Université de Neuchâtel

Faculté des Sciences

Institut d'Informatique

Efficient Memory Management with Hardware Transactional Memory: A Focus on Java Garbage Collectors and C++ Smart Pointers

par

Maria Carpen-Amarie

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. Rachid Guerraoui
École Polytechnique Fédérale de Lausanne, Suisse

Dr. Jean-Pierre Lozi
Université Nice Sophia Antipolis, France

Prof. Nir Shavit
Massachusetts Institute of Technology, États-Unis

Dr. Valerio Schiavoni
Université de Neuchâtel, Suisse

Soutenue le 3 novembre 2017

UNIVERSITÉ DE NEUCHÂTEL

Faculté des Sciences
Secrétariat-décanat de Faculté
Rue Emile-Argand 11
2000 Neuchâtel – Suisse
Tél : + 41 (0)32 718 21 00
E-mail : secretariat.sciences@unine.ch

IMPRIMATUR POUR THESE DE DOCTORAT

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

Madame Maria CARPEN-AMARIE

Titre:

"Efficient Memory Management with Hardware Transactional Memory: A Focus on Java Garbage Collectors and C++ Smart Pointers"

sur le rapport des membres du jury composé comme suit:

- Prof. Pascal Felber, directeur de thèse, Université de Neuchâtel, Suisse
- Dr Valerio Schiavoni, Université de Neuchâtel, Suisse
- Prof. Nir Shavit, MIT, USA
- Prof. Rachid Guerraoui, EPF Lausanne, Suisse
- Dr Jean-Pierre Lozi, Université Nice Sophia Antipolis, France

Neuchâtel, le 7 novembre 2017 Le Doyen, Prof. R. Bshary

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor, Prof. Pascal Felber, for supporting and advising me throughout my work on the PhD thesis. Knowing he is always there to exchange thoughts or give quick feedback every time I needed made this experience so much more enjoyable and insightful. Likewise, I would like to thank the external members of my thesis jury, Prof. Nir Shavit, Prof. Rachid Guerraoui and Dr. Jean-Pierre Lozi for taking the time to review my work and for the exciting discussions on the topic. I'm equally grateful to Dr. Valerio Schiavoni, for being both a friend and a strict reviewer, helping me very much with a cocktail of detailed feedback and good advice. My thanks also go to Prof. Gaël Thomas and David Dice for their constant support and advice.

I am deeply indebted to Dr. Patrick Marlier. Without him this project would have never been started and without his knowledge, patience and enthusiasm during the first year it would have never reached this stage. Thanks, Pat, for a priceless first year of PhD!

During the last four years I had a wonderful time together with all my colleagues and friends at UniNE. I have not forgotten the wonderful moments spent with Anita, Mascha, Emi, Yarco, Heverson, Raziel, before they left the university. Likewise, I'd like to thank the current gang of friends for all the fun coffee breaks, crossword puzzles, boardgames nights and so many other exciting events and debates: Dorian, Rafael, Roberta, Sébastien and everybody else. Of course, special thanks to my terrific officemate Mirco.

Last, but not least, I want to thank all my family. A big hug for my sister Alexandra, who has always been my best friend and a role model for me. Special thanks to my mother, Maria, for always giving me the best possible care. Finally, thank you, Călin, for creating so many moments of happiness and warmth in the last few years.

ABSTRACT

With multi-core systems becoming ubiquitous in the last few years, lightweight synchronization and efficient automatic memory management are more and more demanded in the design of applications and libraries. Memory management techniques try to take advantage of the growing parallelism as much as possible, but the increasingly complex synchronization introduces considerable performance issues.

The goal of this thesis is to leverage hardware transactional memory (HTM) for improving automatic memory management. We address both managed environments (Java) and unmanaged languages (C++). We investigate to what extent a novel synchronization mechanism such as HTM is able to make well-known memory management systems more efficient. We further provide insight into what scenarios benefit most from HTM-based synchronization.

Managed runtime environments generally have a memory reclamation and recycling mechanism integrated, called garbage collector (GC). Its goal is to manage memory automatically and transparently, without impacting the application execution. The Java HotSpot virtual machine features several types of GCs, all of which need to block all application threads, i.e., stop the world, during collection. We propose a novel algorithm for a fully-concurrent GC based on HTM. We implement and optimize transactional access barriers, the core part of the algorithm. Our evaluation suggests that a full implementation of the transactional GC would have comparable performance with Java's low-pause concurrent GC, while also eliminating stop-the-world pauses.

Unmanaged languages use specialized forms of automatic memory management. In this context, we focus on smart pointers in C++. Their logic is based on reference-counting. However, the extra synchronization required by smart pointers has a negative impact on the performance of applications. We propose an HTM-based implementation of smart pointers. We aim to improve their performance by replacing atomic operations on the reference counter with transactions. In case of conflict, the pointer falls back to being a classical smart pointer. We identify favorable scenarios for using transactional pointers and we subsequently specialize them for concurrent data structure traversal.

After studying the downsides of automatic memory management in multiple contexts, we find that HTM is a good candidate for addressing their specific synchronization issues. We show that HTM can successfully support both a feasible pauseless Java GC implementation, as well as more efficient smart pointers in C++.

Keywords: Transactional memory, garbage collection, Java virtual machine, C++, reference counting, data structures.

RÉSUMÉ

De nos jours les systèmes multi-cœurs sont omniprésents. La conception des applications demande de plus en plus un moyen de synchronisation plus léger, ainsi qu'une gestion automatique de la mémoire plus efficace. Les techniques de gestion de la mémoire tâchent de profiter au maximum du parallélisme qui augmente chaque jour. Néanmoins, la complexité de la synchronisation peut entraîner des problèmes de performance majeurs.

Dans cette thèse, nous tirons parti du support matériel pour la mémoire transactionnelle afin d'améliorer la performance de la gestion automatique de la mémoire. Ce travail porte sur des environnements supervisés (Java), ainsi que non-supervisés (C++). Nous étudions la mesure dans laquelle ce nouveau moyen de synchronisation peut rendre des systèmes réputés de gestion de la mémoire plus efficaces. De plus, nous identifions les scénarios où l'on peut utiliser la mémoire transactionnelle matérielle pour maximiser ses bénéfices.

En général, les environnements supervisés ont un mécanisme intégré de récupération et recyclage de la mémoire, appelé le ramasse-miettes. Le but d'un ramasse-miettes est de gérer automatiquement la mémoire sans affecter la performance de l'application. La machine virtuelle Java HotSpot comporte plusieurs types de ramasse-miettes, qui ont tous besoin d'arrêter (ou bloquer) l'application pendant le ramassage. Nous proposons un nouvel algorithme pour un ramasse-miettes entièrement concurrent, employant de la mémoire transactionnelle matérielle pour la synchronisation avec l'application. Nous développons et optimisons une partie fondamentale de notre algorithme, des barrières en écriture et en lecture transactionnelles. Notre évaluation indique qu'après la mise en œuvre complète de notre algorithme, le ramasse-miettes transactionnel aura une performance comparable aux ramasse-miettes existants, mais sans bloquer l'application.

Les langages non-supervisés utilisent parfois des moyens spécialisés de gestion automatique de la mémoire. Dans ce contexte, nous étudions les pointeurs intelligents de C++. La gestion de la mémoire repose sur un algorithme à comptage des références. La performance des applications est parfois dégradée en raison de la synchronisation requise par cet algorithme. Nous proposons une variante des pointeurs intelligents synchronisés avec de la mémoire transactionnelle matérielle. L'idée est de remplacer les opérations atomiques au niveau du compteur des références par des transactions. En cas de conflit, le pointeur transactionnel se transforme en pointeur intelligent classique. Les pointeurs transactionnels se révèlent particulièrement avantageux dans le contexte des structures des données.

Nous analysons les désavantages de la gestion automatique de la mémoire dans plusieurs contextes. Nous montrons que la mémoire transactionnelle matérielle représente une très bonne solution pour les problèmes spécifiques de synchronisation rencontrés par les systèmes de gestion de la mémoire, dans des environnements aussi bien supervisés que non-supervisés.

Mots-clés: Mémoire transactionnelle, ramasse-miettes, machine virtuelle Java, C++, comptage de références, structures des données.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This thesis focuses on ways of improving automatic memory management in Java and C++ with the aid of a novel synchronization technique, hardware transactional memory. In this chapter, we discuss the context of our work, we describe the main contributions and we present the structure of the manuscript.

1.1 Context and motivation

Constant technological advances in computer science in the last few years led to an increased complexity in writing applications that are both correct and efficient. The steady growth in the number of cores per CPU in modern machines makes classical synchronization mechanisms cumbersome and error-prone, marking the need for a more lightweight synchronization option. An elegant solution is offered by the *transactional memory* (TM) paradigm. This synchronization mechanism allows the developer to write code without worrying about concurrent interleavings. The critical sections just need to be encapsulated in "atomic blocks" called *transactions*. These blocks can then execute concurrently in isolation, possibly aborting upon conflict with the speculative execution of another atomic block. Transactional memory can be implemented either in software or in hardware. While software TM mostly remained a research tool due to its considerable overhead, hardware transactional memory (HTM) was included in commodity hardware, starting with the Intel Haswell processor in mid-2013. HTM overcomes STM's drawbacks by providing hardware support for speculative execution of short transactions (size limited by the number of cache lines that can be monitored). It is not suitable for all types

of applications, at least not without a software fallback for long transactions. However, HTM appears to be the perfect solution for tackling specialized concurrency problems that cannot be efficiently solved with locking-based synchronization (also called pessimistic [46]).

Another trend dictated by the occurrence of increasingly complex code is represented by *automatic memory management*. Manually managing memory allocation could sometimes become next to impossible in the context of large multi-threaded applications. The *garbage collector* (GC) is the mechanism responsible for automatic memory reclamation and recycling in managed runtime environments (MRE), such as the Java virtual machine (JVM). Its goal is to identify the chunks of memory that are not used anymore and make them available for further allocation, while disrupting as little as possible the execution of the application. Some unmanaged languages (e.g., C++) also try to facilitate memory management with dedicated data types, such as *smart pointers*. However, a particular downside of garbage collectors and other types of memory management mechanisms is the performance penalty. The overhead (in terms of latency or size) is introduced by the need of synchronization on multi-core systems. This situation presents an opportunity for experimenting with the novel transactional memory mechanism as potential synchronization strategy.

1.2 Contributions

The goal of this thesis is to exploit HTM in the context of automatic memory management systems. We look at both managed and unmanaged systems, with the aim of improving real-life garbage collectors and other memory management structures. We show that HTM is able to provide reliable and efficient synchronization in this context and discuss the challenges that must be addressed in order to reach a suitable performance.

The contributions of this thesis can be summarized as follows:

Extensive study on the performance of Java GCs. The aim of this study over all Java collectors was to identify their major drawbacks, observe their overheads, and decide which GC is most suitable for a transactional implementation. We experimented with all GCs provided by OpenJDK8 HotSpot on a set of benchmarks and a real-life application. The most important observation was that even the most optimized Java GCs, such as ConcurrentMarkSweep (CMS) or GarbageFirst (G1), can cause unacceptably long pauses when confronted with large applications that need significant amounts of memory. This represented our main motivation for devising a GC algorithm that removes the application pauses, while maintaining the same throughput.

HTM-based algorithm for a fully-concurrent Java GC. We designed a novel GC algorithm that exploits HTM, performing parallel collection and a single pass over the object graph. HTM is employed on the application side, protecting mutator load and store accesses from taking place at the same time when the GC moves the accessed object. If the GC and the mutators access the same object, the transaction is aborted and the mutator retries; otherwise, the GC and application threads can work concurrently. We expect aborts to be rare in practice.

We further added two important optimizations to the algorithm. The first one allowed transactional read-barriers only for volatile reads. The second one introduced the concept of *selective transactional barriers*, which defines separate paths depending on the activity

of the GC. More precisely, the application threads adopt a fast-path (without transactional barriers) when the GC is not running, and a concurrent slow-path otherwise. The goal of the optimizations was to reduce transactional accesses as much as possible, while still protecting the application from memory conflicts with the GC.

Feasibility study for a transactional GC. We partially implemented the transactional GC algorithm on top of a real-life Java collector. Specifically, we added the transactional barriers (with their respective optimizations) on top of the Java GC and evaluated what was their overhead. Transactional access barriers represent the core of our algorithm design. Their evaluation alone was sufficient to prove that our GC algorithm was feasible. Preliminary experiments showed that the aforementioned optimizations were necessary for a GC with a suitable throughput. We estimated the performance of a transactional concurrent GC with selective barriers and we found it to be on par with the performance of Java's concurrent low-pause collector.

Transactional C++ smart pointers for concurrent data structures. In addition to our study on Java GCs, we also looked into other automatic memory management systems, more specifically *smart pointers* in C++. This project complements the work on MREs with insights on how to use HTM for optimizing an unmanaged programming framework, namely the C++ standard library.

C++ smart pointers use a *reference counting* mechanism to automatically manage memory. We identified particular situations when the reference counting strategy proves to be costly and unnecessary. Thus, a more lightweight synchronization mechanism could improve the performance of the application. We considered HTM as a good candidate, employing transactions only to guarantee that the pointers are not illegally deallocated by concurrent threads. The transactions are limited in size, thus we expect aborts of any kind to be rare. We further studied which scenarios would benefit most from HTM and transactional pointers. They proved especially effective for faster concurrent data structures traversal.

This work was carried out in collaboration with Prof. Gaël Thomas (Telecom SudParis, France) and with David Dice (senior research scientist at Oracle Labs, USA). The work on the first two contributions was also coordinated by Dr. Patrick Marlier (Cisco, Switzerland), during his post-doctoral research at the University of Neuchâtel. Part of the feasibility study for the transactional Java GC was done in collaboration with Dr. Yaroslav Hayduk, during his PhD studies at the University of Neuchâtel, and with Prof. Christof Fetzer (TU Dresden, Germany).

The contributions of this thesis have been published at the following venues.

Conferences and workshops:

- Maria Carpen-Amarie, Yaroslav Hayduk, Pascal Felber, Christof Fetzer, Gaël Thomas, and Dave Dice. "Towards an Efficient Pauseless Java GC with Selective HTM-Based Access Barriers." In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (ManLang '17), pp. 85-91. ACM, 2017.

- Maria Carpen-Amarie, Dave Dice, Gaël Thomas, and Pascal Felber. "Transactional Pointers: Experiences with HTM-Based Reference Counting in C++." In *Networked Systems. NETYS 2016. Lecture Notes in Computer Science*, vol. 9944, pp. 102-116. Springer, 2016.

- Maria Carpen-Amarie, Dave Dice, Patrick Marlier, Gaël Thomas, and Pascal Felber. "Evaluating HTM for Pauseless Garbage Collectors in Java." In *2015 IEEE Trustcom/ BigDataSE/ ISPA*, vol. 3, pp. 1-8. IEEE, 2015.

- Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. "A Performance Study of Java Garbage Collectors on Multicore Architectures." In *Proceedings of the 6th International Workshop on Programming Models and Applications for Multicores and Manycores* (PMAM '15), pp. 20-29. ACM, 2015.

Technical reports:

- Maria Carpen-Amarie. "Performance Prediction for a Fully-Concurrent Transactional Java GC" In *PhD Workshop Extended Abstracts of the 2017 EBSIS Summer School on Distributed Event Based Systems and Related Topics.* Timmendorfer Strand, Germany, pp.31-32.

- Maria Carpen-Amarie. "Towards Efficient Concurrent Data Structures Traversal with Transactional Smart Pointers" In *PhD Workshop Extended Abstracts of the 2016 EBSIS Summer School on Distributed Event Based Systems and Related Topics.* Sinaia, Romania, pp. 23-24.

- Maria Carpen-Amarie. "Performance Issues in Java GCs: Is HTM the Solution?" In *Technical Report of the 2015 Doctoral Workshop on Distributed Systems.* Le Louverain, Switzerland, pp. 17-20.

- Maria Carpen-Amarie. "TransGC: Concurrent Garbage Collection Using Hardware Transactional Memory." *Euro-TM Short Term Scientific Mission Report.* France, 2014.

- Maria Carpen-Amarie. "TransGC: Concurrent Garbage Collection Using Hardware Transactional Memory." In *Technical Report of the 2014 Doctoral Workshop on Distributed Systems.* Kandersteg, Switzerland, pp. 20-23.

1.3 Organization of the manuscript

This manuscript is organized in four parts.

The first part discusses the background of this thesis. We start by positioning the context of our work with respect to today's systems, demands and existing solutions. We address the two most important concepts combined in this work in Chapter 3 and, respectively, Chapter 4. The former focuses on transactional memory. We study software transactional memory (STM) and what are the associated downsides that motivated the development of hardware transactional memory (HTM). In particular we describe Intel TSX, the main HTM implementation used throughout this work. In the latter chapter we provide some background on automatic memory management. It is split in two parts: one addressing Java GCs, and the other introducing C++ smart pointers.

The second part represents our work on improving Java garbage collectors with HTM. In Chapter 5 we present the literature relevant to this part and argue the novelty and importance of our work in this context. Chapter 6 covers our extensive study on the performance of Java GCs. Then, we define a new algorithm for an HTM-based pauseless GC in Chapter 7. We include here the theoretical aspects of the algorithm, the components needed in the design of such a GC and an informal discussion over the correctness of the algorithm. Its practical implementation is separately addressed in Chapter 8. We describe the existing infrastructure in Java on top of which we (partially) develop our GC and the additions required by the algorithm. A preliminary evaluation indicates the need for a more optimized implementation, which is further provided in Chapter 9. Finally, we extensively evaluate our implementation in Chapter 10. We also conduct a feasibility study, comparing a potential implementation of our GC algorithm with a real-life Java collector.

The third part focuses on C++ smart pointers and our effort towards their improvement with HTM. We define a new type of smart pointer, named transactional pointer, and provide details on the algorithm and implementation in Chapter 11. We also address two basic use cases that justify the usefulness of our approach. We evaluate the implementation of transactional pointers on micro-benchmarks based on these use cases in Chapter 12. Chapter 13 explains how we can specialize the transactional pointers for concurrent data structure traversal and illustrates the required implementation changes. Finally, we evaluate the performance of specialized transactional pointers compared to classical smart pointers in Chapter 14 and discuss the results.

The fourth part summarizes the contributions of the thesis in Chapter 15 and discusses a series of unexplored research challenges revealed by our work in Chapter 16.

Part I

Background

Chapter 2

Context: Multi-core Architectures

Contents

NOWADAYS, multi-core systems are the norm for standalone servers or components of big server farms. They started to be heavily produced for public use in 2001, when scientists considered that CPU clock rate almost reached its limit and not much performance could be gained by increasing it any more [62, 39]. Multi-processor devices evolved since then to the point where today any personal computing device has at least two cores (laptops, smart phones, tablets, etc.). While a machine with a single processor can be efficiently programmed without having in-depth knowledge of its internal architecture, things are different with multi-core machines. Memory architecture, communication between cores, software implementation – these are only a few of the factors that can have a significant impact on the performance of an application in a multi-core context.

A **multi-core system** is composed of multiple **processors**, which are physical (hardware) entities that execute a sequential set of instructions. The group of processors inside a multi-core machine can be either *homogeneous* (all cores are the same) or *heterogeous* (cores are different, e.g., in terms of performance, instruction set, etc.). Processors usually run logical (software) entities called **threads**. The two components (physical and logical) do not directly overlap. Usually, there is a greater number of threads than cores. The execution of threads on a processor is interleaved: when one thread is set aside, another one is scheduled and resumes its own work. A thread can be set aside by a processor for various reasons, such as expired time quantum, long I/O operation, etc. When it is the time for a thread to make progress again, it can be scheduled on a different processor.

Another important aspect of a multi-core architecture is the **interconnect**, i.e., the way in which the processors communicate between themselves and with the memory. There are two

main types of interconnection networks: *symmetric multiprocessing* (SMP) and *non-uniform memory access* (NUMA) [83, Chapter 2]. The former consists of a *bus* that links all the elements. It is the most widely used nowadays, although it presents the risk of being easily overloaded in case of a large number of processors. The latter is composed of multiple nodes, each node connecting multiple processors and a memory. This model is more complex, but also more scalable.

Finally, the processing units exchange data between them over **shared memory**. That means that all entities see the same chunk of memory, write to it and read data from it. The way and the frequency with which the shared memory is accessed has a considerable impact on the overall performance of the system.

2.1 Shared memory

Shared memory is implemented as an array of words, each being 32 or 64 bits in size depending on the platform. Typically, when a thread reads a value from the memory, it provides a one-word address and receives the data associated to that address. Similarly, a write to the memory provides an address and data, the latter being subsequently installed in the shared memory.

However, a memory access is costly and having only a large area of shared memory between multiple cores does not scale well. Therefore, multi-core systems also implement a caching mechanism. The **cache memory** is a smaller memory placed closer to the processor and thus, faster than the main memory. Most of the times, the cache system is implemented on multiple levels: *L1 cache*, separated for data and instructions, both on the same chip as the processor; *L2 cache*, unified for data and instructions, farther away from the processor; and sometimes *L3 cache*, also unified for data and instructions. The data in the cache levels has to be *coherent*, meaning that data has to be uniform across caches of all cores.

The layout of the shared memory plays a critical role in various synchronization mechanisms, such as hardware transactional memory. Generally, the accesses recorded by the transaction are kept in L1 cache (read accesses also in L3 or L2 for some architectures [48, 59]). Thus, in this particular case, the actual size of a transaction for an application is limited by the size of the hardware components. The application logic has to adapt to the given shared memory model of a multi-core system in order to leverage the advantages of a particular synchronization paradigm. Hardware transactional memory is further detailed in Section 3.3.

2.2 Concurrent programming

Since the clock rate of a processing unit reached its performance limits, the next logical step was to increase the number of cores. Ideally, parallelizing a sequential application on `n` cores should improve performance `n` times. However, this is not the case in practice. On the one hand, most applications are not entirely concurrent by nature, their overall performance depending on the parts that cannot be parallelized. On the other hand, computing machines are *asynchronous*, i.e., cores can be interrupted or slowed down by various events that cannot be predicted. In order to have a correct execution, suitable synchronization must be put in place, which might further delay the execution of the program.

The main building block in concurrent applications design is the concept of *critical section*. This refers to a sequence of instructions that can be executed only by one thread at a time,

property called *mutual exclusion*. Most often, mutual exclusion is implemented with *lock* objects that block all threads (except one) at the beginning of the critical section. Typically, a fine-grained locking mechanism implies more parallelism and thus, better performance than a coarse-grained implementation. Two crucial issues that can appear with locking are: *deadlock* – when threads wait for each other's resources to become available and no progress can be made by any of them, and *starvation* – a thread is never able to acquire the lock to the critical section and cannot make any progress. Starting from locks, many other synchronization primitives can be further constructed, e.g., semaphores, monitors.

In order to avoid potential hazards that keep the application from making progress, synchronization strategies turned towards nonblocking algorithms. Informally, the *nonblocking property* guarantees that the delay of a process cannot delay other processes. *Wait-free* and *lock-free* properties are examples of nonblocking progress conditions. The wait-free property guarantees that "any process can complete any operation in a finite number of steps, regardless of the execution speeds on the other processes" [50]. Lock-free algorithms have weaker semantics, guaranteeing that "infinitely often *some* method call [operation] finishes in a finite number of steps" [52]. All wait-free algorithms are also lock-free. Lock-free algorithms can suffer from starvation, but are often more efficient than wait-free algorithms. Generally, designing nonblocking algorithms is complex and error-prone, as all possible thread interleavings and their consequences must be considered.

In this context, while writing correct concurrent applications is essential, designing *efficient* concurrent applications becomes more and more difficult with the increasing complexity of multi-core systems. As such, tools that ease concurrent memory management, such as garbage collectors, and accessible lock-free synchronization tools, like transactional memory, represent a key component of today's concurrent systems landscape.

Chapter 3

Transactional Memory

Contents

TRANSACTIONAL MEMORY is a lock-free synchronization mechanism, typically considered a lightweight alternative to locks. In this chapter we present the main types of transactional memory, both software and hardware. We briefly assess the advantages and disadvantages of using TM in practice. Finally, we describe the TM implementation that is used throughout this work.

3.1 Basics

Ever since multi-core systems started to be heavily used, synchronization methods continued to develop, competing for correctness and efficiency. With the rising number of cores per CPU, classical synchronization (e.g., using locks) is lagging behind and scalable non-blocking synchronization mechanisms are becoming increasingly more important. A notable example in this direction is represented by *transactional memory* (TM).

