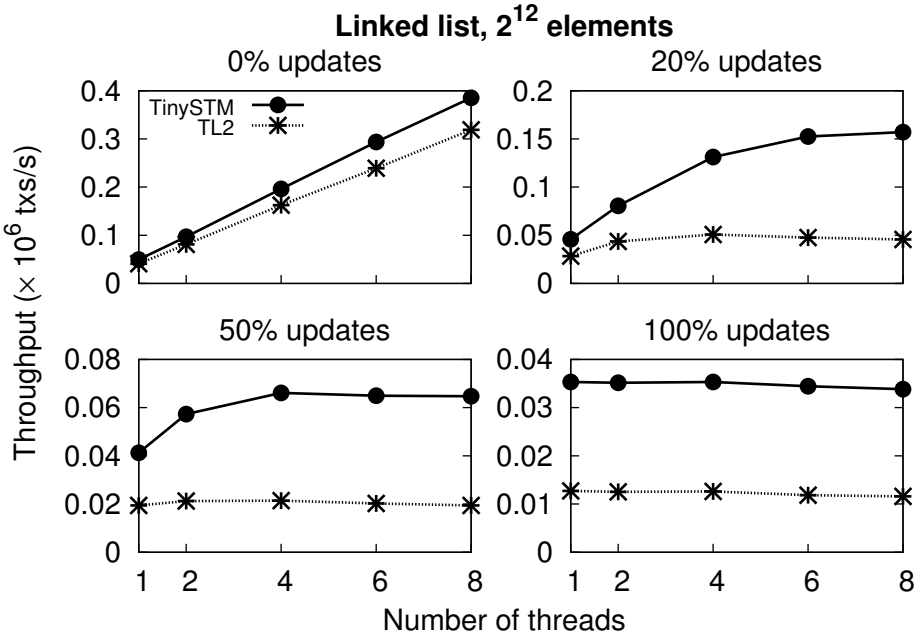


Figure 3.17: Throughput for the linked list micro-benchmarks with different update percentage.

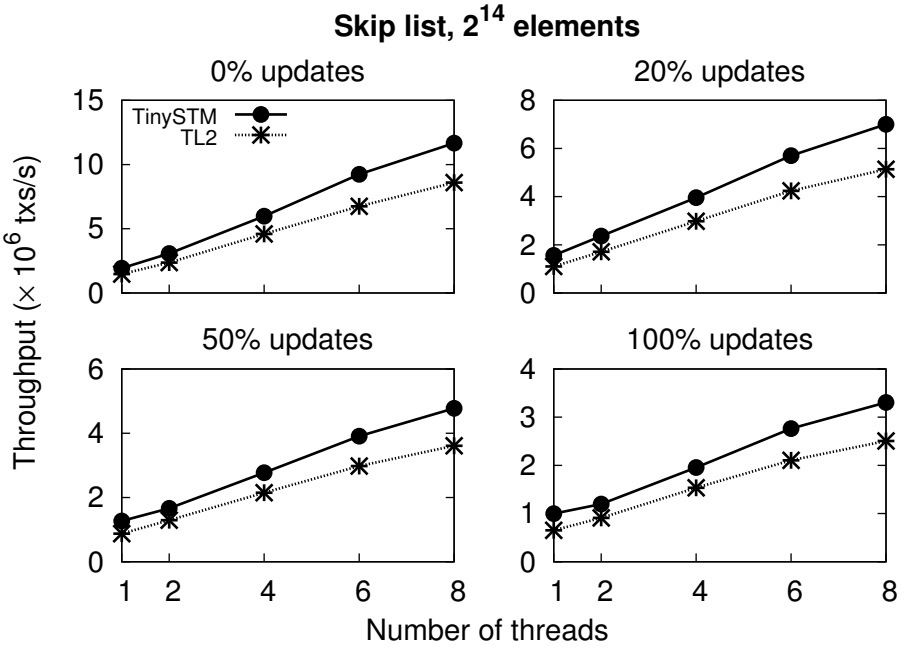


Figure 3.18: Throughput for the skip list micro-benchmarks with different update percentage.

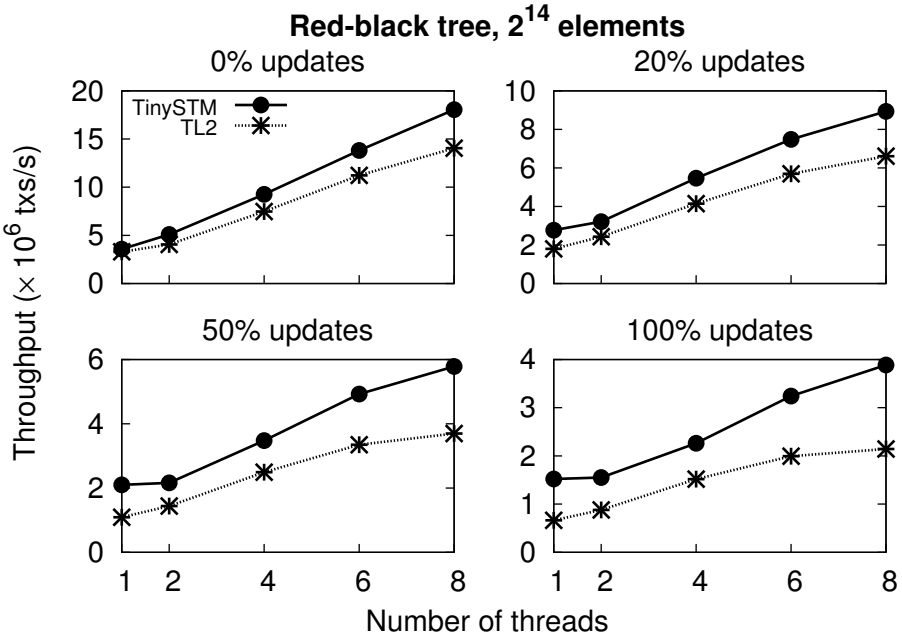


Figure 3.19: Throughput for the red-black tree micro-benchmarks with different update percentage.

The performance of TL2 is slightly lower on most experiments, which can be again explained by the differences in the underlying algorithms.

3.5 Conclusion

In this chapter, we described and implemented a TM algorithm based on timestamps. This algorithm, named LSA, has a high scalability potential for future STMs. Our performance goal drove our implementation choices with a specific attention to CPU constraints such as false sharing. We proposed additional mechanisms to give better transactional guarantees and to ease the library utilization for an adoption of TM by application developers. Finally, the flexibility of our TM library allows extensions for different usages.

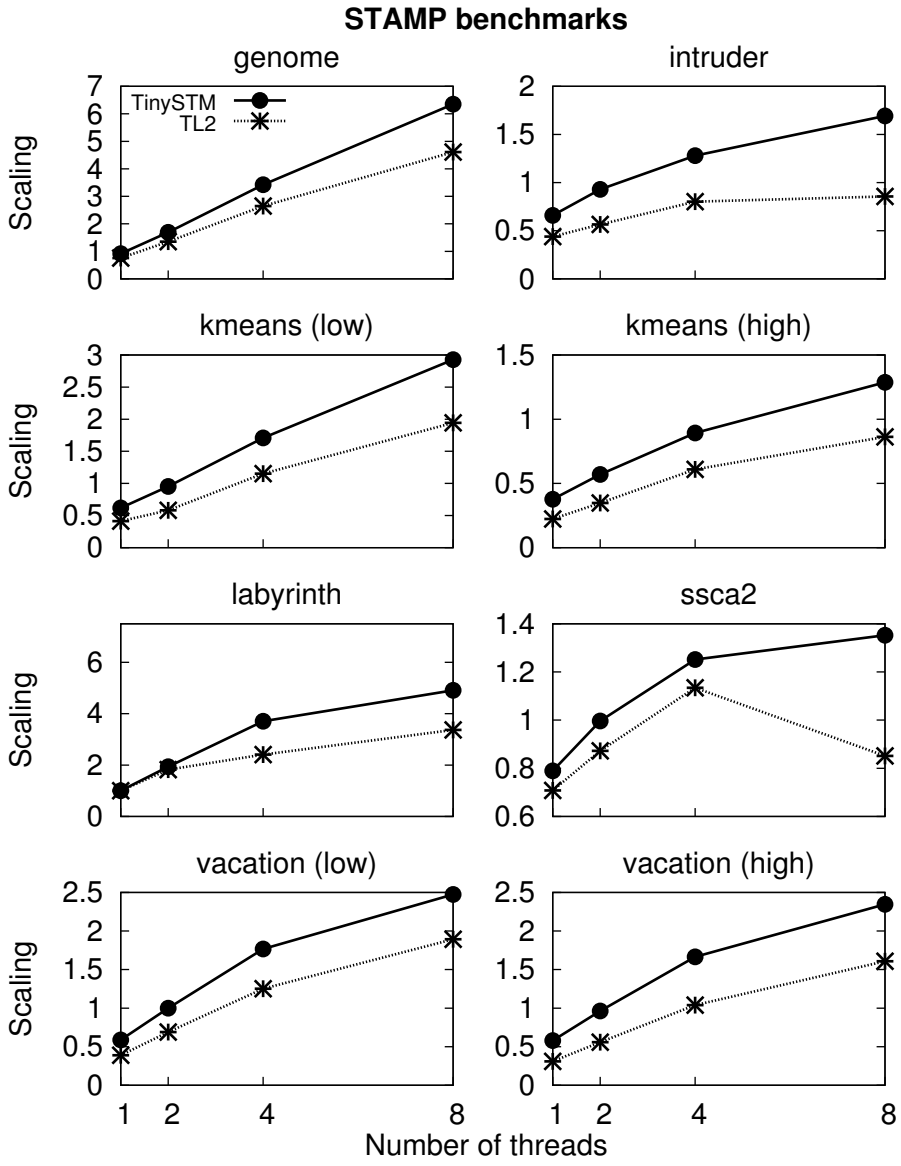


Figure 3.20: Scalability of TinySTM (ETL-WB, WTL-WT, CTL) and TL2 with STAMP benchmark suite.

Chapter 4

Hardware Support for Transactional Memory

The previous chapter presented the design of Software Transactional Memory (STM) with a focus on performance and scalability. Most current TM implementations are software-based [16, 19, 25, 45, 57]. STMs typically reach a good performance compared to fined-grained locking with large number of cores. When the number of cores is low, STMs exhibit significant overheads due to all instrumented accesses. This led some researchers to claim that software transactional memories (STMs) are only a research toy [11]. These overheads can be significantly lowered through hardware support.

While there are at least two industry implementations for hardware support for TM [14, 18], they are not directly available to public. HTM proposals such as [47] involves complex modifications to processor architecture to support any read/write sets size. Among all hardware proposals, hardware-supported transactional memory consists of extensions to current CPUs but with more restrictions such as limited read/write capacities. CPUs tends to maintain ascending compatibility i. e., old programs still run on current CPU and those proposals seem more realistic in the near future.

To obtain the best performance, hardware is used for most transactions and software is used for other transactions to overcome these capacity limitations. To get a chance to be widely adopted, Transactional Memory has to show sufficiently good performance with low number of threads and also a good scalability when the number of threads increases.

First, we survey and discuss the different HTMs proposals. Our approach is based on AMD's ASF ISA extension, a hardware support for transaction proposed by an industrial manufacturer. In Section 4.1, we first evaluate if this hardware extension can help speed up concurrent applications that use speculation. This section is largely based on our conference paper [13]. In Section 4.2, based on this hardware support and on our STM, we propose new hybrid transactional algorithms that use AMD ASF instructions while allowing STM to execute concurrently. These algorithms were published in our conference paper [55] We show in both cases that ASF provides good scalability on several of the considered workloads while incurring much lower overhead than software-only implementations.

4.1 Hardware Transactional Memory

In this section, we first analyze proposed hardware supports for transactions and we select one of them, AMD's Advanced Synchronization Facility (ASF). AMD's ASF [2] is a proposal of extensions for x86_64 ISA. In Section 4.1.2, we continue with a description of the ASF specification and implementations.

Then, our objective is to evaluate if the hardware extension proposed by AMD can help speeding up the speculative execution of atomic blocks. To that end, we used ASF extensions implemented in a near-cycle-accurate AMD64 simulator. This simulator (PTLsim) mimics ASF cycle costs and pipeline interactions that we would expect from a real hardware implementation. We create a TM library that uses ASF and we extend it to use a software-based fallback solution when ASF cannot execute a transactional block (e. g., because of capacity limitations). In Section 4.1.4, we evaluate our implementation with transactional software. Due to the lack of real applications with atomic blocks, we use a set of standard TM benchmarks in our evaluation.

4.1.1 Proposals and related work

The first hardware TM design was proposed by Herlihy and Moss [31]. A separate transactional data cache is accessed in parallel to the conventional data cache. Introducing such a parallel data cache would be intrusive to the implementation of the main load-store path. Microprocessor manufacturers are conservative with modifications to the micro-architecture due to its complexity. Proposals that require little modifications are more likely to have industry uptake.

Shriraman et al. [62] propose two hardware mechanisms intended to accelerate an STM system: *alert-on-update* and *programmable data isolation*. The latter mechanism, which is used for data versioning, relies on heavy modifications to the processor's cache-coherence protocol: the proposed TMESI protocol extends the standard MESI protocol (four states, 14 state transitions) with another five states and 30 state transitions. We regard this explosion of hardware complexity as incompatible with goals of industry for inclusion in a high-volume commercial microprocessor.

Several other academic proposals for hardware TM have been published more recently. To keep architectural extensions modest, proposals primarily either restrain the size of supported hardware transactions (e. g., HyTm [17, 39], PhTM [40]), or limit the offered expressiveness (e. g., LogTM-SE [66], SigTM [10]). Each of these hardware approaches is accompanied by software that works around the limitations and provides the interface and features of STM: flexibility, expressiveness, and large transaction sizes.

Intel's HASTM [58] is an industry proposal for accelerating transactions executed entirely in software. It consists of ISA extensions and hardware mechanisms that together improve STM performance. The proposal allows for a reasonable, low-cost hardware implementation and provides performance comparable to HTM for some types of workloads. However, because the hardware supports read-set monitoring only, it has fewer application scenarios than HTM. For instance, it cannot support most lock-free algorithms.

Sun's Rock processor [18] is an architectural proposal for TM. Unlike previously mentioned proposals, it has been implemented in hardware. It is based on the sensible approach

that hardware should only provide limited support for common cases and advanced functions must be provided in software. Early experiences with this processor have shown encouraging results but also revealed some hardware limitations that severely limit performance. Note that TLB misses abort transactions. Rock also does not support selective annotation, which we described in Section 4.1.2. Finally, Rock does not provide any liveness guarantee, so lock-free algorithms cannot rely on forward progress and have to provide a conventional second code path.

Azul Systems [14] has developed multi-core processors with built-in HTM mechanisms. These mechanisms are principally used for lock elision in Java to accelerate locking. The solution appears to be tightly integrated with the proprietary software stack, so not a general-purpose solution.

4.1.2 AMD's Advanced Synchronization Facility (ASF)

AMD's Advanced Synchronization Facility (ASF) is a public specification proposal of an instruction set extension for the AMD64 architecture [2]. It has the objective to reduce the overheads of speculation and simplify the programming of concurrent programs. ASF has been designed in such a way that it can be implemented in modern microprocessors with reasonable transistor budget and runtime overheads. Diestelhorst and Hohmuth [20] described an earlier version of ASF, dubbed ASF1, and evaluated it for accelerating an STM library. The main difference between ASF1 and the current revision, ASF2, is that ASF1 did not allow dynamic expansion of the set of protected memory locations once a transaction had started the atomic phase in which it could speculatively write to protected memory locations.

ASF has originally been aimed at making lock-free programming significantly easier and faster. We are interested in applying ASF to transactional programming, especially to accelerating TM systems. We present in details the rationale behind ASF and its specification.

The rationale underlying the ASF choice

Although ASF is purely experimental and has not been announced for any future product, it matches our objectives for the uptake of hardware level speculative support.

ASF is an extension of the AMD64 ISA which itself is an extension of the x86 ISA. The x86 architecture is the most widespread and we evaluate our STM on this architecture.

ASF is defined from the ISA [2] perspective and not from the micro-architecture. Such specification enables software developers to rely on a strict definition of instructions and thus let room for experimentation in its implementation in hardware.

Some of the important features are: (1) cache lines are the units of protection and there is no modification to the critical cache-coherence protocol; (2) ASF can be used in kernel or user space or virtualized; (3) we can selectively annotate memory accesses as either transactional or non-transactional; (4) ASF ensures forward progress up to a certain transaction capacity.

By contrast with the first HTM design [31], ASF can be implemented without changes to the cache hierarchy. Azul's HTM [14] also does not support selective annotation like ASF. With Sun's Rock[18], TLB misses abort transactions unlike ASF. By contrast, ASF

```
1  ; DCAS Operation:
2  ; IF ((mem1 = RAX) && (mem2 = RBX)) {
3  ;   mem1 = RDI;      mem2 = RSI;     RCX = 0;
4  ; } ELSE {
5  ;   RAX = mem1;      RBX = mem2;     RCX = 1;
6  ; }  // (R8, R9, R10 modified)
7  DCAS:
8    MOV      R8, RAX
9    MOV      R9, RBX
10 retry:
11   SPECULATE             ; Speculative region begins
12   JNZ      retry        ; Page fault, interrupt, or contention
13   MOV      RCX, 1       ; Default result, overwritten on success
14   LOCK MOV R10, [mem1]  ; Specification begins
15   LOCK MOV RBX, [mem2]
16   CMP      R8, R10      ; DCAS semantics
17   JNZ      out
18   CMP      R9, RBX
19   JNZ      out
20   LOCK MOV [mem1], RDI  ; Update protected memory
21   LOCK MOV [mem2], RSI
22   XOR      RCX, RCX     ; Success indication
23 out:
24   COMMIT
25   MOV      RAX, R10
```

Listing 4.1: ASF example: An implementation of a DCAS primitive using ASF

does ensure forward progress when protecting at-most four memory lines in the absence of contention.

Finally, the ASF evaluation relies on an out-of-order x86 core simulator, giving us high confidence in results obtained.

ASF specification

The complete AMD's ASF specification is available online [2] and we detail some parts for using it in the context of transactions.

ASF adds seven new instructions to the AMD64 ISA for entering and leaving speculative code regions (*speculative regions* for short), and for accessing protected memory locations (i. e., memory locations that can be read and written speculatively and which abort the speculative region if accessed by another thread): `SPECULATE`, `COMMIT`, `ABORT`, `LOCK MOV`, `PREFETCH`, `PREFETCHW`, and `RELEASE`. All these instructions are available in all system modes (user, kernel; virtual-machine guest, host). Figure 4.1 shows an example of a double CAS (DCAS) primitive implemented using ASF.

.

Speculative-region structure. Speculative Regions (SR) have the following structure. The `SPECULATE` instruction marks the start of a region. It also defines the rollback point if the speculative region aborts: in this case, execution continues at the instruction following the `SPECULATE` instruction (with an error code in the rAX register and the zero flag cleared, allowing subsequent code to branch to an abort handler).

The code in the speculative region indicates protected memory locations using the `LOCK MOV`, `LOCK PREFETCH`, and `LOCK PREFETCHW` instructions. The first is also used to load and store protected data; the latter two merely start monitoring a memory line for concurrent stores (`LOCK PREFETCH`) or loads and stores (`LOCK PREFETCHW`).

`COMMIT` and `ABORT` signify the end of a speculative region. `COMMIT` makes all speculative modifications instantly visible to all other CPUs, whereas `ABORT` discards these modifications.

Speculative regions can optionally use the `RELEASE` instruction to modify a transaction's read set. With `RELEASE`, it is possible to stop monitoring a read-only memory line, but not to cancel a pending transactional store (the latter is possible only with `ABORT`). `RELEASE`, which is strictly a hint to the CPU, helps decrease the odds of overflowing transactional capacity and is useful, for example, when walking a linked list to find an element that needs to be mutated.

Aborts. Besides the `ABORT` instruction, there are several conditions that can lead to the abort of a speculative region: contention for protected memory; system calls, exceptions, and interrupts; the use of certain disallowed instructions, e. g., `SYSCALL`. Furthermore, the specific ASF implementation used may enforce aborts for other conditions. Unlike in Sun's HTM design [18], TLB misses do not cause an abort.

In case of an abort, all modifications to protected memory locations are undone, and the 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. 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 context switches, all system components (user programs, OS kernel, hypervisor) can make use of ASF without interfering with one another.

Selective annotation. Unlike most other architecture extensions aimed at the acceleration of transactions, ASF allows software to use both transactional and non-transactional memory accesses within a speculative region. Each `MOV` instruction can be selectively annotated to be either transactional (with `LOCK` prefix) or non-transactional (no prefix); hence the name *selective annotation*. This feature allows reducing the pressure on hardware resources providing TM capacity because programs can avoid protecting data that is known to be thread-local. It also allows implementing STM runtimes or debugging facilities (such as shared event counters) that access memory directly without risking aborts because of memory contention.

Because ASF uses cache-line-sized memory blocks as its unit of protection, software must take care to avoid collocating both protected and unprotected memory objects in the same cache line. ASF can deal with some collocation scenarios by hoisting collocated objects accessed using unprotected memory accesses into the transactional data set. However, ASF does not allow unprotected writes to memory lines that have been modified speculatively and raises an exception if that happens.