First defined by Herlihy and Moss [51], TM became a hot research topic in the last few years. TM avoids the locking mechanism of typical synchronization methods by encapsulating groups of instructions in transactions. A *transaction* is defined as a finite group of instructions that are executed by a single thread. Transactions have to be:

- *serializable*: steps from two transactions cannot be interleaved, other threads are not able to see intermediate results at any time;
- *atomic*: either all changes are applied (if the transaction successfully commits), or none of them (if the transaction aborts).

Depending on the underlying implementation, transactional memory can respect either *weak atomicity* or *strong atomicity*, as defined by Blundell et al. [13]. The two models of atomicity indicate the relationship between transactional and non-transactional code. Strong atomicity implies atomic execution with respect to both transactional and non-transactional code, while weak atomicity permits transactions to be interleaved with the rest of the (non-transactional) code. Even though strong atomicity seems to be a more powerful concept than weak atomicity, in practice the former can potentially break code that would function correctly under the latter. It is recommended that a transactional system always specifies the atomicity model as part of its semantics [13].

Typically, a transaction is able to keep track of the changes due to its two associated logs: a *read-set* and a *write-set*. Their purpose is to track all read and write operations, in order to apply them correctly when the transaction commits, or to discard them if the transaction aborts. A transaction aborts if it conflicts with another thread, i.e., if an outside entity tries to modify any of the objects recorded in the read-set or write-set. An abort can also be deliberately triggered by the application.

The concept of transactional memory covers two main implementations: *software* (STM) and *hardware* (HTM). TM was first implemented in software, then hardware implementation followed closely. Both will be detailed in the next sections.

3.2 Software transactional memory

Transactional memory was first implemented in software (STM). Even though the benefits of TM over classical synchronization methods are significant, notably in terms of ease of use, STM was the subject of long debates whether it is only a research toy [20, 37]. Performance-wise, STM could hardly compete with implementations using locks. This happens because the application is heavily instrumented in order to provide transactional semantics. What is more, most STM systems only guarantee weak atomicity. However, a significant advantage of STM is its portability, being able to run on almost any hardware configuration. In addition to this, STM does not make any assumptions regarding the size of the transaction or its contents. The flexibility of STM led researchers to keep on trying to find practical usages for software transactions and study how their performance can be improved.

A notable aspect regarding the overhead reported by various STM implementations was highlighted by Baldassin et al. [10]. They claimed that the memory allocator could play a significant role in degrading STM performance, by increasing the abort rate through false sharing. The authors conducted an extensive study on multiple applications and observed that the performance of an application can vary a lot depending on the memory allocator. They concluded that the allocator used should always be specified in STM performance evaluations.

It was shown that a viable way of taking advantage of software transactions is to customize an STM system for a particular goal. For example, Meier et al. [64] implemented their own STM mechanism with the sole purpose of replacing the global interpreter lock (GIL) in dynamic

languages, in order for them to allow real concurrency. Their TM system detected conflicts with object granularity and provided optimistic reads, pessimistic writes, complete isolation and serializability. The results of the evaluation on a benchmark suite showed an average speedup of 2.5x on 4 threads compared to the original version featuring the GIL. However, the authors also reported an up to 65.5 % overhead on a single thread for STM and less reliable performance gains when the just-in-time compiler of the PyPy interpreter [15] is enabled.

3.3 Hardware transactional memory

Hardware transactional memory (HTM) first appeared in specialized processors, such as Sun's Rock [34], Azul [23] or IBM Blue Gene/Q [84] chips. It was only made available to public in mid-2013, when Intel released their "Haswell" microarchitecture with fully integrated HTM support. Intel Haswell is the first mainstream CPU that implements HTM, creating a greater opportunity for research in this area.

The Intel Haswell processor benefits of an efficient cache coherence protocol to determine transactional collisions. The cache memory for Haswell is generally split into three levels: level-1 (L1) cache and level-2 (L2) cache, private to each core, and level-3 (L3) cache, shared between all cores. The write-set is tracked in the L1 cache, while the read-set is tracked in the L1, L2 and L3 caches [33]. If any entry of a set is evicted from their respective cache, the hardware transaction aborts. Thus, HTM is limited by the size of the transaction. In addition, cache associativity and hyper-threading also play an important role in transaction failure (e.g., hyper-threading further splits the L1 cache capacity in two).

We mentioned in Section 3.2 that memory allocators were shown to have an important impact on the performance of an STM system; Dice et al. [33] revealed that this finding holds true for HTM as well. It was shown that the allocator placement policies can seriously affect the abort rate due to index conflicts. Their general recommendations were: to use *index-aware* allocators [3] and to shift the stores of shared variables to the end of the transaction. The reason for the latter is given by the cache level where loads and stores are tracked: multiple loads before a store could evict its entry from the L1 cache, while the opposite is less likely to happen (since loads are tracked in the next level caches as well).

In the last few years, HTM has become a hot research topic, especially due to its improved performance over STM. It provided low overhead and excellent scalability when exploited in the implementation of in-memory databases [60, 85]. It was also considered less complex to use than a classical locking scheme in this context. Another area of research that successfully experimented with HTM concerns the evolution of dynamic languages, such as Python, Perl, Ruby [72, 81, 68]. The main goal in this case was to eliminate the GIL and allow these languages to benefit from proper concurrency and scalability. Finally, some work focused in the direction of automatic memory management. This topic will be further addressed in Chapter 5.

3.3.1 HTM limitations

As suggested by the results in the literature, HTM overcomes the aforementioned problems of STM. However, it has its own disadvantages: first, the size of the transactions is limited. Careful planning for the contents of the transaction is needed in order to both avoid overflows and amortize the cost of starting and committing the transaction. Moreover, hardware transactions can be aborted at any time by interrupts, faults and other specific instructions, such as debug

or I/O. Therefore, HTM requires a non-transactional fallback path in case of abort, to ensure progress for the transactions that cannot commit.

HTM is still a novel technology for which the practical implications in various scenarios are yet to be discovered. Even though HTM promotes the idea of an incredible ease of use in directly replacing locks, there are situations in which doing so is both wrong and potentially harmful for the application. A relevant example is represented by the manner in which critical sections are handled, as suggested by Blundell et al. [13]. When directly replacing the locking mechanism for a critical section with a transaction, the semantics of the application change: the transaction is atomic with respect to the global state of the execution, while a lock-based critical section would be atomic with respect to the critical sections guarded by the same lock. What is more, if the hardware transaction is not fitted with a fallback path, the direct replacement of locks can cause a deadlock in an otherwise correct program.

A interesting improvement for HTM, suggested by Herath et al. [49], consists in providing a transactional primitive that permits the transaction to abort and continue, instead of abort and retry. According to the authors, such a primitive could improve the performance of an HTM system, resulting in lower contention and fewer false positives.

HTM is therefore a very powerful tool in a multi-core environment, although not appropriate for all types of applications because of the aforementioned limitations. Nonetheless, it appears to be a suitable solution for tackling specialized concurrency problems, such as concurrent memory management.

3.3.2 Case study: Intel TSX

Intel recently added the *transactional synchronization extensions* (TSX) instruction set to its "Haswell" microprocessor architecture. TSX is a group of instructions that allows the atomic manipulation of memory. They provide two mechanisms of synchronization: hardware lock elision (HLE) and restricted transactional memory (RTM).

Hardware lock elision offers backward compatibility with conventional lock-based mechanisms. Applications using HLE can be used on both legacy and new hardware. It provides two instruction prefixes: `XACQUIRE` and `XRELEASE`. They are perceived as the boundaries of a transactional block on architectures that support TSX, and ignored on older systems, which use classical locks instead. If HTM is supported, the hardware will start a transaction at the `XACQUIRE` instruction instead of taking a lock. If the transaction aborts, then the system automatically returns to using locks. In a nutshell, the HLE interface is particularly convenient for heterogeneous clusters, where only a part of the machines might have HTM support.

Restricted transactional memory implements HTM with a new instruction set, not available on legacy computers. It allows more flexibility in defining transactional regions; however, it always needs a fallback path in case the transactional execution is not successful. In this work we rely on RTM for all transactional implementations. RTM defines the following instructions: `XBEGIN`, `XEND`, `XABORT` and `XTEST`. The first two indicate the start and the end of a transactional region. When `XEND` is called, the transaction commits all the changes to the memory. If the transaction is aborted, either by the interaction with another thread or by the explicit `XABORT` instruction, the execution jumps to a fallback handler specified when calling `XBEGIN`. The reason for the abort is saved in the `EAX` register. This allows developers to handle different types

of aborts and to switch to another synchronization strategy if necessary. Finally, the `XTEST` instruction indicates whether the processor is executing a transactional region or not.

Both HLE and RTM instructions are fully supported in several compilers, notably gcc, LLVM/clang and Intel's and Microsoft's C++ compilers via built-in functions. The Linux kernel supports specific performance counters for TSX in order to easily evaluate and diagnose transaction behavior.

3.4 Summary

In this chapter we introduced the concept of transactional memory, with its two major implementations: software and hardware. We detailed the advantages and downsides for both STM and HTM. Finally, we focused on the HTM instruction set further used in this work, defined by the Intel Haswell microprocessor architecture.

Chapter 4

Automatic Memory Management

Contents

As software tries to keep pace with the rapid increase in concurrency in today's multi-core systems, manual memory handling becomes more and more complex. This chapter presents automatic memory management alternatives that are often employed in large scale software systems. We describe garbage collection in Java, with its most representative implementations, and another form of memory management in C++, namely smart pointers.

4.1 Garbage collection in Java

The **garbage collector** (GC) is a fundamental component of any managed runtime environment (MRE), such as the Java Virtual Machine (JVM). The goal of the GC is to automatically manage the memory dynamically allocated by an application. Garbage collection should respect the following important principles [56]:

- it must be *safe*: the GC is not allowed to reclaim live data;
- it must be *comprehensive*: the GC is not allowed to leave uncollected garbage in the heap.

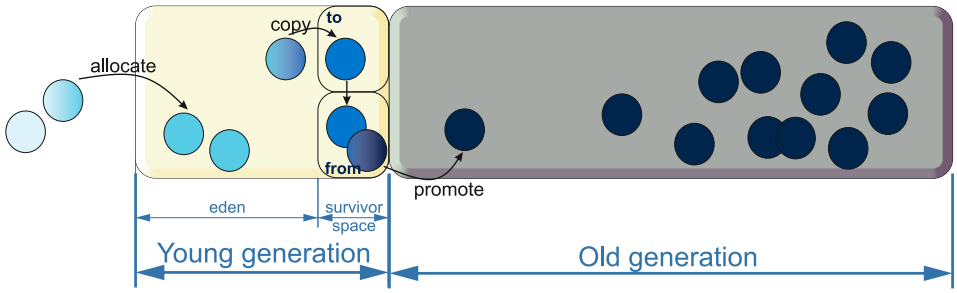


Figure 4.1 – Java GC heap: structure and promotion across generations.

The overhead of a garbage collector is defined as the percent of the total execution time the application spends collecting. Other factors that describe the performance of a GC are: the duration of the GC pauses, the memory overhead, the impact on the throughput and responsiveness of an application. A GC pause (also called *stop-the-world* pause) is defined as the interval of time for which the application threads are suspended during the collection. This blocking phase can have the same length as the whole collection or as a part of it. Stop-the-world pauses are implemented with a *safepoint* mechanism. Java uses safepoints for all VM operations that cannot be executed concurrently with the application threads (also called *mutators*). Some of these VM operations, and arguably the most frequent ones, are related to the GC. Other blocking VM operations handle less costly tasks such as code deoptimization, code cache flushing, debug operations, etc.

4.1.1 Basics

HotSpot JVM is one of the most widely used MREs. In the current version (8.0), it features seven different GC algorithms, which can be characterized along multiple axes:

Generational. The *generational GC strategy* dictates splitting the heap into two or more separate regions (called *generations*) according to the age of the objects. The generations are collected at different frequencies. The *young generation* is collected most often, as recommended by the generational hypothesis which states that *most objects die young*. Older generations are collected less often (or not at all). Young generation collections are called *minor collections*, while old generation collections are also called *major* or *full collections*. The number and repartition of the generations depends on each MRE. Garbage collectors in Java HotSpot divide the heap in three main spaces: the young generation, the old generation and the permanent generation. New objects are generally allocated in the young generation. The young generation is further split into three parts: *eden*, *from-space* and *to-space* (also called *survivor spaces*). All objects are allocated in the eden space, while the survivor spaces start out empty. When eden fills up, a minor GC takes place and moves live objects to the first survivor space (from-space), while the other objects are discarded. At the next minor GC, the second survivor space is used for the live objects. The first survivor space is emptied and all objects are copied to the second one. For all subsequent young generation collections, the two survivor spaces just switch places, so that after each collection one of them and eden are cleared. The objects that survive a few minor collection are *promoted* to the old generation. This process continues until the old generation fills up as well. At that point, a major collection takes place and the same process repeats from

the beginning. These operations are illustrated in Figure 4.1. Finally, the permanent generation consists of Java class metadata. It was permanently removed starting with Java 8.

Tracing. All HotSpot GCs are tracing collectors. They initiate a collection whenever the allocation of new objects fails in order to regenerate the live objects set and discard the garbage. They identify all reachable objects by starting from the roots and following pointers. Unreachable objects are considered garbage and made available for recycling. The *roots* are locations that hold references to heap data and that the application can manipulate directly: registers, stack variables, global variables. Moreover, most of the HotSpot garbage collectors are *copying* (also *moving* or *compacting*) collectors. They move the live objects during a collection in order to avoid memory fragmentation.

Stop-the-world vs. concurrent. Most HotSpot GCs are parallel and span multiple threads to collect the memory. However, some of them perform at the same time (concurrently) with the mutators, others suspend the mutators for the entire duration of the collection (stop the world). All concurrent collectors also have short phases during which they need to suspend the mutators. As such, Java HotSpot does not feature any *fully concurrent* GC. Stop-the-world GCs provide a good application throughput, but usually suffer from significantly long GC pauses. Concurrent GCs, on the contrary, aim at very short pauses at the expense of throughput and simplicity.

HotSpot JVM defines the following GC algorithms:

- **Serial:** stop-the-world, copying collector that uses a single thread for collection;
- **ParallelScavenge:** stop-the-world copying collector that uses multiple threads for collection (i.e., parallel collector);
- **ConcurrentMarkSweep (CMS):** low-pause, parallel, mostly-concurrent collector. It is not a compacting collector;
- **ParallelNew (ParNew):** a version of ParallelScavenge, especially modified to work together with ConcurrentMarkSweep. It provides the necessary synchronization mechanisms to run safely during the concurrent phases of CMS;
- **SerialOld:** single-threaded, stop-the-world, mark-sweep-compact collector;
- **ParallelOld:** multi-threaded, stop-the-world, compacting collector;
- **GarbageFirst (G1):** low-pause, parallel, mostly-concurrent, compacting collector.

Some of these algorithms are mainly used for collecting the young generation, while others are especially tailored for the old generation, or cover both (in G1's case). The seven GC algorithms can be combined in six ways to provide the following generational GC options that can be selected for HotSpot:

- **SerialGC:** uses Serial for the young generation and SerialOld for the old generation;
- **ParNewGC:** uses ParNew for the young generation and SerialOld for the old generation;

- **ConcMarkSweepGC:** uses ParNew for the young generation and CMS for the old generation. In case the full collection cannot proceed concurrently, the collector falls back to SerialOld instead of CMS;
- **ParallelGC:** uses ParallelScavenge for the young generation and SerialOld for the old generation;
- **ParallelOldGC:** uses ParallelScavenge for the young generation and ParallelOld for the old generation;
- **G1GC:** uses G1 for both generations.

In this work, we mostly focus on three out of the six GC combinations of HotSpot JVM, namely *ConcMarkSweepGC*, *ParallelOldGC* and *G1GC*. Given the growing demand of multi-core systems and applications, we considered the GCs that involved serial algorithms less relevant. Moreover, the three chosen GCs present interesting characteristics from different classes of algorithms, each with their specific way of execution and advantages.

4.1.2 ParallelOld GC

The combination of ParallelScavenge and ParallelOld algorithms for the young and, respectively, the old generation constitutes the default garbage collector of OpenJDK8. It is a stop-the-world, throughput-oriented GC. In order to achieve a good performance, it provides several important aspects of GCs: it is a generational, copying and compacting collector. Since the GC stops the world for any collection, these aspects are easily integrated in its mechanism.

ParallelOld GC works as follows: whenever allocation fails in the young generation, a VM operation is scheduled on the unique VM thread. As such, the VM thread starts a safepoint and suspends all mutators, allowing GC threads to collect without any interaction. At the beginning of any young generation collection, the eden contains the objects allocated since the last collection, to-space is empty, while from-space contains objects that survived the previous collection. Starting with this configuration, the young generation collection copies all live objects from eden to to-space and promotes the objects in from-space to the old generation. The copying process starts with root scanning, then proceeds to do a *breadth-first search* (BFS) on the reference graph, from the roots. The GC threads balance their work based on a work-stealing algorithm. After this, eden and from-space will only contain garbage, and the two survivor spaces are swapped. When the GC threads finish the parallel collecting phase, a termination protocol is run, before the mutators can resume work [42].

In order for the parallel GC threads to synchronize correctly when moving objects, they use a *Brooks-style forwarding pointer* [18]. When an object is copied to a new location, the copy at that location becomes the *primary copy* of the object. A forwarding pointer normally points to the primary copy of an object (see Figure 4.2). If the object is not forwarded, then the pointer is self-referential, i.e., it points to itself. The forwarding pointer leads other threads to the primary copy of an object when they still have a reference to the old copy. This approach allows for lazy updating of the references that point to the old version of a moved object.

If the allocation fails during a young generation collection, or a large object cannot be fit in the old generation by the mutators, a full collection is requested. The old generation collection runs a parallel two-phase mark-compact algorithm. As the name indicates, during the first phase the GC threads mark in parallel the live objects in the old generation starting from the roots;

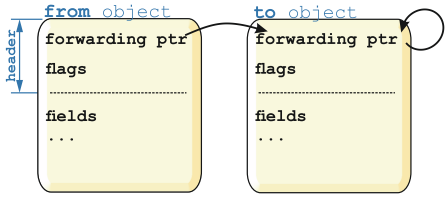


Figure 4.2 – Brooks forwarding pointers.

the second phase represents the parallel compaction, where live objects are placed in the spaces previously occupied by dead objects.

According to Gidra et al. [42], ParallelOld GC suffers from two major bottlenecks: **lack of NUMA-awareness** and **a contended lock during the parallel work of the GC threads**. The former refers to NUMA architectures, where the objects may be preferentially allocated on a single node that gets overloaded. Another significant downside of the ParallelOld collector (not addressed in the aforementioned work) concerns **GC pauses** that can reach prohibitively long durations.

4.1.3 ConcurrentMarkSweep GC

HotSpot's concurrent collector aims at low GC pauses, by letting the application threads work at the same time as the GC during old generation collections. For young generation collections, it uses the same stop-the-world copying algorithm as ParallelOld GC (See Section 4.1.2).

The old generation collector, CMS [31], executes four major distinct phases:

- **Initial mark.** This is a stop-the-world phase. The GC needs to suspend all application threads in order to reliably scan their stacks for object references. Basically, this is the part where the roots are found and scanned;
- **Concurrent mark.** The collector allows the application threads to resume their work while it concurrently scans the rest of the reference graph starting from the roots it found in the first phase. However, since the mutators are running at the same time, they can alter the memory while it is being collected. Thus, the live objects set discovered during this mark phase is not final. There might be undiscovered live objects at the end of this phase;
- **Remark.** This is a stop-the-world phase. The collector suspends all mutators in order to identify the last live objects that were left unscanned after the concurrent phase. This phase correctly concludes the marking of live objects;
- **Concurrent sweep.** After all the objects are adequately marked, the GC identifies dead objects and declares them as free memory (by adding them to a *free-list*). This is done concurrently with the application threads.

CMS evolved from another mostly-concurrent version proposed by Boehm et al. [14] for unmanaged languages, such as C/C++. Similar to the latter, the marking algorithm has to respect the *tri-color invariant*. As mentioned before, the marking starts from the roots of the

reference graph and follows the links until all live objects have been marked. The objects in the heap can have one of the following three colors: **white** – nodes that have not been visited yet and, at the end of the marking phase, are considered garbage; **black** – nodes that have been processed and their entire subgraph of references has been visited as well, the GC will not visit them again; **gray** – in the process of being visited by the collector, either their immediate descendants have not been processed yet or the rest of the graph has been altered by a concurrent mutator. The invariant states that there should be no direct link from a black object to a white one. At the end of a collection all reachable nodes are black, while all white nodes are garbage and can be reclaimed.

There are two main ways in which an object's *mark* can be implemented: it can either *be part of the header of the object* (i.e., of the forwarding pointer as presented in Section 4.1.2), or *use a bitmap*. CMS adopts the latter option, in order to prevent the interaction with the mutators over the header of objects when running concurrently. More precisely, CMS uses an array of mark bits that correspond to every four-byte word in the heap [31].

CMS defines a set of threads especially dedicated for collections. These particular threads are intentionally not labeled as GC threads, so that they get suspended together with the mutators during young generation collection. They are often called *background threads*. This design has several benefits: it increases the concurrency on some systems, it avoids slowing down the young generation collector and minimizes the synchronization effort.

Finally, it is important to note that the concurrent mode for CMS may fail. This happens if the old generation is filling up with live data faster than it is collected by the concurrent collector. Fragmentation is another factor that could cause this failure. In this case, CMS falls back to a mark-sweep-compact stop-the-world algorithm. Given the fact that the mutators are suspended during the collection and compaction of a whole heap, these failures are considerably expensive.

4.1.4 GarbageFirst GC

G1 GC is the newest garbage collector included in HotSpot JVM. G1 is logically classified as *generational*, while practically its algorithm only has a few characteristics in common with classical generational GCs. The main goal of G1 GC is to provide very low pause times, while maintaining high throughput, for large multi-core systems with large heaps and allocation rates [30]. In order to achieve this, it splits the whole heap into regions of equal size. At any point in time there exists a *current allocation region* from which memory is being allocated. When there is no more space in that region, the allocation continues with the next region. Empty allocation regions are kept in a linked list, in order to provide constant time allocation.

G1 GC is mostly concurrent. In order to identify garbage, G1 applies a concurrent marking algorithm, very similar to CMS. The algorithm has a stop-the-world **initial marking** phase, a **concurrent marking** phase, and a stop-the-world **final marking** phase. The marking phases are followed by a concurrent **live data counting** phase and a stop-the-world **cleanup** phase. The former identifies the amount of live data that was previously marked. The counting process is finished in the cleanup phase, which also completes the marking, sorts the heap regions according to their efficiency and reclaims empty regions. Additionally, G1 stops the world during so-called *evacuation pauses*, when it performs compaction. The evacuation pauses are planned ahead and applied only to a chosen subset of heap regions. Regions are selected for future evacuation collections based on the result of the aforementioned sorting, done according to

specific heuristics regarding the time to collect and the free space left when collecting. The information on live data in each region is processed, and all live objects moved (evacuated) to other regions. Thus the collected regions are now considered empty.

Given the layout of the G1 heap, it cannot be considered a truly generational GC. However, it mimics the behavior of a generational GC by designating some of the regions as *young*. This adds the region to the next collection set. Further, there are two types of evacuation pauses with respect to the generational aspect: *fully young* or *partially young*. The former builds collection sets containing only regions marked as young, while the latter may add non-young regions if the requested pause time allows. As an optimization, the stop-the-world marking phases are often piggy-backed on a young evacuation pause.

4.2 C++ smart pointers

Automatic memory management is split into two representative approaches: *reference counting* and *tracing*. Tracing algorithms are most often used in high performance settings (see Section 4.1), while the reference counting strategy is usually avoided due to its major drawbacks. A critical downside is represented by its considerable overhead over tracing, estimated at 30 % on average [76]. The concept of reference counting is simple: it keeps track of the number of references for each object, updating a counter when references are removed or new ones are added. The object is destroyed only when the count reaches zero. However, this means that each pointer mutation has to be tracked and intercepted, making a naive implementation of reference counting very expensive. Recently, reference counting techniques were reconsidered and optimized, becoming comparable to tracing in terms of performance [76, 12, 77]. A noteworthy memory management mechanism that depends on reference counting is illustrated by C++ smart pointers.

A **smart pointer** is an abstract data type that encapsulates a pointer while providing additional features, such as automatic memory management or bounds checking. These features were added to classical pointers in order to reduce programming bugs (created by manually managing the memory), while keeping the same efficiency. Smart pointers can prevent most memory leaks and dangling pointers.

4.2.1 Types of smart pointers

There are three major types of smart pointers in C++, all of them dedicated to automatic memory management of dynamically allocated objects. Even if they slightly differ, all of them manage the lifetime of objects in a way that guarantees object destruction at an appropriate moment, so as to avoid memory hazards. According to how the ownership of the object is handled, C++ smart pointers are divided in the following categories [65]:

std::unique_ptr provides *exclusive ownership* semantics. They have the same size as a raw pointer and support the same instructions. A `std::unique_ptr` always owns the object it points to. It defines two separate forms for individual objects and arrays, in order to avoid any ambiguity when accessing the entity it points to. Whenever an `std::unique_ptr` is moved, the destination pointer acquires full ownership of the object while the initial pointer becomes `null`. Because of the exclusive ownership property, a smart pointer of this kind cannot be

copied. When the smart pointer is destroyed, it automatically destroys the resource it owns. The destructor usually uses `delete` to eliminate the object, but `std::unique_ptr` also supports *custom deleters*, i.e. user-defined functions that are called when the resource is about to be destroyed. The custom deleter is indicated in the constructor of the smart pointer. Either way, the `std::unique_ptr` ensures the safe removal of the object exactly once along every path through the program (including the case of exceptions). However, a downside of using custom deleters is the increase in size of the `std::unique_ptr`, which can thus become significantly large.

std::shared_ptr provides *shared ownership* semantics. That is, multiple owners have to synchronize to ensure a safe and suitable destruction at the right moment. `std::shared_ptr` handles that by keeping a reference count for the owned object. When the resource is no longer of use, the last smart pointer that stops pointing to it will also destroy the object. An important feature of C++ smart pointers is that an `std::unique_ptr` can be easily and efficiently converted into a `std::shared_ptr`, in case the logic of the application demands shared ownership for a specific resource. More details regarding the internals of the `std::shared_ptr` will be provided in Section 4.2.2.

std::weak_ptr extends `std::shared_ptr` with some extra features. More precisely, it is very similar to a `std::shared_ptr`, but it does not participate in any way to the shared ownership or modify the reference count of the shared resource. This type of smart pointers cannot be dereferenced or tested if they contain a `null` value. However, they are allowed to *dangle* and this can be directly tested from the `std::weak_ptr` API. `std::weak_ptr`s are always created from `std::shared_ptr`s and share the same information on the resource. One of the most relevant application of this kind of smart pointer is to prevent `std::shared_ptr`s cycles. For example, in a cycle of two `std::shared_ptr`s they both have a reference count of at least 1, since they both point to each other. That could easily result in a memory leak when there are no more references towards this cycle of pointers. However, if a `std::weak_ptr` would be used for one of the edges, then nothing would stop the `std::shared_ptr` to destroy the resource when it is not needed anymore and the `std::weak_ptr` to detect that it is dangling.

4.2.2 Case study: `std::shared_ptr`

In this thesis we focus on `std::shared_ptr` in particular. The goal of the smart pointers of this kind is to combine automatic memory management with efficient access and execution times. They implement a type of *reference counting* garbage collection. More precisely, they keep track of the number of references of an object with the help of a shared counter. The shared reference counter is incremented every time a new `std::shared_ptr` to the object is created, and decremented when a reference is destroyed. When the counter reaches 0, the object is destroyed along with the last `std::shared_ptr` pointing to it. As mentioned before, no specific `std::shared_ptr` owns the object. All the smart pointers that point at it collaborate to safely maintain an accurate reference count.

A `std::shared_ptr` contains two pointers: the raw pointer to the resource it encapsulates and a raw pointer towards a *control block* (Figure 4.3). That makes the `std::shared_ptr` twice as big as a raw pointer or a `std::unique_ptr`. However, as opposed to the latter, the size is constant with respect to custom deleters, i.e., the size of a custom deleter would not influence

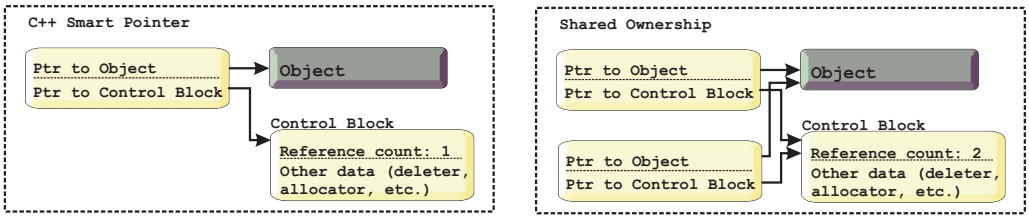


Figure 4.3 – C++ smart pointer components and reference counting mechanism.

in any way the `std::shared_ptr` size. All additional information needed to correctly manage memory (reference count, custom deleters and/or allocators, weak reference count, etc.) is stored in the control block. The control block is created at the same time as the first `std::shared_ptr` to a specific object. All subsequent `std::shared_ptr`s are linked to the same control block.

As in the case of traditional garbage collectors, automatic memory management pays the price in terms of performance. In particular, `std::shared_ptr`s have the following downsides:

- **Size.** Compared to raw pointers `std::shared_ptr`s are twice as big. In addition to this, there is also the variable size of the control block (one per shared resource), which is allocated separately;
- **Synchronization for reference counting.** One of the most import features of smart pointers is that they correctly manage shared resources in multi-threaded environments. That means that multiple threads might want to modify the reference count of an object at the same time. Thus, their accesses need to be synchronized in order to have consistent results. In the GNU C Compiler 5 (gcc5) implementation of `std::shared_ptr`s, there are two ways of implementing the synchronization for incrementing and decrementing the counter in a thread-safe manner. One of them employs a **mutex** and safely alters the counter in a critical area. The second one, which is also the default, uses **atomic operations**. Atomic operations are faster than locks, but still slower than non-atomic operations. Thus, reading and writing the reference count has a non-trivial cost;
- **No cycles.** Conceptually, `std::shared_ptr`s do not allow for a correct implementation of cyclic structures. If two `std::shared_ptr`s point to each other, then their reference counts will never decrease below 1. That means they will never be destructed and the memory they point to will be leaked (there would be no way to access it, but it would not be freed either);
- **Shared resource cannot be an array.** Unlike `std::unique_ptr`, `std::shared_ptr`'s API is only designed for pointers to single objects, offering no `operator[]`.

While having some notable disadvantages, `std::shared_ptr`s still have a reasonable cost for the functionality they provide, necessary in a variety of scenarios. If default deleters and allocators are used, then the size of the control block is of around three words. The atomic operations on the shared reference counter usually map on hardware instructions and have an acceptable overhead in general. Moreover, dereferencing a `std::shared_ptr` has the same cost as dereferencing a normal raw pointer.

4.3 Summary

This chapter presented some background on automatic memory management, from the perspective of both a managed runtime environment, Java VM, as well as an unmanaged language, C++. On the one hand, we considered general properties of Java garbage collectors and briefly described the seven GCs currently implemented in HotSpot JVM. Further on, we detailed the three GCs that are most used in this work (ParallelOld, ConcurrentMarkSweep and G1). On the other hand, we introduced C++ smart pointers, as tools for automatic memory management in an otherwise unmanaged environment. We focused on the `std::shared_ptr` implementation in particular, which represents a type of memory management based on reference counting.

Part II

Exploiting HTM for Concurrent Java GC

Chapter 5

Garbage Collection: Related Work

Contents

Garbage collectors' performance has been an important topic ever since the managed runtime environments started to be heavily used both in research, as well as in production. The topic became hot once the GCs had to be adapted for multi-core systems and the algorithms turned challenging. With the growth in the number of cores of today's computing machines, a simple serial collector is not suitable any more. However, because of the synchronization needed when moving objects, the current GCs are not fully concurrent either. Thus, lots of studies and improvements have been proposed in this area in the last few years.

Most attempts to improve GC algorithms focus on optimizing their concurrency and reduce their pauses. In this category, we find: **concurrent reference counting algorithms**, such as the one defined by Brandt et al. [17] – which reports very low pauses, but an increased memory overhead; as well as **concurrent tracing algorithms** – best represented by ConcurrentMarkSweep GC [31]. In addition to CMS, the Sapphire GC [53] aims to reduce the GC pauses required by a concurrent copying algorithm, by avoiding to stop the world when adjusting thread stacks to account for copied objects. However, the algorithm is only a proof-of-concept, never being thoroughly tested. Further on, Azul systems devised the C4 collector [82], another concurrent parallel compacting GC. The novelty here is that it performs concurrent collection in all generations. C4 is further improved with a non-blocking lock-free work-sharing algorithm [54]. Finally, a non-proprietary GC for Java, Shenandoah, describes a region-based low-pause parallel and concurrent compacting algorithm for large heaps [40]. One important drawback is that it requires more memory space than other algorithms. While it shows promising results during preliminary evaluation, the presented implementation is only incipient.

Another category of GC optimizations addresses only specific parts of collectors. For example, Kliot et al. [58] devise an algorithm for *lock-free concurrent stack scanning*, thus improving one of the last parts that required concurrent collectors to stop the world for a correct execution. Sagonas and Wilhelmsson [74] aim to improve the classical mark-sweep algorithm by entirely eliminating the sweep phase in their *mark-split* collector. Choi et al. [22] defined a *biased allocator* for generational GCs, that allocates long-lived objects directly in the old generation, thus decreasing GC pause time with 4.1 % on average.

There is also work that attempts to improve GC algorithms with respect to a specific type of application. The Yak GC is developped especially for collecting Big Data applications [66]. Yak achieves high throughput and low latency by dividing the heap into separate spaces for control and data, and using different algorithms for the two. Another example is the *data structure aware* GC presented by Cohen et al. [24]. In a nutshell, they improve collection times for programs that rely on large long-lived data structures, such as database servers. They propose allocating the nodes of the data structure separately from the rest of the heap and consider for collection only those that are explictly marked as deleted by the application. The authors claim that these changes require only a handful of modifications in the code, but result in significant improvement for the considered applications.

Finally, on top of the aforementioned optimizations, real-time GCs and TM-based collectors are making efforts to reduce the GC pauses even more, or to eliminate them entirely. These two categories are presented in detail in the next sections.

5.1 Real-time garbage collectors

A domain that truly requires fast and uninterrupted garbage collection is represented by *real-time systems*. In this case, the implementation is less focused on improving the throughput and more on decreasing as much as possible the GC pauses. Thus, the work conducted in this area is relevant to our context when reasoning about ways to eliminate GC pauses. The classes of GCs presented in this section are specially implemented for specific devices, such as embedded systems. They do not always address the same requirements as general-purpose Java VM GCs.

Integrating automatic memory management in real-time systems is notoriously difficult. Usually, the GC is at fault, due to its unpredictable stop-the-world pauses and various barriers that are needed for synchronization. Most of the times, real-time collectors are restricted to uniprocessors [57, 7, 5, 6], or rely on special hardware [8]. The first concurrent compacting real-time GC for multi-core architectures, called Stopless, was proposed by Pizlo et al. [69]. It is implemented on top of C#. Further on, the authors also addressed Java GCs in the real-time context and provided Schism, a *fragmentation-tolerant* concurrent GC for multi-processor systems [71, 70]. Schism achieves hard real-time bounds together with good throughput by using a combination of a concurrent mark-region GC for fixed-size objects and a replication-copying GC to manage array metadata. Another interesting real-time GC for Java is implemented by Siebert [79]. It is a concurrent and parallel mark and sweep collector that only handles fixed-size blocks in memory. It is not a compacting GC. However, on small-sized heaps it is shown to have good scalability and little overhead over the single-threaded implementation.

Even if not especially dedicated to real-time systems, G1 GC [30] achieves real-time goals with high probability on large heaps and large workloads. It partitions the heap in fixed-size regions and any set of regions can be chosen for collection in order to ensure the time requirements.

5.2 TM-based garbage collectors

An early effort in implementing garbage collectors with transactional memory was made by McGachey et al. [63]. They used STM to implement a new concurrent GC designed to enforce an under 1 ms pause for 90 % of collections, under 10 ms for 90 % of the rest and under 100 ms for what is left. They successfully reached their goals, thus making a first step towards combining TM with GC.

Once HTM has been supported in some specialized processors and large systems (e.g., Azul, Sun), various proposals have studied the potential benefits of HTM in concurrent scenarios [34, 38]. A particularly interesting work in this direction has been proposed by Iyengar et al. [55], who implemented an HTM-based Java GC on a specialized multi-core CPU build by Azul systems. The Collie algorithm represents a first contribution in this field. Its evaluation shows that, thanks to the use of the HTM, Collie has a better responsiveness and throughput than C4 [82], their baseline copying and concurrent collector without HTM. However, the paper does not compare Collie with a throughput-oriented collector such as ParallelOld in HotSpot, and it is thus difficult to know whether using HTM in a concurrent collector can help to achieve high throughput.

In 2013, Intel made available their HTM implementation to the public in the Haswell CPU. This created a widely available platform for studying and applying HTM to concurrency problems. Alistarh et al. [4] leveraged HTM for adding automatic memory reclamation for typical C/C++ data structures, such as hash table, linked list, non-blocking queue, etc. According to the evaluation, the performance of their algorithm, called StackTrack, is superior to other memory reclamation mechanisms. However, it can also degrade the throughput of the data structure by up to 50 %.

Ritson et al. [73] conducted a performance study on three main GC algorithms of the Jikes research virtual machine (RVM), implemented with HTM. They assess the benefits of using transactions for parallel semispace copying collection, concurrent replicating GC and parallel bitmap marking. They find that HTM significantly improves the first algorithm (by 48 % to 101 %), while the other two do not have any benefits, nor drawbacks. They explain that it is difficult to plan the work of the GC such that the cost of the transactions is fully amortized.

We consider that our contribution complements this work with the study of the impact of HTM on *real-life* Java garbage collectors and the proposal of a novel HTM-based algorithm.

5.3 Summary

This chapter summarized the state of the art in the GC field, covering a few relevant examples of optimizations in the literature. We insisted on GCs implemented for real-time systems, since they have the strictest constraints regarding stop-the-world pauses. We then looked at GCs employing transactional memory, both software and hardware. We commented in detail the approaches related to our work, marking their downsides, advantages and parts that leave room for improvement.

Chapter 6

Performance Study of Java GCs

Contents

This chapter surveys current Java GCs and their performance implications. We study to what extent the GC can affect the application performance and give some pointers on what configurations can have an important impact on the application execution. We assess the performance in terms of both responsiveness (i.e., the delay between the generation of an event and its acknowledgment) and throughput (i.e., the amount of work performed in a given period of time). We also measure the duration of the GC pause times, i.e., the period of time when none of the application threads is progressing. We experiment with the OpenJDK8 Java Virtual Machine, being one of the most extensively used language runtimes. All collectors in this JVM are generational; this means that the heap is split in two parts: the young and the old generation. As an optimization for multi-core systems, a *thread local allocation buffer* (TLAB) is used in the young generation (i.e., chunks of the young generation are pre-allocated

for each thread, which subsequently takes care of the local allocations). Thus, most of the time there is no need for synchronization support when allocating new objects. We exhaustively test the efficiency of all current GCs both in an academic environment (with a set of benchmarks in the DaCapo suite), as well as in a real-life scenario (running the massively scalable database Apache Cassandra). The benchmark experiments are split in two categories: (1) when a system (full) GC is forced between the benchmark iterations and (2) when no system GC takes place and the memory is collected only when needed.

Overall, our study tends to show that sometimes the GC might have an unexpected behavior (e.g., longer pauses for smaller young generation size, TLAB decreases performance in some cases). We detail below the main issues we investigated.

- **Total execution time:** we measure the total execution time for the benchmarks in our subset with different GC and heap sizes. G1 GC has the worst throughput among the GCs for test case (1), while for (2), the GCs perform in a similar manner;
- **GC pause time:** we study the length of the application pauses caused by the GC activity, for all benchmarks. We observe that the pause length varies significantly for all GCs. Overall, ParallelOld seems more stable than G1 and ConcurrentMarkSweep;
- **GC statistics**: we observe that sometimes a smaller young generation size results in a bigger *average pause time* for the same heap size (e.g., in the case of ConcurrentMarkSweep and ParNew GCs);
- **TLAB influence:** we check whether enabling the TLAB leads to a performance improvement for each GC and benchmark. We consider a variation of 2 % from the average total execution time. We find that most of the time the TLAB does not have any influence (either good or bad), but there are cases in which it even degrades the performance;
- **GC ranking:** we order the GCs according to the best execution times of the benchmarks in our experiments. Based on this simple ranking and the previous tests we find that the default GC, ParallelOld, is constantly performing well, while both G1 and ConcurrentMarkSweep tend to degrade the application performance.

For the client-server experiments we use Cassandra DB as the server and the YCSB benchmark on the client-side. We discuss the impact of three most often used GCs on multi-core machines (ParallelOld, ConcurrentMarkSweep and G1).

- **On the server side:** a detailed analysis of Cassandra's logs indicates application pauses of up to 4 minutes for ParallelOld, and of 3–5 seconds for G1 and ConcurrentMarkSweep. Even though the latter is much smaller than the former, it is not negligible in systems where responsiveness is critical;
- **On the client side:** we study the client response time and observe that most of the peaks in the response time correspond to the moments when a GC took place. The result shows that the server is unable to progress during the pause time. This degrades the user experience and, possibly, the progress on other nodes (that are either waiting for the response or suspect that the collecting server is faulty).

Based on these observations, we draw the conclusion that there are workloads that are executed better with particular GCs and sometimes the assumptions on the GC activity do

not hold true for all scenarios. We show that, on the DaCapo benchmark suite, Java's default GC gives the best overall performance out of the three main GCs, while on a memory-intensive database server, it introduces unacceptable pauses and affects the response time on the client. Moreover, even the concurrent low-pause GC, ConcurrentMarkSweep, and G1 GC that ensures bounded pause time, end up stopping the application threads for a few seconds. This is still a significant pause that can affect the execution of the application.

6.1 Experimental setup

The benchmark suite and client-server system further described in this section are used in experiments throughout Part II of the thesis. We introduce here all DaCapo benchmarks with their characteristics and some internal details of the Cassandra DB server. The following infrastructure configuration is only used for the performance study in the current chapter.

6.1.1 Infrastructure details

We mainly perform the experiments on a 48-core server with 64 GB RAM, running Ubuntu Linux on 64 bits. The cores are distributed over 4 sockets: 2 NUMA nodes per socket, each having 6 CPUs. Each core benefits of a 1.5 MB Level-1 cache (different for instructions and data) and a 6 MB Level-2 cache. A 12 MB Level-3 cache is available per NUMA node. Additionally, we use a 16-core machine with 8 GB RAM for the database client.

We use OpenJDK8 for all the tests. We consider as **baseline** the default GC configuration used by Java: ParallelOld GC, with a maximum heap size of ∼16 GB, maximum young generation size ∼5.6 GB and TLAB enabled. We set both the minimum and maximum heap size at the same value, so that we have a fixed heap size.

6.1.2 DaCapo benchmark suite

The DaCapo benchmark suite [11] is a memory-management benchmarking tool for Java. It consists of the following set of open source, real world applications:

- ***avrora:*** single external thread, but internally multi-threaded;
- ***batik:*** mostly single-threaded both externally and internally;
- ***eclipse:*** single external thread, internally multi-threaded;
- ***fop:*** single-threaded;
- ***h2:*** multi-threaded (one client thread per hardware thread);
- ***jython:*** single external thread, internally using one thread per hardware thread;
- ***luindex:*** single external thread, internally it uses some helper threads to a limited degree displaying limited concurrency;
- ***lusearch:*** multi-threaded, one client thread per hardware thread;

- ***pmd:*** single client thread, internally multi-threaded using one worker thread per hardware thread;
- ***sunflow:*** multi-threaded, driven by a client thread per hardware thread;
- ***tomcat:*** multi-threaded, driven by a client thread per hardware thread;
- ***tradebeans:*** multi-threaded, driven by a client thread per hardware thread;
- ***tradesoap:*** same as tradebeans;
- ***xalan:*** multi-threaded, driven by a client thread per hardware thread.

When executing a DaCapo benchmark with the default options, it only runs once, returning the execution time at the end. However, a number of parameters can be fed to DaCapo. An important option is the number of iterations the benchmark is required to execute. By default, the application will perform a system GC between every two iterations. This feature can also be disabled. When running multiple iterations, all except the last one represent warm-up rounds; the last iteration is the actual run of the benchmark. Besides this property, one can also indicate the number of threads for the current execution. This option will overwrite the default setting of having one client thread per hardware thread.

There are 14 benchmarks in the most recent DaCapo suite, released in 2009. Out of these, 3 benchmarks crashed on every test: *eclipse*, *tradebeans* and *tradesoap*. For other benchmarks, e.g., *avrora*, the execution time from one iteration to the next varied significantly. Thus, in order to identify a subset of stable benchmarks, we ran each 20 times, with the baseline Java configuration. A run consisted of 10 iterations. After each iteration a system GC takes place. We computed the variation as the *relative standard deviation*, which indicates how tightly is the data clustered around the mean. We take as metrics for stability the following two characteristics:

- **The duration of last iteration:** we do not take into consideration the duration of warm-up rounds, expecting the actual run duration to be stable;
- **The total execution time:** even though the duration for each iteration varies separately, we check if the total execution time, i.e., the sum of the durations of all iterations, remains constant.

Based on the information gathered we selected the benchmarks listed in Table 6.1. All other benchmarks in the DaCapo group showed variations of more than 5 % in both measured times: execution time and final round. We accepted in our experiments the benchmarks that are stable for at least one characteristic. The rest of this work only focuses on the selected subset of benchmarks.

6.1.3 Apache Cassandra database server

Cassandra [1] is a distributed on-disk NoSQL database. Its architecture is based on Google's BigTable [21] and Amazon's Dynamo [29] databases. It provides no single point of failure, and is meant to be scalable and highly available. Data is partitioned and replicated over the nodes. Durability in Cassandra is ensured by the use of a commit log where all the modifications are recorded. Exploring the whole commit log to answer a request is expensive; thus, Cassandra also

Table 6.1 – Relative standard deviation for the final iteration and total execution time for a subset of DaCapo benchmarks.

Benchmark	Final iteration (%)	Total execution time (%)
h2	2.67	1.71
tomcat	1.37	1.08
xalan	2.41	4.01
jython	5.79	3.43
pmd	2.75	0.78
luindex	8.6	3.21
batik	10.75	2.54

has a cache of the state of the database. This cache is partially stored to disk and partially stored in memory. After a crash, a node has to rebuild this cache before answering client requests. For this purpose, it rebuilds the cache that was stored in memory by replaying the modifications from the commit log. Through a couple of configuration files, one can select the GC, heap and young generation sizes and modify the amount of memory used for the cache and for the commit logs.

6.1.4 Workload generator: YCSB client

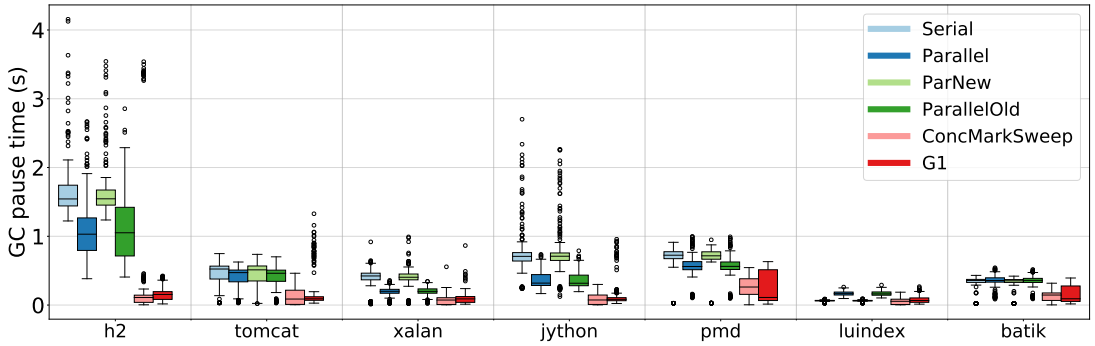
A good way to test the responsiveness of the database is to run a benchmark on the client-side, such as Yahoo Cloud Serving Benchmark (YCSB) [25]. YCSB Client is a workload generator, having predefined core workloads that can be further extended according to the user's needs. The workloads define the number of *read*, *insert* and *update* operations executed on the database, the number of records that will be affected, the execution time, etc. It can be used in two states: the loading phase and the running phase. The former only populates the database (loads the data), while the latter executes the specified workload against the database. At the end, it returns statistics about the execution.