CPU A mode	CPU A operation	CPU B cache line state	
		Prot. Shared	**Prot. 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 4.1: Conflict matrix for ASF operations ([2], §6.2.1).

Isolation. ASF provides strong isolation: it protects speculative regions against conflicting memory accesses to protected memory locations from both other speculative regions and regular code concurrently running on other CPUs.

In addition, all aborts caused by contention appear to be instantaneous: ASF does not allow any side effects caused by misspeculation in a speculative region to become visible. These side effects include non-speculative memory modifications and page faults after the abort, which may have been rendered spurious or invalid by the memory access causing the abort.

Eventual forward progress. ASF architecturally ensures eventual forward progress in the absence of contention and exceptions when a speculative region protects no more than four 64-byte memory lines[1]. This enables easy lock-free programming without requiring software to provide a second code path that does not use ASF. Because it only holds in the absence of contention, software still has to control contention to avoid livelocks, but that can be accomplished easily, for example, by employing an exponential-backoff scheme.

An ASF implementation may have a much higher capacity than the four architectural memory lines, but software cannot rely on any forward progress if it attempts to use more than four lines. In this case, software has to provide a fallback path to be taken in the event of a capacity overflow, for example, by grabbing a global lock monitored by all other speculative regions.

Conflict resolution Conflict resolution in ASF follows the "requester wins" policy (i. e., existing SRs will be aborted by incoming conflicting memory accesses). Table 4.1 summarizes how ASF handles contention when CPU A performs an operation while CPU B is in a SR with the cache line protected by ASF [2].

[1] *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.

Operations ordering The ordering guarantees that ASF provides for mixed speculative and non-speculative accesses are important for the correctness of our algorithms, and are required for the general-purpose synchronization techniques listed in Section 4.2.1 to be applicable or practical. In short, aborts are instantaneous with respect to the program order of instructions in SRs. For example, aborts are supposed to happen before externally visible effects such as page faults or non-speculative stores appear. A consequence is that memory lines are monitored early for conflicting accesses (i. e., once the respective instructions are issued in the CPU, which is always before they retire). After an abort, execution is resumed at the `SPECULATE` instruction. Further, atomic instructions such as compare-and-set or fetch-and-increment retain their ordering guarantees (e. g., a CAS ordered before a `COMMIT` in a program will become visible before the transaction's commit). This behavior illustrates why speculative accesses are also referred to as "protected" accesses.

ASF implementation variants

ASF was designed to propose a reasonable extension to current CPUs and different designs of integration were proposed. The minimal capacity requirements for an ASF implementation (four transactional cache lines) are deliberately low so existing CPU designs can support simple ASF applications, such as lock-free algorithms or small transactions, with very low additional cost. On the other side of the implementation spectrum, an ASF implementation can support even large transactions efficiently.

In this section, we present two basic implementation variants which are the ones present in the simulator we used in our evaluation (described in Section 4.1.3).

LLB-based implementation. The first ASF implementation variant introduces a new CPU data structure called the *locked-line buffer* (LLB). The LLB holds the addresses of protected memory locations as well as backup copies of speculatively modified memory lines. It snoops remote memory requests, and if an incompatible probe request is received, it aborts the speculative region and writes back the backup copies before the probe is answered.

The advantage of an LLB-based implementation is that the cache hierarchy does not have to be modified. Speculatively modified cache lines can even be evicted to another cache level or to main memory.

Because the LLB is a fully associative structure, it is not bound by the L1 cache's associativity and can ensure a larger number of protected memory locations. However, since fully associative structures are more costly, the total capacity typically would be much smaller than the L1 size.

We are testing two different reasonable sizes: LLB-8 and LLB-256.

Cache-based implementation. The second variant is to keep the LLB-based approach for all speculative-write combined with a cache-based approach for all speculative-reads.

It uses the L1 cache to monitor the speculative region's read set, and the LLB to maintain backup copies of and monitor its write set. So it assumes that the L1 cache is not shared by more than one logical CPU (hardware thread).

Each cache line needs only one speculative-read bit. When a speculative region protects data cached in a given line, the speculative-read bit is turned on. Whenever a cache line that

has this bit set needs to be removed from the cache (because of a remote write request or because of a capacity conflict), the speculative region is aborted.

When the speculative region modifies a protected cache line, the backup data is copied to the LLB. Thus, dirty cache lines do not have to be backed up by evicting them to a higher cache level or main memory.

When a speculative region completes successfully, all speculative-read bits and LLB's are flash-cleared, i. e., set to zero.

On the contrary of a pure cache-based implementation where writes are also in cache, this design minimizes changes to the cache hierarchy, especially when the all caches participate in the coherence protocols as first class citizens: the CPU core's L1 cache remains the owner of the cache line and can defer responses to incompatible memory probes until it has written back the backup data, without having to synchronize with other caches.

The advantage over a pure LLB-based implementation is the much higher read-set capacity offered by the L1 cache. However, the capacity is limited by the cache's associativity.

We are testing the same LLB sizes as pure LLB-based implementation and we call them: LLB-8 w/ L1, LLB-256 w/ L1.

4.1.3 ASF simulator

For our evaluation of ASF, we rely on simulation because ASF is only an ISA extension proposal with no implementation in hardware. Fortunately, AMD has developed an extension of PTLsim [67], called PTLsim-ASF for proposing an evaluation of the ASF proposal.

PTLsim is a cycle accurate x86 microprocessor simulator for the x86 and x86-64 instruction sets. Thanks to the x86 simulation, it allows us to easily reuse the existing compiler infrastructure, binaries, and compiled operating system kernels. Using the same binary code will generate more relevant performance predictions and comparable numbers for native and simulated execution.

PTLsim can work with the Xen hypervisor to provide full system x86-64 simulation. ASF, for example, aborts ongoing speculative regions whenever there is a timer interrupt, task switch, or page fault. These events are controlled by the OS and potentially have a large impact on performance perceived by code using ASF. To assess this impact, it is therefore necessary to closely model their behavior, which is best done by putting the operating system into the simulation, too.

PTLsim features a detailed timing model that models an out-of-order core and an associated cache hierarchy in a near-cycle-accurate fashion. It also models the interactions between *multiple* distinct processor cores and memory hierarchies with good tracking of native results [20]. The simulator has been configured to match the general characteristics of a system based on AMD Opteron processors formerly codenamed "Barcelona" (family 10h), with a three-wide clustered core, out-of-order instruction issuing, and instruction latencies modeled after the AMD Opteron microprocessor [3]. 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.

Of course, detailed simulation models are slower than native execution by several orders of magnitude, because simulating a single cycle usually takes much more than one cycle on the host machine (1ms simulated requires about 650s of computation for 16 cores). Fortunately, PTLsim allows execution of uninteresting parts of the benchmark runs, such as OS boot and benchmark initialization, at native speed by providing a seamless switchover between native and simulated execution.

Currently the two implementation variants introduced in Section 4.1.2 are implemented: LLB-based implementations of varying capacity, and implementations that combine the L1 cache for read-set tracking and an LLB for write-set tracking.

4.1.4 Evaluation of AMD's ASF in a transactional context

Implementation of ASF-based TM

ASF-TM is our TM library that uses ASF instructions and also implements the Transactional Memory Application Binary Interface (ABI). The TM-ABI is generic interface that permits to use TM library with transactional compilers and that will be detailed later. It adds (1) some features that are required by the ABI but are not part of ASF and (2) a fallback execution path in software. We need a fallback path in case ASF cannot commit a transaction because of one of ASF's limitations (e. g., capacity limitations or a transaction executing a system call; see Section 4.1.2).

In our evaluation of ASF, we chose to just provide a serial-irrevocable mode as the software fallback. This mode already exists in most STMs as the fallback path for execution of external, non-isolated, or irrevocable actions. It is also required by the ABI. If a transaction is in this mode, it is not allowed to abort itself, but the TM ensures that it will not be aborted and that no other transaction is being executed concurrently. If no transaction is in this mode, all transactions execute the normal TM algorithm (in our case, ASF speculative regions). Our measurement shows that ASF can handle most of our current workloads directly in hardware (see Section 4.1.4).

ASF-TM needs to make sure that conflicting accesses by concurrent transactions are detected. To do so, it uses ASF speculative loads and stores for these accesses. This is implemented using ASF assembly code in ASF-TM. Note that this code will get inlined if we link ASF-TM statically to the application. The compiler only uses transactional memory accesses for data that is potentially shared with other threads. Therefore, accesses to a thread's stack are not speculative or transactional unless the address of a variable on the stack has been shared.

As we explained previously, we want ASF-TM to be compatible with the existing TM ABI, so we cannot rely on the compiler to insert a `SPECULATE` instruction into the application code. Instead, transactions are started by calling a special "transaction begin" function that is a combination of a software `setjmp` implementation and a `SPECULATE` instruction. Because ASF does not restore CPU registers (except the instruction and stack

pointers), we use the software `setjmp` to checkpoint and restore CPU registers[2] in the current thread's transaction descriptor.

When ASF detects a conflict, it aborts by rolling back all speculatively modified cache lines and resuming execution at the instruction that follows the `SPECULATE` instruction, which is located in the "transaction begin" function. Transaction restarts are then emulated by letting the application return from this function again, thus making it seem as if the previous attempt at running the transaction never happened. The function returns a (TM-ABI defined) value that indicates whether changes to the stack that have not been tracked by ASF have to be rolled back, and which code path (e. g., ASF or serial-irrevocable mode) has to be executed. The TM compiler adds code that performs the necessary actions according to the return value.

Before starting the ASF speculative region, the `begin` function additionally initializes the tracking of memory management functions and performs simple contention management if necessary (e. g., use exponential back-off). ASF transactions that fail to execute a certain number of times or experience ASF capacity overflows will get restarted in serial-irrevocable mode by employing an alternative code path generated by the compiler on this purpose.

To commit an ASF transaction, it is sufficient to call a commit function of the ASF-TM library that contains an ASF `COMMIT` instruction.

Evaluation

Current high-performance x86 microprocessor designs are highly complex and, hence, performance prediction through simulation is nontrivial. To support the validity of our evaluation of ASF, we start by assessing the accuracy of the PTLSIM/ASF simulator. That is, we measure the deviation between simulated performance and performance of native execution on a real machine. A close match between simulated and real executions supports our overall approach because it indicates how well the simulator models a realistic processor micro-architecture. It also increases the confidence that we can have in the overall evaluation.

We then evaluate ASF itself by using (1) applications from the STAMP (see Section 2.3.5) TM benchmark suite[3] and (2) the well-known integer set micro-benchmarks (see Section 2.3.5). We use the standard STAMP configuration for simulator environments [9].

Integer set micro-benchmark runs *search*, *insert*, and *remove* operations on an ordered set of integers, and is implemented either using a linked list, a skip list, a red-black tree, or a hash table. The principles behind these benchmarks resemble the description of the integer-set benchmarks in [18]. Operations are completely random and on random elements. The initial size of a set (i. e., the number of elements it contains) is half the size of the key range from which elements are drawn. No insertion or removal happens if the element is already in or not in the set, respectively. All these programs use several threads and implement synchronization using atomic blocks (i. e., C/C++ transaction statements, see Chapter 5).

[2]The calling convention that is used in the application code determines which registers have to be restored.

[3]We exclude the Bayes and Yada applications in our measurements. We have observed nonreproducible behavior for Bayes with several TM implementations and Yada has extremely long transactions and does not show any scalability with any of the TMs we analyzed.

Features	Description
Processor	AMD Opteron "Barcelona"
Number of cores	8 cores
Clock speed	2.2 GHz
RAM size	1 GB
Operating System	Xen/Linux 2.6.20

Table 4.2: Parameters for the simulated system.

We used the Dresden TM Compiler[4] to compile the applications and used ASF-TM as the TM library. To reduce impact from the memory allocator, we have selected the allocator with best scalability out of glibc 2.10 standard malloc, glibc 2.10 experimental malloc, and the Hoard memory allocator [8] for the presented results. Runs marked as sequential are single-threaded executions of these programs with no synchronization mechanism in use and no instrumentation added.

Following the performance evaluation, we additionally investigate ASF runtime overheads and the effects of different ASF capacities.

We use PTLsim-ASF (as described in Section 4.1.3) as our simulation testbed. The simulated machine has eight CPU cores, each having a clock speed of 2.2 GHz as defined in Table 4.2. Because PTLsim does not yet model limited cross-socket bandwidths, these eight cores behave as if they were located on the same socket, resembling future processors with higher levels of core integration.

We evaluate ASF using four implementations: (1) with an LLB of 8 lines; (2) with an LLB of 256 lines; (3) with LLB w/ L1 of 8 lines; and, (4) with LLB w/ L1 of 256 lines. They are denoted by LLB-8, LLB-256, LLB-8 w/ L1, and LLB-256 w/ L1, respectively.

For our STM measurements, we use our implementation TinySTM[5] in write-through mode.

Simulator accuracy. Figure 4.1 shows the difference in runtimes between execution on a real machine[6] and a simulated execution within PTLsim-ASF, in which we adapted the available parameters of the simulation model to match the characteristics of the native micro-architecture. For five out of the eight STAMP benchmarks, PTLsim-ASF stays within 10–15% of the native performance, which is in line with earlier results for smaller benchmarks [20]. Vacation and K-Means seem to exercise mechanisms in the micro-architecture that perform differently in PTLsim-ASF and in our selected native machine. Clearly, PTLsim cannot model all of the performance relevant micro-architectural subtleties present in native cores, because many of them are not public, highly specific to the revision of the microprocessor, and difficult to reproduce and identify.

One source of the inaccuracies we observed might be a PTLsim quirk: although PTLsim carefully models a TLB and the logic for page-table walks, it only consults them for loads. Stores do not query the TLB and therefore are not delayed by TLB misses, do not

[4]DTMC version used in these experiments is based on LLVM 2.6.

[5]For these measurements, we used a forked version of TinySTM which is compatible with DTMC.

[6]AMD Opteron processor formerly codenamed "Barcelona," family 10h, 2.2 GHz.

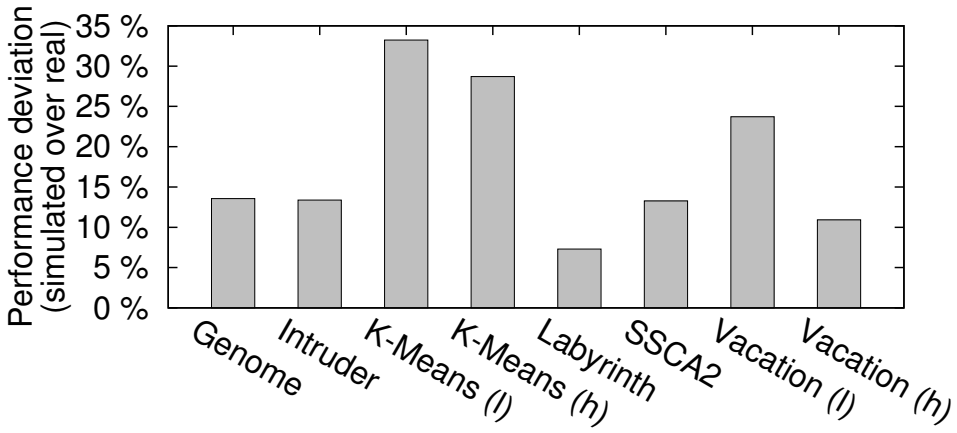


Figure 4.1: PTLSim accuracy for the runtime of the STAMP benchmarks (no TM, no ASF, one thread) for simulated with respect to native execution.

update TLB entries, and are not stalled by bandwidth limitations in the page-table walker. The effect on accuracy likely is minor since translations for many stores already reside in the TLB because of a prior load.

Despite these differences, we think that PTLsim models a realistic micro-architecture and captures several novel interactions in current microprocessors. For our main evaluation we conduct all experiments—including the baseline STM runs—inside the simulator to make sure that our results are not affected by the discrepancies.

ASF performance. Figure 4.2 presents scalability results for selected applications from the STAMP benchmark suite[7]. We also compare the performance of ASF-based TM to the performance of a finely tuned STM (TinySTM) and to serial execution of sequential code (without a TM).

We observe that ASF-based TMs show very good scalability and much better performance than STM for some applications, notably genome, intruder, ssca2, and vacation. Other applications such as labyrinth do not scale well with LLB-8 and LLB-256 because the TM uses serial-irrevocable mode extensively, yet performance is still significantly better than STM. As expected, the applications that do not scale well are those with transactions that have large read and write sets (according to Table III in [9]).

For applications with little contention and short transactions, all four ASF variants perform well. For other applications, LLB-256 usually outperforms the other implementation variants because LLB-8 suffers from limited buffer size and LLB w/ L1 is susceptible to cache-associativity limitations. Yet, it is interesting to note, even the LLB-8-based implementation provides benefits for many applications.

To summarize, the ASF-based TMs have a significantly smaller single-thread overhead than the STM and scale well for many benchmarks. The STM-based variants scale as well,

[7]We added appropriate padding to the entry points of the main data structures to avoid unnecessary contention aborts due to false sharing of cache lines.

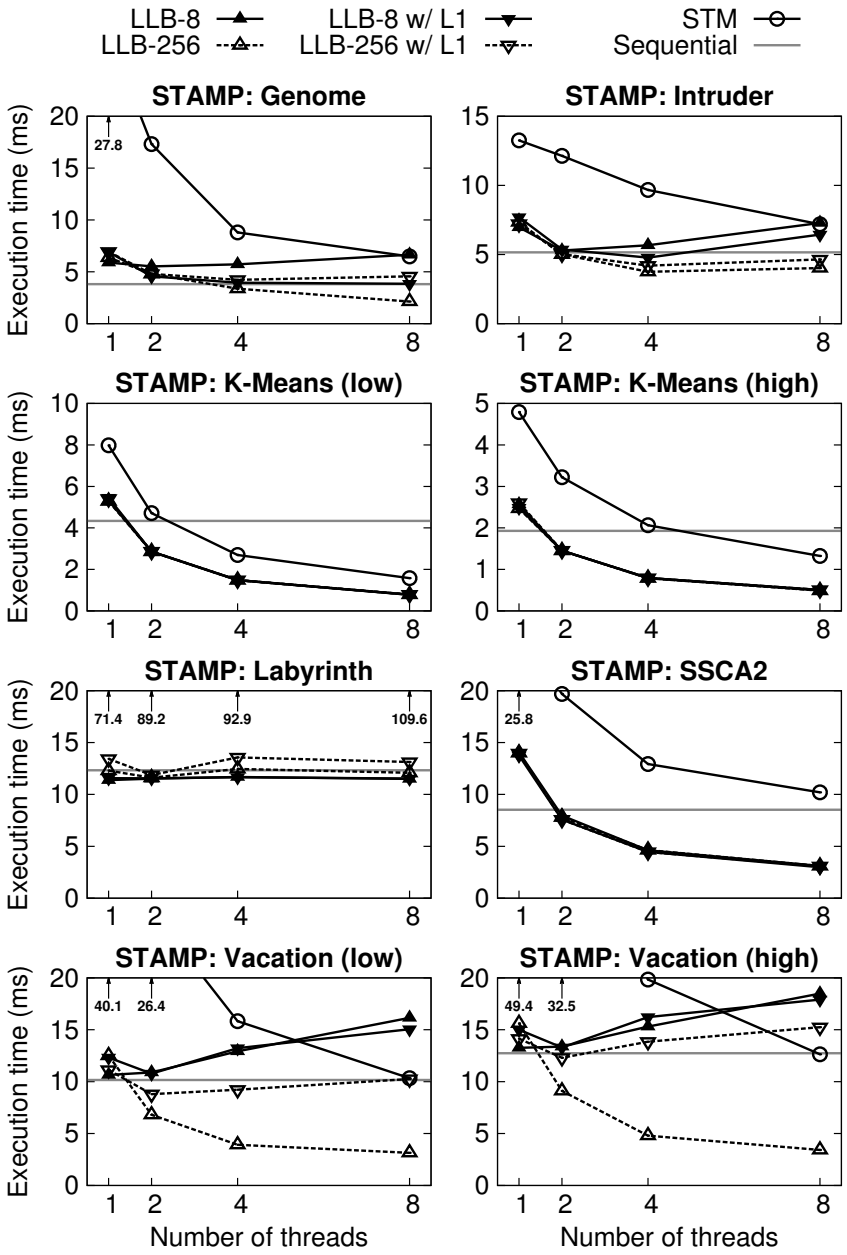


Figure 4.2: Scalability of applications, with four ASF implementations and varying thread count (execution time; lower is better). The arrows indicate STM values that did not fit into the diagram. The horizontal bars show the execution time for execution of sequential code (without a TM).

but they outperform serial execution only with many threads. In general, the ASF-based TMs outperform the STM by almost an order of magnitude.