6.2 Experiments on benchmark applications

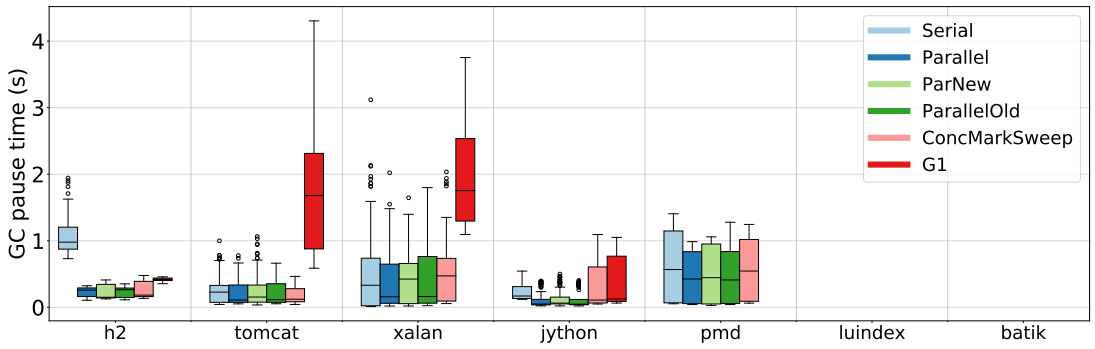
Starting from the baseline configuration, we run benchmarks of the DaCapo suite with all the supported GCs. We vary the maximum heap size from the baseline to the maximum amount of memory supported by the machine, i.e., 64 GB. We also vary the young generation size from the baseline to the heap size. Finally, we test separately when the TLAB is enabled or disabled. Typically, we use the default DaCapo configuration for the benchmarks. This configuration exploits the maximum number of hardware threads available on the machine. We generally configure 10 iterations for each benchmark. We also switch between having a system GC after each iteration and no system GC inserted by the benchmark.

6.2.1 GC pause time

All currently implemented GCs in OpenJDK8 need to stop the application threads in at least one of their collection phases. We compare the impact caused by GC pauses for the selected



(a) System GC



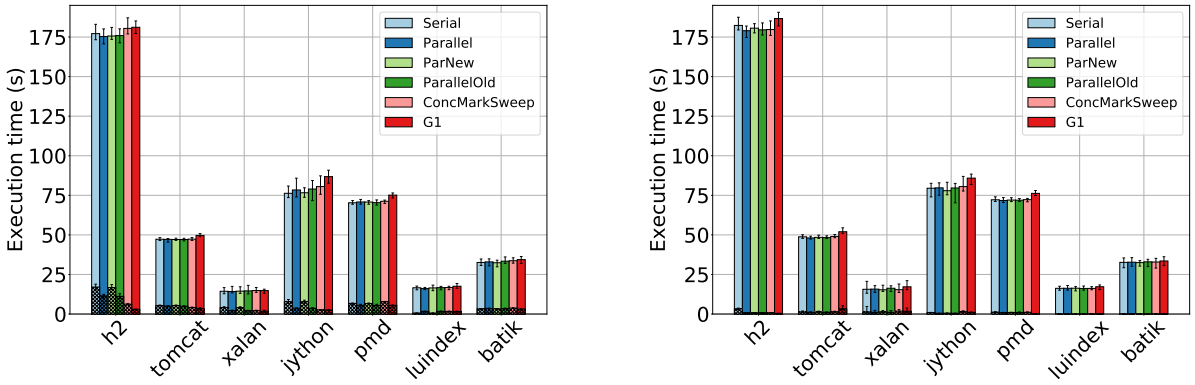
(b) No system GC

Figure 6.1 – GC pause times for all Java GCs and DaCapo benchmarks.

benchmarks on multiple axes, as follows: Figure 6.1 shows the distribution of GC pause times across benchmarks; Figure 6.2 shows the total execution time of the benchmarks for all supported GCs; finally, Figure 6.3 indicates the total number of GCs during the execution of each benchmark. In all cases we illustrate the results having system GC (which forces a full collection) activated between iterations, and when this feature is deactivated. By default, the JVM uses a sequential GC for explicit system GCs. In order to fully study each respective GC, being especially interested in the performance of the concurrent ones, we forced the system GCs to be executed with the selected old generation GC. This is accomplished by adding the `-XX:+ExplicitGCInvokesConcurrent` runtime option to the JVM.

In all charts the X axis represents the benchmarks that were executed, while the Y axis indicates the pause duration (in seconds), the total execution time (in seconds), and, respectively, the number of GCs. Basically, these plots cluster the results for all benchmarks and all GCs for different characteristics. All tests are conducted with the baseline configuration for the heap, young generation and TLAB. We repeat each experiment 20 times. In Figures 6.2 and 6.3, the bars represent the median value of the respective feature over all runs. We use error bars to indicate the variations of the measured attributes, representing the minimum and the maximum values. The error bars help us draw meaningful conclusions regarding the GC impact. Moreover, the pattern-filled area in the execution time plots represents the fraction of time the GC takes from the total execution time of the benchmark.

We start by analyzing the GC pauses in Figure 6.1. The figure indicates the median pause time, the variation from the median (the box extends from the first quartile to the third quartile,



(a) System GC

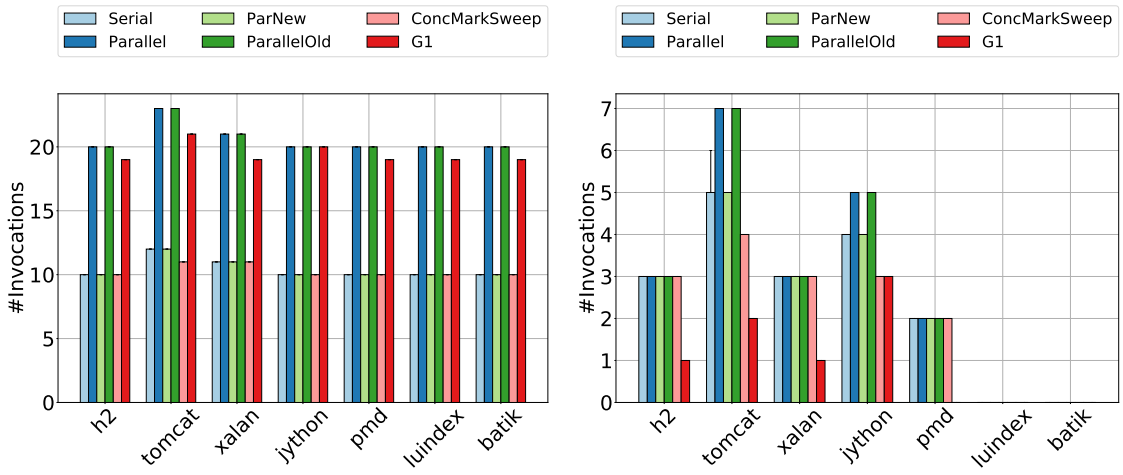
(b) No system GC

Figure 6.2 – Execution time for all DaCapo benchmarks with various GCs enabled (total GC duration marked).

the whiskers extend with $1.5 \cdot IQR$ (Interquartile Range) from the ends of the box), and the outliers. We observe that in general the GC pause length varies significantly. We consider first the runs that have a system GC between all iterations (Figure 6.1(a)). In this case, the distribution of the pause length corresponds to the following pattern: most of the GC pause times cluster around the median, with a few pauses considerably longer than the average (e.g., the outliers in the figure). Despite having the shortest median pause times in all cases, CMS and G1 compete closely with the other GCs for the longest maximum pause. For example, we see a few pauses of more than 3 seconds for CMS and the *h2* benchmark (where the total execution time of the benchmark is close to 3 minutes, as shown in Figure 6.2(a)). Similarly , G1 generates pauses close to or over 1 second for three other benchmarks (out of which *xalan*, which is executed entirely in less than 20 seconds). ParallelOld GC is neither the best nor the worst in any situation. While its pause times vary as much as for the other GCs, it has fewer spikes, being more stable.

On the other hand, when no system GCs are planned between iterations, no GCs at all are performed for two benchmarks (i.e., *luindex* and *batik*) with the aforementioned memory configuration. G1 also avoids collection in the case of the *pmd* benchmark. We observe that **G1 is consistently the collector with the smallest number of pauses, but also with the most prominent spikes in the pause length**. The number of pauses can be seen in Figure 6.3(b). We observe pauses of over 3 seconds for the same benchmarks that were affected in the other scenario: *tomcat* and *xalan*. We notice that in this case CMS and ParallelOld have comparable performance, with CMS being usually slightly worse than ParallelOld (that is, having either the median or the maximum pauses greater).

Looking at the benchmark execution time (Figure 6.2(a)), we note that **G1 GC has the worst throughput** when system GC is activated and it is forced to have multiple full collections. Generally the execution time of the application is longer when using G1 than any other GC. It goes up to 16.7 % longer than for all the other GCs, in the case of the *jython* benchmark. Generally, one of the best performing GCs is ParallelOld GC, which is also the default GC for Java. Another interesting remark is that the total execution time of the benchmarks remains roughly the same when system GC is enabled or disabled, even though in the latter case fewer GCs take place and the total time spent collecting is smaller. We believe this happens because in



(a) System GC (b) No system GC

Figure 6.3 – Number of GC invocations per application run for all DaCapo benchmarks.

the former version there is a system GC before the first iteration, which prepares the benchmark to proceed with a clean view of the memory. In the other case, the benchmark starts the iterations immediately after setting up all data structures that are necessary for its run. We observe that when system GC is disabled the first iteration takes more time to complete the same amount of work. This extra time amortizes to some extent the lack of system GCs and the benchmarks end up with similar execution times as the version with system GCs between iterations.

Figure 6.3 shows how many collections take place for each benchmark, with the two settings regarding the system GC between iterations. The values in the figure are the ones reported by each GC in their log file. Some of the GCs count a full collection as two (one for the young generation and another one for the entire heap). This is the case for the GCs that consistently report 20 or more invocations for the system GC scenario (ParallelGC and ParallelOld GC). Practically, there are few situations in which the GCs need to collect more than those 10 full collections imposed by the benchmark settings (i.e., at most 3 more times for most GCs). A special case is G1, which constantly reports 9 – 11 extra invocations. The explanation is that this GC considers the two separate blocking phases of its collection as being 2 invocations (sometime one of them is not necessary). Thus, basically, G1 does not collect more than during the system GCs between iterations either. When there is no forced full GC (Figure 6.3(b)), we observe that all benchmarks strictly need fewer collections than the number imposed between iterations. As mentioned before, G1, followed by CMS, have the smallest number of invocations in this case.

Next, we evaluated the correlation between the pauses caused by the GCs and the sizes of the heap and the young generation. We expected to find the relation described by Blackburn et al. [61, 77]: the **total pause increases with the decreasing young generation size**, in applications that spend most of the time in young generation collections. This happens because the total number of minor collections increases, when the young generation is smaller. However, the **average pause should decrease with the decreasing young generation size**.

Given the fact that some benchmarks did not show any GC with our heap configuration, we ran the experiments once more with the following settings:

Table 6.2 – Statistics for the *h2* benchmark with varying heap and young generation sizes (CMS GC).

Heap-young gen. size	#pauses(full)	Avg. pause time (s)	Total pause time (s)	Total execution time (s)
64 GB – 6 GB	3 (0)	1.46	4.6	190.05
64 GB – 12 GB	2 (0)	0.51	1.04	186.72
64 GB – 24 GB	2 (0)	0.47	0.94	185.55
64 GB – 48 GB	2 (0)	0.37	0.74	187.9
1 GB – 200 MB	67 (1)	0.08	5.26	183.66
1 GB – 100 MB	135 (1)	0.07	9.03	189.8
500 MB – 200 MB	74 (7)	0.11	8.31	189.64
500 MB – 100 MB	136 (3)	0.07	9.02	188.64
250 MB – 200 MB	629 (342)	1.35	848.4	1103.52
250 MB – 100 MB	144 (89)	1.09	160.33	771.6

- heap sizes of 1 GB, 500 MB, 250 MB;
- young generation sizes of 200 MB, 100 MB.

Based on both sets of experiments, we observed that our expectation does not hold in all cases. We can take as an example the *h2* benchmark and its results for the ConcurrentMarkSweep collector (Table 6.2). In the upper side of the table we keep the heap size constant, at 64 GB and vary the young generation size from 6 GB to 48 GB. We observe that the **average pause** duration for the smallest young generation size is significantly longer than for a bigger young generation size. Likewise, the average pause duration for a young generation of 24 GB is longer than the one for 48 GB. In the second part of the table, we list the results for the small heap and young generation sizes. Having such a small amount of memory for this benchmark results in hundreds of garbage collections during the execution. It is interesting to point out that in the case of a small heap the total pause time can represent more than 50 % of the total execution time. Another important fact is that in the same conditions the ParallelOld collector behaves as expected in both situations.

6.2.2 TLAB influence

The **Thread Local Allocation Buffers (TLAB)** represent chunks of young generation, one buffer per thread, where the new objects are first allocated. This allows for faster memory allocation, since the thread is able to allocate memory inside the buffer without a lock.

We consider that the TLAB has a positive impact for a GC when the total execution time for that GC was smaller than the same experiment without TLAB. To account for the variation in the execution times, we computed a 2 % deviation from the average execution time. Thus, if the difference between the total times with and without TLAB is included in the interval [-deviation,deviation], we say that enabling the TLAB does not bring either improvement or deterioration (=). If the total execution time without TLAB is greater than the execution time with TLAB (plus the deviation), it means that enabling the TLAB results in an improvement in the execution time (+). Otherwise, we mark it as negative influence (–).

Based on the results obtained for the baseline configuration of the heap and young generation size, we observe that in most cases the TLAB does not have a particular influence (Table 6.3).

Table 6.3 – TLAB influence over all GCs and the selected subset of benchmarks.

Benchmark	ConcMarkSweep	G1	ParNew	Parallel	ParallelOld	Serial
h2	=	=	=	=	=	=
tomcat	=	=	=	=	=	=
xalan	+	=	=	+	+	–
jython	=	=	=	=	=	=
pmd	=	=	=	=	=	=
luindex	–	=	=	=	=	–
batik	=	=	=	=	=	=

However, there are cases when enabling the TLAB leads to a decreased performance (e.g., for CMS and Serial GCs). We observe that for the *xalan* benchmark, enabling the TLAB varies between improvement and deterioration depending on the GC used. On the other hand, for the *luindex* benchmark the TLAB is at most indifferent, resulting in decreased performance if CMS or Serial GCs are used. We correlate the deterioration of *luindex* with the fact that this benchmark is mostly single-threaded, so by default it should not gain much from enabling the TLAB. Also, we suspect that the lack of improvement when enabling the TLAB is related to the small size of the benchmarks, which do not stress the memory so much that the TLAB can make a difference. Even so, the result is surprising.

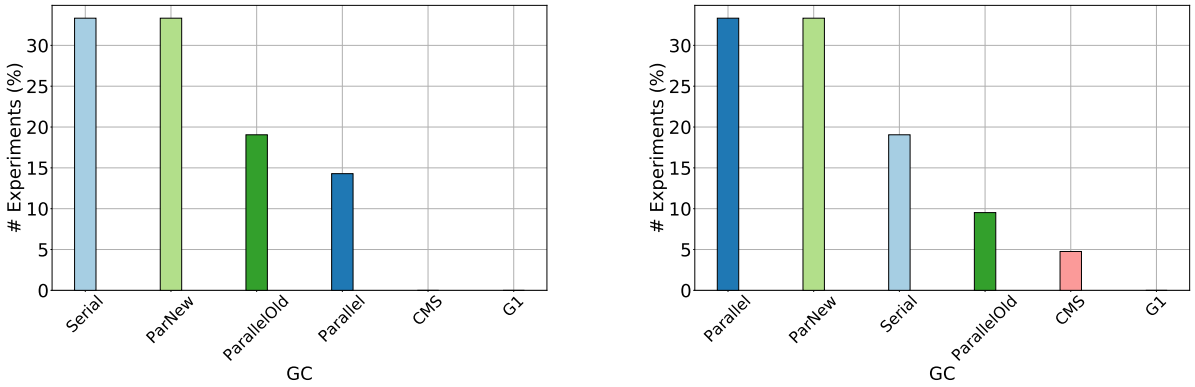
6.2.3 GC ranking

In order to have an overall idea on the GC influence over the DaCapo benchmarks, we finally classify them according to the number of experiments in which they perform the best. An experiment is defined by a benchmark, a heap size and a young generation size. We make this ranking based on the following combinations (heap size, young generation size):

- **(**16 GB**,** 6 GB**),** the baseline configuration;
- **(**64 GB**,** 12 GB**),** tests run on a big heap and young generation;
- **(**500 MB**,** 200 MB**),** tests run on a small heap and young generation.

We choose these values in order to have both a configuration which stresses the GC (as seen in Table 6.2), and one on the other size of the spectrum. In total, there are 21 individual experiments, executed for all GCs over 20 runs. For each experiment we consider the run with the shortest execution time as the best. The Y axis in Figures 6.4(a) and 6.4(b) shows the percent of experiments for which each GC was employed in the best run. The GCs (on the X axis) are sorted according to their ranking.

When the system GC is enabled between all iterations (Figure 6.4(a)) there is no column for the G1 or CMS collectors. That means that **G1 and CMS did not perform better than all other GCs in any of the experiments**. The baseline collector is average, being best at almost 20 % of the experiments. Surprisingly, the Serial collector has the greatest ranking, together with ParNew, both of which stop the world entirely for old generation collections. However, we believe that having a system GC between all iterations reduced the size of the collections; the



(a) System GC

(b) No System GC

Figure 6.4 – GC ranking according to the number of experiments in which they performed the best.

concurrent GCs (i.e., G1 and CMS) ended up with introducing more synchronization overhead than the time spent collecting if the world was stopped.

If we disable the system GC (Figure 6.4(b)), CMS GC improves, being the best in around 5 % of the experiments. However, CMS and G1 are still the worst according to this metric. G1 GC is ranked last, with 0 %. We also observe that, in this case as well, the default GC (ParallelOld) is better than the concurrent GCs.

Based on these results, we conclude that the ParallelOld GC is a fitting choice for a default GC, since it proves to be stable and has satisfactory results. Running either of the concurrent collectors, G1 and ConcurrentMarkSweep, results in both longer pause times and execution time. The next section debates the impact of these three most important GCs on a memory-intensive real-life application. We show that in this case, ParallelOld has significant drawbacks, as compared to the other two collectors.

6.3 Experiments on client-server applications

We started this study by analyzing the impact of garbage collector activities on a set of benchmarks. The next step is to check if the previous results hold on a real-life environment, such as a client-server system. Our experiments follow two main directions:

- The duration of the GC-induced pauses on the server;
- The impact on the response time at the client side.

For both experiments we use Cassandra DB on a single node as the server in combination with the YCSB benchmark as the client. On the server we set the heap size at 64 GB and the young generation size at 12 GB. This experiment also aims to evaluate the behavior of GCs with a very big heap and young generations size. In the previous section we were unable to perform this test, since for these values the DaCapo benchmarks were not filling up the memory and there were no GCs. We choose this size for the young generation according to the JVM recommendations to be around one fourth of the total memory.

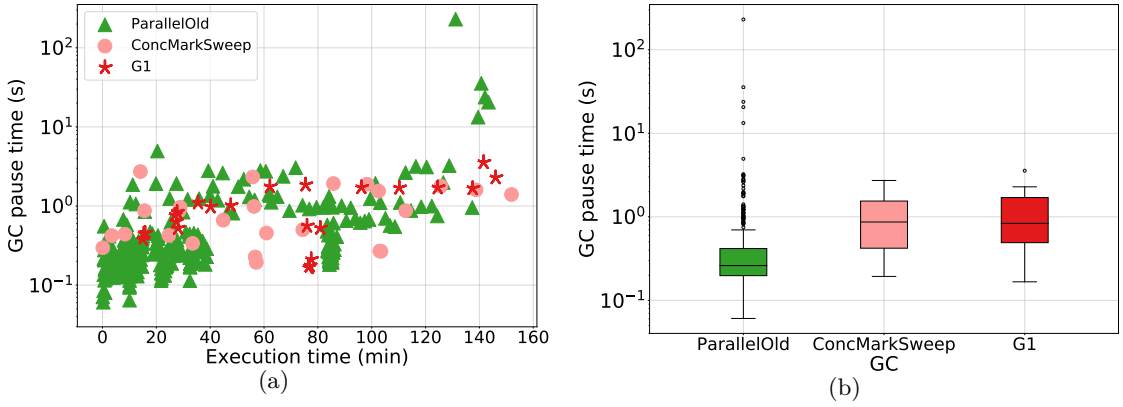


Figure 6.5 – GC pauses for the three main GCs (ParallelOld, ConcurrentMarkSweep and G1) for the stress-test configuration on Cassandra server.

6.3.1 GC impact on the server side

We first study the performance of the default GC, ParallelOld. We based our main experiments on two different Cassandra configurations. In both cases, the YCSB client is used in the *loading phase*, i.e., it continuously populates the database with records, for a specified amount of time.

- **Default configuration.** We run the experiment on 100 client threads in parallel for one hour and two hours, respectively. The shorter test case ends with no full GC; nonetheless the collection of the young generation reaches a peak pause of around 17 seconds. The latter stresses the memory even more by loading records for an extra hour. This results in a full GC that stopped the application threads for more than 160 seconds. Moreover, the young generation collections take up to 25 seconds;
- **Stress test configuration.** We take advantage of Cassandra's internal data structures and configure it to flush as rarely as possible its records to disk and stress the memory until the server is saturated. That is, we set up both the commit log and the internal caching structure of Cassandra (called *memtable*) to have the same size as the heap, which means that everything is always kept in memory. Moreover, we load the database with records beforehand, so that when it starts the memory is already partially occupied. Then, we run the same workload as before for two hours. This experiment results in a full GC lasting around 4 minutes.

Then, we experiment with CMS and G1 on the stress test configuration of Cassandra. Figure 6.5 compares the performance of all three GCs in heavy-workload conditions (100 % write). We study the evolution of the GCs in time in Figure 6.5(a) and we look at the distribution of GC pauses in Figure 6.5(b). Both figures are presented in logarithmic scale. The former shows the resulting pauses caused by the GC activity on the Y axis, while the X axis indicates the execution time. Even though the YCSB client itself ran for a fixed amount of time (two hours), the total execution time in the chart is longer. It also contains the loading step of Cassandra: because of our configuration, the server must first bring into memory all commit logs and replay already executed transactions. Only then we start the actual benchmark. Figure 6.5(b) shows the GC pause times (on the Y axis) for each of the 3 GCs (on the X axis). The boxes and whiskers have the same configuration as for the DaCapo benchmarks (Section 6.2.1).

Strictly regarding ConcurrentMarkSweep and G1 GCs, we observe that their performance is comparable in terms of stop-the-world pause duration. Both of them reach pauses of more than 2 seconds, going up to 3.5 seconds for G1. When compared to the default GC, the first observation is that ParallelOld's pauses are up to two orders of magnitude longer than for the other two GCs (hence the logarithmic scale on Y axis). Looking at the GC evolution during the entire run (Figure 6.5(a)), ParallelOld shows much more activity than the others, accounting for a considerable total pause time per application. Another interesting fact is that the smallest pauses are also provided by ParallelOld, while the pauses for G1 and CMS keep to the same duration, without varying too much. In the first 30–40 minutes of the run (corresponding to the replay of the commit logs), we observe a cluster of low-pause invocations for ParallelOld. In the same interval, the concurrent GCs only collect a few times but the collections have higher latency. Next, all 3 GCs perform similarly for the most part of then run. However,towards the end of the experiment, when the memory gets filled, ParallelOld has several enormous pauses. More precisely, almost all pauses after the longest pause showed in the plot are over 10 seconds long. Both ConcurrentMarkSweep and G1 continue on the same trend as before until the end, with only a slight spike up to 3.5 seconds on the last G1 collection.

In addition, Figure 6.5(b) summarizes statistical aspects of the GC pauses in this context. It confirms the above observations. ConcurrentMarkSweep and G1 are both more stable and very similar performance-wise. We find only an outlier for G1, corresponding to the aforementioned spike at the end of the experiment. ParallelOld has the median pause time much lower than the other two, but the variation is greater, as well as the maximum pause times.

Even if the pauses caused by ConcurrentMarkSweep and G1 are considerably smaller than those resulted from the use of ParallelOld, they can still be unacceptable in certain circumstances. Apache Cassandra is a distributed database, supposed to run on multiple nodes for an indefinite amount of time. In only two hours we succeeded to saturate the server and obtain garbage collections lasting for a few seconds with G1 and ConcurrentMarkSweep, respectively for a few minutes with ParallelOld. Moreover, in a distributed system, even a lag of a few seconds might result in the current node being considered down and the initiation of a cumbersome synchronization protocol.

6.3.2 GC impact on the client side

We use a custom workload for the client-side experiments: 50 % *read* and 50 % *update* operations. We run the experiment three times, with the three main garbage collectors supported by Cassandra: ParallelOld, ConcurrentMarkSweep and G1.

The charts resulted from the above experiment are illustrated in Figure 6.6. In order to make the charts more readable and to reduce the size of the images, we only plot the highest 10000 points of every chart. The shared X axis represents the execution time passed since the beginning of the experiment, in minutes. We run each experiment for two hours. The Y axis indicates the latency of the aforementioned operations on the client during the experiment. On the same plot we mark in red the moments in time when GCs took place on the server and the length of the GC pause in that point (on the Y axis). This way, we aim to determine if there is any correlation between GC pauses on the server and delayed response time on the client.

We make several important observations based on this figure. First, most of the points are following a well defined low latency line; moreover, for the *update* operations the line of points is constant for all three GCs, while for the *read* operation, the line has some increasing "steps".

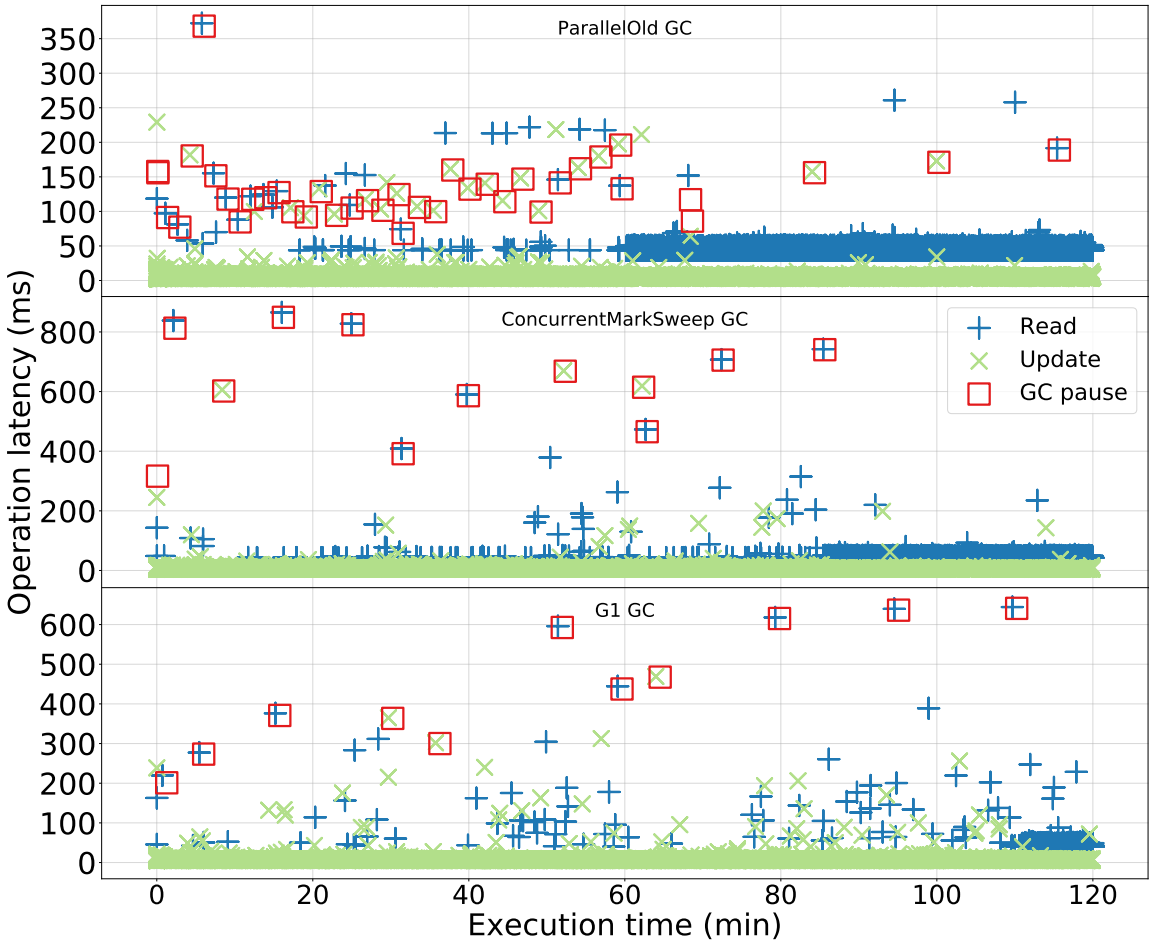


Figure 6.6 – Correlation between operation latency on the client and GC pauses on the server with the three main GCs: ParallelOld, ConcurrentMarkSweep and G1.

Apart from the constant line of values, there are the spikes that we expected to see: a few tens of points with greater latency, for both kinds of operations. Second, by plotting the GC pause on the same chart as the operation latencies, we observe that the highest latencies correspond to the moments when a collection took place. In most cases, the length of the GC pause corresponds to the latency of the operation as well. Thus, we conclude that the latency peaks are strongly related to the garbage collection activity.

Finally, we note again the visible difference between the number of pauses generated by each of the GCs: despite having a lighter workload than before, ParallelOld has around 4 times more pauses than the other two GCs. They do not match all the latency spikes, but most of them can be correlated with GC pauses on the server.

6.4 Summary

This chapter presented an extended study over all Java GCs and their properties. We started with lining up some expectations on the GC behavior in general and checked them one by one. We experimented with the DaCapo benchmark suite and the Apache Cassandra server. They

Table 6.4 – Advantages and disadvantages of the 3 main GCs, according to our experiments.

GC	Experiment	Throughput	Pause Time
ParallelOld	DaCapo	good	short
	Cassandra	good	unacceptable
CMS	DaCapo	fairly good	acceptable
	Cassandra	fairly good	significant
G1	DaCapo	bad	unacceptable
	Cassandra	fairly good	significant

offer different testing conditions in terms of memory and load and, thus, different perspectives on how the GCs react. We found that the GCs do not always behave as expected. We generally focused more on the three most widely used GCs in Java: ParallelOld, ConcurrentMarkSweep and G1. We analyzed the results and roughly summarized the benefits and disadvantages of these three GCs in Table 6.4.

Chapter 7

Transactional GC Algorithm

Contents

While garbage collectors significantly simplify programmers' tasks by transparently handling memory, they also introduce various overheads and sources of unpredictability. Most importantly, GCs typically block the application while reclaiming free memory, which makes them unfit for environments where responsiveness is crucial, such as real-time systems. There have been several approaches for developing concurrent GCs that can exploit the processing capabilities of multi-core architectures, but at the expense of a synchronization overhead between the application and the collector. In this chapter, we introduce **a novel approach to implementing pauseless moving garbage collection using hardware transactional memory**. We first elaborate on the various components that play an essential role in the algorithm. Then, we describe the design of a moving GC algorithm that can operate concurrently with the application threads. Finally, we give an informal intuition on the correctness of the concurrent system presented.

7.1 Introduction

Generally, managed runtime environments use *garbage collectors* to automatically reclaim memory. For today's servers characterized by large memory requirements and running on multi-core architectures, a GC that suspends the application during an entire collection is not adequate anymore. With a heap size of tens of GB, the pause caused by a single collection cycle can sometimes reach minutes, dramatically degrading server responsiveness [41, 19].

Significant efforts have been put in the last few years into improving concurrent GCs, since usually concurrency comes at the expense of possible consistency issues. For example, when an object is copied, but not all its incoming references are updated to the new version of the object, the application may be able to access both the old and the new version of the object, in case of a faulty implementation. In order to avoid this issue, concurrent moving collectors use fine grained synchronization mechanisms between the GC and the application. These mechanisms are applied to memory reads or writes performed by the application. While current moving concurrent collectors actually improve the responsiveness of the application, they tend to globally degrade its throughput. They are slowed down by the extra code executed at each memory read or write (also called *barrier*).

7.1.1 Motivation

The significant impact of concurrent moving GCs on application throughput and responsiveness together with the possible solution offered by transactional memory motivate this work. Exploratory research studying the impact of garbage collection on application performance is presented in Chapter 6. Briefly, we considered two scenarios: *a benchmark suite* and *a client-server system*, and two main metrics: the application throughput and the duration of GC pauses. We evaluated the GCs featured in OpenJDK8's HotSpot, while varying the heap size, young generation size and other properties. In what follows, we only focus on the three main GCs considered in this work: ParallelOld, ConcurrentMarkSweep and G1.

To summarize our experiments, we have found that, in terms of throughput, ConcurrentMarkSweep and G1 perform poorly on the benchmark suite, whereas ParallelOld runs consistently well. This results show that a non-moving concurrent algorithm, such as ConcurrentMarkSweep, already introduces an overhead. For a moving collector, we can reasonably suspect that this overhead will be significantly worse because of the synchronization required to avoid inconsistencies between the old and the new versions of the same object.

Then, in term of responsiveness, we have found that the use of ParallelOld results in unacceptable pauses of the application for a client (e.g., up to 4 minutes in our experiments); ConcurrentMarkSweep and G1 have significantly better results than ParallelOld, which shows that concurrency actually helps in improving responsiveness. Nonetheless, their pause lengths go up to 3.5 seconds because of the big heap/young generation size that was continuously kept saturated by the client. These pause times can still be unacceptable on certain systems. For instance, in a distributed environment (such as Cassandra when deployed on multiple nodes) the response time on a node might be critical to keep the system from considering it faulty or crashed. Moreover, Cassandra DB should be able to run without interruption for an indefinite period of time: we only ran it for two hours and all 3 GCs finished with pauses longer than 1 s.

In this work, we study whether using HTM to synchronize the application and the GC could remove the application pauses while maintaining an acceptable throughput. To this end,

we propose a new concurrent moving GC algorithm that uses transactions in read and write barriers to synchronize the application with the GC. When the GC needs to move an object, it marks the object as busy. If an application thread tries to access the object at the same time, the transactional barrier notices the busy state and aborts the transaction. Otherwise, the application threads can work concurrently with the GC threads.

7.1.2 Design choices

We consider as starting point a classical concurrent copying collector algorithm. We will call it *the base collector* of our approach. We extend it with transactions executed during application reads and writes. We use them to avoid inconsistencies when objects are moved. In practice, we looked at Java's **ConcurrentMarkSweep collector (CMS)**, real-life implementation that fits the aforementioned theoretical aspects. We further elaborate on the implementation details in Chapter 8.

There are two different strategies of implementing garbage collection with transactional memory: one could either instrument the GC (such as in the solution offered by the Collie collector [55]), or the application accesses (load and store). We apply the latter approach in our algorithm. We choose this design for the following reasons: first, we avoid implementing a state-of-the-art GC from scratch (i.e, the Collie collector), being able to work directly on the ConcurrentMarkSweep HotSpot collector. Moreover, we directly eliminate the need for the two traversals the object graph, one of the main downsides of the Collie algorithm. Finally, when inserting HTM on the application side, we can easily estimate the transaction size (encapsulating only a single load or store of an object), a crucial aspect when using HTM.

7.2 Our approach

As mentioned before, stop-the-world GCs have high throughput, but they have considerable pauses with large workloads, such as a fully loaded Cassandra server. In order to decrease this pause time while maintaining comparable throughput, we present a novel GC algorithm that should benefit from HTM. We start by describing the most important components of the baseline GC that are used or modified in our design.

7.2.1 Brooks forwarding pointer

The algorithm that essentially stands at the base of Java's ConcurrentMarkSweep young generation collector (briefly presented in Section 4.1.3) is a variation on a classical incremental copying algorithm defined by Henry Baker [9]. Brooks [18] added an important optimization over the original algorithm, namely the *Brooks forwarding pointer*. Initially, Baker's algorithm featured a costly read-barrier to test if an object needs to be forwarded. In order to avoid the barrier, the objects in a Brooks-style implementation are not referred directly, but through an indirection field in their header. If the object is not yet copied, this field indicates its original address address, in from-space. Otherwise, it indicates the address of the copy, in to-space. The indirection field is also called forwarding pointer. In other words, a forwarding pointer generally points to the *primary copy* of an object (see also Section 4.1.2). If the object is not forwarded, we say that the pointer is self-referential, i.e., it points to itself. The goal of a forwarding pointer is to directly lead the mutators to the primary copy of an object when the mutator still has a

reference to the old copy. This approach allows for lazy updating of the references that point to the old version of a moved object. The cost of this optimization is that Brooks objects require additional space for the forwarding pointer.

The ConcurrentMarkSweep collector in HotSpot already features a forwarding pointer in the object's header. Since the old generation collector is not a copying collector, the forwarding pointer is mostly used when the young generation collector (ParNew) copies objects between survivor spaces or promotes them to the old generation. While this happens during a stop-the-world pause in the original implementation, the presence of the forwarding pointer is extremely important for the correctness of our concurrent transactional algorithm.

7.2.2 Access barriers

Garbage collectors evolved to be concurrent with the main purpose of reducing pause times. However, concurrency also implies synchronization with the mutator, since the application threads would be allowed to work and perform changes at the same time as the concurrent GC threads. This calls for the need to synchronize the access activity of the two groups of threads so as to avoid races. This is done by guarding all mutator accesses (both load and store) with *access barriers*. The barriers between mutators and the heap can either trap and process read accesses (*read-barriers*), or write accesses (*write-barriers*). Not all GC algorithms have both types of barriers, the choice depending on multiple factors [56], such as:

- the frequency of read and write accesses;
- the frequency of barrier invocations;
- the amount of work done inside the barrier.

Usually the cost of read-barriers is greater than that of write-barriers. The latter are mostly used in the context of mark-sweep collectors, indicating where the marking should be done. The former are typically used with copying collectors, where they trap read accesses in order to copy objects to to-space.

In our practical case, the barriers are supposed to identify the access to an object in the process of being moved by the GC, wait for the copy to complete, and redirect the mutator to the new location of the object through the forwarding pointer.

7.2.3 Algorithm design

We propose a new concurrent GC approach that uses transactional memory to improve GC responsiveness and overall shorten stop-the-world pauses. It involves several major alterations to the base copying algorithm. These modifications involve the concepts discussed in previous sections, i.e., forwarding pointers and access barriers.

First of all, the access barriers, both read and write, are redefined to allow safe concurrent access for mutators as well as for the GC threads. For this purpose the typical instructions defined by the barriers (loads and stores, respectively) are encapsulated in hardware transactions. In this approach the execution of the GC is prioritized over mutators' accesses. In other words, mutators trying to access an object in the process of being changed by the GC will always abort and retry. They will succeed in accessing the object when the GC action is finished. This behavior is facilitated by the use of transactional memory.

Algorithm 1 Access barrier.

```
1: function ACCESS-BARRIER(oop)
2:     TX-BEGIN
3:     data ← oop.metadata
4:     if IS-BUSY(data) then
5:         TX-ABORT
6:     end if
7:     value ← LOAD-FIELD(f.field1)
8:     TX-COMMIT
9: end function
```

Algorithm 2 Busy-bit handling.

```
1: function COPY-OBJECT(oop, size)
2:     newoop ← ALLOCATE(size)
3:     SET-BUSY-ATOMIC(oop)
4:     COPY(oop, newoop, size)
5:     oop.FORWARD-TO(newoop)
6:     UNSET-BUSY-ATOMIC(oop)
7:     UNSET-BUSY-ATOMIC(newoop)
8: end function
```

Figure 7.1 – Pseudocode snippets illustrating relevant algorithm parts both on the mutator side (left) and on the GC side (right).

Whenever an object is moved, the GC alters its forwarding pointer to refer the new location. Thus, the forwarding pointer is an important indication of conflict between the GC and a mutator that would potentially try to access the old copy of the object. As such, reading the forwarding pointer right after starting the transaction is an essential step of the algorithm: this way, its memory address will be automatically monitored in hardware during the entire memory access of the mutator. HTM would notice any conflict with the GC and force the mutator to retry its access.

Further on, we add a flag to each object, indicating if the object is being currently modified by the GC. We call it *busy-bit*. Before copying an object, the GC sets the busy-bit as enabled, indicating that the mutator must wait before accessing the object. The GC then copies the object and updates the forwarding pointer to the new location of the object. The busy-bit is unset, communicating to the mutator that it is now safe to access the primary (new) copy of the object. On the mutator side, we check the state of the busy-bit in the transactional barrier, before accessing the object. If the current object "is busy", we explicitly abort the transaction, letting the mutator retry when the copy is finished. Finally, if the current object is neither marked as busy nor moved by the GC in the meantime, the mutator is free to access it, concurrently with a GC that works in other memory areas.

These steps are illustrated in Algorithm 1. The *oop* in the pseudocode stands for *ordinary object pointer*, denoting a managed pointer to an object. For simplicity, let us assume that an object conceptually consists of metadata and the actual contents. Both the forwarding pointer and the busy-bit are located in the metadata section of the object. Line 3 represents the part where we load the metadata of the object of interest, right after starting a transaction. Thus, we have the forwarding pointer monitored in hardware and we can also check for the busy-bit as described (line 4). *Field1* represents an object field that needs to be loaded in the particular case presented in Algorithm 1. The accessed field is loaded in a register called *value*. If the transaction aborts, either explicitly or due to a conflict, the mutator retries it, after having observed that the busy-bit is unset. We expect aborts to be rare in practice.

Algorithm 2 shows the necessary changes on the GC side, as explained above. The copying activity of the GC is announced by the enabled busy-bit. Care must be taken when disabling the flag: since the object is typically copied byte-by-byte to the new location, the new instance will be marked as busy from the start. Thus, the busy-bit must be disabled on both objects at the end, in order to allow mutators to access them (line 7).

This approach has two immediate consequences:

- The algorithm should allow the moving phase of the collection to take place concurrently with mutator activity due to the non-blocking synchronization through hardware transactions;
- In a practical implementation, we expect transactional operations to add some overhead, which might result in loss of throughput. The overhead comes from the latency of starting and committing a hardware transaction (specifically, the run of the operations XBEGIN and XEND). It is machine specific, but generally comparable to an uncontended atomic `cmpxchg` (atomic compare-and-swap) operation.

7.2.4 Discussion

We need to systematically analyze any possible interleavings or interactions that could result in inconsistent or faulty outcomes. The algorithm has to identify and avoid conflicts between GC activity when copying an object and a memory access to the same object performed by a mutator. In this case, the mutator could access the old version of the object while it is copied, leading either to a read of a stale value, or to a write to the old version of an already copied object. For this reason, the application thread encapsulates any memory access into a transaction.

We identify the following cases:

- *The mutator and the GC work at the same time, on the same memory area: GC starts to copy the object before a transaction is started.* In other words, lines 3–4 in Algorithm 2 happen before line 2 in Algorithm 1. In this case, we further find the following sub-cases:
  - the whole GC function ends before reading object metadata in the access barrier: then there is no conflict, the mutator will follow the newly installed forwarding pointer and finish correctly;
  - once the metadata is read inside the transaction, the mutator will eventually abort either when the forwarding pointer is installed, or when the busy-bit is unset. Upon retry, the mutator will be correctly forwarded to the new copy of the object;
  - the busy-bit check is done while the GC is still in the copy function (line 4): the mutator will explicitly abort and retry.
- *The mutator and the GC work at the same time, on the same memory area: the mutator starts a transaction and reads object metadata, then the GC starts to copy the object.* This is equivalent to lines 3–4 in Algorithm 2 happening between line 3 and line 4 in Algorithm 1. Since the part of the object that contains the busy-bit is already monitored in hardware when the busy-bit is set by the GC, the alteration of this flag will cause the transaction to abort. Upon retry, the mutator will find the GC either busy (and therefore explicitly abort) or finished (thus correctly continuing the memory access);
- *The mutator and the GC work at the same time, on the same memory area: the mutator starts accessing the object, then the GC function starts.* In this case, the application thread finds the object not busy during the check so it proceeds with the access. We identify the following potential interleavings:

  - the mutator finishes before the GC set the busy-bit (line 3 in Algorithm 2), i.e., before the object is altered in any way. This scenario implies a correct successful execution on both sides;
  - the GC sets the busy-bit before the transaction in the mutator is committed. The modification of the busy-bit protected by the transaction aborts the mutator and all the changes done during memory access are rolled back. As before, the GC will first finish copying the object, then the mutator will be free to try again on the correct data.

- *The mutator and GC work at the same time, but access non-overlapping memory areas.* In this case, their executions can run correctly in parallel, each of them being able to finish their work successfully.

After carefully considering all the ways of interaction between application threads and GC in the proposed concurrent algorithm, we conclude that it provides a correct GC functionality, leaving no room for inconsistencies and faults.

7.3 Summary

This chapter provided a detailed description of a novel GC concurrent algorithm based on HTM. We built our solution on top of a typical concurrent GC algorithm. We started by bringing some insight into which components of the chosen base GC play an important role in our algorithm. We elaborated on forwarding pointers and access barriers. The transactional GC algorithm uses the former to identify conflicting accesses to objects, by enhancing the latter with transactions. In short, we presented an algorithm that defines a functional pauseless concurrent GC, where notable stop-the-world pauses are eliminated with the help of hardware transactions.

Chapter 8

Initial Implementation

Contents

Before proceeding with the full implementation of a pauseless GC based on the algorithm presented in Chapter 7, we start by incrementally adding the relevant parts of the algorithm to our target base GC (ConcurrentMarkSweep). **The main goal of our work is to answer the question of whether using HTM in GCs is a viable option.** HTM became only recently part of mainstream processors, so indications on what scenarios would benefit most from it are scarce. As mentioned before, HTM seems to be a perfect match for tackling specialized synchronization problems, such as garbage collection. However, there is no guarantee that this is the case.

Starting from the chosen base GC, a *full implementation* of a pauseless collector consists of the following steps:

1. change parts of the base implementation, as required by the algorithm;
2. add necessary pieces on top of the base implementation;
3. remove previous (blocking) synchronization from the base implementation.

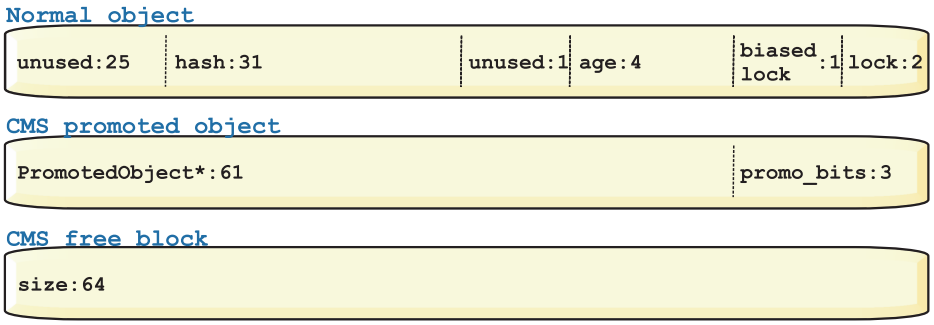


Figure 8.1 – Bit format for an object header, big endian, most significant bit first.