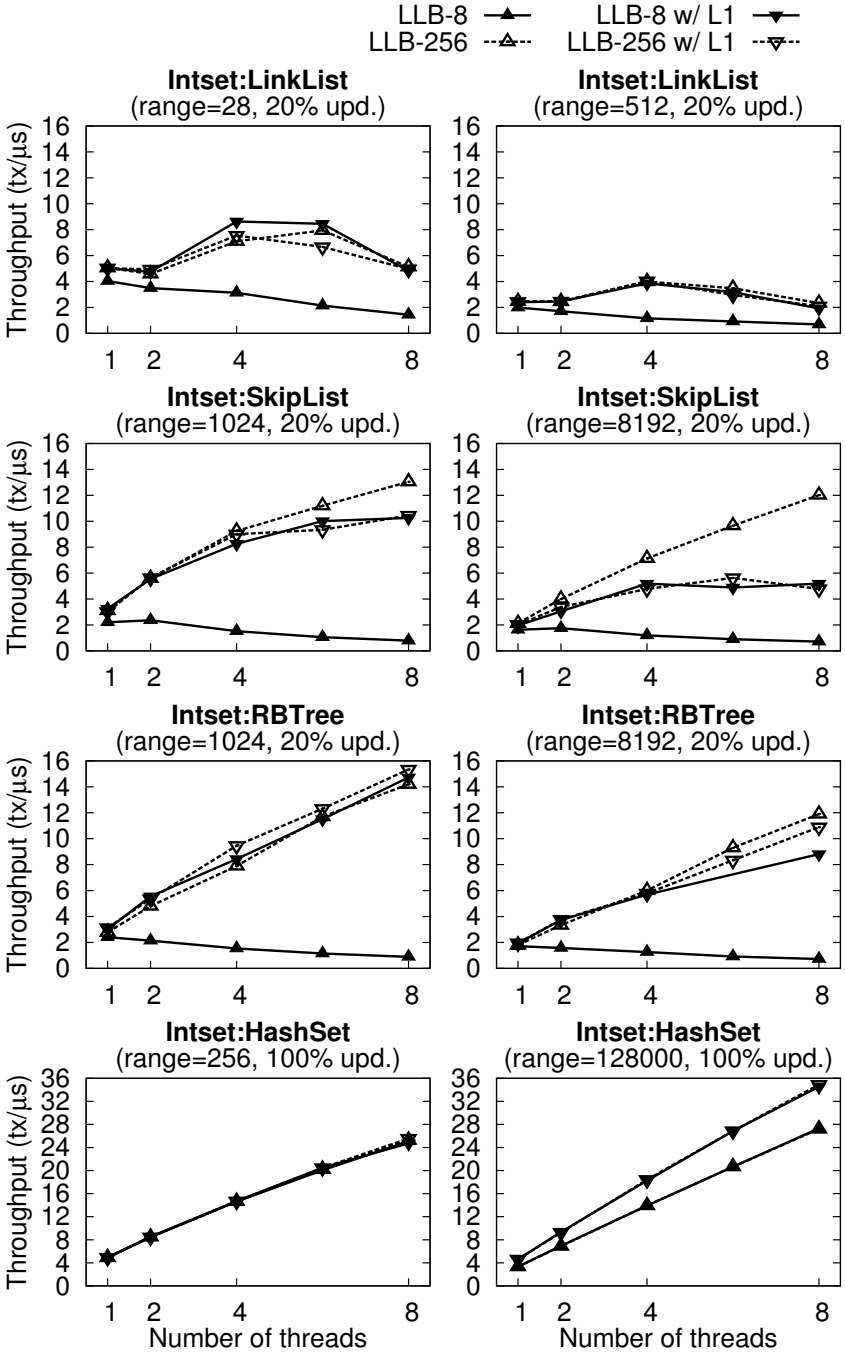


Figure 4.3: Scalability of IntegerSet with linked list, skip list, red-black tree, and hash set, with four ASF implementations and varying thread count and key range (throughput; higher is better).

Scalability. Figure 4.3 presents scalability results for the integer set micro-benchmark. We vary the key range between $\{0 \dots 28\}$ and $\{0 \dots 128000\}$.

In all integer set variants except the hash-set-based one, the LLB-8 implementation performs poorly because its capacity is insufficient for holding the parts of the data structure that are accessed, leading to constant execution of the software fallback path. This fallback path is serial-irrevocable mode and suffers from contention if used excessively by many threads. The cache-based implementations generally perform equally well, indicating that the write set of all transactions is smaller than 8 cache lines. LLB-256 (without the L1 cache) never performs significantly worse than the cache-based implementations, indicating that the read set always fits into 256 cache lines, and occasionally outperforms them because it is not susceptible to cache-associativity limitations. The performance drop observed for the linked list with more than four threads results from the increased likelihood of conflict in the sequentially traversed list. In general, the hash-set variant performs best and can tolerate the largest key range and the largest update rates because it has the smallest transactional data set and very few conflicts.

ASF abort reasons. Figure 4.4 provides a breakdown of the abort reasons in the STAMP applications with different ASF implementations. Unsurprisingly, the implementation with the small dedicated buffer (eight-entry LLB) suffers from many capacity aborts for most benchmarks, while the larger dedicated buffer (256-entry LLB) usually has the least capacity aborts. Adding the L1 cache for tracking transactional reads ("+L1") does not always reduce capacity aborts, but actually increases them for several benchmarks. Three reasons contribute to the increase. First, although the L1 cache has a large total capacity, it has limited associativity (two-way set associativity) and therefore usable capacity is dependent on the address layout. Second, our current read-set-tracking implementation does not modify the cache-line displacement logic. Non-speculative accesses may displace cache lines used for tracking the read set. Finally, cache lines may be brought into the cache out of order and purely due to speculation of the core. These additional cache lines may further displace lines that track the transaction's read set.

Since displacement of cache lines with transactional data causes capacity aborts, the large number of those is not only caused by actual capacity overflows, but may be caused by disadvantageous transient core behavior. For our current study, we fall back to serial mode to handle capacity aborts, therefore reducing contention aborts for benchmarks with high capacity failures. To leverage the partially transient nature of capacity aborts, one could also retry aborting transactions in ASF and hope for favorable behavior. Furthermore, we will tackle the issue from the hardware side by containing the random effects and ensuring that we meet the architectural minimum capacity.

ASF capacity. Figure 4.5 presents the scalability in terms of transaction size versus throughput for runs with eight threads. We vary the transaction size (i. e., the number of memory locations accessed) by initially populating the linked list with different amounts of elements. LLB-8 is not sufficient to hold the working set for larger transactions. Transactions have to be executed in software fallback mode for most linked-list transactions with more than eight elements. For the red-black tree, the tree height is most determining for the

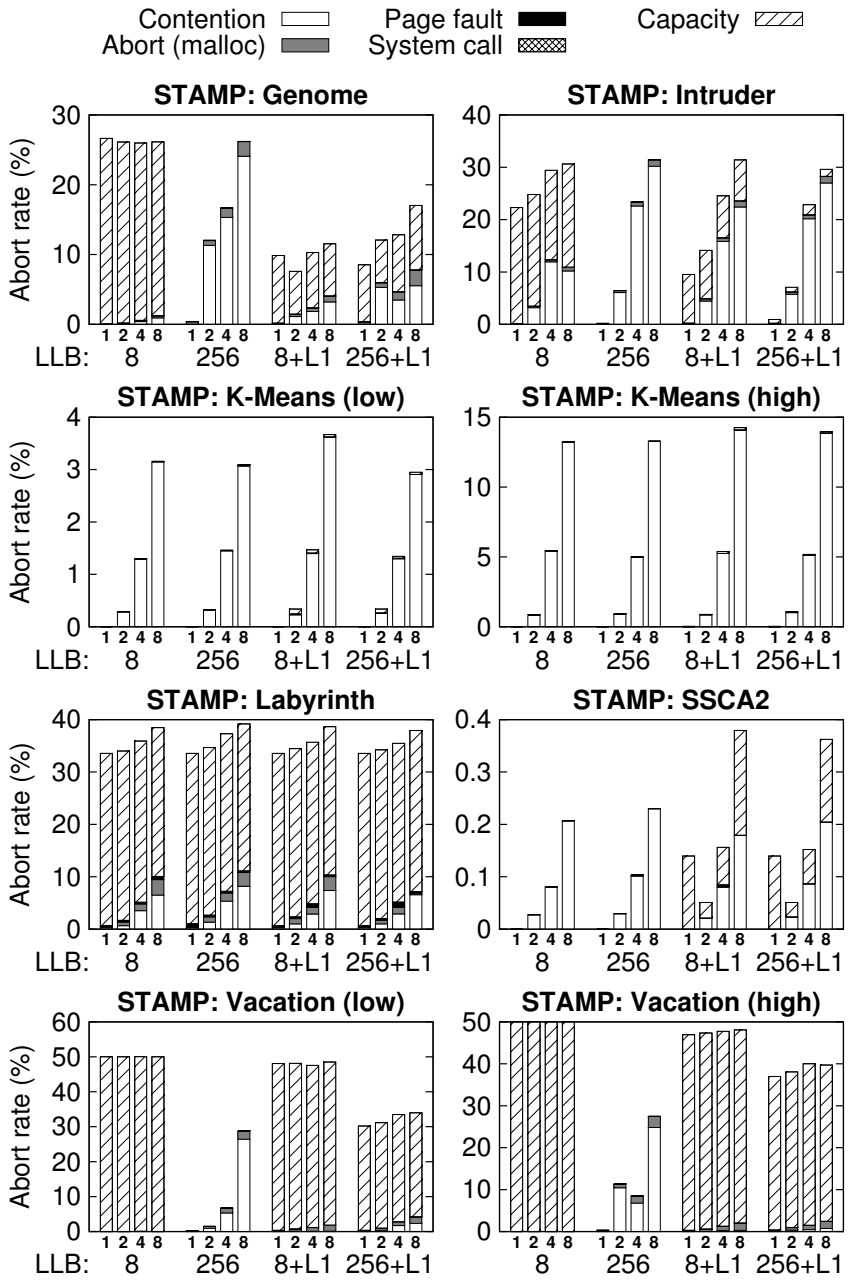


Figure 4.4: Abort rates of applications, with four ASF implementations and varying thread count. The different patterns identify the cause of aborts.

Application	LLB_8	LLB_256	LLB_8 w/L1	LLB_256 w/L1
Intruder	80.55%	100%	–	99.45%
Genome	73.41%	100%	89.92%	90.92%
K-means low	100%	100%	100%	100%
K-means high	100%	100%	99.96%	99.96%
Labyrinth	67.12%	67.12%	67.12%	67.12%
SSCA2	100%	100%	99.99%	99.99%
Vacation low	50.01%	100%	55.44%	76.43%
Vacation high	50.02%	100%	54.90%	65.61%

Table 4.3: Percent of STAMP transactions that fit inside an ASF speculative region.

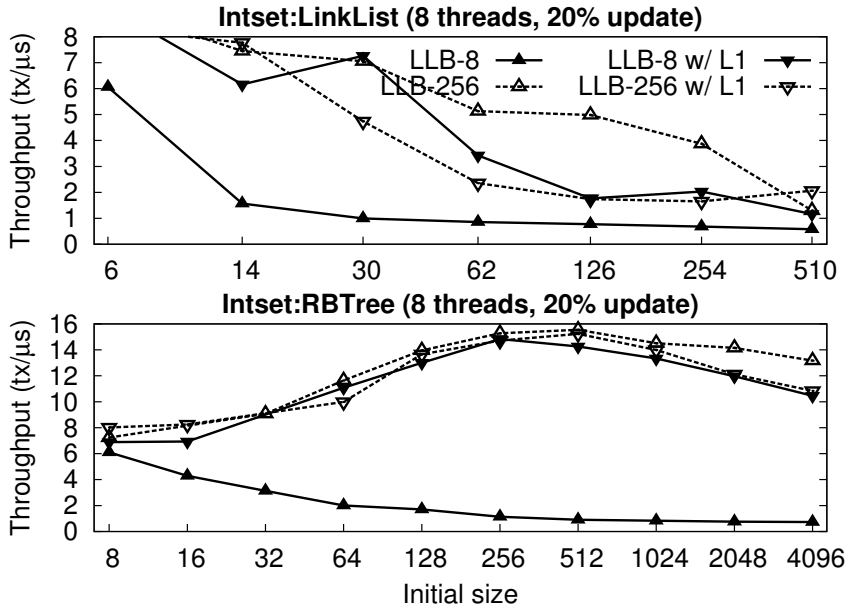


Figure 4.5: Influence of ASF capacity on throughput for different ASF variants (red-black tree and linked list with 20% update rate with eight threads).

App. / % updates / size	linked list / 20% / 128			skip list / 20% / 128		
	ASF	STM	Ratio	ASF	STM	Ratio
Non-instr. code	0	0	–	0	0	–
Instr. app. code	1368105	1747385	1.28	1107561	1793351	1.62
Abort/restart	0	0	–	0	0	–
Tx load/store	1029659	31024930	30.13	652817	10073146	15.43
Tx start/commit	1322509	1087201	0.82	1276152	1176545	0.92
App. / % updates / size	**red-black tree / 20% / 128**			**hash set / 100% / 128**		
	ASF	STM	Ratio	ASF	STM	Ratio
Non-instr. code	0	0	–	9738	0	0.00
Instr. app. code	2039471	281328	0.13	78822	87368	1.11
Abort/restart	0	0	–	426147	0	0.00
Tx load/store	233246	7623913	32.69	533696	5013248	9.39
Tx start/commit	1306687	1033154	0.79	1263550	1316656	1.04

Table 4.4: Single-thread breakdown of cycles spent inside transactions for ASF-TM (with LLB-256) and TinySTM.

transaction size. At around 256 elements, almost all transactions run in serial-irrevocable mode for LLB-8.

The overall throughput for the list benchmark decreases with problem size because traversing longer lists increases conflict ratio, work per transaction, and chance for capacity overflow. LLB-256, LLB-8 w/ L1, and LLB-256 w/ L1 behave similarly with this benchmark.

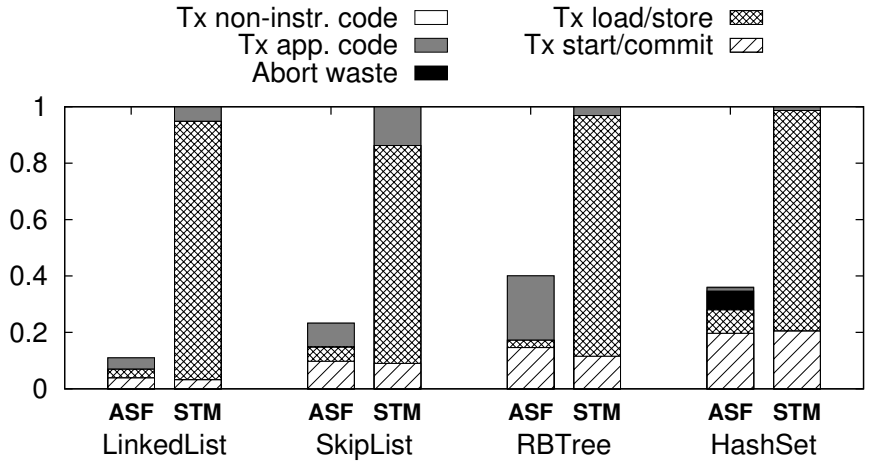


Figure 4.6: Single-thread overhead details for ASF-TM (with LLB-256) and TinySTM. All values normalized to the STM results of the respective benchmark.

ASF single-thread overheads. To quantify the performance improvement seen with ASF, we have inspected some benchmark runs more closely and broke up the spent cycles into categories. Because adding online timing analysis adds bookkeeping work, interferes with compiler optimization steps, increases cache traffic, and impairs pipeline interaction, we refrained from adding the statistics code into the application or run-time. Instead, we manually annotated the compiled final binaries—marking assembly code line-by-line with one of the categories "TX entry/exit," "TX load/store," "TX abort," and "application"—and extended our simulator to produce a timed trace of the execution. We then produced the cycle breakdown by offline analysis and aggregation of the traces, without any interference with the benchmarks execution.

Figure 4.6 and Table 4.4 present the details of the composition of the overhead imposed by the TM stack based on ASF or on STM. The results are for single-threaded runs of the IntegerSet benchmark on the LLB-256 implementation. Because there is only one thread, there are no aborts caused by memory contention. All aborts reported for the hash-set variant occur because of page faults, which require OS-kernel intervention and therefore abort the ASF speculative regions.

The overhead of starting and committing transactions is similar for ASF and STM in single-thread executions, largely due to the additional code that is run for entering a transaction. As described in Section 4.1.4 and details in Chapter 5, we had to add code to the ASF implementation that provides the semantics of the ABI on top of the `SPECULATE` instruction. For small transactions, this cost can be the dominant overhead in comparison to the uninstrumented code.

Although transactional loads and stores are much more costly in general in an STM, we were surprised by the difference in improvement for different benchmarks. If we compare the red-black tree and the hash set, we find that there is almost a factor of 33× speed-up for transactional loads and stores for the tree, and only 9× for the hash-set. On closer inspection we found that this can be attributed to cache effects: the hash set has many cache misses, because its data access pattern is mostly random and all accesses update the set, which in total is larger than the first and second level caches (2^{17} buckets, with 16 bytes/bucket). With out-of-order execution, a large part of the STM's constant additional computation and memory traffic overhead can be effectively interleaved with the cache misses and in general has less impact on the incurred relative slowdown.

4.1.5 Conclusion

In this section, we presented the use of AMD's ASF as a hardware support for transactional memory on the x86 architecture. We introduced our runtime library ASF-TM that leverages this hardware support for hardware transactions and proposes a serial irrevocable mode as a fallback solution when hardware transactions cannot execute. Our evaluation indicates that the availability of hardware support improves performance compared to previous software-only solutions by a significant margin, and provides good scalability with most workloads.

4.2 Hybrid Transaction Memory

Most HTM proposals support a limited capacity in the write set of the transactions they can support. ASF is an example of such, where the number of cache lines that can be accessed in a transaction can be as low as four in order to lower its hardware implementation costs. Thus hardware support has to be complemented with software fallback solutions that execute in software the transactions that cannot run in hardware. A simple fallback strategy, used in Section 4.1.4, is to execute software transactions serially, i. e., only one at a time. However, this approach limits performance when software transactions are frequent. It is therefore desirable to develop *hybrid* TM (HyTM) in which multiple hardware and software transactions can run concurrently.

Most previous HyTM proposals [17, 39] have assumed HTMs in which every memory access inside a transaction is speculative, that is, it is transactional, isolated from other threads until transaction commit and will be rolled back on abort. In contrast, ASF provides selective annotation (see Section 4.1.2), which means that non-speculative memory accesses are supported within transactions (including non-speculative atomic instructions) and speculative memory accesses have to be explicitly marked as such.

In this section, we present new *hybrid TM* algorithms that can execute HTM and STM transactions concurrently and can thus provide good performance over a large spectrum of workloads. The algorithms belong to the class of time-based TM designs and exploit the ability of some HTMs to have both transactional and non-transactional memory accesses within a transaction to decrease the transactions' runtime overhead, abort rates, and hardware capacity requirements. We evaluate implementations of these algorithms using micro-benchmarks and transactional applications.

4.2.1 Contributions

In this section, we present a family of *novel HyTM algorithms* that use AMD's ASF as HTM. We make heavy use of non-speculative operations in transactions to construct 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 (this is further discussed in Section 4.2.3).

Our HyTM algorithms use the LSA algorithm (see Section 3.2), for software transactions. As in the previous section, we evaluate the performance of our algorithms on a near-cycle-accurate x86 simulator with support for several implementations of ASF (see Section 4.1.2) that differ notably in their capacity limits.

Non-speculative operations are useful beyond HyTM optimizations. In this section, we present two *general-purpose synchronization techniques*: (1) Monitor metadata but read data non-speculatively; (2) Use non-speculative atomic read-modify-write operations to send synchronization messages. These techniques are all combinations of both transaction-based synchronization and classic non-transactional synchronization using standard atomic instruction.

The first technique can reduce HTM capacity requirements and has similarities to lock elision [50], whereas the other one is about composability with non-transactional

synchronization. We will explain the techniques further in Section 4.2.4. To make them applicable, the HTM does not only have to allow non-speculative operations but it must also provide certain ordering guarantees. These conflict resolution rules, as described in Section 4.1.2, are important for understanding how our HyTM algorithms work and why they perform well.

The rest of the section is organized as follows. In Section 4.2.2, we provide background information about ASF and TM in general, and in Section 4.2.3 we discuss related work on HyTM designs. We present our new HyTM algorithm in Section 4.2.4, evaluate it in Section 4.2.5, and conclude in Section 4.2.6.

4.2.2 Background

Our objective is to investigate the design of *hybrid transactional memory algorithms* that exploit hardware facilities for decreasing the overhead of transactions in good cases while composing well with state-of-the-art software transactional memory algorithms.

Algorithm 2 Common transaction start code for all HyTMs.

```
 1: hytm-start():
 2:     if hytm-disabled() 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()                    ▷ 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()                            ▷ we are in a hardware transaction
11:     return true                            ▷ execute HTM codepath
```

The compiler generates separate STM and HTM code paths for each transaction. A common transaction `start` function (see Algorithm 2) takes care of selecting STM or HTM code at runtime. A transaction first tries to run in hardware mode using a special ASF `SPECULATE` instruction (line 4). This instruction returns a non-zero value when jumping back after an abort, similarly to `setjmp/longjmp` 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 has been decided, only STM or HTM code will be executed (functions starting with `stm-` or `htm-`, respectively) during this attempt to execute the transaction.

In the rest of this section, we give an overview of the hardware TM support used for our hybrid algorithms and we discuss related work.

4.2.3 Related work

In the literature, different techniques are proposed for hybrid transactional memory.

Some HyTMs [32, 40] proposed to not run software transactions concurrently with hardware ones. The advantage of the approach is to reduce at the maximum the required

capacity for the HTM. The major problem is that the HTM cannot run all transactions in hardware. When software transactions have to be used, the performance of the system decreases drastically. HyTM should be able to run concurrently hardware and software transactions.