The rest of this work only focuses on evaluating and optimizing the practical implementation of the aforementioned transactional algorithm (steps 1 and 2), in order to lead it to a state that makes a full implementation of the GC worthwhile. If the simple addition of algorithm pieces results in a significant overhead in practice, either because of the incipient form of HTM or other practical reasons, then a full implementation (step 3 above) would be useless. More precisely, we have implemented the transactional barriers, both at the interpreter level and at the JIT level, but we have not modified the CMS algorithm itself: the young generation collector still stops the world, and the old collector is still concurrent without copying the objects. Modifying CMS to fully implement our concurrent moving algorithm would require removing all the GC safepoints and redesigning the initial marking (roots identification) and remarking phases. Hence it would demand a time-consuming implementation whereas potential negative results could already indicate that it may only deliver inferior performance.

8.1 Implementation details

The main additions to the original CMS algorithm concern the busy-bit logic and the access barriers. The theoretical illustration of the transactional barriers seems to have an equally easy and relatively straightforward practical implementation (simply encapsulating each operation in transactions). However, in practice we had to overcome a few difficulties, further described in the sections that follow. The most notable issue is that the transactional algorithm requires read barriers, while, by default, HotSpot only injects write barriers.

8.1.1 Java object's header

While reasoning about the transactional algorithm presented in Chapter 7, we imagined the objects as being composed of a *metadata* part, where information about the object is stored, and the actual *payload*. CMS objects have a conceptually similar composition: a *header* that contains all the details necessary to access and manage the data and the instance *data*. The header is further split into the *mark word* and *metadata*. The latter contains a pointer to the class instance of the objects. The former has the fixed size of a memory word and describes the format of the object. The forwarding pointer is encoded in the mark word. Other information consists of the locking state of the object, age and hash value. The internal structure of the mark word varies based on multiple factors. We only consider objects on 64 bits, no compressed-pointers [2] or

biased-locking [32, 35]. We adopted the latter two options in order to keep the implementation as simple as possible in terms of pieces that can interact or behave unexpectedly. We consider the 64-bit setting acceptable, since this is the norm nowadays both for servers and desktop computers. Figure 8.1 illustrates different mark word layouts. The promoted object layout describes the object after promotion. It mostly consists of a pointer to the promoted object. The last layout, showing a CMS free block, is specific to the objects that are not yet allocated or that are recycled. CMS does not have a compacting old generation collector. It manages free space through lists of free memory chunks, ordered by size, called *freelists*. When dead objects are collected, they are added to a freelist, as CMS free blocks.

8.1.2 Busy-bit implementation

A first approach to implementing the busy-bit involved using one of the unused bits in the mark word. In this way, reading the forwarding pointer and the busy-bit together inside the access barriers would have been straightforward. However, given the various layouts of the mark word, even after eliminating compressed pointers and biased locking, there is no common bit that can be reserved for the busy flag.

Finally, we placed the busy-bit as a separate field of the object header, aligned immediately after the object metadata. This entails the following deviations from the theoretical algorithm: on the one hand, the busy-bit and forwarding pointer are not together at the same address anymore, hence they should be read separately after starting a transaction in the access barriers. On the other hand, reading the busy-bit in order to check its value later in the transaction, implies monitoring the flag independently. The value of the busy bit changes both at the beginning and at the end of the GC task, ensuring by itself alone the correctness guarantees discussed in Section 7.2.4. Thus in the practical implementation, we can safely remove the forwarding pointer read inside the transaction and only rely on the busy-bit.

Several new methods were added to the original implementation of the object header. First, we need to handle the values of the busy-bit. Accepted values are 0 and 1, and they are set atomically. We use atomic operations defined in an architecture-specific library in OpenJDK source code. Since whenever we have to enable or disable the busy-bit its value changes to 1, respectively 0, regardless of its previous value, we used an *atomic exchange* operation instead of an atomic compare-and-swap (CAS). Internally, this essentially translates to the

```
xchgl (val),dest
```

assembly instruction, where *(val)* represents the contents of the variable holding the new value and *dest* is the destination, in our case the busy-bit. This instruction implicitly locks the bus while the operation takes place. The new value is returned.

The busy-bit has to be initialized to 0, i.e., no object is considered busy when it is first allocated. For this, we made sure to identify all types of allocated objects (which can be roughly split into normal objects and array objects) and initialize the bit as appropriate.

Another important method that needed to be defined returns the offset of the busy-bit inside the header. It is based on an already existing function `offset_of(class, field)` that computes the offset of a field in a given class using pointer arithmetics. We will further explain the purpose of this method in the next sections.

Listing 8.1 – Interpreter: busy-bit handling.

```
1   static void do_oop_store(Address dest, Register oop, /*other parameters*/){
      // ....
2     txbegin(L_fallback);
3     Address busy_val(oop, oopDesc::busy_offset_in_bytes());
4     movl(rdx, busy_val);
5     cmpl(rdx,0x1);
6     jcc(Assembler::equal, L_retry_label);
      // copy the oop to destination
7     txend();
      // ....
8     bind(L_retry_label);
9     xabort();
      // ....
10  }
```

8.1.3 Access barriers in the Java interpreter

HotSpot JVM takes advantage of a *template-based interpreter*, containing the translation between assembly and bytecode. Basically, it generates assembly instructions for each bytecode at runtime. Then, it executes any requested bytecode based on the generated template-table. The *template table* is architecture-specific. While a typical interpreter implementation featuring a sort of switch-statement would be significantly slower, the template-based implementation has a considerable complexity, measured in code size (separate implementation for different configurations being necessary), as well as code's inner complexity (support for dynamic code generation makes debugging and improvement notably difficult).

It is important to note that the code that populates the template table is still typical C++ code, while the instructions with which the table is populated are in the assembly language. In order to make it easily readable, the translator provides C++ functions that transparently implement the associated assembly instructions. For example, the C++ function

```
void Assembler::movl(Register dst, Address src);
```

is implemented as

```
emit_int8((unsigned char)0x8B);
emit_operand(dst, src);
```

These operations directly emit the assembly language instruction associated to a simple `mov` between a register and an address.

We worked directly on the code of the template table, the lowest level of interaction with the application threads. First, we modified the main **store operation** (called `do_oop_store`) to handle the necessary parts from our transactional algorithm. The method previously contained various barriers, depending on the GC employed. Since we worked at assembler code level, the busy-bit could not be simply read and tested as in a high-level language. Listing 8.1 shows relevant code snippets. The lines were renumbered compared to the original code, for better visibility. Excluded lines are marked with comments and dots, mostly consisting of variable initialization, saving and retrieving registers on or from the stack, etc. Thus, we start our

Listing 8.2 – Interpreter: starting a transaction.

```
1   void txbegin(Label& L_fallback){
      // push registers on the stack
2     movl(r8, RTMRetryCount);
3     bind(L_rtm_retry);
4     if (RTMRetryCount > 0){
5       decrementl(r8);
6       testl(r8, r8);
7       jcc(Assembler::notZero, L_tx);
        // restore registers
8       jmp(L_fallback);
9     }
10    bind(L_tx);
11    xbegin(L_rtm_retry);
      // restore registers
12  }
```

Listing 8.3 – Interpreter: committing a transaction.

```
1   void txend(){
2     xtest();
3     jcc(Assembler::zero, L_no_tx);
4     xend();
5     bind(L_no_tx);
6   }
```

example at the beginning of the hardware transaction (line 2). We find the exact address of the busy-bit, by computing its offset with the aforementioned specialized static method. We save the word found at that address in a register (called `rdx` in this example) and test if it is equal to 1. The `cmpl` operation is equivalent to a subtraction, where the result is discarded. If the two operands are equal, then the *zero flag* is set to 1 and the jump condition in line 6 evaluates to `true`. If the busy-bit is found 1, the transaction should retry. Therefore, depending on the outcome of the `cmpl` instruction, we either continue with the store operation (line 6), or jump to `L_retry_label`. Line 6 shows that, if we found the busy-bit enabled, we explicitly abort the transaction, roll back the changes, and cause the mutator to restart the transaction. If the busy-bit is not enabled, we proceed with the store operation and commit the transaction (line 7).

Functions `txbegin()` and `txend()` in Listing 8.1 are wrappers over the native functions defined by the assembler that handle transactions, `xbegin()` and `xend()`. The implementation of `txbegin()` is illustrated in Listing 8.2. The native implementation of `xbegin()` takes a label as parameter (see line 11). The label should point to a location in the code where the execution can continue in case of abort. This allows the developer to specify a fallback path, usually required for HTM. In our case, we *bind* the label before the call to `xbegin()`, so that the transaction restarts as needed. We then check if a constant specifying the number of retries, i.e., `RTMRetryCount`, was defined (line 4). If not, we retry for an unlimited number of times. Given the fact that in our case there is virtually no reason for infinite aborts (the transaction cannot overflow and there are no forbidden instructions inside), this implementation would be sufficient. However, we also provide a mechanism that allows for a fallback path in case of excessive aborts. Thus, if a value is defined for our constant, we save it into a register and decrement the value in the register each time the transaction restarts (line 5). While the retry count is greater than 0, the program jumps to the `xbegin()` instruction and keeps trying to start a transaction. If the number of retries is exausted, we jump to the fallback path, hence avoiding to start a transaction anymore. The `txbegin()` function receives as parameter a label that points to a potential fallback path. Since our implementation does not yet remove the original synchronization mechanisms of the original GC and we expect aborts to be rare in

practice, at this point our fallback consists of letting the program to follow its normal execution, instead of the transactional one. However, if all classical synchronization of the GC is relaxed, a more complex fallback path code may be required.

Similarly, Listing 8.3 shows the implementation of the `txend()` function. It basically consists of a check if a hardware transaction is running (line 2). If yes, it commits the current transaction by calling the native function `xend()`; otherwise, it means that the transaction was aborted and a potential fallback path was followed, so the function simply returns.

A similar implementation was used for **load operations**. However, since in the original GC implementation there are no read-barriers injected in the code, we defined our own `do_oop_load()` function to handle the transactional barriers. Then, we pointed all individual load wrapper functions at the newly defined function. More precisely, instead of calling the native implementation of the load function, they call `do_oop_load()`, which, in turn, calls the assembler load implementation encapsulated in transactional barriers. Otherwise, there are no major differences from the store operation implementation presented in Listing 8.1.

8.1.4 Access barriers in the JIT compiler

In addition to the interpreter, HotSpot JVM also benefits from a *just-in-time* (JIT) compiler. The JIT compiler provides faster and more efficient machine code generation. It typically handles "hot", performance critical, methods, by optimizing their execution. Basically, the methods corresponding to this criteria are compiled to machine code directly, and not interpreted at runtime anymore. The JVM is able to identify what methods would take advantage the most from being compiled by continually monitoring the code and looking at *method entry* and *loop back branches* counts. The threshold over which a method or loop is considered performance critical is set at run-time. HotSpot JVM features two different compilers:

- **C1 (client) compiler:** recommended for applications with a graphical interface, it provides a quick startup and strong optimizations;
- **C2 (server) compiler:** typically used for long-running server-side applications, it provides even more aggressive optimization and better overall performance.

The two modes of operation can be individually enabled at run-time, or they can be used together in what is called a *tiered compilation*. We implemented our transactional barriers on top of the C2 compiler. The reason for this decision is the end goal of testing on real-life large-scale server applications, such as Cassandra DB.

While the client compiler relies on basic blocks for building a control flow graph, the server compiler uses a data structure as intermediate representation (IR) [75]. The resulting graph is more complex, having both *data* and *control dependence* edges from the definition of a value to its usage. The compiler starts by parsing the bytecode and creates a platform-independent representation, the *Ideal* graph. A number of optimizations are applied at this stage, e.g., global value numbering, dead code elimination, etc. The next phase splits the Ideal graph into subtrees and creates architecture-specific files containing instruction definitions and their costs. The costs are then used for ordering instructions inside basic blocks. The basic blocks that are thus created are used for building a control flow graph, called the *MachNode graph*. The next step represents register allocation which assigns physical registers through a graph coloring algorithm. Finally, machine code is emitted by iterating over the newly-formed graph.

Listing 8.4 – Transactional barriers specification in the architecture-description file.

```
instruct membar_txbegin() %{
  match(MemBarTxBegin); // link to the associated MemBarTxBeginNode class
  format %{ "xbegin␣L_rtm_retry\n\t"
            "L_rtm_retry:␣(membar)\n" %}
  ins_encode %{
    // define labels and save registers
    // the same fallback logic as in the Interpreter (see Listing 8.2)
    xbegin(L_rtm_retry);
    // restore registers
  %}
  ins_pipe(pipe_slow);
%}

instruct membar_txend() %{
  match(MemBarTxEnd); // link to the associated MemBarTxEndNode class
  format %{ "xend␣(membar)\n" %}
  ins_encode %{
    // define labels and save registers
    xtest();
    jccb(Assembler::zero, L_no_tx);
    txend();
    bind(L_no_tx);
  %}
  ins_pipe(pipe_slow);
%}
```

For installing our transactional barriers into the C2 compiler code, we modified its parsing phase, the architecture-specific file that holds instruction definitions and added new nodes into the Ideal graph. The implementation of the transactional barriers in the JIT is less elaborate than the one in the interpreter. Given the complexity of the C2 compiler, we first completed the barriers in an incipient form, without busy-bit handling or explicit aborts. This served the purpose of estimating the potential cost of these barriers even in their simplest form.

We allocated two new graph nodes for our barriers. All nodes are implemented as instances of a subclass of the `Node` class. The edges between nodes are implemented simply as pointers. The inputs for each node are represented as an array of `Node` pointers. There is only one output, consisting of either a single value, or a tuple of results. The compiler already had nodes of the type `MemBarNode`, employed for executing memory barriers. We extended this type and defined `MemBarTxBeginNode` and `MemBarTxEndNode`, by simply further calling the constructor of a typical `MemBarNode`. The barriers are inserted during the parsing phase of the compiler. More precisely, when the bytecodes for load and store operations are encountered and the Ideal graph is built, we add our custom Nodes.

The instructions executed by the new Nodes are defined in an *architecture description* file, using a specific language. We modify the description file for the AMD64 architecture. The file generally contains definitions for registers, encoding classes, necessary C++ functions, operands, resources, architecture's pipeline behavior and all kinds of instructions (branch, move, add, etc.). The description of instructions has the following configuration blocks:

- **match** – states which machine-independent subtree may be replaced by this instruction;
- **ins_cost** – the estimated cost of this instruction is used by instruction selection to identify a minimum cost tree of machine instructions;
- **format** – a string describing the disassembly for this instruction;
- **ins_encode** – the code executed by the defined instruction, given as a list of encoded classes with parameters;
- **ins_pipe** – specifies the stages in which input and output are referenced by the hardware pipeline.

Our implementation of the transactional barriers in JIT is illustrated in Listing 8.4. We omit from this listing some of the parts already explained in previous sections. Blocks of instructions are defined between pairs `%{` – `%}`. First we correlate our instruction with the previously defined Node in line 2 and, respectively, line 13. Then we define the contents of the instruction block (lines 5 – 8 and 15 – 20). Finally, we instruct the pipeline to force serialization with the `pipe_slow` class. Based on this file, our transactional barriers code is generated for every load and store operation compiled by the JIT.

8.2 Early results

Having a first implementation of our transactional barriers ready, we proceed to briefly test this version in order to learn what performance penalty they introduce. For testing we use an Intel i7-5960X CPU @ 3.00 GHz (Haswell) with fully integrated HTM support. We experiment with the DaCapo benchmark suite. Specifically, we run the subset of benchmarks that we discovered to be stable (see Section 6.1.2). All selected benchmarks have small variations (less than 5 % in most cases) for both the total execution time and for the final round. One exception is represented by the *batik* benchmark, which is only stable for the total execution time, varying with more than 10 % for the final iteration. Since we only focus on the execution time of the actual run of the benchmark (the final round), we remove this application from our subset.

We start by measuring the overhead introduced by the read and write barriers and the execution time in interpreter mode of the last iteration of each benchmark with the new implementation. We gather statistics using the `perf stat` Linux tool for the following events: `tx-start`, `tx-abort`, `cycles` and `cycles-t`. The command outputs the counter values for how many transactions were started during the benchmark execution, how many of these were aborted, the total number of CPU cycles and the number of cycles inside transactional regions. Based on the first two counters we compute the percent of aborts for each benchmark, while from the last two numbers we obtain the percent of transactional cycles. We execute each benchmark 20 times. We measure the overhead in the following situations:

- transactional barriers are disabled (the default access barriers are in place), vanilla ConcurrentMarkSweep is running;
- only write transactional barriers are enabled (there is no read barrier, same as in the default implementation);
- only read transactional barriers are enabled;

Table 8.1 – Independent overhead for transactional read and write barriers (interpreter).

Benchmark		h2	tomcat	xalan	jython	pmd	luindex
Execution time (s)	no-tx	127.67	8.44	5.05	76.97	8.04	11.82
	write-tx	128.79	8.48	5.1	79.19	8.41	11.97
	read-tx	172.87	9.56	6.11	94.33	9.49	15.13
	all-tx	174.77	9.8	6.18	96.72	9.86	15.33
Overhead (%)	no-tx	–	–	–	–	–	–
	write-tx	0.88	0.49	0.99	2.88	4.6	1.26
	read-tx	35.4	13.33	21.01	22.55	18.04	28.06
	all-tx	36.89	16.16	22.51	25.65	22.65	29.74
Tx-cycles (%)	no-tx	–	–	–	–	–	–
	write-tx	2.93	2.23	1.75	4.96	6.85	3.52
	read-tx	49.52	30.4	27.41	35.58	27.97	39.95
	all-tx	50.93	31.52	28.18	37.98	30.88	41.99
Tx-aborts (%)	no-tx	–	–	–	–	–	–
	write-tx	0.12	0.38	0.67	0.03	0.25	0.05
	read-tx	0.04	0.75	0.74	0.02	0.34	0.06
	all-tx	0.02	0.7	0.69	0.04	0.32	0.01

- both types of transactional barriers are enabled.

Table 8.1 illustrates the overhead of the read/write barriers for the tested benchmarks. The reported values are obtained as the mean over 20 runs. First of all, we observe that the write barrier has an almost negligible effect on the performance of the application in interpreter mode (less than 5 % in all cases). However, the read barrier raises the overhead to an unacceptable level. The simple addition of the read barriers increases the execution time with more than 10 %, going up to 35.4 % overhead. This result is closely connected to the percent of CPU cycles executed in transactional regions. When the read barriers are enabled, more than 25 % of the total cycles are inside transactions. In the case of the *h2* benchmark around 50 % of the cycles are transactional, which leads to an execution time with ∼35 % longer than the original implementation.

We also notice that the proportion of transactional aborts is close to 0 in all cases. This result confirms our assumption that aborts would be a rare event in our approach. However, it also indicates that the overhead we observe comes from the implementation of the algorithm itself, rather than the amount of aborts. In fact, this outcome was to be expected, since we added extra barriers for all load accesses. We also have to take into account that the content of the transactions is relatively small, e.g., a simple load instruction. The consequence is twofold: on the one hand, there are no overflow aborts; on the other hand, the contents of the transaction may not entirely amortize the cost of starting and committing it. These observations together with the additional barriers inserted in the code lead to a prohibitively large overhead.

Figure 8.2 illustrates more visibly the difference between the overhead introduced by the read barriers and write barriers for all DaCapo benchmarks (on the X axis). The overhead is

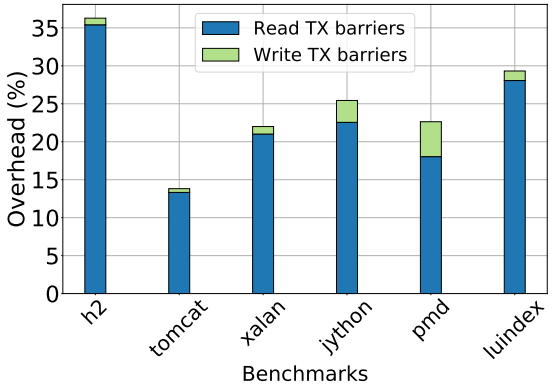


Figure 8.2 – Overhead of transactional read barriers and write barriers, enabled individually.

marked on the Y axis. Each of the stacked bars represents the value of the overhead for a type of barriers (they are not cumulative). We only show the overhead of read and write barriers when enabled separately. We observe that the overhead of write barriers is mostly negligible compared to that of read barriers.

8.3 Summary

There are two different approaches for implementing a GC with HTM support: one can either instrument the GC or the application accesses. In our algorithm we followed the latter approach, and replaced the blocking access barriers of the mutators with transactions. We built our implementation on top of HotSpot's ConcurrentMarkSweep collector. We implemented the transactional barriers in the interpreter and in the JIT compiler. Before embarking on fully implementing a new GC, we focused on assessing the overhead introduced by our transactional implementation. The goal was to verify if the approach is feasible or not. We performed preliminary testing on a widely-used benchmark suite, DaCapo, using the most stable benchmarks in terms of execution time. We found that by encapsulating all read and write accesses in transactional barriers, the overhead goes up to 37 % in interpreter mode. The overheads indicated by our early experiments would make a transactional GC inferior to any existing GC from the start. As such, we considered critical to further optimize the performance of the transactional barriers before continuing with a complete and thorough evaluation.

Chapter 9

Optimized Implementation

Contents

Preliminary testing of the first version of transactional barriers indicates a significantly increased overhead over the original implementation of the base Java GC, ConcurrentMarkSweep. At this point, a fully-working GC implementation featuring transactional barriers would clearly have much lower performance than any state-of-the-art GC. Therefore, we further focus on improving the transactional barriers with the aim to reduce the overhead to an acceptable level. This chapter presents two optimizations that we add incrementally to the current implementation. The optimized barriers are evaluated in the next chapter.

9.1 Volatile-only read barriers

Previous observations indicated that the extra read barriers added to the original implementation are the main source of latency for the new GC (see Section 8.2). As such, we first attempt to decrease their overhead. We leave write barriers aside at this point, since their overhead is already negligible. There are various characteristics that could influence the performance of

Listing 9.1 – Interpreter: volatile load check.

```
1   movl(reg, flags);
2   shrl(reg, is_volatile_shift);
3   cmpl(reg, 0x1);
4   jcc(Assembler::notEqual, L_notVolatile);
5   do_oop_load(dest, src, 0x1);
6   jmpb(L_afterVolatileSet);

8   bind(L_notVolatile);
9   do_oop_load(dst, src, 0x0);

11  bind(L_afterVolatileSet);
```

transactional read-barriers: the size, the contents, their frequency. Out of these, the last one is the only feature that can be improved. The other two have already been addressed when the barriers were first implemented: the transaction typically encapsulates the smallest unit possible, i.e., a simple load machine instruction. Consequently, we tried to optimize (i.e., to reduce) the frequency of the transactional read-barriers.

9.1.1 Preliminary observations

The transactional algorithm presented in Chapter 7 assumes access barriers for all load and store operations. This appeared as necessary in a multi-threaded context, in order to avoid stale values. More precisely, when the GC works concurrently with the mutators, there is a possibility that the mutator accesses an old copy of an object that is just being moved by the GC at the same time. Our current algorithm design does not permit this kind of outcomes.

The *Java memory model* (JMM) [43, Chapter 17] represents a set of specifications describing threads' interaction with the memory. It provides strict rules concerning single-threaded executions, as well as multi-threaded scenarios. In the case of a single thread, it allows for any kind of compiler optimizations, instruction reordering, etc., as long as it respects *as-if-serial* semantics. When multiple threads are involved, the rules are more complex, accounting for the fact that threads can have different views of the same data if not synchronized properly. Thus, the JMM defines concrete guidelines on what values are allowed to be returned when data is read.

Given the practical implementation of our algorithm in HotSpot JVM, its execution needs to comply with the JMM. At this point, with all the barriers in place as required by the algorithm, the current implementation enforces a stricter order than required. The JMM dictates that only volatile reads must preserve the order of the operations, i.e., they must always return the last value written to a variable. All other loads are also allowed to return an older value. In most applications, volatile reads represent only a fraction of the total read accesses. Thus, inserting transactional read barriers only for volatile reads considerably reduces the incidence of additional barriers compared to the original implementation, while still respecting the JMM.

9.1.2 Optimization details

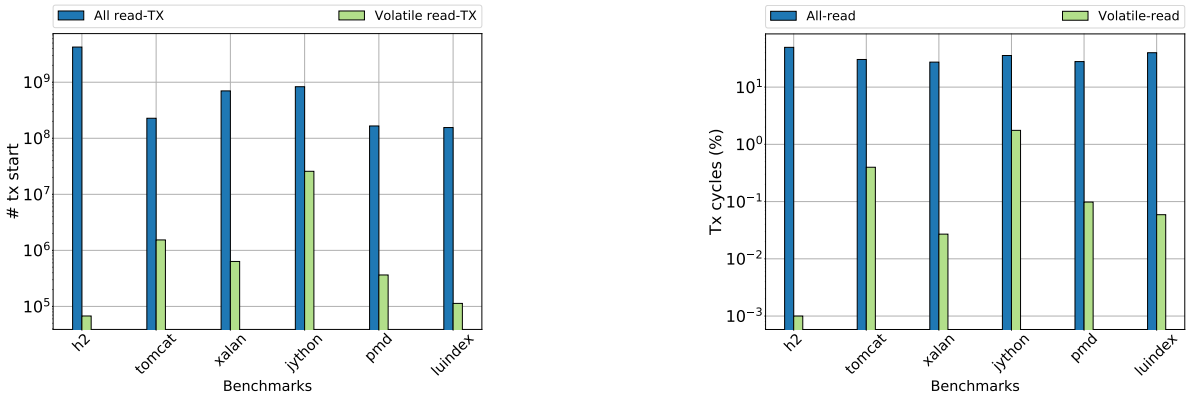
We modify our first implementation to identify volatile accesses for all load operations. We start with the alteration of the interpreter. The new `do_oop_load()` function receives an additional argument that indicates whether the current read is volatile (argument equal to 1) or not (argument equal to 0). The volatile check illustrated in Listing 9.1 is added to the current implementation to call the load function with the correct argument value. The volatile state is encoded as a flag. In this additional code snippet we retrieve the flags associated with the object of interest and obtain the value of the volatile flag (by executing a bitwise shift with a predefined number of positions, called `is_volatile_shift`), as shown in line 2. Then, the state of the flag is checked. If not set, the execution jumps further in the code and calls the load function with the volatile parameter set to 0 (lines 8 – 9). If we are dealing with a volatile read, the execution continues without branching (line 5), loads the object while protecting it with transactional barriers, and jumps at the end of the snippet. Thus, with this mechanism in place for all object loads, we reduce the occurrence of the transactional read barriers to a fraction of times compared to the first implementation.

Implementing the volatile optimization in the JIT compiler is even more straightforward than in the interpreter. During the parsing phase volatile checks are already in place for loading and storing fields of objects. These checks internally verify the same flags that we manually fetch for the check in the interpreter. We employ the available methods to check if a particular load operation applies to a volatile field and only insert transactional barriers if this is the case.

9.1.3 Optimization validation

Our optimization is based on the idea that volatile reads are significantly fewer than non-volatile reads. In order to validate this assumption and the efficiency of the optimization, we compare the amount of transactional cycles with and without the optimization, as well as the number of started transactions. We measure these features in the same conditions as the preliminary experiments on the unoptimized version of the transactional barriers (see Section 8.2). More precisely, we test on the DaCapo benchmarks, with one iteration per benchmark. We run the benchmarks on an Intel Core i7-5960X (Haswell) CPU at 3.00 GHz and 32 GB RAM. We only showcase the interpreter implementation.

Figure 9.1 shows the aforementioned transactional features for the first version of the transactional barriers (i.e., inserted for all read accesses) and in the case of the volatile optimization. More precisely, Figure 9.1(a) illustrates the number of transactions started in each case, while Figure 9.1(b) represents the percent of CPU cycles that are executed inside transactions. We choose to represent the data on a logarithmic scale (on the Y axis in both figures) because of the considerable difference between the results in the two implementations. The volatile optimization reduces by 2–3 orders of magnitude both the amount of transactions and the percent of transactional cycles. Depending on the frequency of volatile operations for each application, some benchmarks exhibit a negligible (close to 0) percent of cycles in transactional regions. Another interesting observation is that, while the benchmark results are comparable for transactional barriers for all reads, they become significantly different when only volatile reads are taken into account. This result in not surprising: having benchmarks of the same proportions, they execute read accesses in the same order of magnitude; however, the amount of volatile loads depends on the internal structure of each application. Hence the variation between benchmarks.



(a) Tx start

(b) Tx cycles

Figure 9.1 – Transactional parameters when inserting read-barriers for all reads vs. for volatile reads only (Java interpreter).

This quick evaluation confirms that the proposed optimization greatly reduces the incidence of the transactional read barriers in the execution of the application, while still respecting the JMM. Thus, we expect the overhead of our algorithm to decrease proportionally.

9.2 Selective transactional barriers

We further try to improve the performance of transactional barriers with another optimization. We propose *selective access barriers* to reduce the remaining overhead of the previous algorithm to an adequate value. We call them *selective* because they select a particular path depending on the state of the execution. Essentially, the path is chosen based on whether the GC is running or not. This optimization is orthogonal to the one presented in Section 9.1. Its main goal is to further reduce the frequency of transactional barriers to a minimum: they are entirely removed during typical stand-alone application execution, while being inserted during concurrent execution with the GC *for volatile reads only*. The expected result is close-to-zero overhead when the GC is not running and a manageable overhead otherwise, amortized by the lack of stop-the-world pauses when collecting.

9.2.1 Algorithm

The general idea is straightforward and easy to integrate with the previous version of the algorithm. We define a global flag called the *in-collection flag*. When the GC becomes active, we set the in-collection flag, wait for each thread to acknowledge the change, then proceed to the collection. On their side, application threads have a check for the value of the flag for each load or store operation. Based on this value, they choose to follow either *the slow path* (featuring transactional barriers) or *the fast path* (normal execution). However, their acknowledgment, when the flag is set, is critical for the correctness of the algorithm. Let us consider the following scenarios:

- *GC sets the in-collection flag and immediately proceeds to copy the object.* It is possible that some of the mutators just passed the aforementioned in-collection check and found the

flag unset. In this case they will continue on the fast path. However, the fast path is not suitable for concurrent execution with the GC, since mutator accesses are not protected in any way from interacting with the collection. Thus, this scenario is error-prone and could lead to inconsistent results;

- *GC sets the in-collection flag, waits for acknowledgment from mutators, then copies the object.* In order for all threads to be aware of the modification, they are forced to traverse a short safepoint. The safepoint is launched by the GC before it starts operating on objects. The GC only starts copying the objects after the safepoint ends, when mutators are also allowed to continue their work concurrently. At the time when mutators are released, the in-collection flag is already set and visible for all threads. This measure guarantees that the application threads will take the slow path whenever they work concurrently with the GC. Then, when the GC finishes its intervention, we have the following sub-cases for unsetting the in-collection flag:
  - *GC unsets the in-collection flag, awaits for acknowledgment from mutators.* This corresponds to introducing another short safepoint at the end of the GC activity. Similar to enabling the in-collection bit, the GC unsets it, stops the world and at the end of the pause the new value of the flag will be visible for all mutators. Thus, mutators can directly resume the usage of the fast-path;
  - *GC unsets the in-collection flag, stops execution.* Mutators are not expected to notice the value change immediately. They are allowed to continue on the slow-path for another few iterations before all of them go back on the fast-path. This approach is correct, since it translates to extra synchronization, rather than lack thereof (as it is the case when enabling the flag without a safepoint).

After carefully analyzing the above options, we select the following combination: in order for all threads to be aware of the modification, we force them to traverse a short safepoint. Since the correctness of the algorithm does not necessarily require this check point at the end of the collection, we consider the potential overhead of a few extra slow-path invocations to be smaller than a safepoint. After all of the application threads "checked in", they can continue running concurrently with the GC using the slow path featuring transactional barriers. Application threads need to check if the flag is set before every load and store access, so that they take the correct path. However, the flag typically stays in the cache of the cores, hence this check does not introduce a significant overhead.

9.2.2 HotSpot JVM internals

The first step towards the implementation of the selective barriers was an in-depth study of the VM internals and GC functioning. This is necessary in order to accurately identify when the GC is considered to be working, how it is launched and stopped and at what level it is best to insert our modifications.

In a nutshell, all GC operations (and any other operation that needs to be run at a safepoint) are executed by the VM thread. There is only one VM thread running at any point during the execution of the application. Application threads can request the VM thread to execute a particular operation, blocking until it is done. The VM thread employs a synchronized queue to schedule the operations, inserted in the queue in no particular order (i.e., sequential requests

Listing 9.2 – VM thread main loop.

```
1  void VMThread::loop() {
2    while (true){
3      op = VMOperationQueue.get_next();
4      if (op.runs_at_safepoint()) {
5        SafepointSynchronize::begin(); // start safepoint
6        evaluate_operation(op);
7        SafepointSynchronize::end(); // end safepoint
8      }
9      else
10        evaluate_operation(op);
11     VMOperationRequest_lock.notify();
12   }
13 }
```

Listing 9.3 – GC operations scheduling in the VM thread.

```
1 void VMThread::execute(VM_Operation op) {
2   op.doit_prologue();
3   VMOperationQueue.add(op);  // schedule operation
4   VMOperationRequest_lock.wait(); // wait to be released by the VM thread
5   op.doit_epilogue();
6 }
```

from different threads can be interleaved). Then, in a loop lasting as long as the application is running, the operations are removed from the queue and executed according to requirements: if a safepoint is needed, all mutators are subsequently suspended and the current operation executed in isolation; otherwise, the operation runs concurrently with the other threads. The pseudocode in Listing 9.2 shows these steps executed by the VM thread.

All GC-related operations that can be scheduled by the VM thread require a safepoint. Typically, scheduling an operation by a Java thread follows the sequence of steps in Listing 9.3. The execution of GC operations is surrounded by helper functions called the *prologue* and *epilogue* of the execution (lines 2 and 5). They are needed for specific settings before and after the collection. The mutator schedules its operation in the queue of operations and waits until the operation is evaluated. It is released by the VM thread after the evaluation (Listing 9.2, line 11).

The GC operation corresponding to the collection that we want to make concurrent (the young generation moving objects between spaces or promoting them to the old generation) is called `VM_GenCollectForAllocation`. It is scheduled by an application thread when it is unable to allocate memory, i.e., the allocation request is bigger than the available space in the young generation. It is associated to the GC cause called `_allocation_failure` and it usually triggers a young generation collection. Along the path from scheduling the operation to the actual GC collection, the execution passes through multiple phases: while mutators are suspended, the ParNew collector launches multiple tasks and worker threads; the GC threads first process the roots and then try to copy the object to a survivor space. If the copy does not succeed, the object is promoted to the old generation.

Listing 9.4 – In-collection flag check.

```
1 movptr(rdx,ExternalAddress((address) & (TxGCGlobal::inCollection)));
2 cmp(rdx,0);
3 jcc(Assembler::equal, L_continue_label);
```

9.2.3 Implementation details

We define a new class covering the necessary logic behind the in-collection flag. The flag is a field of the class, being handled through a few member methods, that initialize, get and update it as required. Then, we consider the two levels where in-collection flag logic has to be added: GC side and application side (both in the interpreter and in the JIT, as before).

As previously mentioned, the execution of the `VM_GenCollectForAllocation` GC operation is delimited by a *prologue* and an *epilogue* method. They further call two functions named `notify_gc_begin()` and `notify_gc_end()`, which clearly mark the starting and final points of the collection. We piggyback our solution on these methods, by setting the in-collection flag in the former method and unsetting it in the latter. This way, we make sure that the in-collection flag is enabled during the GC execution, not sooner or later.

The application threads check the in-collection flag for every load and store operation. The assembly language implementation is similar for the interpreter and JIT. Listing 9.4 shows how the check is implemented. This fragment is inserted at the beginning of the load and store barriers (Listing 8.1), in order to jump over their execution when the GC is not active, i.e., take the fast path. The value of in-collection is obtained in line 1. Threads can safely check the value concurrently without risking to find an inconsistent value during the GC execution, since the flag is only modified by the VM thread at a safepoint. If the mutator finds the flag set to 0 (no GC is running), it avoids the barrier and advances directly to the memory access operation (line 3).

This implementation guarantees the removal of any transactional overhead when the GC is not active. When the GC is running, the algorithm described in Chapter 7 allows the application threads to continue their execution and ensures correctness. It follows that the same number of operations are spread uniformly over a shorter execution time, that now excludes the blocking phases (during which no operations are executed). The concurrent operations have an overhead, but we expect it to be amortized by the increased concurrency. These considerations are further analyzed in Chapter 10, followed by an in-depth discussion over the feasibility and utility of a full implementation of the transactional GC.

9.3 Summary

This chapter presented two optimizations that aimed to improve the performance of the original HTM-based GC algorithm. Both optimizations focused on reducing as much as possible transactional accesses, especially for load operations. As such, we confined the use of transactions to the concurrent portions of the execution, where they are absolutely necessary for a correct and consistent outcome. As permitted by the JMM, we only protected volatile read operations with transactions. Since the optimized algorithm reduces the overhead of the transactions to a minimum while it increases concurrency, we expect performance to be greatly improved compared to our preliminary results (Section 8.2).

Chapter 10

Evaluation

Contents

After having optimized the transactional barriers, we re-evaluate their overhead. This chapter describes in detail the experiments and the results. It shows that the first optimization (volatile reads) alone is not enough to make the new GC algorithm feasible and, thus, the second one (selective barriers) is necessary. We estimate the performance of a transactional GC implementation with selective barriers and we find it would be comparable with that of the ConcurrentMarkSweep collector. Finally, we briefly consider several challenges in implementing the transactional GC.

10.1 Experimental setup

All experiments described in this chapter are run with the following configuration: 8-core (16 threads) Intel Core i7-5960X CPU @ 3.00 GHz (Haswell), 32 GB RAM, with fully integrated HTM support. We enable separately the interpreter alone or both the interpreter and the JIT

compiler together for our tests. In all experiments we run the original and the modified versions of the CMS collector of Java HotSpot in OpenJDK8. We evaluate the potential improvement of our optimized GC over the initial transactional implementation with a series of experiments, in two scenarios: a benchmark suite and a client-server system.

As benchmark suite we use the six DaCapo benchmarks that we discovered to be stable, as described in Sections 6.1.2 and 8.2. Each benchmark has ten iterations (nine warm-up rounds and the actual run). All reported results represent the mean over 20 consecutive runs. Each benchmark uses 8 concurrent threads for running its workload (equal to the number of cores on the machine).

For the client-server scenario, we run the Apache Cassandra 3.9 database server with the YCSB client 0.12.0 benchmarks (described in Sections 6.1.3 and 6.1.4). The benchmarks have two phases: load and run. When *load* is selected, the benchmark simply inserts new records in the database (100 % write). For the *run* phase, the benchmarks define different combinations of read, update, insert operations as described by Cooper et al. [25]. We configure the benchmarks to execute a fixed number of 10^7 operations on 100 client threads. We consider the results for *load* in general and for the *run* phase of each benchmark separately. All results represent the mean over 5 runs. Between any two runs of the benchmarks, the server's page cache is cleared, the database files are erased and the database is populated again. This is necessary in order to start all benchmarks with the same state and have reproducible experiments.

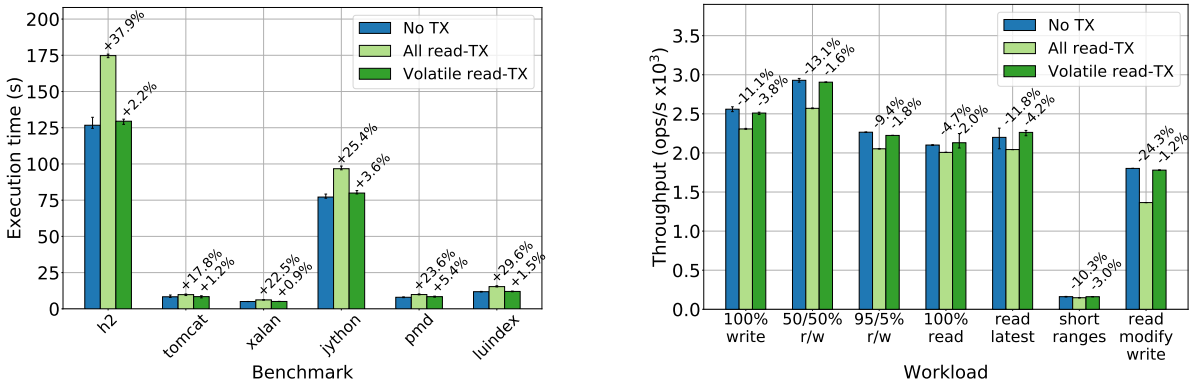
10.2 Experimental results

First, we focus on evaluating the overhead of the barriers optimized to use transactions only for volatile reads. This is essential in order to understand to what extend the second optimization is necessary for an acceptable GC implementation. It also shows the methodology and the exact steps we followed during the development of transactional barriers. In what follows we use the term *optimized* barriers to denote the barriers that are applied only for volatile reads. After the second optimization (Section 9.2) the barriers are called *selective* barriers. The evaluation of selective barriers is addressed in the next section.

10.2.1 Transactional barriers overhead in the interpreter

In Section 8.2 we presented the overhead of the unoptimized transactional barriers in the Java interpreter. These preliminary results indicated a significant slowdown due to the barriers, which convinced us that optimizations are needed before further testing them with the JIT compiler. Thus, we start with the evaluation of the optimized transactional barriers in the interpreter, in order to be able to directly compare the results with our early experiments. The interpreter is enabled with the Java runtime option `-Xint`.

Figure 10.1(a) illustrates the execution time of the DaCapo benchmarks for the original implementation, the unoptimized transactional barriers and their optimized version. The benchmarks are configured as indicated in Section 10.1. The X axis shows the mean execution time as reported by each benchmark, in seconds. Longer execution time means a slower application, thus in this plot lower bars are better. We represent the minimum and maximum values for the execution time with error bars. The implementation featuring transactional barriers for all read and write accesses is called *All read-TX* in the figure, while the optimized implementation is called *Volatile read-TX*. All write accesses are encapsulated in transactional



(a) DaCapo (b) Cassandra

Figure 10.1 – Overhead of transactional barriers in the Java interpreter, unoptimized and optimized versions.

barriers in both cases. The overhead of the transactional implementations is marked above the bars, making it easier to grasp the difference in performance and determine if the new overhead is acceptable. Thus, we clearly observe the positive effect of the volatile-only optimization: the overhead was reduced to under 6 % for all benchmarks. The difference is significant, since the initial overhead was ∼25 %, going up to 37 % for one benchmark. This result shows a pronounced improvement over the initial implementation of the algorithm.

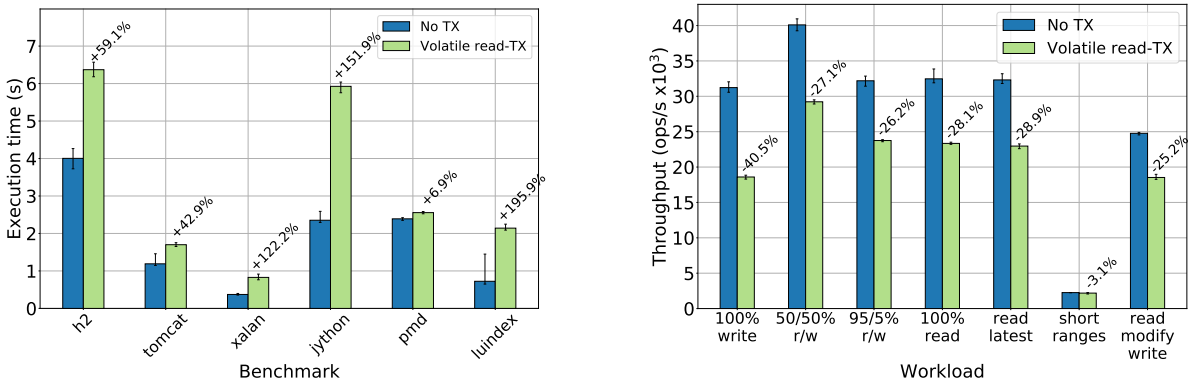
After finally having a suitable performance in the Java interpreter for the DaCapo benchmarks, we also introduced the Cassandra server in our experiments. Figure 10.1 shows the throughput of the application for the original GC implementation, the transactional version and the optimized barriers. We look at the impact of transactional barriers on the client-server communication. Greater values for the throughput are better, meaning that the fixed number of operations was executed in less time. We observe that transactional barriers for all reads (*All read-TX* in the figure) have a serious effect on the client's throughput, with the majority of the workloads incurring more than 10 % overhead. The overhead goes up to almost 25 % for the *read-modify-write* workload. As in the case of DaCapo benchmarks, the only-volatile optimization (represented by *Volatile read-TX* in the figure) considerably reduces the overhead. After the optimization all the workloads have a throughput overhead of less than 5 % in the interpreter.

In conclusion, the optimization proves to be effective in both testing scenarios, lowering the overhead of our HTM-based barriers to acceptable values in interpreted code.

10.2.2 Transactional barriers overhead in the JIT

We further test our volatile-only barriers in the JIT compiler. Basically, we execute our experiments without the Java option that forces the execution to be interpreted, thus using the default combination of interpreter and JIT. We enable transactional barriers for both the interpreted and compiled code.

These measurements allow us to compare the relative overhead of our implementation in the interpreter with the results for the JIT implementation. Running interpreted code alone has a larger latency than when it is combined with compiled code. However, the latency of starting and committing transactions is constant, it does not depend on the way the code is



(a) DaCapo (b) Cassandra

Figure 10.2 – Overhead of transactional barriers in the JIT compiler, unoptimized (no TX) and optimized (volatile read-TX) versions.

executed. Therefore, the transactional overhead appears smaller in the interpreted execution. Despite showing a reasonable overhead in the interpreter, we expect the actual overhead of volatile-only barriers to be significantly greater. In what follows we exclude the unoptimized version of the transactional barriers from our experiments, as having too high an overhead even for the interpreter-only tests. Section 10.2.1 proved that the volatile-only optimization is necessary and can be further considered as the only transactional implementation worth using for evaluation. Moreover, we consider that the latency in the JIT corresponds to the real value of the overhead, because this is the default way of executing real-life Java applications.

Figure 10.2(a) confirms our intuition. First, it is clear from the values on the Y axis that there is a considerable difference in the execution times with the interpreter alone and with the JIT. In some cases, the execution in the JIT is with two orders of magnitude faster. However, the fairly constant transactional overhead, together with a shorter execution time, result in prominent overheads. The transactional execution time for half of the benchmarks is doubled or almost tripled in the case of *luindex*. Only one benchmark (*pmd*) has the overhead under 10 %. This figure proves that in real-life conditions, even optimized, our algorithm can still introduce prohibitively large delays.

Further experimenting in a large-scale setting, with the Cassandra server, we observe that the overhead is not so pronounced as for the DaCapo benchmarks (Figure 10.2(b)). However, for 6 workloads out of 7, the throughput on the client is reduced by 20 % with the optimized barriers compared to the original CMS implementation. For a database server working in real-world conditions, this represents a significant impact on the system's overall performance. An interesting case is the *short-ranges* workload, which does not suffer from a considerable change in overhead when switching from the interpreter to the JIT. This is the only write-intensive workload where short ranges of records are queried, instead of individual records. However, the CPU utilization is greatly increased in the JIT executions. Basically, in this particular case, the structure of the workload favors more work to be done in parallel on the server, instead of impacting the communication with the client. This particularity of the *short-ranges* workload is more evident in the following section.

After enabling the JIT compiler in our experiments we observe that the optimized version of the transactional barriers is insufficient for the default Java configuration for real-life applications.

This leads us to the conclusion that we need to remove all unneeded transactional accesses, thus devising selective barriers.

10.3 Selective barriers evaluation

The main goal of this group of experiments is to pinpoint the potential performance gain with the selective barriers. More precisely, we measure how many transactions (that are costly) we save in all scenarios. For simplicity, in what follows, we define the transactional overhead as the difference between the execution times of a GC implementation that uses transactional barriers and the original implementation of that GC. Typically, if a memory access (either load or store) is encapsulated in a transaction, it has a longer execution time. This delay contains the time to start and commit (or worse, abort) a hardware transaction, in addition to the overhead generally associated to operations executed inside transactions. If we sum up all overheads incurred by individual memory accesses, we obtain the transactional overhead associated to an application. Since the individual overheads vary based on many factors, we settle for the minimum common overhead for all accesses, that of starting and stopping a hardware transaction. Based on this, we aim to give an intuition of how much of the initial transactional overhead is eliminated by simply removing the measured number of transactions when enabling the selective barriers. Thus, we find an upper bound for the transactional overhead of the selective barriers. Further, we compute an estimation of the actual CPU time of a transactional GC implementation using selective barriers based on our experimental results.