Some others HyTMs [39, 42] proposed an object based design but many papers [16, 19, 25] showed that approaches based on indirection have significant overhead.

Another approach is to use hardware support to accelerate STMs as proposed by Saha *et al.* in [58]. Unfortunately, transactional stores in this proposal do not use hardware-acceleration. The limited usage of hardware for transactional operation limits the performance improvement.

The HyTM proposed by Damron *et al.* [17] combines a best-effort HTM with a word-based STM algorithm. It relies on a HTM that transactified all accesses to memory when in transaction. The hardware transaction ensures that both transactional application data and TM metadata are atomic even if data are thread-local. This strong guarantee reduces the number of transactions that fit the hardware capacity.

Dalessandro *et al.* describe a HyTM [15] based on the NOrec STM [16]. The NOrec approach uses a global versioned lock to synchronize software transactions. In the hybrid version, HyNOrec uses an additional versioned lock to serialize software transactions with hardware transactions. It offers low runtime overhead and low capacity requirements but the usage of a single global lock creates much contention on the same cache-line. It makes this approach affordable only for low number of cores when the contention is low.

The new HyTM algorithms that we present in this thesis improve on previous designs. In the class of HyTMs with ownership records, HyLSA features either lower HTM capacity requirements or a smaller runtime overhead.

4.2.4 The Hybrid Lazy Snapshot Algorithm

Our first algorithm extends the Lazy Snapshot Algorithm (LSA) (see Chapter 3). LSA is a time-based STM algorithm that uses on-demand validation and a global time base to build a consistent snapshot of the values accessed by a transaction. For clarity, we reproduce here the single version, word-based LSA algorithm already presented in a more general form in Chapter 3. The basic version of the LSA algorithm is shown in Algorithm 4 and the state of the algorithm are shown in Algorithm 3.

Transaction stores are buffered until commit. The consistency of the snapshot read by the transaction is checked based on versioned locks (ownership records, or *orecs* for short) and a global time base, which is typically implemented using a shared counter. The orec protecting a given memory location is determined by hashing the address and looking up the associated entry in a global array of orecs. Note that, in this design, an orec protects multiple memory locations.

To install its updates during commit, a transaction first acquires the locks that cover updated memory locations (line 27) and obtains a new commit time from the global time base by incrementing it atomically (line 32). The transaction subsequently validates that the values it has read have not changed (lines 34 and 44–49) and, if so, writes back its updates to shared memory (lines 37–38). Finally, when releasing the locks, the versions of the orecs are set to the commit time (lines 41–43). Reading transactions can thus see the virtual commit

Algorithm 3 LSA STM state (encounter-time locking/write-back variant)

```
 1: Global state:
 2:     clock ← 0                                                    ▷ global clock
 3:     orecs: word-sized ownership records, each consisting of:
 4:       locked: bit indicating if orec is locked
 5:       owner: thread owning the orec (if locked)
 6:       version: version number (if ¬ locked)

 7: State of thread:
 8:     lb: lower bound of snapshot
 9:     ub: upper bound of snapshot
10:     r-set: read set of tuples ⟨addr, val, ver⟩
11:     w-set: write set of tuples ⟨addr, val⟩
```

time of the updated memory locations and use it to check the consistency of their read set. If all loads did not virtually happen at the same time, the snapshot is inconsistent.

A snapshot can be extended by validating that values previously read are valid at extension time, which is guaranteed if the versions in the associated orecs have not changed. LSA tries to extend the snapshot when reading a value protected by an orec with a version number more recent than the snapshot's upper bound (line 14), as well as when committing to extend the snapshot up to the commit time, which represents the linearization point of the transaction (line 34).

We now describe the hybrid extensions of LSA using eager conflict detection (shown in Algorithm 5). A variant with lazy conflict detection is also presented. Note that the HyTM decides at runtime whether to execute in hardware or software mode, as explained in Section 4.2.3 and Algorithm 2.

Transactional loads first perform an ASF-protected load of the associated orec (line 6). This operation starts monitoring of 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 *non-speculative* load operation (line 9) to read the target value. Note that ASF will start monitoring the orec before loading from the target address (see Section 4.1.2). 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.

This represents an application of the first of the synchronization techniques listed in Section 4.2.1: We only monitor metadata (i. e., the orec) but read application data non-speculatively. This enables the HyTM to influence the HTM capacity required for transactions via its mapping from data to metadata, which in turn can make best-effort HTM useable even if transactions have to read more application data than provided by the HTM's capacity. In turn, the HTM has to guarantee that the monitoring starts before the non-speculative load.

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). The operation effectively ensures *eager* detection of

Algorithm 4 LSA STM algorithm (encounter-time locking/write-back variant)

```
 1: stm-start():
 2:     lb ← ub ← clock
 3:     r-set ← w-set ← ∅
 4: stm-load(addr):
 5:     ⟨orec, val⟩ ← ⟨orecs[hash(addr)], *addr⟩                ▷ post-validated atomic read
 6:     if orec.locked then
 7:         if orec.owner ≠ p then
 8:             abort()                                          ▷ orec owned by other thread
 9:         if ⟨addr, new-val, *⟩ ∈ w-set then
10:             val ← new-val                                    ▷ update write set entry
11:     else
12:         if orec.version > ub then                            ▷ try to extend snapshot
13:             ub ← clock
14:             if ¬ validate() then
15:                 abort()                                      ▷ cannot extend snapshot
16:         val ← *addr
17:         r-set ← r-set ∪ {⟨addr, val, orec.version⟩}          ▷ add to read set
18:     return val
19: stm-store(addr,val):
20:     orec ← orecs[hash(addr)]
21:     if orec.locked then
22:         if orec.owner ≠ p then
23:             abort()                                          ▷ orec owned by other thread
24:     else
25:         if ⟨addr, *, ver⟩ ∈ r-set ∧ ver ≠ orec.version then
26:             abort()                                          ▷ read different version earlier
27:         if ¬ cas(orecs[hash(addr)] : orec → ⟨true, p⟩) then
28:             abort()                                          ▷ cannot acquire orec
29:     w-set ← w-set \ {⟨addr, *⟩} ∪ {⟨addr, val⟩}              ▷ add to write set
30: stm-commit():
31:     if w-set ≠ ∅ then                                        ▷ is transaction read-only?
32:         ub ← atomic-inc-and-fetch(clock)                     ▷ commit timestamp
33:         if ub ≠ lb + 1 then
34:             if ¬ validate() then
35:                 abort()                                      ▷ cannot extend snapshot
36:         o-set ← ∅                                            ▷ set of orecs updated by transaction
37:         for all ⟨addr, val⟩ ∈ w-set do                       ▷ write updates to memory
38:             *addr ← val
39:             o-set ← o-set ∪ {hash(addr)}
40:         end for
41:         for all o ∈ o-set do
42:             orecs[o] ← ⟨false, ub⟩                           ▷ release orecs
43:         end for
44: stm-validate():
45:     for all ⟨addr, val, ver⟩ ∈ r-set do                      ▷ Are orecs free and version unchanged?
46:         orec ← orecs[hash(addr)]
47:         if (orec.locked ∧ orec.owner ≠ p) ∨
               (¬ orec.locked ∧ orec.version ≠ ver) then
48:             abort()                                          ▷ inconsistent snapshot
49:     end for
```

Algorithm 5 HyLSA — Eager variant (extends Algorithm 4)

```
 1: State of thread:                                          ▷ extends state of Algorithm 3
 2:     o-set: set of orecs updated by transaction

 3: htm-start():
 4:     o-set ← ∅

 5: htm-load(addr):
 6:     LOCK MOV : orec ← orecs[hash(addr)]                   ▷ protected load
 7:     if orec.locked then
 8:         ABORT                                             ▷ orec owned by (other) software transaction
 9:     val ← addr                                            ▷ nonspeculative load
10:     return val

11: htm-store(addr,val):
12:     LOCK MOV : orec ← orecs[hash(addr)]                   ▷ protected load
13:     if orec.locked then
14:         ABORT                                             ▷ orec owned by (other) software transaction
15:     LOCK PREFETCHW orec                                   ▷ watch for concurrent loads/stores
16:     LOCK MOV : addr ← val                                 ▷ speculative write
17:     o-set ← o-set ∪ {hash(addr)}

18: htm-commit():
19:     if o ≠ ∅ then                                         ▷ is transaction read-only?
20:         ct ← atomic-inc-and-fetch(clock)                  ▷ commit timestamp
21:         for all o ∈ o-set do
22:             LOCK MOV : orecs[o] ← ⟨false, ct⟩
23:         end for
24:     COMMIT                                                ▷ commit hardware transaction
```

conflicts with concurrent transactions. Finally, the updated memory location is speculatively written (line 16).

Algorithm 6 HyLSA — Lazy variant (extends Algorithm 4)

```
 1: State of thread:                                              ▷ extends state of Algorithm 4
 2:     o-set: set of orecs updated by transaction

 3: htm-start():
 4:     o-set ← ∅

 5: htm-load(addr):
 6:     LOCK MOV : orec ← 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):
12:     LOCK MOV : addr ← val                                                 ▷ speculative write
13:     o-set ← o-set ∪ {hash(addr)}

14: htm-commit():
15:     if o ≠ ∅ then                                                  ▷ is transaction read-only?
16:         ct ← clock + 1                                        ▷ optimistic commit timestamp
17:         for all o ∈ o-set do
18:             LOCK MOV : orec ← orecs[o]                                    ▷ protected load
19:             if orec.locked then
20:                 ABORT                          ▷ orec owned by (other) software transaction
21:             LOCK MOV : orecs[o] ← ⟨false, ct⟩                             ▷ speculative write
22:         end for
23:         t ← clock
24:         if ct ≤ t then                                  ▷ was optimistic timestamp valid?
25:             ct ← t + 1                                    ▷ use conservative timestamp
26:             for all o ∈ o-set do
27:                 LOCK MOV : orecs[o] ← ⟨false, ct⟩                         ▷ speculative write
28:             end for
29:             t ← clock
30:         if ct > t then
31:             atomic-inc(clock)
32:     COMMIT                                                ▷ commit hardware transaction
```

Upon commit, an update transaction first acquires a unique commit timestamp from the global time base (line 20). This will be ordered after the start of monitoring of previously accessed orecs, but will become visible to other threads before the transaction's commit (see Section 4.1.2). Next, it speculatively writes all updated orecs (lines 21–23), and finally tries to commit the transaction (line 24). 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). 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 and 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 non-speculatively 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 Section 4.2.1. Because ASF provides non-speculative atomic read-modify-write (RMW) operations, hardware transactions can very efficiently send such messages. In contrast, using speculatively stores would lead to frequent aborts caused by consumers of those messages. If using just non-speculative stores instead of RMW 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 non-speculative atomic RMW operations are essential because it allows hardware transactions to send messages after monitoring data and before commit or monitoring further data.

Another hybrid extension of LSA, HyLSA-lazy, is shown in Algorithm 6. It uses lazy conflict detection: upon store, we do not read nor watch the orec associated with the accessed memory location, but instead we speculatively write to the target location (line 12). 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 17–22).

Proofs of correctness for these algortihms are described in [55].

4.2.5 Evaluation of HyLSA

To evaluate the performance of our HyTMs, we use a similar experimental setup as in our previous study in Section 4.1.4. We simulate a machine with sixteen x86 CPU cores on a single socket, each having a clock speed of 2.2 GHz. We evaluate three ASF implementations (see Section 4.1.2), LLB-8 w/ L1, LLB-8 and LLB-256.

The STM implementations that we compare against are "LSA" (a version of TinySTM, detailed in Section 3.2.7 using write-through mode, eager conflict detection, similar to Algorithm 1). The baseline HTM ("HTM") uses serial-irrevocable mode as simple software fallback like in ASF-TM evaluation 4.1.4. The HyTM implementations have the same names as the respective algorithms (e. g., Algorithm 5 is denoted "HyLSA") and use the LSA implementations for their software code paths.

As benchmarks, we use selected applications from the STAMP TM benchmark suite [9] and the typical integer set micro-benchmarks (IntegerSet). The latter are implementations of a sorted set of integers based on a skip list, a red-black tree, a hash table, and a linked list. During runtime, several threads use transactions to repeatedly execute insert, remove, or contains operations on the set (the type of operations and accessed elements are chosen

Benchmark	Range	Commits on hardware code path (%)		
		LLB-8	LLB-8 w/L1	LLB-256
SkipList-Large	8192	<1%	60–70%	100%
SkipList-Small	1024	<1%	90–95%	100%
RBTree-Large	8192	0–2%	70–90%	100%
RBTree-Small	1024	2–10%	85–95%	100%
HashTable	128000	100%	95%	100%
LinkedList-Large	512	1–3%	100%	100%
LinkedList-Small	28	30–60%	100%	100%

Table 4.5: Integer set micro-benchmarks and approximate ratio of HTM commits for HyLSA to total number of commits for single-thread execution.

randomly). All set elements are within a certain key range, and the set is initially half full. Table 4.5 shows the configurations that we consider. In HashTable all transactions are update transactions (insert or remove operations), in all other benchmarks the update rate is 20%. However, these operations only insert (remove) an element if it is absent from (part of) the set, so the actual percentage of update transactions can be smaller. We use the Hoard memory allocator [8] in HashTable and glibc 2.10 standard malloc in the other benchmarks.

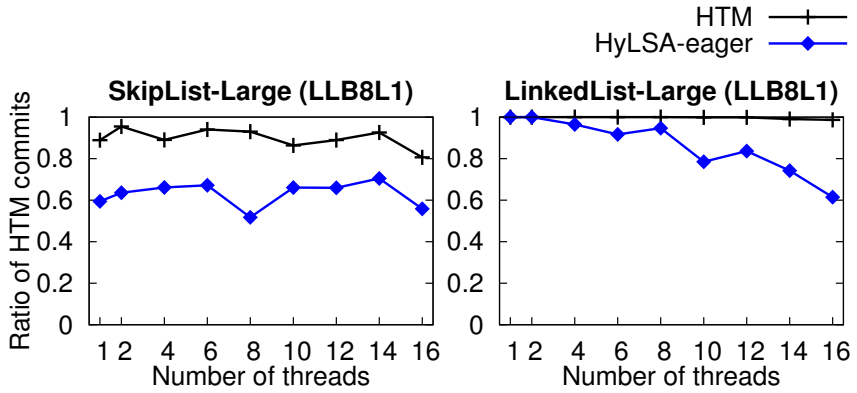


Figure 4.7: Ratio of HTM commits to total number of commits.

Table 4.5 shows which percentage of transaction commits happen on the hardware code path in comparison to the total number of commits. LLB-256 provides sufficient capacity to execute all transactions in our IntegerSet configurations in hardware. In contrast, LLB-8's capacity is most often too small. Note that in our implementations, only permanent ASF abort reasons like exceeding ASF's capacity make the HyTM switch to the software code path. Contention 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 we show is essentially independent of the level of contention in a workload.

Figure 4.8 shows the performance of three HyLSA variants. Unfortunately, the current version of the ASF simulator does not provide the ASF ordering guarantees for non-speculative

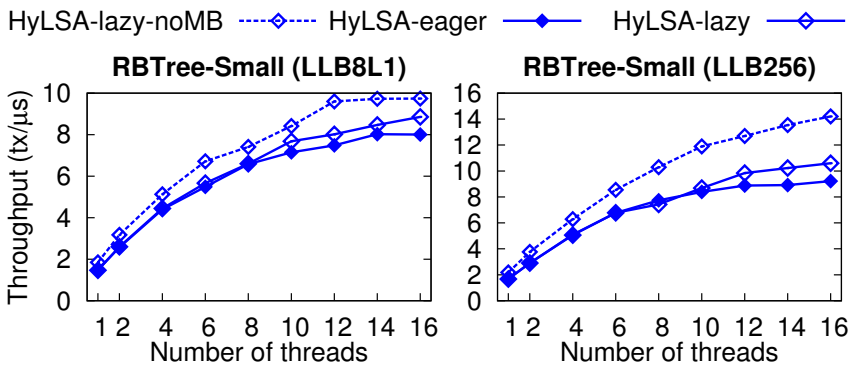


Figure 4.8: Comparison of HyLSA algorithm variants.

accesses in all cases. To be able to run the same HyLSA TMs in all benchmarks, we had to add memory barriers (i. e., an `lfence` instruction) between the speculative load of an orec and the non-speculative load of data (e. g., lines 6 and 9 in Algorithm 5). HyLSA-lazy-noMB is Algorithm 6 without such barriers, which shows their runtime overhead. However, scalability remains similar in our benchmarks. HyLSA-lazy can scale slightly better than HyLSA-eager, which is likely due to lazy conflict detection. Because both perform similar in many situations, we will focus on HyLSA-eager.

Unfortunately, the current version of the ASF simulator does not provide the ASF ordering guarantees for non-speculative accesses in all cases. To be able to run the same HyLSA TMs in all benchmarks, we had to add memory barriers (i. e., an `lfence` instruction) between the speculative load of an orec and the non-speculative load of data (e. g., lines 6 and 9 in Algorithm 5).

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. In our implementations, word-sized data are mapped to word-sized orecs (i. e., we discard the lower three bits of an address and select with the remaining bits 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 map 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 4.5 and Figure 4.7 show that HyLSA is already likely to hit capacity limitations just because it needs twice the capacity for stores, so it is important for HyLSA to read data non-speculatively. Figure 4.9 illustrates this point further, showing that when accessing data and orecs speculatively (HyLSA-eager-SDL), less transactions can execute the hardware code path.

Figure 4.10 presents a concluding overview of TM performance with the integer set micro-benchmarks. We show configurations that are representative or that highlight interesting properties. HashTable performs similarly on all ASF implementations and scales very well, but ultimately suffers from external bottlenecks (e. g., the memory allocator).

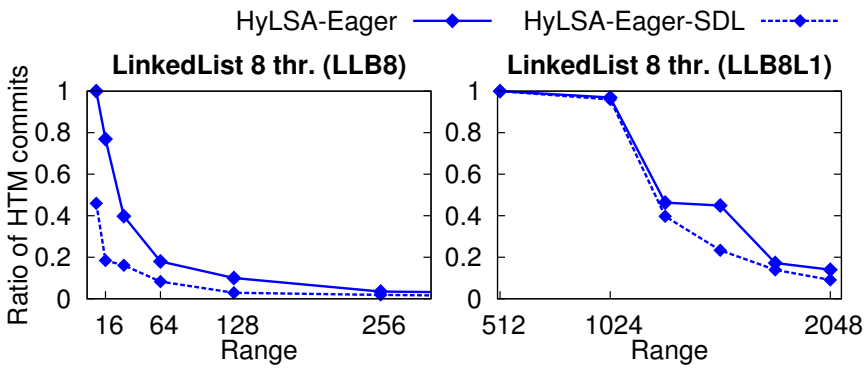


Figure 4.9: Ratio of HTM commits to total number of commits for 8 threads and read-only LinkedList of various sizes (range), when accessing data speculatively (SDL) or not.

LLB-8 is (on all other benchmarks) not sufficient to run many transactions in hardware, and STMs perform slightly better than HyTMs because the latter try to first execute in hardware (unsuccessfully).

Pure HTM has the lowest overhead but its simple fallback mode (serial execution) can quickly decrease performance. HyLSA has higher runtime overhead than Pure HTM but typically scales well.

Benchmark	**LLB-8**	**LLB-8 w/ L1**	**LLB-256**
Genome	HTM: 65% HyLSA: 38–42%	HTM: 90–95% HyLSA: 75%	100%
KMeans-Hi KMeans-Lo	HTM: 100% HyLSA: 25%	95–100%	100%
Vacation-Hi	0%	HTM: 13–14% HyLSA: 3–5%	100%
Vacation-Lo	0%	HTM: 8–11% HyLSA: 1–2%	100%
SSCA2	99–100%		

Table 4.6: Approximate ratio of HTM commits to total number of commits in STAMP.

In the small LinkedList on LLB-256, all transactions can execute in hardware but HyTMs and HTM do not scale. The reason for this behavior is that ASF's conflict detection is on the granularity of cache lines, whereas STMs can use smaller granularities (word-sized in LSA), 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 hash function 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.

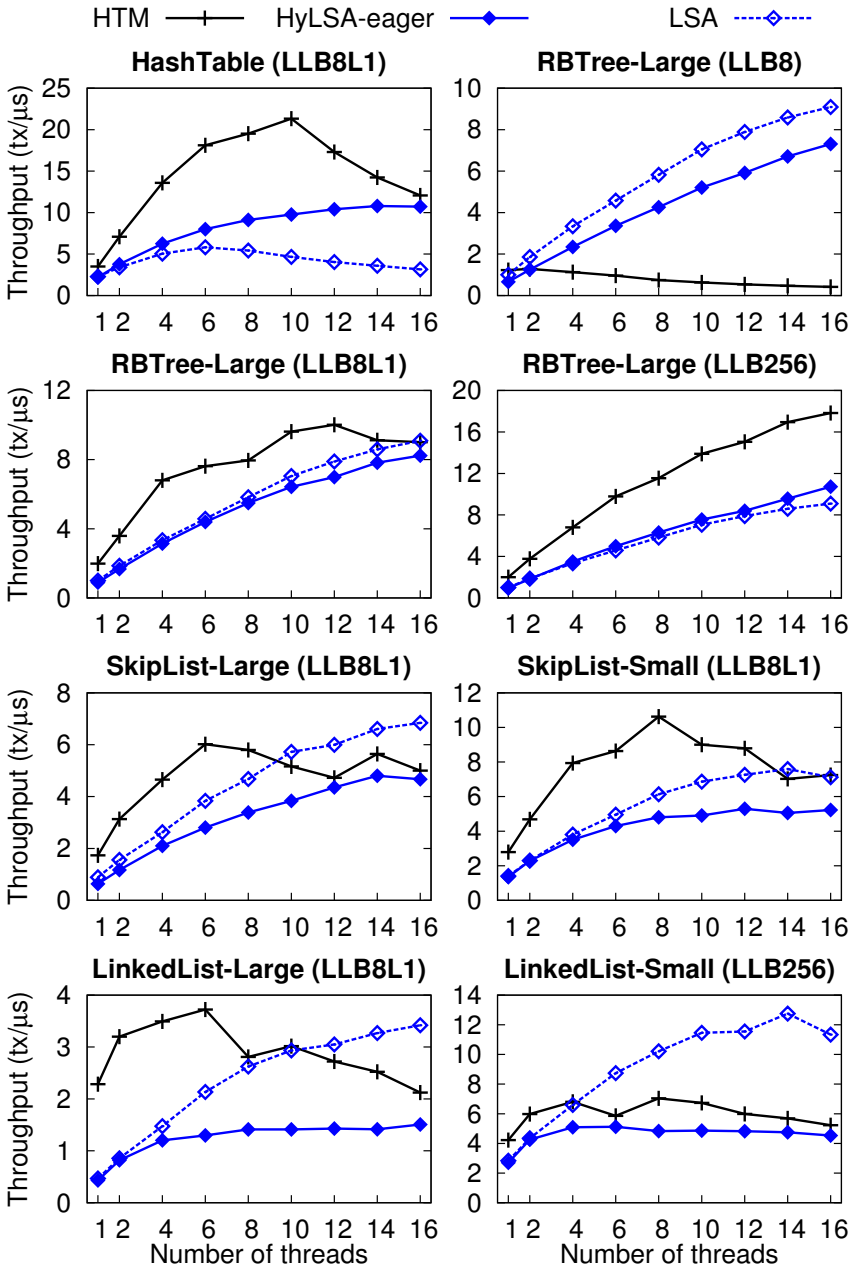


Figure 4.10: Overview of scalability of TMs with IntegerSet.

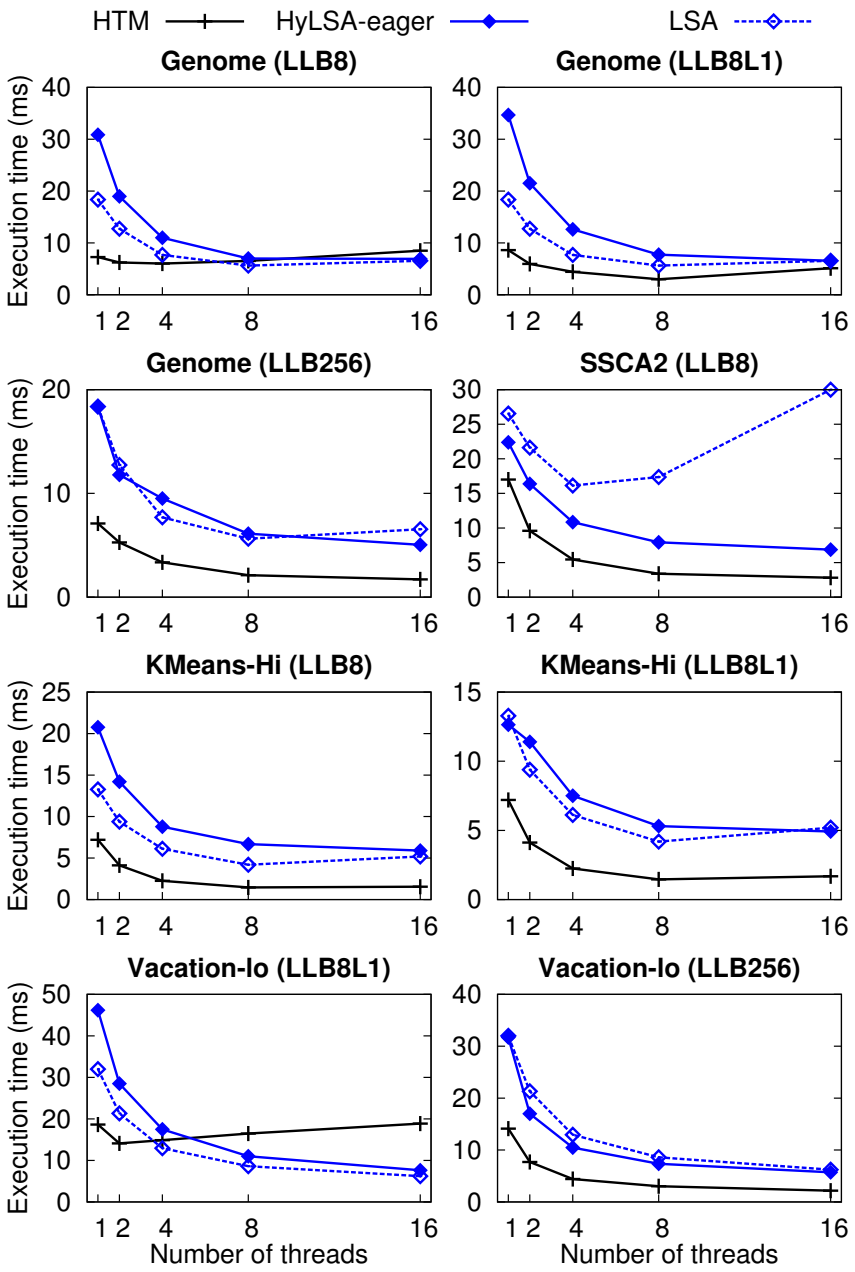


Figure 4.11: Overview of scalability of TMs in selected STAMP benchmarks. SSCA2 performs similar on all ASF implementations. KMeans-Lo performs roughly similar to KMeans-Hi, and KMeans-Hi on LLB-8 w/L1 is similar to LLB-256. Vacation-Hi performs similar to Vacation-Lo, and Vacation-Lo on LLB-8 is similar to LLB-8 w/L1.

To conclude the evaluation, we show performance results for selected applications from STAMP (see Table 4.6) in Figure 4.11. We chose benchmarks that are stable and have parallelism in their workloads, and executed them using STAMP's standard parameter configurations for simulator environments. LLB-256 is again sufficient to execute all transactions in hardware. Genome on LLB-8 also exhibits this behavior. HyLSA's larger capacity requirements for stores decrease the HTM ratio as well.

4.2.6 Conclusion

In this section, we proposed and evaluated novel hybrid software/hardware transactional memory algorithms. They improve upon previous HyTM algorithms by either 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. We confirmed this through experimental evaluation on a near-cycle-accurate x86 simulator with support for AMD's ASF hardware extensions.

While previous HyTM designs have used non-speculative memory accesses inside of hardware transactions, we show that this has a much larger potential and importance if algorithms also make use of non-speculative atomic read-modify-write instructions. We also found it very useful that ASF monitors speculatively accessed locations eagerly for conflicting accesses by other threads. We believe that the general-purpose techniques that we used in our algorithms apply not just to HyTM but can be useful in general for concurrent algorithms based on new synchronization hardware such as ASF.

4.3 Summary

In this chapter, we have tackled the problem of overhead with Software transactional Memory by the use of a hardware support for transactions. We evaluated AMD's ASF in the context of Hardware Transactional Memory to show that hardware facilities enable to reach good performance even with a low number of threads.

Unfortunately, hardware limitations and in particular capacity constraints prevent us from using it in all cases. Henceforth, a fallback solution must complement the hardware support. The serial irrevocable is a solution but it is a performance killer. We proposed and evaluated a new Hybrid Transactional Memory based on ASF. Mixing Software and Hardware Transactional Memory provides a good solution to maintain low overhead hardware transactions without sacrificing concurrency when some software transactions must be executed.

Chapter 5

Integration of a Full Transactional Memory Stack

The support of TM lies at all levels of a usual system stack. We describe in this chapter the integration of Transaction Memory mechanisms at all levels of this stack.

5.1 Challenges of Transactional Memory integration

TM-based programming has been proposed as a promising alternative to lock-based programming, introducing the abstraction of transaction into programming languages. However, up to recently the transaction support has been mostly provided in terms of software libraries and not as a programming language construct. Coding transactions using TM libraries hampers software development since: it is cumbersome and time consuming (the programmer needs to specify all the transactional accesses through library calls), the readability of the code is low (simple memory accesses that need to be transactional become all library calls), it is not portable (the written software stays specific to the TM library for which it is written).

The very fact that a program using a specific TM library is not portable is an important issue for the integration of TM into programming languages. TM libraries perform differently depending on the workload. Programmers may wish to compile the same code with different TMs to find out the best performing TM for a given application, especially if the hardware support for transaction is available. It is thus desirable for a compiler to allow selecting among different TM libraries for compilation.

The ideal solution would be to have a standard TM library interface so that any high level language providing higher-level transactional language constructs could use a single TM library interface. Providing such an interface to the TM library programmer allows the easy integration of any TM library implementation with a programming language.

The integration of TM libraries into programming languages using a standard interface is unfortunately not enough to integrate transactional behavior. To complete the integration, it is required to specify a clear syntax and associated semantic to be exposed to the application programmer. The ideal syntax to express a transaction is a block of code (we call such a block transaction block) in which the enclosed code execute according to a specified transactional behavior. The syntax of such an ideal transaction block is simple: it encloses code as would any other traditional block (e. g., function body, if statement body, loop body etc.). Although

the syntax of the ideal transaction block is that simple, its semantics do not always fit with the semantics of the statements that could be enclosed in a transaction block, introducing difficulties of interoperability of the ideal transaction block with other language constructs.

Currently, the range of language level constructs the TM can control is limited. This is partly an implementation issue but the main problem is that several language constructs (such as system calls, I/O calls, synchronization constructs, exceptions etc.) cannot be rolled back and/or repeated (as transaction semantics would require). Hence, their interoperability with a transaction block needs to be solved at the language level rather than at the TM implementation level.

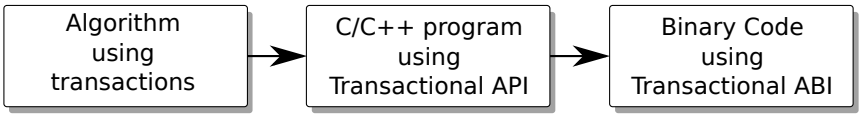


Figure 5.1: Evolution from a transactional algorithm to transactional binary code

Figure 5.1 describes the creation process of a transactional program. The first step is to express the usage of transactions inside an algorithm. Then the algorithm is implemented in a programming language. Finally, the compiler translates the source code to a machine code.

The specification of a standard TM library interface and transactional language constructs are the basic steps for the complete integration of TM, programming languages, and other levels of the system stack. To achieve integration, these goals should be fulfilled with appropriate compilation and runtime tools.

5.1.1 Contributions

In this chapter, we discuss the different aspects of TM integration into programming languages and provide solutions targeting each of these aspects. Part of the solution requires the specification of different interfaces: an interface at the TM library level (denoted as ABI in Figure 5.2) and a second at the programming language level (denoted as API in Figure 5.2). The interface at the TM library level aims at standardizing the interface for TM library calls (to be used by TM library programmers) while the programming language interface aims at describing the syntax and associated semantics of language extensions for transactional constructs (to be used by application programmers). The rest of the integration solutions lies in the implementation of the interface inside a TM runtime library.

5.1.2 Outline

Section 5.2 first describes the specification of programming language interfaces for transactional execution, denoted as API in Figure 5.2. First, we detail the fundamental semantics of atomic blocks and the fundamental transactional constructs. Then, we present the types of transactional guarantees and the requirements on functions for using them within transactional code. Section 5.3 describes the standard TM library interface named Application Binary interface (ABI in Figure 5.2). Section 5.4 presents compilers with transactional support,

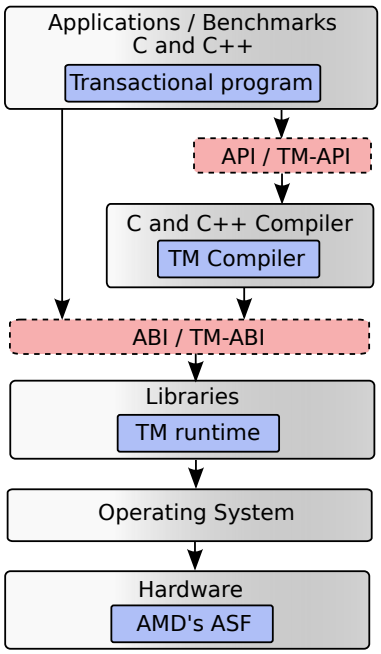


Figure 5.2: A TM-enabled software stack

i. e., API and ABI compliance. It also details the implementation required for making the transactional language constructs operational. It is followed by implementation details and some important features for the performance of the stack. The chapter is concluded with an evaluation of the transactional stack built with transactional applications.

5.2 C/C++ language extensions

So far, programmers have mostly used transactional memory in the form of STM libraries with preprocessor macros to develop transaction of programs for some time. However, when using these libraries, they must explicitly inject calls to the transactional memory libraries in their code (for example, when accessing shared memory locations). This method has severe disadvantages:

- It is harder to program and understand and is error-prone (for example, it's easy to forget to annotate a memory access).
- The compiler does not get insight into a transaction's actions and access patterns. The library cannot perform important optimizations such as detecting which data is shared or thread-private. In contrast, compilers can perform many transaction optimizations at compilation time.
- Close interaction between transactional memory and compiler-generated code such as exception handling is difficult to impossible.

Transactional Memory researchers often define transactions with the "atomic" block but its associated semantic is usually too vague for a practical implementation: it only specifies that the code block must appear atomic. It does not define for instance the behavior of the transaction. The definition of the transactional block for a language must come with a complete specification.

OpenTM [6] was the first tentative of a language extension which was based on pragmas for transactions and OpenMP for parallelization. The main drawbacks are that the usage of pragmas is error-prone and difficult to maintain. It also reduces the effectiveness of compiler optimizations.

We detail in this section the first concerted "draft specification for transactional constructs in C++" [35]. This specification was designed by several partners including some companies such Intel and Sun. Note that the full transactional constructs specification supports transaction nesting and exceptions but we do not describe these points in this thesis.

5.2.1 Fundamental transactional semantics

Although the specification details the semantics of the various TM language constructs, all of these constructs share fundamental semantic characteristics, which we denote as the transactional behavior.

The transactional behavior is based on the definition of a transaction, which is a set of actions delimited, and distinguished from regular code, by a starting action (generally called start) and a terminating action (generally called commit). The code that appears inside the atomic block (between start and commit actions) is defined as transactional code. Any code that is not transactional is called non-transactional code. Any memory access that takes place in transactional code is called transactional access, while a memory access performed in non-transactional code is called non-transactional access.

Transactional behavior

The transactional behavior defines the guarantees required for the execution of the transactional code. A transaction guarantees the transactional behavior if the following properties are enforced for transactional code:

Atomicity: either all the program state modifications performed by a transaction are visible at once to the other concurrent transactions (i. e., the transaction commits), or not visible at all (i. e., the transaction aborts).

Isolation: A transaction executes as if there is no other concurrent transaction in the system.

Consistency: A transaction always acts on a consistent state (even if it would abort in the end).

C++ language extension specifications aim to provide at least this transactional behavior to the application programmer. Note, however, that the transactional behavior is enforced only among transactions; the concurrent interaction of transactional and non-transactional code on the same data is left unspecified and it will be refined as an option in the following.

Irrevocable transactional behavior

The transactional behavior, as defined above, is not always enough for programming languages because some programming statements (such as I/O accesses, system calls or synchronization actions) cannot be rolled back. Such statements are called *transactional unsafe*. Code including irrevocable statements is thus defined as transactional unsafe. The use of transactional unsafe statements in transactional code requires different semantics, called irrevocable transactional behavior. These semantics enforce the atomicity, isolation and consistency properties with respect to other transactions (as in transactional behavior) as well as the requirement of a transaction to be executed only once (for the correct execution of statements that cannot be rolled back).

5.2.2 Fundamental transactional constructs

The specification allows three language constructs to be executed in a transaction: compound statements, expressions and functions. Transactional compound statements are called transaction statements, transactional expressions are called transaction expressions and transactional functions are called function transaction blocks. The keyword that allows performing the execution of these three language constructs in a transaction is `__transaction`. The syntax for each of the transactional constructs is as follows:

Transaction statement: `__transaction compound-statement`

Transaction expression: `__transaction ( expression )`

Function transaction block:

```
function-signature __transaction { function-body }
```

In general, it is common to associate a transaction to a block, where the beginning and the end of the block are the start and commit actions and the content is the set of actions for which transaction guarantees are ensured. Among the three constructs described above, the transaction statement corresponds to a transaction block while the transaction expression and function transaction blocks can be seen as derivations of the transaction statement. A function transaction block is merely a reusable transaction statement, while a transaction expression can be considered as the transactional computation of an expression that is equivalent to the following transaction statement:

```
__transaction { T temp = expression }
```

A transaction expression computes an expression in a transaction and stores the result of the expression in a temporary object.

5.2.3 Types of transactional guarantees

The `__transaction` keyword can be followed by one of two transactional attributes while declaring a construct transactional: `[[atomic]]` or `[[relaxed]]`. These attributes specify that the following guarantee is ensured for the described transactional construct:

- The `[[atomic]]` attribute requires that the described transaction enforces the transactional behavior (e. g., it enforces only atomicity, isolation and consistency). This attribute forbids the use of statements that cannot be rolled back.
- The `[[relaxed]]` attribute suggests that the described transaction executes according to irrevocable transactional behavior if it includes transactional unsafe code, otherwise it executes according to transactional behavior.

A transaction that is assigned an `[[atomic]]` attribute is called an atomic transaction, while a transaction assigned a `[[relaxed]]` attribute is called a relaxed transaction. Syntactically these attributes just need to follow the `__transaction` keyword to make the distinction between the different transactional guarantees as follows (although below the syntax is given for transaction statement the same construction applies to transaction expression and function transaction blocks):

- Atomic transaction: `__transaction [[atomic]] compound-statement`
- Relaxed transaction: `__transaction [[relaxed]] compound-statement`

The explanations of `[[atomic]]` and `[[relaxed]]` attributes imply that atomic transactions enforce stricter guarantees. It is desirable to control that this stricter guarantee is respected at compilation time. The specification denotes any code that can be enclosed inside a relaxed transaction but not inside an atomic transaction as unsafe. More specifically a statement that is used in a transaction is deemed unsafe if any of the following applies:

- The statement is a transaction statement with the `[[relaxed]]` attribute.
- The statement performs any use of a volatile object (initialization, assignment or a read).
- The statement is a function call that is
  - either not explicitly declared safe with the `[[transaction_safe]]` attributes (see Section 5.2.4 for this attribute),
  - or not a virtual function, but contains any of the unsafe statements defined above.

```
__transaction [[atomic]] {
    a = a + 1;
    b = b + 1;
}

__transaction [[relaxed]] {
    printf("%d %d", a, b);
}
```

Listing 5.1: Examples of use for transaction type attribute

A statement is called safe if none of the above applies. According to this definition of safety, atomic transactions are transactions that contain only transactional safe code, i. e., undoable code as defined in the first transaction of Listing 5.1. Since only relaxed transactions can contain transactional unsafe code, irrevocability mechanisms for transactional memory can only be used for implementations of relaxed transactions. Note that relaxed transactions

provide transactional behavior if their content does not include unsafe actions. If a relaxed transaction encloses irrevocable actions, it is only then the relaxed transaction provides irrevocable transactional behavior (where the basic transactional behavior guarantees are still possible while it is only the concurrency of the execution that is hampered). The second transaction of Listing 5.1 shows an example of a transaction with irreversible action, i. e., the `printf` function.

5.2.4 Use of functions in transactional code

Function calls in atomic or relaxed transactions will be treated differently depending on whether the code they contain is safe. If a function contains only safe code it is said to be a safe function. A function can be declared safe using the attribute `[[transaction_safe]]`. It is also possible to explicitly declare a function unsafe using the `[[transaction_unsafe]]` attribute. Function safety attributes are syntactically located in front of the function signature, e. g., a function `f()` returning void can be annotated with one of the above safety attributes as follows:

- `[[transaction_safe]] void f()`
- `[[transaction_callable]] void f()`
- `[[transaction_unsafe]] void f()`

```
[[transaction_safe]] int checkConsistency() {
    return a == b;
}

__transaction [[atomic]] {
      IsOk = checkConsistency();
}
```

Listing 5.2: Example of use for transactional_safe attribute

```
[[transaction_unsafe]] int printf(const char *format, ...);

__transaction [[relaxed]] {
    printf("%d %d", a, b);
}
```

Listing 5.3: Example of use for transactional_unsafe attribute

In Listing 5.2, the `transaction_safe` attribute indicates that the function `checkConsistency` can be called in transaction i. e., all memory accesses can be transactified. The `transaction_unsafe` attribute in Listing 5.3 indicates that the function `printf` does irreversible actions and cannot be called in an atomic transaction.

Function pointers can also be declared safe or unsafe with the same attributes. This is useful to forbid the assignment of an unsafe function or function pointer to a safe function pointer.

In case of class inheritance, member functions preserve safety attributes declared for their base class. For virtual functions the overriding restrictions are as follows: A virtual

function explicitly declared safe could only be overridden by a virtual function also explicitly declared safe.

For simplicity of assigning attributes to functions, the specification allows C++ classes to be annotated with a function safety attribute. Such a class attribute acts as the default attribute for all the member functions declared in the class unless overridden explicitly by the member function declaration. Also class attributes do not apply to functions inherited from a base class or functions included in the class via the "using" declaration.

If a function is not explicitly declared safe, it can still be inferred to be safe from its definition if it contains only safe statements. This is useful especially for template functions that do not take function safety attribute, because their safety can only be determined at compilation time when the template parameters are known. In such a case, the compiler can decide on the safety of the function by analyzing the body of the function.

A last function attribute is the `[[transaction_callable]]` attribute but it has no safety implications. This attribute allows programmers to indicate that the function may execute unsafe code but it is considered safe until it reaches the unsafe statement e. g., the unsafe statement is unlikely because it is located inside a conditional ("if") block. Listing 5.4 shows that the function `checkConsistency` could executed safely in an atomic transaction but can also execute an unsafe statement, e. g., the `printf` function, if the condition is true. Such an attribute can be used if the function is specifically written and optimized for use in relaxed transactions.

```
[[transaction_callable]] int checkConsistency() {
    if (a != b) {
        printf("Inconsistent state");
        return 0;
    }
    return 1;
}
__transaction [[relaxed]] {
    IsOk = checkConsistency();
}
```

Listing 5.4: Example of use for transactional_callable attribute

5.3 Transactional Application Binary Interface

The input of the TM compiler is a C/C++ program, which can be enhanced by the Transactional C/C++ Constructs. This program will be transformed to machine instructions and all transactional operations will be mapped to library function calls. The independence of the TM implementation is important to support multiple underlying TM implementations. The objective is that the same user code can be linked against different TM implementations without requiring recompilation, e. g., when a new algorithm or a hardware TM is released. That way, all the management of transactions and concurrency is performed by an external library: the transactional memory runtime library (TM runtime).

Historically, the first TM runtime implementations for C/C++ have provided APIs to programmatically declare, start, commit, and abort transactions, as well as read from and write to transactional (shared) memory. This is the case for our STM, for instance, which proposes a simplified and historical ABI.

```
sigjmp_buf *stm_start(stm_tx_attr_t *attr);
int stm_commit();
stm_word_t stm_load(volatile stm_word_t *addr);
void stm_store(volatile stm_word_t *addr, stm_word_t value);
```

Listing 5.5: Minimal TinySTM ABI to create transactions

Listing 5.5 shows the minimal ABI that allows to start and commit a transaction, and also to perform transactional accesses by read and write.

An essential aspect of C/C++ TM support is the standardization of the TM runtime library interface that specifies how the compiler maps transactional operations to the underlying TM runtime. Having a standardized interface is intended to be as general as possible so whenever possible it avoids restrictions on the TM library implementation. It allows us to use any complying compiler with any complying TM runtime.

Such convention between the compiler and the TM runtime has been named as the Application Binary Interface (ABI) and has been defined jointly with several partners and companies [34]. This section describes the specification of this interface in detail.

5.3.1 From transaction to code engineering

As mentioned in Section 5.2.1, the main transaction operations are start, commit, load and store, but these have to be converted to real machine code. The conversion process is illustrated by Figure 5.3.

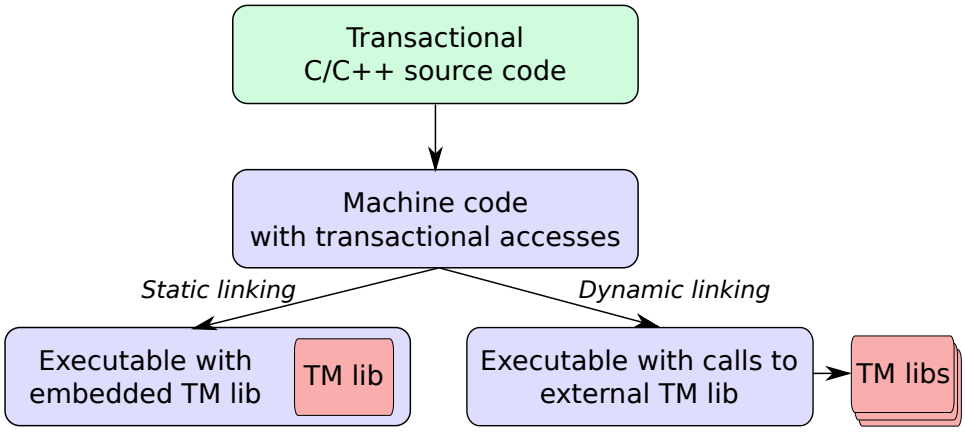


Figure 5.3: Compilation process with Transactional Memory support.

An example of this transformation process using a Transactional C/C++ API is given in Figure 5.4: Note that this transformation is hidden to the developer and happens internally inside the compiler. The result is a compiled binary code (object code in .o files). All functions names are prefixed with `_ITM_` to avoid clashes with any of the other library or user defined function names.

In the rest of this section, we first describe the main ABI functions and then we introduce other ABI functions accompanied with the rationale for their existence.

```
int a,b,c;
__transaction {


  a = b;

  if (c == 1)
    __transaction_abort;


}
```

```
int a,b,c;
ret = _ITM_beginTransaction(pr_instrumentedCode);
if (ret != a_abortTransaction) {
  int tmp = _ITM_RU4 (&a);
  _ITM_WU4 (&b, tmp);
  tmp = _ITM_RU4 (&c);
  if (tmp == 1) {
    _ITM_abortTransaction ();
  }
  _ITM_commitTransaction ();
}
```

Figure 5.4: Example of transformation from API to ABI.

5.3.2 Main ABI functions

This section describes the required functions to have a minimal transactional system.

Starting a transaction

The start of transaction defines the behavior and the properties for the transaction. To that end, the properties that the compiler detected, e. g., irrevocable mode are passed as arguments to the start function and the start returns some attributes to indicate how the transaction must behave.

```
uint32_t _ITM_beginTransaction(uint32_t, ...);
```

The compiler can give properties to the transaction by passing them to `_ITM_beginTransaction`:

- `pr_instrumentedCode`, `pr_uninstrumentedCode` indicate if the instrumented or the uninstrumented code path is available. `pr_multiwayCode` is defined for convenience purposes.
- `pr_hasNoXMMUpdate` (also called `pr_hasNoVectorUpdate`) indicates that the transaction is not using vector registers (i. e., MMX/SSE for x86 CPU).
- `pr_hasNoAbort` indicates that the transaction has no user abort.
- `pr_hasNoRetry` indicates that the transaction has no user retry.
- `pr_hasNoIrrevocable` indicates that the transaction will not attempt to become irrevocable.
- `pr_doesGoIrrevocable` indicates that the transaction has to switch the serial irrevocable mode.
- `pr_aWBarriersOmitted` and `pr_RaRBarriersOmitted` indicate that the compiler has omitted "after Write" or "Read after Read" barriers.
- `pr_undoLogCode` indicates that the transaction has only undo logging, i. e., only restore thread private variables on abort and no other barriers.
- `pr_preferUninstrumented` indicates that the uninstrumented code path is the best choice, e. g., the compiler cannot perform optimizations on the instrumented code path.

- `pr_exceptionBlock` indicates that the transaction contains an exception block.

```
typedef enum
{
   pr_instrumentedCode       = 0x0001,
   pr_uninstrumentedCode     = 0x0002,
   pr_multiwayCode           = pr_instrumentedCode | pr_uninstrumentedCode,
   pr_hasNoVectorUpdate      = 0x0004,
   pr_hasNoAbort             = 0x0008,
   pr_hasNoRetry             = 0x0010,
   pr_hasNoIrrevocable       = 0x0020,
   pr_doesGoIrrevocable      = 0x0040,
   pr_aWBarriersOmitted      = 0x0100,
   pr_RaRBarriersOmitted     = 0x0200,
   pr_undoLogCode            = 0x0400,
   pr_preferUninstrumented   = 0x0800,
   pr_exceptionBlock         = 0x1000,
} _ITM_codeProperties;
```

The result of the begin function describes what actions to take:

- `a_runInstrumentedCode` and `a_runUninstrumentedCode` indicate either to run the instrumented code path or the uninstrumented one.
- `a_saveLiveVariables` and `a_restoreLiveVariables` indicate either to save or restore local variables.
- `a_abortTransaction` indicates to leave completely the transaction.

```
typedef enum
{
   a_runInstrumentedCode       = 0x01,
   a_runUninstrumentedCode     = 0x02,
   a_saveLiveVariables         = 0x04,
   a_restoreLiveVariables      = 0x08,
   a_abortTransaction          = 0x10,
} _ITM_actions;
```

Committing a transaction

There are several functions for committing a transaction. These correspond to the different cases:

- `_ITM_commitTransaction` is the regular commit function;
- `_ITM_tryCommitTransaction` is used in the case of C++ program to commit a transaction when there is a try/catch block.

```
void _ITM_commitTransaction (void);
bool _ITM_tryCommitTransaction (void);
```

Read memory barriers

Since the TM runtime library needs to know when the memory is read to manage conflicts, the ABI also incorporates the read function. Unfortunately, it is not enough to have only

one read function because many types exists in C, so to cover all kind of type sizes, many functions are defined for sizes from 1 byte to 8 bytes and for different other types (floating point and complex value: `float`, `double`, `long double`).

```
uint8_t _ITM_RU1(const uint8_t *);
uint16_t _ITM_RU2(const uint16_t *);
uint32_t _ITM_RU4(const uint32_t *);
uint64_t _ITM_RU8(const uint64_t *);

float _ITM_RF(const float *);
double _ITM_RD(const double *);
long double _ITM_RE(const long double *);

__m64 _ITM_RM64 (const __m64 *);
__m128 _ITM_RM128 (const __m128 *);

float _Complex _ITM_RCF(const float _Complex *);
double _Complex _ITM_RCD(const double _Complex *);
long double _Complex _ITM_RCE(const long double _Complex *);
```

Note that even if a read function returns a unsigned int, it works fine if the return type declared is a signed int because the compiler will not try to convert the value.

Write memory barriers

As for read barriers, the transactional library needs to be informed of all writes to the shared memory to manage concurrent accesses. Similarly again to reads, a collection of write barrier functions are needed for accesses that correspond to different size and types. Write barrier functions have 2 parameters: the address to be written and the value.

```
void _ITM_WU1 (uint8 *, uint8);
void _ITM_WU2 (uint16 *, uint16);
void _ITM_WU4 (uint32 *, uint32);
void _ITM_WU8 (uint64 *, uint64);

void _ITM_WF (float *, float);
void _ITM_WD (double *, double);
void _ITM_WE (long double *, long double);

void _ITM_WM64 (__m64 *, __m64);
void _ITM_WM128 (__m128 *, __m128);

void _ITM_WCF (float _Complex *, float _Complex);
void _ITM_WCD (double _Complex *, double _Complex);
void _ITM_WCE (long double _Complex *, long double _Complex);
```

Aborting a transaction

The ABI function `_ITM_abortTransaction` enables a transaction to abort explicitly. Different parameters indicate the type of abort. Note that this function never returns.

```
typedef enum {
    userAbort = 1, userRetry = 2,
    TMConflict= 4, exceptionBlockAbort = 8,
    outerAbort = 16
} _ITM_abortReason;

void _ITM_abortTransaction(_ITM_abortReason);
```

On the contrary, _ITM_rollbackTransaction returns (no longjmp) and it rolls back a transaction to the innermost nesting level.

```
void _ITM_rollbackTransaction (void);
```

Changing execution mode

A transaction may have to run in a different execution mode to meet its properties. For example, a transaction has to run in serial and irrevocable mode if an operation is not undoable.

```
typedef enum
{
    modeSerialIrrevocable,
} _ITM_transactionState;

void _ITM_changeTransactionMode (_ITM_transactionState);
```

The argument to _ITM_changeTransactionMode indicates in which mode to run. It can be extended to allow new modes of execution.

5.3.3 Extended ABI functions

To allow advanced functions such as memory allocation in transactions and to allow optimizations of the TM runtime, several ABI functions have been added to the basic ones.

Local variables accesses

In a transaction, local variables, which are outside of the transaction block can be accessed and also modified but if the transaction conflicts and must roll back, these local variables must be restored. The purpose of these ITM_L* functions is to save the variable before it is modified by the transaction. Of course, all accesses of local variables could be done with ITM_RU/ITM_WU but using these specific functions save a call to ITM_RU and is expected to be less costly than a regular store.

```
void _ITM_LU1 (const uint8 *);
void _ITM_LU2 (const uint16 *);
void _ITM_LU4 (const uint32 *);
void _ITM_LU8 (const uint64 *);
void _ITM_LF (const float *);
void _ITM_LD (const double *);
void _ITM_LE (const long double *);
void _ITM_LM64 (const __m64 *);
void _ITM_LM128 (const __m128 *);
void _ITM_LCF (const float _Complex *);
void _ITM_LCD (const double _Complex *);
void _ITM_LCE (const long double _Complex *);
```

_ITM_LB permits to log an arbitrary size of memory.

```
void _ITM_LB (const void*, size_t);
```

Optimized loads and stores using compiler hints

In order to alleviate performance issues encountered by Transactional Memory, the ABI incorporates some specific functions for accesses to the same address.

One optimization is to instrument code so as to avoid unnecessary calls for the same memory location. An expression like `v = v + 42` in a trivial implementation would add `'v'` first to the read set and then later to the write set. In an optimized implementation with a sophisticated TM runtime, the compiler could add code to announce that the variable is now read but later written to.

The TM compiler can easily detect if the same address is accessed. It thus optimizes the TM instrumentation by injecting four of these combined notifications of the TM runtime:

- Write-after-read (WaR): as described above.
- Read-after-write (RaW): the other way round. The variable also gets added to the read set.
- Read-after-read (RaR): a second read dominates the second.
- Read-for-write (RfW): the value is read to be stored.

The complete list of optimized load and store functions for data sizes of 8 bytes is given below. The same convention name applies to other data types.

```
uint64 _ITM_RaRU8 (const uint64 *);
uint64 _ITM_RaWU8 (const uint64 *);
uint64 _ITM_RfWU8 (const uint64 *);
void _ITM_WaRU8 (uint64 *, uint64);
void _ITM_WaWU8 (uint64 *, uint64);
```

Optimized block accesses

Accessing consecutive addresses using the regular read function can be costly because it requires using many calls to the `_ITM_RU` function. In order to improve this situation, transactional versions of `memset`, `memcpy` and `memmove` have been defined.

`memcpy` and `memmove` have the same optimized barriers as loads and stores, but different versions of these functions follow the abbreviations stated below as a naming convention that determines whether the source or the destination can be accessed in a non-transactional manner:

- “R” indicates read.
- “W” indicates write.
- “n” indicates non-transactional region.
- “t” indicates transactional region.
- “aR” indicates after read access.
- “aW” indicates after write access.

The complete list of different transactional versions of the `memcpy` function is as follows:

```
void _ITM_memcpyRnWt(void *, const void *, size_t);
void _ITM_memcpyRnWtaR(void *, const void *, size_t);
void _ITM_memcpyRnWtaW(void *, const void *, size_t);
void _ITM_memcpyRtWn(void *, const void *, size_t);
void _ITM_memcpyRtWt(void *, const void *, size_t);
void _ITM_memcpyRtWtaR(void *, const void *, size_t);
void _ITM_memcpyRtWtaW(void *, const void *, size_t);
void _ITM_memcpyRtaRWn(void *, const void *, size_t);
void _ITM_memcpyRtaRWt(void *, const void *, size_t);
void _ITM_memcpyRtaRWtaR(void *, const void *, size_t);
void _ITM_memcpyRtaRWtaW(void *, const void *, size_t);
void _ITM_memcpyRtaWWn(void *, const void *, size_t);
void _ITM_memcpyRtaWWt(void *, const void *, size_t);
void _ITM_memcpyRtaWWtaR(void *, const void *, size_t);
void _ITM_memcpyRtaWWtaW(void *, const void *, size_t);
```

The `memmove` function has the same transactional versions as `memcpy`.

The `memset` function is simpler because it has only the destination parameter. The different transactional versions available for `memset` are:

```
void _ITM_memsetW(void *, int, size_t);
void _ITM_memsetWaR(void *, int, size_t);
void _ITM_memsetWaW(void *, int, size_t);
```

Transaction descriptor extension

All ABI functions require locating the transaction descriptor of the current transaction using Thread Local Storage (TLS), which can be costly in some cases (e. g., shared dynamic library requires calls to pthread library for all TLS accesses). In order to improve performance, another ABI version with the transaction descriptor as extra argument exists.

```
typedef struct { } _ITM_transaction;
_ITM_transaction * _ITM_getTransaction (void) ;
uint32_t _ITM_beginTransaction(_ITM_transaction *, uint32_t,  ...);
void _ITM_commitTransaction (_ITM_transaction *);
```

5.3.4 ABI functions available to the user

The user of transaction blocks may need to do specific actions while in a transaction, so some functions are available directly from the ABI. This is particularly useful to deal with external actions as described in Section 5.4.6.

The application may need to perform specific actions upon transaction commit or rollback. `_ITM_addUserCommitAction` adds an action to the commit log.

```
typedef void (* _ITM_userCommitFunction) (void *);
void _ITM_addUserCommitAction(_ITM_userCommitFunction, _ITM_transactionId_t, void *)
```

`_ITM_addUserUndoAction` adds an action to the undo log that will be executed if the transaction aborts.

```
typedef void (* _ITM_userUndoFunction)(void *);
void _ITM_addUserUndoAction(_ITM_userUndoFunction, void *)
```

The user can get the unique thread number that the transactional library generates.

```
int _ITM_getThreadnum(void)
```

In some particular cases, the application may need to unprotect previous transactional accesses, e. g., for a weaker transactional memory model.

```
void _ITM_dropReferences (void *, size_t)
```

From a user perspective, it is important to check that the current version of the transactional library is compatible with its application. Two functions are defined to check the compatibility and to get the name of transactional library.

```
int _ITM_versionCompatible (int);
const char * _ITM_libraryVersion (void);
```

The function `_ITM_error` allows raising a fatal error while in a transaction:

```
void _ITM_error(const _ITM_srcLocation *, int errorCode);
```

The user may propose alternative code if the transaction is in regular or irrevocable mode. To that end, `_ITM_inTransaction` returns the status of the current transaction.

```
_ITM_howExecuting _ITM_inTransaction(void);

typedef enum
{
    outsideTransaction = 0,
    inRetryableTransaction,
    inIrrevocableTransaction
} _ITM_howExecuting;
```

The application may need to get a transaction identifier, e. g., for debugging purposes.

```
typedef uint64_t _ITM_transactionId_t;
_ITM_transactionId_t _ITM_getTransactionId(void);
```

5.4 Integrating transactional support

In addition to language constructs, an essential aspect of C/C++ TM support is to support the ABI, which specifies how the compiler maps transactional operations to the underlying TM library calls. The ABI specification has been described in the previous section.

The TM stack follows the specifications being defined as part of the standardization process and provides implementations that match the interface specifications (both for the ABI and programming language constructs).

Below, we present the tools that implement these specifications for the C and C++ languages, and their specificities. Then, we discuss some challenges of integration and propose appropriate solutions that we apply to our TM library.

5.4.1 Transactional Memory compilers

Due to the nature of TM, which requires a supporting runtime environment to implement software or hybrid TM, and the tight integration of TM into the OS's ABI it is essential to coordinate the efforts among the parties working on TM support for a given platform.

TM compiler has an essential role because the usability of TM depends on the availability of programming language extensions. This requires collaboration at different level and particularly the ABI and the platform.

Several companies such as Red Hat and Intel are collaborating closely on the TM-ABI specification to make their compilers compatible with it. Among available TM compilers[1], three of them support the x86 platform. We introduce them in the following paragraphs.

GCC with Transactional Memory support

The GNU Compiler Collection (GCC) is a free and open source compiler, which is considered as an industry-standard compiler. Its frontend supports a variety of languages, but its primary targets remains C and C++. Its backend supports a significant number of processors, including x86 and a great number of platforms, including GNU/Linux. The open source nature of GCC encourage everyone to contribute. Several companies, such as Red Hat, IBM, Novell, Google, Apple, Intel and AMD are involved in the development of GCC.

The support for transactional memory in the GCC Compiler has been developed by Red Hat in the context of the VELOX Project. Its development is still in progress. It should be merged with future releases of GCC when the TM support will be deemed stable. GCC is the most popular C/C++ compiler for open source platforms. It is likely to reach a much greater developer base than Intel or Sun prototype compilers.

It supports the draft specifications for C/C++ transactional construct which are summarized in Section 5.2. One difference from the specifications is the attribute assignation, as GCC does not support C++0x type attributes yet. For example, the attribute `[[transaction_safe]]` should be written `__attribute__((transaction_safe))`. The implementation follows the TM ABI for Linux, as summarized in Section 5.3. Additionally, GCC has specific optimizations which are detailed in the following paragraph.

Optimizations Instrumenting code to implement transactional semantics means adding significant overheads. All memory accesses have to be considered and possibly made through to the TM runtime. There are several types of memory accesses for which no instrumentation is necessary because there can be no conflict with another thread.

Accesses to thread-local storage (TLS) do not have to be annotated because other threads are guaranteed to not have access to this memory. Only undo operations have to be performed upon cancellation. Note that this is not even necessary for transaction-local memory locations.

[1]Throughout this thesis, we focused our TM library on x86 CPU so both TM compilers from Sun and IBM are excluded from our study as they support respectively SPARC CPU and AIX operating system. However, all mechanisms are very similar and can be transposed to other platforms.

GCC can also optimize accesses to newly allocated memory (e. g., using malloc). As long as the compiler can determine that the memory has not escaped to other threads only minimal instrumentation is necessary for the undo operation.

GCC supports optimized load and store barriers as described in Section 5.3.3 and evaluated in Section 5.5.2.

Finally, GCC also detects if the transaction performs read-only accesses. In this case, a flag is set as an attribute of the transaction. This information is then used for optimizations that boost the speed of the read-only transaction.

Dresden TM Compiler

The Dresden TM Compiler (DTMC) is a compilation tool that supports transactions in C and C++ programs. DTMC supports a large subset of the transactional language constructs for C++ (transaction statements, function attributes; see Section 5.2).

DTMC transforms transactional C/C++ programs in a multi-pass process. It is based on the LLVM compiler framework [12], which allows the construction of highly modular compilers. LLVM's compiler front-end for C/C++ (`llvm-gcc`) parses and transforms source code into LLVM's intermediate representation (IR). To support transaction statements, the TM support code that Red Hat engineers developed for the GNU Compiler Collection (`gcc-tm`) was ported it to `llvm-gcc`. The output of the modified `llvm-gcc` is thus LLVM IR in which transaction statements are visible.

DTMC maps transaction statements of the LLVM IR to calls to a TM runtime library. It uses a compiler pass that transforms LLVM IR with transaction statements so that (1) memory accesses in transactions are rewritten as calls to load and store functions in the TM runtime library, (2) transactions are started and committed using calls to the TM library, and (3) function calls inside transactions are redirected to "transactional" clones of the original functions. This compiler pass is a significantly improved and extended version of Tanger [26].

DTMC performs many TM optimizations but in particular, it can detect when transactional accesses are word-aligned (see Section 5.4.5). It also takes the advantage of Link Time Optimization (LTO) provided by LLVM to precisely detect non-shared variable and to allow the inlining of the TM library functions (see Section 5.4.7).

Intel C/C++ STM Compiler

The Intel C++ STM Compiler is a prototype for x86 platforms based on the production Intel C/C++ compiler 11.0. It was the first compiler to support the draft specification of Transactional Language Constructs for C++ (see Section 5.2).

Due to its closed source, we just have limited information about it. It fully supports the TM-ABI including optimized transactional load and store barriers.

5.4.2 TinySTM and ABI Compatibility

TinySTM has been developed initially to evaluate the performance of the Lazy Snapshot Algorithm. It shows good scalability with transactional benchmarks and applications.

This led us to extend TinySTM to be compatible with the TM ABI. This compatibility allows using our Transactional Memory Library with available TM compilers such as GCC, DTMC and Intel Compiler. All latest applications such as RMS-TM [38] uses transaction blocks instead of manual instrumentation. This ABI compatibility allows testing TinySTM with a larger spectrum of applications. It also enables us to have a fair comparison with other TM libraries.

The support of the ABI has required modifications to TinySTM implementation on the following aspects:

- Function name adaptation to match ABI names.
- Support for transactional loads and stores of different size.
- Support for context saving on transaction start.
- Support for several features required by the ABI specification, such as irrevocability (both serial and concurrent).
- Support for optimized transactional loads and stores.

5.4.3 Memory management

In a transaction, the memory is managed differently because (1) if a transaction aborts the memory allocated during the transaction execution has to be freed, and (2) memory freed during the transaction execution has to be protected until commit against concurrent accesses and the actual free operation should be performed only when the transaction commits.

Memory management is not part of the official ABI but it needs anyway to be managed since most of applications use dynamic memory allocation in transaction. All TM compilers have different ways to manage it as defined in Listing 5.6 and in Listing 5.7.

```
void *_ITM_malloc (size_t);
void *_ITM_calloc (size_t, size_t);
void _ITM_free (void *);
```

Listing 5.6: transactional memory allocation function with GCC and DTMC

```
void *malloc._$TXN (size_t);
void *calloc._$TXN (size_t, size_t);
void free._$TXN (void *);
```

Listing 5.7: transactional memory allocation function with the Intel STM Compiler

The default `new` and `delete` operators of the C++ language can be wrapped by using transactional `malloc`, transactional `free` or by using dedicated transactional calls but this is not yet standardized. TM compilers manage new and delete operators by wrapping the original operator to the transactional one as defined in Listing 5.8 for GCC.

```
/* void *operator new(unsigned int) */
void *_ZGTtnwj(unsigned int);
/* void *operator new[](unsigned int) */
void *_ZGTtnaj(unsigned int);

/* void *operator new(unsigned long) */
```

```
void *_ZGTtnwm(unsigned long);
/* void *operator new[](unsigned long) */
void *_ZGTtnam(unsigned long);

/* void operator delete(void*) */
void _ZdlPv(void *);
/* void operator delete[](void*) */
void _ZdaPv(void *);
```

Listing 5.8: new and delete transactional operators with GCC

5.4.4 Clones and indirect functions

The compiler is responsible for instrumenting code within a transaction block, including function calls.

Cloned functions In order to allow function calls in a transaction, the TM compiler creates instrumented clones of functions. The clone is created if the compiler detects that a function can be called in a transaction or if the function is annotated to be used in transaction, e. g., with the `transaction_safe` attribute.

The compiler uses a name mangling convention for the transactional (i. e., instrumented) clone of a function. This convention for transactional clone allows identifying cloned functions and linking different code objects together. Unfortunately, the naming convention is not defined in the TM-ABI which leads to different implementation as follows:

GCC prepends `_ZGTt` to the name of cloned functions.

DTMC prepends `tanger_txnal_` to the name of cloned functions.

Intel STM Compiler appends `_$TXN` to the name of cloned functions.

Indirect function calls In the case of the use of function pointers, a compiler cannot statically identify the version of the function called. It thus relies on a runtime technique to select which function to effectively call i. e., the clone in case of atomic transaction and the original in case of irrevocable transaction. In case of an atomic transaction, the TM runtime returns the cloned function pointer and if no clone exists, the TM runtime stops execution. In case of a relaxed transaction, it returns the cloned function pointer and if no clone exists, the TM runtime changes the transaction mode to serial irrevocable and returns the pointer to the original function.

GCC and DTMC work the same way for dealing with indirect function calls but with different names as described in Listing 5.9 for GCC and in Listing 5.10 for DTMC. The compiler adds a translation table to the TM runtime that records original function pointers and also pointer to their clones.

```
void *_ITM_getTMCloneOrIrrevocable (void *);
void *_ITM_getTMCloneSafe (void *);
void _ITM_registerTMCloneTable (void *, size_t);
void _ITM_deregisterTMCloneTable (void *);
```

Listing 5.9: Runtime functions required by GCC for indirect function calls

```
void tanger_stm_indirect_register_multiple(void* nontxnal, void* txnal, uint32_t version);
void *tanger_stm_indirect_resolve_multiple(void *nontxnal_function, uint32_t version);
```

Listing 5.10: Runtime functions required by DTMC for indirect function calls

The current version of the Intel STM Compiler does not support transactional indirect function calls through function pointers. It supports function pointers in relaxed transactions by switching to serial irrevocable mode.

To allow compilers interoperability, the future version of the ABI should define these convention names for cloned functions and for clone registration.

5.4.5 Store barriers

TM compilers face some problems that cannot be solved at compile time, e. g., "does the write happen in stack?", "is the write address word-aligned?". It is thus the reponsibility of the TM runtime to handle these problems, which make TM-ABI store barriers more complex than regular store barriers.

Stores into the stack The compiler can fall in a situation where the pointer may escape the control of compiler or the pointer cannot be determined statically. As seen previously, the TM-ABI specifies functions to manage stack-local variables, e. g., `ITM_LU8`, but a regular write barrier, e. g., `ITM_WU8` may also write into the stack.

The TM-runtime must thus detect when the address is into the stack memory and write it straightaway to avoid possible stack corruption. The following example shows a situation that leads to stack corruption.

```
1  extern [[transaction_safe]]
2  int mod(int *, int *);
3
4  int bar() {
5    int l3 = 3, l4 = 4;
6    return mod(&l3, &l4);
7  }
8
9  void foo() {
10   int l1 = 1, l2 = 2;
11   __transaction {
12     l1 = bar();
13     if (l1 == 0)
14       rollback();
15   }
16 }
```

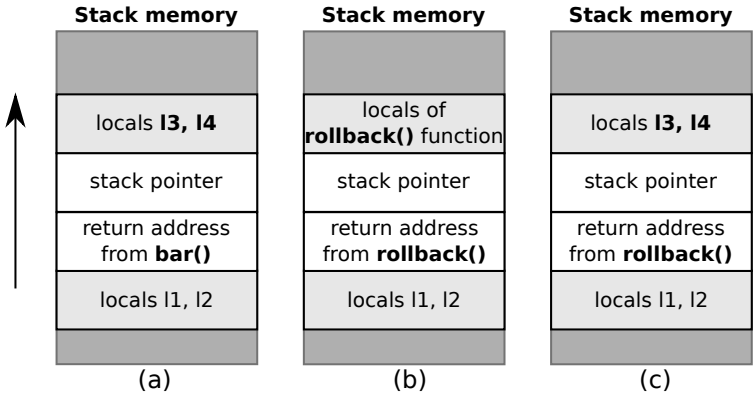


Figure 5.5: An example which leads to stack corruption if store barriers do not check address to be written. The left part shows the C code and the right part shows the stack content at different steps of execution: (a) at line 6, (b) line 14 and (c) at line 14 at the end of the `rollback` function.

In Figure 5.5, we can see at line 6 (a) and line 14 (b), the locals of `bar` and the locals of `abort` are at the same position in the stack. At line 14, the rollback of the transaction

restores the values of l3 and l4 to their original values. Unfortunately, this rollback erases the current local variables of the `rollback` function, i. e., it corrupts the stack and leads to a faulty execution, as described in (c).

Misaligned stores In the TM-ABI specifications, store barriers need to deal with accesses that are not occurring on their natural alignment, i. e., that are misaligned. The TM library may need to handle this specific case because the memory address can overlap two different protected areas.

As the compiler may detect aligned accesses, the TM library can provide additional load and store barriers for faster access with aligned memory. DTMC provides such possible optimization.

5.4.6 Support for external actions

An external action is a function or library used by an application, which is not under the control of the application developer. All functions of an external library or system calls are external actions. The support for external actions is crucial to allow the wide adoption of transactional memory.

Transactional memory has to deal with these in the case where applications are using external libraries or system calls to make progress. There is one exception, if the external library can be transactified and thus proposes transaction-safe functions; in that case, the function call will not be the external action but a transaction-safe function.

We propose different solutions to handle such actions:

- Switching to irrevocable mode;
- Proposing an alternative action;
- Keeping the external action as it is;
- Deferring the external action to transaction commit;
- Compensating the external action on transaction abort.

All these solutions can be easily implemented using the Transactional C/C++ Constructs (TM API) and the TM ABI.

Irrevocability modes for external actions

The easiest way to deal with an external action is to use the irrevocable mode (also called inevitable). In irrevocable mode, a transaction is protected from aborting and will commit eventually. In its simplest form, irrevocable mode is implemented by executing an irrevocable transaction alone while no other transaction is in progress (serial mode). Even though this approach is simple and safe, it does not provide any concurrency and should be reserved as a fallback mechanism for special situations.

The TM API provides a specific semantic to that end. A “relaxed” transaction indicates that the transaction can go irrevocable (See Listing 5.11). Marking a function with the attribute `transaction_unsafe` indicates an unsafe statement and thus forces the transaction to switch to irrevocable mode before it proceeds to the call.

```
__attribute__((transaction_unsafe))
void external();

__transaction [[relaxed]] {
  external();
}
```

Listing 5.11: Example of an unsafe statement using the TM-API

Unfortunately, the irrevocable mode is too pessimistic and doesn't allow concurrency with any other transaction. In the TM API, we propose a new irrevocable mode, named *concurrent irrevocable* mode, that allows an irrevocable transaction to execute concurrently with other non-irrevocable transactions. Once a transaction has acquired the irrevocable status, no other update transaction is allowed to commit. Since an irrevocable transaction is guaranteed to never abort, in case of a conflict, the other optimistic transaction will systematically abort. This irrevocable mode provides a higher level of concurrency. Read only transactions can run and are allowed to commit, update transactions can run but delay their commit phase until after the completion of the irrevocable transaction.

Proposing alternative actions

Another solution to support an external action is to propose a replacement function that is transaction safe. The TM API proposes a specific definition (See Listing 5.12), `transaction_wrap`, which allows the definition of a replacement function of an otherwise transaction unsafe function. This replacement function is called inside the transaction instead of the unsafe avoiding the switch to irrevocable mode.

```
__attribute__((transaction_wrap(external),transaction_safe))
void external_safe() {
  // Safe replacement for the external action
}

int main() {
  __transaction {
    external();
  }
}
```

Listing 5.12: Example definition of a transaction-safe alternative of an external function

Keeping the external actions

In some specific actions, the action itself, even if it is external, is safe to be executed in a transaction. This is typically the case if the external action is stateless. For example, the function or the system call `getpid`, which returns the process ID of the current process, can be called transparently. The TM API provides a specific attribute, `transaction_pure`, which allows marking functions that have this behavior (See Listing 5.13).

```
__attribute__((transaction_pure))
pid_t getpid(void);

__transaction {
  getpid();
}
```

Listing 5.13: Example of the marking of a function as pure

Deferring external actions to transaction commit

Another mechanism to deal with an external action is to delay and execute it at commit time. An example can be the `printf` function. On way to delay the execution is that all `printf` calls in a transaction append their output to a buffer and the output is only effective when the transaction commits (See Listing 5.14). The TM ABI allows such behavior using the `_ITM_addUserCommitAction` function.

```
#define BUFFER_SIZE         (65536)
__thread char buffer[BUFFER_SIZE];

void printf_commit(void *arg) {
  printf("%s", buffer);
  buffer[0] = '\0';
}

void printf_abort(void *arg) {
  buffer[0] = '\0';
}

__attribute__((transaction_pure, transaction_wrap(printf)))
int wrap_printf(const char *format, ...) {
  int ret;
  char local_buffer[BUFFER_SIZE];
  va_list ap;

  va_start(ap, format);
  ret = snprintf(local_buffer, BUFFER_SIZE, format, ap);
  va_end(ap);
  strncat(buffer, local_buffer, BUFFER_SIZE);

  _ITM_addUserUndoAction(printf_abort, NULL);
  _ITM_addUserCommitAction(printf_commit, 1, NULL);

  return ret;
}
```

Listing 5.14: Example of a delayed action for printf

Compensating external actions on transaction abort

Finally, another way to deal with external actions in transactions is to compensate external actions on abort. An example could be the `chdir` function, which changes the current directory of the process. The `chdir` is made effective in the transaction but if the transaction needs to abort, a compensation function must reset the directory to the previous path. The TM ABI provides the `_ITM_addUserUndoAction` to that end (See Listing 5.15).

```
#define BUFFER_SIZE   (65536)
__thread char buffer[BUFFER_SIZE];

void chdir_abort(void *arg) {
  chdir(buffer);
  buffer[0] = '\0';
}

void chdir_commit(void *arg) {
  buffer[0] = '\0';
}

__attribute__((transaction_safe, transaction_wrap(chdir)))
int chdir_wrap(const char *path);

int chdir_wrap(const char *path) {
  /* Only get the directory the first time */
  if (buffer[0] == '\0') {
    if (getcwd(buffer, BUFFER_SIZE) == NULL) {
      return -1;
    }
    _ITM_addUserUndoAction(chdir_abort, NULL);
    _ITM_addUserCommitAction(chdir_commit, 1, NULL);
  }
  return chdir(path);
}
```

Listing 5.15: Example for compensation action for chdir()

These mechanisms help to manage external actions also for the hardware integration.

5.4.7 Integrating AMD's ASF efficiently

ASF-TM is our Transactional Memory library that uses AMD's ASF as hardware support to speed up the TM system (see Chapter 4 for more detail). It also implements the TM-ABI to be used with TM compilers and thus by TM applications.

We have used a TM compiler, DTMC (see Section 5.4.1), and ASF-TM to execute such transactional C and C++ programs with the help of ASF.

```
extern long cntr;
void increment() {
  __transaction {

    cntr = cntr + 5;

  }
}
```

```
extern long cntr;
void increment() {
  _ITM_beginTransaction(...);

  long l_cntr = (long) _ITM_RU8(&cntr);
  l_cntr = l_cntr + 5;
  _ITM_WU8(&cntr, l_cntr);
  _ITM_commitTransaction();
}
```

```
; mem1 for cntr

SPECULATE
JNZ      handle_abort
LOCK MOV RCX, [mem1]
ADD      RCX, 5
LOCK MOV [mem1], RCX
COMMIT
```

Figure 5.6: An example of how C code with a transaction statement (left) is transformed to targeting a TM library ABI (middle) and finally to native code that uses ASF (right). Note that additional code around `SPECULATE` for providing full semantics of `_ITM_beginTransaction` has been omitted for brevity.

Inlining of TM library In LLVM, the intermediate representation of the code is still available at the final linking stage when creating the application's executable code. This allows the compiler to perform whole-program optimization and code generation, which includes inlining the functions in the TM library if this library is linked statically. This generally results in code of the same quality as if the compiler inserted the TM instrumentation code directly. So, we use link-time optimization to reduce or even eliminate run-time overheads due to function calls.

Multiple code path The TM compiler can also create different code paths for a transaction. These code paths use functions for different runtime modes of the TM, and the TM library determines at runtime (i. e., when starting or restarting a transaction) which code path will be executed. For example, an STM and an ASF code path can coexist and can be optimized independently.

Figure 5.6 shows the transformation stages of a simple atomic block in C code. There is no dependence on ASF before ASF-TM is linked to the application (last transformation stage), still link-time optimization can inline ASF instructions. Please note that several implementation details have been omitted for clarity (e. g., ASF-TM requires more software support to begin and commit transactions).

Safely executing non-speculative code There are a few challenges when implementing ASF-TM. ASF permits non-speculative memory accesses within transactions. This allows the reduction of the read-set size of a transaction and, hence, larger transactions can be executed with ASF. However, as a consequence, we need to take care of non-speculative code called from within an ASF speculative region.

ASF requires programmers or compilers to explicitly mark memory accesses within a transaction that are speculative. If a non-transactional function f (e. g., within an external library) were to be called within an ASF speculative region, all memory accesses of this function would be non-speculative. These non-speculative memory updates of f could cause consistency issues if the region is aborted.

Transactions might call external functions for several reasons: for example, memory management or exception handling. STMs therefore deal with calls to external functions in different ways as described in Section 5.4.6.

ASF-TM uses the "alternative function" approach, for example, for a transactional `malloc` function. Because the semantics of this function are known, the transactional version can be built so it is robust against asynchronous aborts by ASF. This is particularly easy to ensure for functions that only operate on thread-local data. For example, ASF-TM uses a custom memory allocator for in-transaction allocations to avoid having to abort and execute in serial-irrevocable mode. This allocator still uses the default allocator internally, but executing the standard `malloc` function in a speculative region would not be safe because of potential incomplete updates to the memory allocation metadata. Note that non-transactional functions can be aborted at any point when using ASF (e. g., if a memory location that has been speculatively read is modified in another thread). Such asynchronous aborts make alternative functions difficult to implement correctly.

When compiling for ASF-TM, DTMC will always use the "irrevocable transaction" approach (i. e., switch a transaction to serial-irrevocable mode) before calling a function for which there exists no ASF-safe version.

We used static linking and link-time optimization when creating the application code evaluated in Chapter 4.

5.5 Evaluation of the transactional software stack

5.5.1 Cost of the standardized ABI

As described in Section 5.4.5, TM-ABI loads and stores have a higher overhead than naive TinySTM due to extra checks.

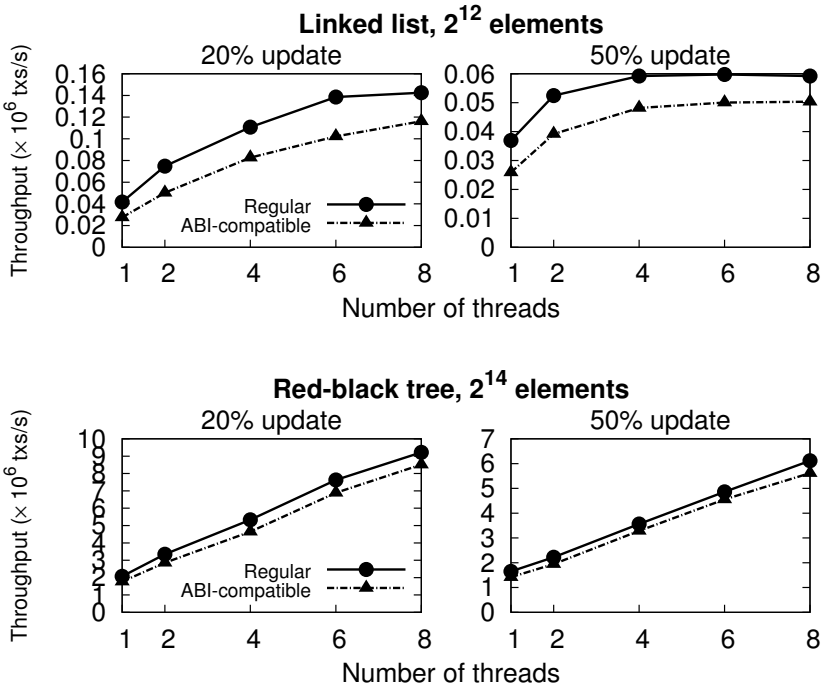


Figure 5.7: Performance comparison between regular loads/stores and enhanced loads/stores.

Figure 5.7 indicates that this overhead is low. The additionnal check, which consists in verifying whether the address is on the stack, could be avoided with specific barriers for aligned memory accesses.

5.5.2 Evaluation of complex memory barriers

As previously explained in Section 5.3.3, loads and stores barriers of the TM-ABI can be optimized for some specific patterns on memory accesses.

	Improvements	Code size	Speed improvement
ITM_RU8	–	486 bytes	–
ITM_RaRU8	Avoid adding again to the read set	286 bytes (41.2%)	77 cycles (8.3%)
ITM_RaWU8	Avoid checking for concurrent write access and direct access to the write set	80 bytes (83.6%)	77 cycles (15.4%)
ITM_RfWU8	Acquire the write lock and no need to add to the read set	884 bytes* (30.2%)	140 cycles* (9.1%)
ITM_WU8	–	780 bytes	–
ITM_WaRU8	None in this case	780 bytes (same)	147 cycles (same)
ITM_WaWU8	Avoid checking for concurrent write access and direct access to write set	104 bytes (86.7%)	77 cycles (15.4%)

Table 5.1: Improvements with TinySTM using optimized memory barriers.
* For the read plus the write operation, compared to regular read plus write.

Indeed, the recognizing of "afterRead", "afterWrite", and "forWrite" access patterns allows the compiler to call special versions of the TM library. The ABI specification defines for many of the variants the interfaces to inform the runtime about the read and write sets, e. g., the `_ITM_RU4` function to add a 4-byte integer to the read set has an optimized `_ITM_RfWU4` companion. The runtime can then make appropriate decisions right away (like early aborting the transaction).

```
extern long cntr;
void increment() {
  __transaction {

    cntr = cntr + 5;

  }
}
```

```
extern long cntr;
void increment() {
  _ITM_beginTransaction(...);

  long l_cntr = (long) _ITM_RfWU8(&cntr);
  l_cntr = l_cntr + 5;
  _ITM_WaWU8(&cntr, l_cntr);
  _ITM_commitTransaction();
}
```

Figure 5.8: An example of how C code with a transaction statement (left) is transformed to targeting a TM library ABI with optimized barriers.

Table 5.1 presents examples of improvements achieved with optimized load and store functions using an experimental TinySTM with early conflict detection and write back strategy on a 64-bit machine. Optimized barriers can reduce up to a 86.7% the code size and improve up to 15.4% the execution speed compare to regular barriers.

5.5.3 Transaction descriptor variants

The TM library requires a transaction descriptor to keep information about the current transaction. Usually, a transaction is bound to a thread since the TM-API does not allow interleaved atomic blocks. There are two possibilities:

- The transaction descriptor is explicitly added into the binary program. This solution has two drawbacks: (1) it is intrusive for the program; (2) it can use one register to keep it;
- The transaction descriptor is implicit. The burden to locate the descriptor is then let to the TM library. This solution has one major drawback; the TM library requires a mechanism to store data in the thread descriptor. The most common solution is to use the pthread library (indirectly via thread local mechanism of the compiler). The best solution is to use the reserved space for TM which is available in the GNU libc since version 2.8.

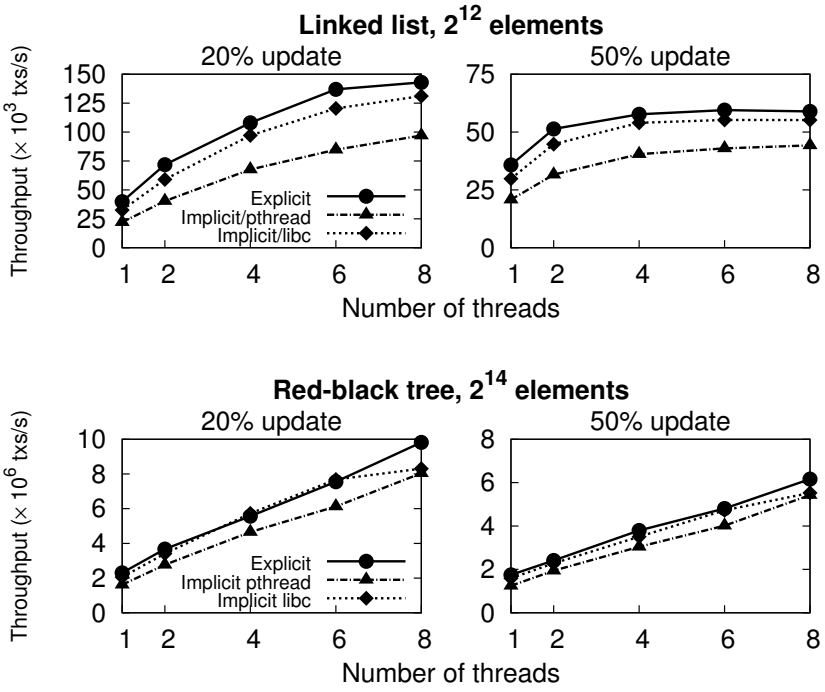


Figure 5.9: Performance comparison for implicit and explicit transaction descriptor.

Figure 5.9 evaluates the different approaches with TinySTM. Integer set benchmarks perform better with explicit transaction descriptor approach than implicit one. However, it is more intrusive since the descriptor is kept inside application binary code. The figure shows that the libc extension is really efficient compared to the regular pthread implementation and close to the explicit approach.

5.5.4 Testing compilers with STAMP benchmarks

To compare the different compilers, we used the same STM library and we tested them using the STAMP benchmarks. We adapted our STM library to the different compilers to match specificities in the TM-ABI, e. g., memory allocations (see Section 5.4.3).

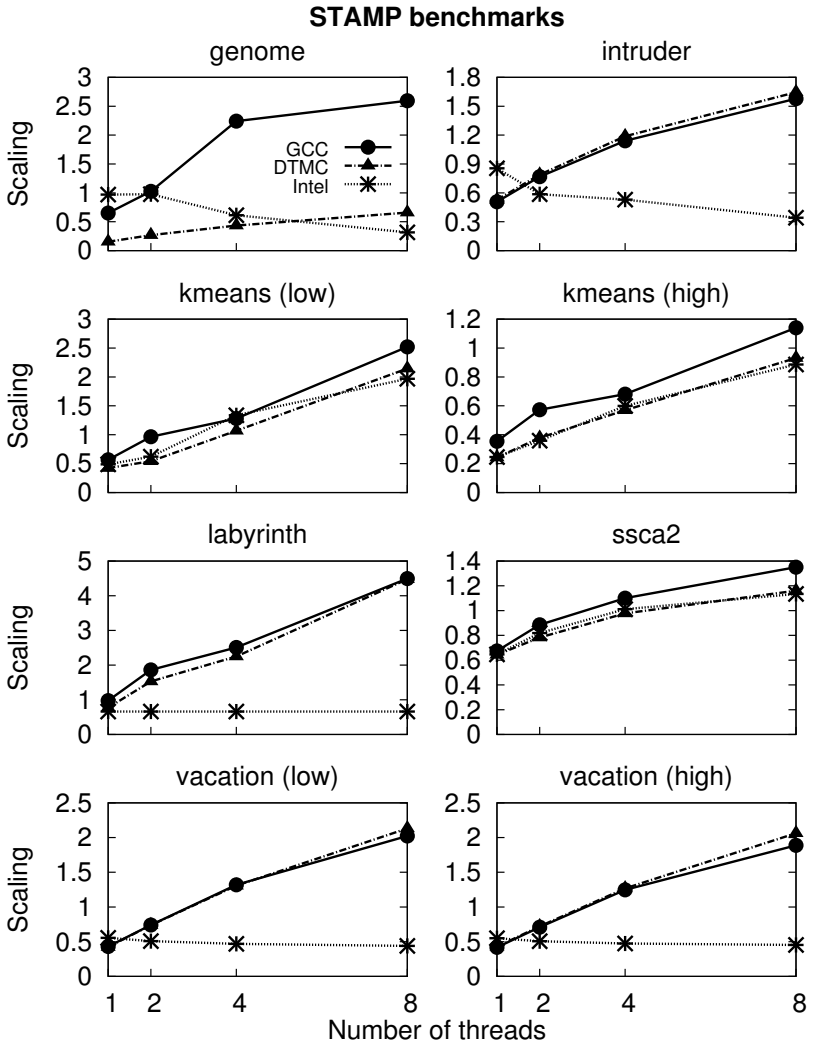


Figure 5.10: Overview of TM scalability with different compilers in selected STAMP benchmarks.

Figure 5.10 shows that the Intel STM Compiler has scalability issue in Genome, Intruder, Labyrinth, and Vacation benchmarks. In these benchmarks, the Intel STM Compiler suffers from the missing support for indirect function calls. It forces the TM library to execute with serial irrevocable mode, which reduces drastically the parallelism of the applications.

In all benchmarks but Genome, DTMC and GCC perform almost at the same speed. It shows that the LTO optimization has a little effect on performance for Software Transactional

Memory. Note that the TM library did not support the optimized memory barriers in the tested version.

GCC uses a shared dynamic library for the TM library whereas the Intel STM Compiler and DTMC uses static dynamic library. The major advantage of the dynamic library is to be able to change the TM library at the runtime without recompiling the application. The shared dynamic library requires a position independent code, which adds some extra overhead in the generated code. We show that this overhead is minor in the overall performance.

5.6 Conclusion

This chapter presented the different layers for a transactional software stack. First, we presented how the transactional memory paradigm can be integrated into the traditional C/C++ programming language. We detailed the high-level API for transactional programming.

Second, we explained how the transactional API is transformed to binary code through the transactional ABI. We explained how the compiler generates transactional code and how the TM library is integrated with it. We adapted our TM runtime to be used with any transactional compiler that complies with the TM ABI specification.

Finally, we addressed some issues with transaction integration. We presented the differences between operational C/C++ compilers with transactions support, the integration of hardware transaction support, the management of external actions, and various optimizations.

That enable practical and efficient transactional programming within an unmanaged environment.

Chapter 6

Conclusion

Multi-core is now ubiquitous in current hardware, from servers to mobile devices. The number of available cores will continue to increase. Multi-core programming has to evolve to become easier. Transactional Memory promises a new paradigm that is both scalable and practical.

In this thesis, we developed new implementations for transactional memory that optimize the software transactional memory, that benefit from hardware support, and that integrate fully all transactional facilities to a software stack.

6.1 Summary of contributions

The contributions of this thesis can be summarized as follows:

Efficient software transactional memory. First, we designed a time based implementation of transactional memory library that scales well with the number of cores. We proposed improvements for performance that allow reducing overheads compared to sequential code. We also extended our TM for usability in order to ease the adoption of transactional memory. The evaluation of our STM library conveyed the scalability and performance of TinySTM on real multi-core CPUs.

Using hardware support for transactions. We presented a hardware support proposal for transactions, ASF, coming from an industry manufacturer. We evaluated it in a transactional context. It proved to be scalable and to have reduced overheads for transactional loads and stores. We also observed that this hardware support has some limitations such as limited load/store capacities. To overcome these limitations, we designed a Hybrid TM that mix hardware and software transactions based on a time-based algorithm. This Hybrid TM retains the promises of HTM but with a parallel fallback solution when hardware transaction cannot be executed.

Integrating transactional memory in a software stack We presented language specifications for transactions in C/C++. We also presented the application binary interface for transactional memory library, which enables a generic interface for all TM libraries. We discussed extensions to a TM library in order to follow the specifications of the language and

of the binary interface. Finally, we described the integration of hardware transactions into a software system stack.

6.2 Perspectives

We conclude by presenting the research and uptake perspectives opened by the work presented in this thesis.

Uptake by the open source community. We have worked on improving software transactional memory for performance and usability. The GNU GCC Compiler is on the way to support transactional memory in C/C++. It is shipped with a basic transactional memory library. Unfortunately, this transactional memory library is quite minimal. We envision that our work could be leveraged to benefit the GCC community.

Validate hardware simulations. Hardware manufacturers are working on processor extensions for advanced concurrency support. No hardware support for transactions is announced for next generation CPUs but difficulties for multi-core programming are real. Whilst internal prototypes are likely, the effectiveness of our Hybrid TM algorithm could be validated on real hardware. This unique opportunity could lead to continue developing new algorithms and systems that would be widely applicable in the computer industry.

Porting transactional memory to managed environment. In this thesis, we targeted the support of C/C++ platforms and we managed to successfully integrate a complete TM stack ranging from applications to hardware support. However, on the Java platform for example, the generated code is meant to execute in a managed environment rather than directly on the physical processor. The thesis contribution on integration can help with the adaption of transactional memory to a managed environment, possibly as a system-level component below the VM instead of an application-level library as proposed so far. The availability of both unmanaged and managed environments has the highest potential for enabling wide adoption of transactional memory.

Appendix A

Publications

W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J.L. Lawall, 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), January 2010.

D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, E. Riviere.
Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack.
In *Proceedings of the 5th ACM SIGOPS EuroSys European Conference on Computer System* (EuroSys 2010), April 2010.

P. Felber, C. Fetzer, P. Marlier, and T. Riegel.
Time-based Software Transactional Memory.
In *IEEE Transactions on Parallel and Distributed Systems* (TPDS), June 2010.

P. Felber, C. Fetzer, P. Marlier, M. Nowack, T. Riegel.
Brief Announcement: **Hybrid Time-Based Transactional Memory.**
In *Proceedings of the 23rd International Symposium on Distributed Computing* (DISC'10) September 2010.

P. Felber, E. Riviere, W. Maldonado, D. Harmanci, P. Marlier, S. Diestelhorst, M. Hohmuth, M. Pohlack, A. Cristal, I. Hur, O. Unsal, P. Stenstrom, A. Dragojevic, R. Guerraoui, M. Kapalka, V. Gramoli, U. Drepper, S. Tomic, Y. Afek, G. Korland, N. Shavit, C. Fetzer, M. Nowack, and T. Riegel.
The Velox Transactional Memory Stack.
In *IEEE Micro* Volume 30 Issue 5, 2010.

W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller and E. Riviere.
Kernel-Assisted Scheduling and Deadline Support for Software Transactional Memory.
In *Proceedings of the Conférence Française en Systèmes d'Exploitation* (CFSE), May 2011.

P. Felber, C. Fetzer, P. Marlier, M. Nowack and T. Riegel.
Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations.
In *Proceedings of the 23rd Annual ACM Symposium on Parallel Algorithms* (SPAA'11), June 2011.

W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller and E. Riviere.
Deadline-Aware Scheduling for Software Transactional Memory.
In *Proceedings of the International Conference on Dependable Systems and Networks* (DSN 2011 DCCS), June 2011.

References

[1] Martin Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, feb 2009.

[2] Advanced Micro Devices, Inc. *Advanced Synchronization Facility - Proposed Architectural Specification*, 2.1 edition, mar 2009.

[3] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 10h and 12h Processors*, 3.13 edition, feb 2011.

[4] Yehuda Afek, Ulrich Drepper, Pascal Felber, Christof Fetzer, Vincent Gramoli, Michael Hohmuth, Etienne Riviere, Per Stenstrom, Osman Unsal, Walther Maldonado Moreira, Derin Harmanci, Patrick Marlier, Stephan Diestelhorst, Martin Pohlack, Adrian Cristal, Ibrahim Hur, Aleksandar Dragojevic, Rachid Guerraoui, Michal Kapalka, Sasa Tomic, Guy Korland, Nir Shavit, Martin Nowack, and Torvald Riegel. The velox transactional memory stack. *IEEE Micro*, 30:76–87, September 2010.

[5] Utku Aydonat and Tarek Abdelrahman. Serializability of transactions in software transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, feb 2008.

[6] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The opentm transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 376–387. IEEE Computer Society, Los Alamitos, CA, USA, Sep 2007.

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick 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] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2000.

[9] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[10] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[11] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11), 2008.

[12] Chris Lattner and Vikram 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.

[13] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne 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, Paris, France, 2010. ACM.

[14] Cliff Click. Azul's experiences with hardware transactional memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, jan 2009.

[15] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael 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.

[16] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP '10: Proc. 15th ACM Symp. on Principles and Practice of Parallel Programming*, New York, NY, USA, jan 2010. ACM.

[17] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, San Jose, CA, USA, 2006.

[18] Dave Dice, Yossi Lev, Mark Moir, and Daniel 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. ACM, mar 2009.

[19] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In Shlomi Dolev, editor, *DISC '06: Proc. 20th International Symposium on Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, sep 2006. Springer-Verlag Lecture Notes in Computer Science volume 4167.

[20] Stephan Diestelhorst and Michael Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods April, 2008*, Apr 2008.

[21] Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. Stretching transactional memory. In *PLDI '09: Proc. 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 155–165, jun 2009.

[22] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[23] Pascal Felber, Christof Fetzer, Patrick Marlier, Martin Nowack, and Torvald Riegel. Brief announcement: hybrid time-based transactional memory. In *Proceedings of the 24th international conference on Distributed computing*, volume 6343 of *DISC'10*, pages 124–126, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based Software Transactional Memory. *IEEE Trans. Parallel Distrib. Syst.*, 21:1793–1807, December 2010.

[25] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, Salt Lake City, UT, USA, feb 2008.

[26] Pascal Felber, Christof Fetzer, Torvald Riegel, Martin Susskraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[27] Vincent Gramoli, Derin Harmanci, and Pascal Felber. Toward a theory of input acceptance for transactional memories. In *Proceedings of the 12th International Conference On Principles Of Distributed Systems (OPODIS'08)*, volume 5401 of *LNCS*, pages 527–533. Springer-Verlag, Dec 2008.

[28] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 258–264, New York, NY, USA, Jul 2005. ACM Press.

[29] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 197–206, Seattle, WA, USA, 1990.

[30] Maurice Herlihy. SXM: C# Software Transactional Memory. Unpublished manuscript, Brown Univ. `http://www.cs.brown.edu/~mph/`, may 2005.

[31] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, USA, May 1993.

[32] Owen S. Hofmann, Christopher J. Rossbach, and Emmett 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. ACM, mar 2009.

[33] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM '06: Proc. 5th International Symposium on Memory Management*, pages 74–83, jun 2006.

[34] Intel. *Intel Transactional Memory Compiler and Runtime Application Binary Interface*. Intel, 1.0.1 edition, Nov 2008.

[35] Intel. *Draft Specification of Transactional Language Constructs for C++*. Intel, IBM, Sun, 1.0 edition, Aug 2009.

[36] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol 2, Instruction Set Reference*, May 2011.

[37] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol 3, System Programming Guide*, May 2011.

[38] Gokcen Kestor, Srdjan Stipic, Osman S. Unsal, Adrián Cristal, and Mateo Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[39] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, Mar 2006. ACM Press.

[40] Yosef Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, Portland, OR, USA, aug 2007.

[41] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[42] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, May 2004. Massachusetts Institute of Technology.

[43] Walther Maldonado, Patrick Marlier, Pascal Felber, Etienne Riviere, Julia L. Lawall, and Gilles Muller. Deadline-aware scheduling for software transactional memory. In *Proceedings of the 41st International Conference on Dependable Systems and Networks*, DSN '11, jun 2011.

[44] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles 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, jan 2010. ACM.

[45] Virendra J. Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, feb 2008.

[46] Alex Matveev, Ori Shalev, and Nir Shavit. Dynamic identification of transactional memory locations. Unpublished Manuscript, Tel-Aviv University, 2007.

[47] Njuguna Njoroge, Jared Casper, Sewook Wee, Yuriy Teslyar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. Atlas: A chip-multiprocessor with transactional memory support. In *Proceedings of the Conference on Design Automation and Test in Europe*. IEEE Computer Society, Apr 2007.

[48] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07: Proc. 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375, sep 2007.

[49] Mathias Payer and Thomas Gross. adaptSTM - an online fine-grained adaptive stm system. Technical report, ETH Zurich, 2010.

[50] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, Washington, DC, USA, Dec 2001.

[51] Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. From causal to z-linearizable transactional memory (brief announcement). In *PODC '07: Proc. 26th ACM symposium on Principles of distributed computing*, pages 340–341, aug 2007.

[52] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, Sep 2006.

[53] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.

[54] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Jun 2007.

[55] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the 23nd ACM symposium on Parallelism in algorithms and architectures*, San Jose, CA, USA, 2011. ACM.

[56] Christopher Rossbach, Owen Hofmann, and Emmett Witchel. Is transactional memory programming actually easier? In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, jan 2010.

[57] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*, pages 187–197, Mar 2006.

[58] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196. IEEE Computer Society, 2006.

[59] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[60] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[61] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug 1995.

[62] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, San Diego, CA, USA, Jun 2007.

[63] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *ICPP '08: Proc. 37th International Conference on Parallel Processing*, sep 2008.

[64] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30, March 2005.

[65] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proc. twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, jun 2008.

[66] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*, Feb 2007.

[67] Matt 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.

[68] Rui Zhang, Zoran Budimlić, and William N. Scherer III. Commit phase in timestamp-based STM. In *SPAA '08: Proc. twentieth annual symposium on Parallelism in algorithms and architectures*, pages 326–335, jun 2008.