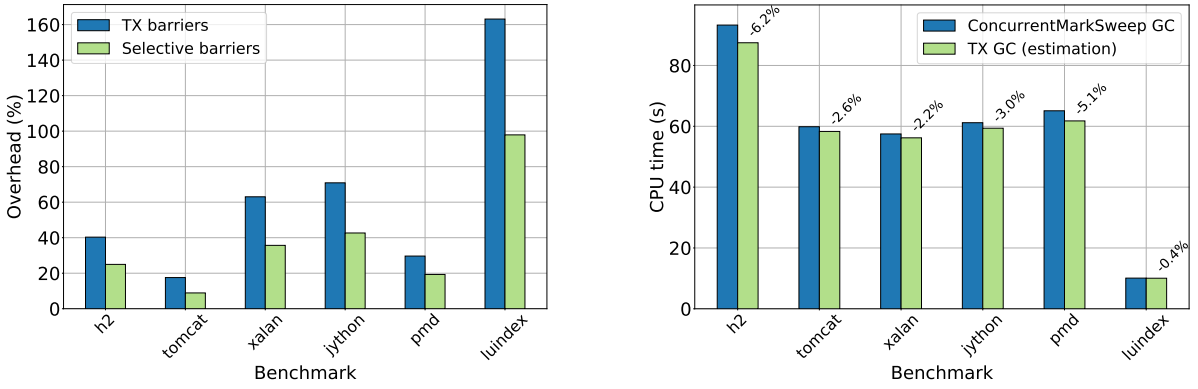
10.3.1 Transaction duration

We define the cost of a hardware transaction as the time it takes to start and commit the transaction. The minimum run time of an empty transaction gives us a lower bound for the cost. This helps us estimate a lower bound for the overhead eliminated by the selective barriers; this means that the savings are even more significant in practice than in our experiments.

Given the fine measurement we need to do, we rely on the CPU's time-stamp counter, through the native `RDTSC` (read time-stamp counter) instruction, which has nanosecond accuracy. In order to have a stable and conclusive result, we repeatedly start and stop an empty transaction in a loop of 10^9 iterations and measure the total time of the loop. From this, we subtract the time it takes to process the loop itself, with no body. The transactional operations (begin and end) are inserted in assembly language, to reduce the overhead of function calls. We also pin the thread to one of the cores and we fix the frequency of the CPU to the maximum frequency admitted, 3 GHz. The results represent the mean over 50 runs. Thus, we consider that a hardware transaction takes 29.92 CPU cycles and 9.97 ns, on average, on our Intel Haswell machine. We use these values to estimate the savings in terms of execution time provided by the selective access barriers in various scenarios.

10.3.2 Benchmark suite

First, we test our new selective barriers on the DaCapo benchmarks. We measure the total number of transactions that are eliminated by the selective barriers per application execution. However, since all benchmarks execute their workload concurrently on multiple threads, a part of the transactions take place in parallel. In order for the results to be as accurate as possible, we



(a) (b)

Figure 10.3 – DaCapo: Overhead upper-bound for selective barriers vs. initial transactional overhead (left) and estimated CPU time for a transactional GC with selective barriers (right).

Table 10.1 – Estimated time savings (DaCapo).

Benchmark	CPU time (s)		Total #tx ($x10^7$)	Min. saved time	
	no tx	tx		(s)	(%)
h2	93.28	130.89	143.57	14.31	10.94
tomcat	59.86	70.37	51.94	5.18	7.36
xalan	57.49	93.72	157.68	15.72	16.77
jython	61.18	104.55	173.28	17.28	16.52
pmd	65.1	84.41	67.66	6.75	7.99
luindex	10.09	26.55	66.02	6.58	24.79

consider using the **CPU time** as metric, rather than the actual wall-clock execution time. The CPU time represents the amount of time (CPU cycles) for which a processing unit is used by the application, excluding all idle times, such as waiting for I/O operations. This measurement is cumulative over all threads, i.e., for a multi-threaded program, the CPU time is expected to be greater than the wall-clock execution time. As the CPU time correctly defines the total execution time over all threads, we can directly associate it with the total number of transactions executed by the application. We retrieve the CPU time for an application with the `perf` Linux tool, using the option `-e cycles`. Thus, we obtain the CPU time for the original implementation using CMS and for the initial transactional implementation, as well as the overhead of the latter over the former. Next, we multiply the measured number of transactions with the transaction cost indicated in Section 10.3.1, to estimate the time saved with the selective barriers. Table 10.1 shows the number of transactions and the estimated savings.

We further employ these results to study the overhead of our selective barriers. A considerable barrier overhead was the initial cause that rendered the transactional implementation impractical. Therefore, the overhead represents our main evaluation concern. Figure 10.3(a) illustrates an estimated upper-bound of the selective barriers overhead, as compared to the transactional overhead of the volatile-only implementation. Given the minimum values used for the overhead computation, e.g., for the transaction cost, we consider this upper-bound generous, expecting

Table 10.2 – Estimated time savings (Cassandra).

Workload	CPU time (min)		Total #tx ($x10^9$)	Min. saved time	
	no tx	tx		(min)	(%)
100% write	54.29	100.94	121.08	20.12	19.93
50/50% r/w	55.16	68.21	66.87	11.11	16.29
95/5% r/w	60.11	74.35	69.54	11.56	15.54
100% read	74.18	88.31	79.24	13.17	14.91
read-latest	60.51	76.08	72.41	12.03	15.82
short-ranges	215.61	777.87	1006.09	167.18	21.49
read-modify-write	84.54	104.93	100.91	16.77	15.98

close-to-zero overhead in practice. The purpose of the figure is to show to what extent the overhead is reduced only by introducing a fast-path with no transactional barrier. Basically, even in this severely underestimated scenario, the initial overhead is decreased by up to 50 %.

After studying the potential improvement obtained by removing part of the transactions, we look next at a rough estimation of the CPU time of a fully-concurrent GC with selective barriers. In order to compute this, we need the CPU time spent in GCs, in addition to the information already gathered. This information is readily available in the details printed by the GC during the execution. Using all the collected data, we estimate the new CPU time according to the following formula:

```
est_time = original_time - gc_time + est_gc_time - fast_access_time,
```

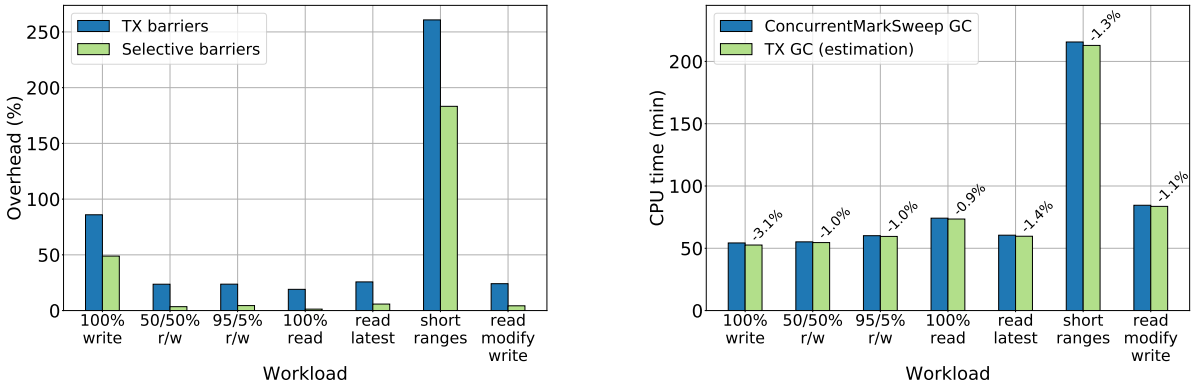
where `orig_time` stands for original time, i.e., garbage-collected with CMS; `gc_time` represents the part of the GC that would be made concurrent in our setting, that is, the ParNew collection duration; `est_gc_time` is the sequence that would replace the stop-the-world execution, i.e., the concurrent slow path of the selective barriers, its duration being estimated based on how many accesses could be made during the GC and their cost; finally, we need to subtract the time taken by the accesses that become concurrent, so that we don't count them twice (both on the fast and slow paths), `fast_access_time`.

In a nutshell, this formula translates to: *synthetically replace the GC stop-the-world time with the estimated time on the slow path and avoid counting the slow-path accesses on the fast-path.* We illustrate the result in Figure 10.3(b). The estimated CPU time of our potential transactional GC with selective barriers is comparable with the original implementation using CMS. The difference is indicated by the percentage labels above the TX GC bars. This result suggests that the performance of a transactional GC enhanced with selective barriers would be on par with that of state-of-the-art GCs, while also eliminating most of the stop-the-world pauses.

10.3.3 Client-server scenario

We also experiment with our selective barriers in a real-life scenario, using the Apache Cassandra 3.9 database server, together with the YCSB client 0.12.0 benchmarks.

Table 10.2 shows the server CPU time with the original CMS implementation, as well as with the volatile-only transactional barriers, total number of transactions that are eliminated due to



(a) (b)

Figure 10.4 – Cassandra: Overhead upper-bound for selective barriers vs. initial transactional overhead (left) and estimated CPU time for a transactional GC with selective barriers (right).

the selective barriers and a lower bound for the time saved by simply removing the indicated number of transactions. It is important to observe that for all the reported execution times, the number of operations completed on the client is constant. This means that the transactional overhead directly impacts the throughput on the client. Therefore, a shorter CPU time on the server also results in performance improvement on the client side.

Figure 10.4(a) indicates how much we decrease the transactional overhead only by removing the cost of starting and committing a transaction for all transactions eliminated by the selective barriers. The gains are not uniform over the benchmarks. We believe this is explained by the type of workload that is executed on the client. Let us take as example the 100 %-*read* workload. Since a load operation does not increase the overhead of the transaction significantly, the cost of these accesses is close to our estimation for an empty transaction. Thus, the simple addition of a fast path in the initial algorithm results in at least 93 % less overhead for a read-only workload. At the opposite end, the 100 %-*write* and *short-ranges* workloads are both write-intensive, leading to enormous overheads. The cost of an empty transaction is less fitting for estimating these accesses, resulting in less apparent savings, with a lower bound of 30 – 40 %. The rest of the benchmarks, featuring various ratios of read/write operations, reduce the transactional overhead with at least 80 %. Except for the write-intensive benchmarks, the upper bound for all overheads with selective barriers is under 5 %. We also notice here the difference between the overheads with respect to the throughput versus the total CPU time for the *short-ranges* workload. This discrepancy comes from the fact that there is not much overhead added on the communication between the client and server, but rather on the computations on the server alone. This results in more parallelism and CPU utilization, without critically affecting the throughput. Finally, we observe that for the other workloads, the associated overheads for the throughput and for the CPU time, respectively, are comparable.

Finally, we apply the same logic as for the DaCapo benchmarks (Section 10.3.2) to estimate the execution time of a concurrent GC with selective barriers. Figure 10.4(b) illustrates the comparison between our estimation and the actual CPU time of the Cassandra server configured to use CMS GC. The outcome of this scenario confirms the previous results: a transactional GC with selective barriers can be expected to have a suitable throughput, comparable to real-life collectors, and be almost fully concurrent at the same time.

10.4 Discussion: removing GC pauses

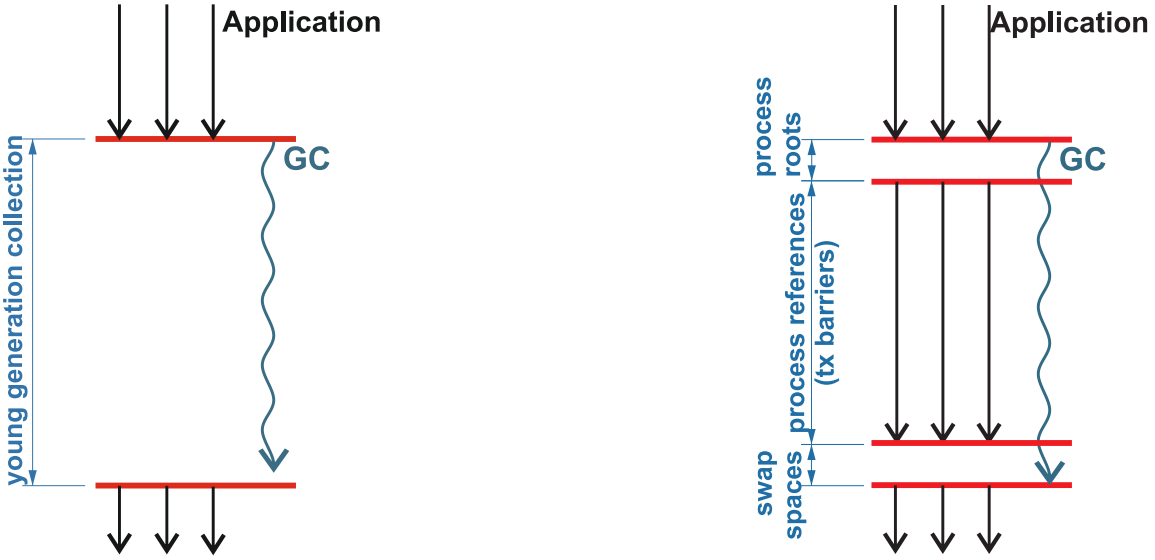
Given the promising estimations for the performance of the selective access barriers, the next step is to change the GC safepoint logic to allow the mutators to run at the same time. We further devise a detailed plan of the modifications needed to relax the GC. We specifically aim to make concurrent the young generation collection and its moving phases: (1) copying objects inside the young generation; and (2) promotion to the old generation. Currently, the GC stops the world during the execution of these operations. Removing the entire safepoint is not straightforward.

We highlight here the most important aspects that need to be addressed and adapted in order to relax the GC:

- **Root scanning.** In most GC algorithms, this particular operation of the garbage collector needs to be executed at a safepoint in order to ensure correctness. The CMS collector, used for the old generation, already has a specific blocking phase for scanning the roots. However, the collector for the young generation, ParNew, used to block the mutators during the whole collection. While there is previous work [58] that designs a lock-free concurrent stack-scanning algorithm, we consider that in our case the complexity of implementing this would surpass the potential benefits. Moreover, based on our experiments the latency of the root scanning operation is negligible. The total CPU time spent in the CMS initial mark phase never exceeds 0.5 seconds for the Cassandra workloads. In comparison, the CPU time spent in young generation collections is around 45 seconds in average for most workloads, going up to over 11 minutes for the *short-ranges* workload. Thus, in our design the root scanning represents the only part of the GC for which the mutators are blocked, in both ParNew and CMS;

- **Tri-color invariant.** Most tracing collectors implement Dijkstra's tri-color marking invariant [36]. In a nutshell, all objects have to be part of one of the three sets: *white set*, objects that will be collected, *black set*, objects that are scanned and reachable from the roots, will not be collected, and *gray set*, containing objects in the process of having their references scanned. The invariant states that no black object can have a reference to a white object. In our setting, some scenarios may not preserve this invariant. For example, let us assume we have an object A that was already copied (in the black set), and an object B still in the white set, referenced by an object C in the process of being moved (gray set). If at this point B stops being referenced by C, but A creates a reference to B, object B will be collected even though it is alive. This happens because A's references will not be processed again, while C does not have a reference to B anymore. This scenario is possible because the application is allowed to modify objects concurrently with the moving GC. To ensure that this does not happen in our design, we consider the following solution: whenever a black object receives concurrently a reference that corresponded to a white object, the white object is directly marked as gray;

- **Stale references.** Copying an object means creating new copies for its references as well. However, when mutators run concurrently with the GC threads, other objects that share the same references may not know about the new copies and still see the old references. This potential issue is handled by the forwarding pointer that is already present in the header of the object when using CMS. The busy-bit prevents access to objects that are in



(a) (b)

Figure 10.5 – Phases of a concurrent transactional GC with selective barriers (right) compared to the original implementation (left).

the process of being copied; the forwarding pointer ensures that all subsequent accesses to a copied object are written in its primary copy;

- **Allocation in to-space.** This is an optimization that avoids useless copies of an object or extra synchronization when the application allocates new objects concurrently with the GC collection. The young generation is split in three spaces: eden, to-space, from-space. If we allocated the new object in eden as per custom, we would have to copy it to to-space anyway during the collection; if we allocated in from-space, we would have to synchronize with the GC to ensure that the object is collected.

With these considerations in mind, we suggest modifying the current implementation with selective barriers as follows: split the safepoint that protects the entire young generation collection into multiple parts, configured to run at a safepoint or concurrently depending on their task (Figure 10.5). The root scanning is the first phase and it is run at a safepoint. We expect this pause to be short, as mentioned earlier. On this safepoint we can also piggyback the alteration of the in-collection flag that enables the selective barriers. Then we pass to the next mode of collecting, which will be concurrent. This covers the processing of the references (also moving objects between survivor spaces or promoting them to the old generation), correctness being ensured by the transactional barriers enabled in the previous step. Finally, since we plan to allocate new objects in to-space, a short safepoint may be necessary for swapping survivor spaces (from-space and to-space). These general guidelines were followed when computing the estimated CPU time for a transactional GC with selective barriers.

A full implementation of a transactional concurrent GC requires two major steps: **developing an efficient synchronization mechanism** and **relaxing current synchronization**. We consider that once the new synchronization mechanism was proved to be correct and practical,

and a detailed plan was laid down for removing the previous mechanism, the rest is mostly an engineering effort. However, the first part is critical for making the latter worthwhile. Thus, we leave the remainder of the implementation as future work.

10.5 Summary

This chapter presented an extensive evaluation of our implementation of transactional barriers and subsequent optimizations. We experimented with a widely used benchmark suite, as well as with a real-life setting of a server-client database system. We started by evaluating the overhead of the optimized transactional barriers in the Java interpreter, where they were proved to have an acceptable performance. By enabling the JIT compiler, we further observed that the relative transactional overhead is much more pronounced in the new setting. This represented the motivation for implementing the selective-barriers optimization. We aimed to predict the performance of a concurrent transactional GC that used selective barriers. The results indicated that most overhead would be removed and the performance would be on par with state-of-the-art GCs. We further gave detailed guidelines on how to achieve almost-full concurrency in the case of our transactional GC algorithm.

Part III

Exploiting HTM for Improved C++ Smart Pointers

Chapter 11

Transactional Smart Pointers

Contents

The most popular programming languages, such as C++ or Java, have libraries and data structures designed to automatically address concurrency hazards in order to run on multiple threads. This trend has also been adopted in the memory management domain. However, automatic concurrent memory management also comes at a price, leading sometimes to noticeable overhead. In this thesis, we experiment with C++ smart pointers and their automatic memory-management technique based on reference counting. More precisely, we study how we can use hardware transactional memory (HTM) to avoid costly and sometimes unnecessary atomic operations. This chapter presents details about the algorithm and the implementation of transactional smart pointers. We then introduce specific use cases that could particularly benefit from transactional pointers.

11.1 Motivation

With the increasing degree of concurrency in today's hardware, lock-free implementation of applications and data structures gained extensive attention in the last few years. In this context,

using classical synchronization mechanisms based on locks (such as mutexes, barriers, etc.) tends to become more and more complex and error-prone. Transactional memory offers an elegant solution to implementing lock-free synchronization. Until recently, TM algorithms were mostly reserved to the research environment. However, the emergence of hardware transactional memory in mainstream processors overcame the performance pitfall, while conserving the benefits in scalability and correctness.

Automatic memory management mechanisms often suffer from performance drops due to their synchronization strategies. A notable example is represented by the **smart pointer implementation in the C++ standard library**. Smart pointers use *reference counting* to protect a raw pointer from being illegally deallocated and to avoid any other memory hazards. They are thread-safe and the operations on the shared reference counter are atomic. This provides adequate and safe memory management for multi-threaded programs. Nonetheless, the reference counting strategy is costly and sometimes unnecessary, e.g., when manipulating copies of a smart pointer with a reference count that never drops below 1 and hence never needs to release memory.

In this work, we explore possible scenarios where HTM could improve the performance of applications that use C++ smart pointers. The aim is to avoid executing the unnecessary atomic operations required by the reference counting strategy. First, **we expect HTM to improve the performance of smart pointers over the original implementation**. Then, by adding this low abort-rate HTM fast-path, **we are also addressing concurrency problems related to smart pointer handling**. Gottschlich et al. [44] show that template-based generic structures, such as C++ smart pointers, are deadlock-prone, among other synchronization issues. They also propose the use of TM in their implementation. These problems motivated our extensive study on the benefits of HTM for C++ smart pointers.

11.2 Related work

Considering the ever increasing interest in transactional memory in the last few years, a reasonable amount of effort has been focused on integrating TM with mainstream programming languages, such as C++. Crowl et al. [26] present a general design that would permit the insertion of transactional constructs into C++. They identify the main issues that need to be addressed and propose a new syntax that could be incrementally adopted in the existing code base. These guidelines do not address a particular class of transactional memory (STM or HTM). The authors identify I/O operations and exceptions as being the trickiest to handle, which generally proved to be the case for HTM since it appeared in specialized processors.

Ni et al. [67] go even further and implement a fully working STM system that adds language constructs for transactional programming in C++. The system includes new C++ language extensions, a compiler and an STM runtime library. They conduct an extensive evaluation on 20 parallel benchmarks ported to use their C++ language extensions. The results show that the STM system performs well on all workloads, especially in terms of scalability. In the same context, Shpeisman et al. address the integration of TM semantics in C++ [78]. The authors argue that generally TM semantics fit C++ logic well. However, specific features like nested transactions, user-level aborts or even legacy code on the C++ side, need extensive reasoning. As such, they provide two new sets of language constructs for transactional memory in C++: one for the common cases that need atomicity, also allowing user aborts; and one that handles legacy code. Another effort in this direction is made by Dalessandro et al. [27]. They design a

library-based software transactional memory API and build several applications with it. The API successfully handles all kinds of specific C++ features, such as inheritance, macros or generics. In order to cover all these features and provide intuitive semantics for C++, the authors implement the STM library on top of C++ smart pointers. Basically, they hide all metadata manipulation behind smart pointer operators (for construction, assignment, dereference). Despite the API's capabilities, the authors claim that TM cannot simplify concurrent programming if implemented as a library, language and compiler support being indispensable for reaching this goal.

A more focused work is presented by Gottschlich and Boehm [44] regarding the need for transactional memory in generic programming. They give as example C++ shared pointers and similar constructs, and indicate that implementing them with transactions would avoid deadlocks and other synchronization issues. In this case, the authors do not explore the performance of a potential transactional implementation, but the correctness of such a strategy. They mention that any type of transactional memory would be an improvement, while hinting that HTM would probably have a good impact on the performance as well.

Dragojević et al. [38] explain why C++ needs a form of automatic memory management. Their reasoning is based on the lack of lock-free data structures in C++, as opposed to garbage-collected languages such as Java. They make a case for HTM as having a good potential for improving this situation. They present a few algorithms of dynamic-sized concurrent data structures synchronized with HTM and show that, in general, they perform better than non-HTM algorithms or require less space. The authors conclude that it is considerably easier to design correct concurrent algorithms with HTM.

However, opinion is divided on the performance benefits of HTM for synchronizing concurrent data structures. On the one hand, David et al. [28] report an increase in throughput of at most 5 % when using HTM for concurrent search data structures, considering the improvement as negligible. They partly blame the current HTM implementations for this shortcoming, envisioning stronger results when this technology matures.

On the other hand, Bonnichsen et al. [16] present a concurrent ordered map implementation with HTM that performs up to 3.9 times faster than the state of the art. They also formulate a set of guidelines for applying HTM efficiently. The recommendations mostly address spatial locality, specific instructions (e.g., system calls) and transaction size. The authors design an ordered map data structure (called BT-tree) with these considerations in mind and find that the HTM-based data structure has a very good performance in terms of space, time and energy.

In our work on C++ smart pointers, we apply the guidelines that recommend enhancing them with transactional support, thus avoiding specific concurrency issues, and evaluate the potential performance improvement when using HTM on concurrent data structures. We show that, depending on how the HTM logic is applied, it is able to improve significantly the classical implementation of C++ smart pointers.

11.3 Algorithm

The original algorithm for C++ smart pointers is straightforward: when the pointer is created, it contains a raw pointer and a control block for this reference. The reference count is initialized with 1. As previously explained in Chapter 4 (Figure 4.3), when a new copy of the same pointer is created, they will have in common the reference and the reference count field, which is updated by atomic increment. Every time a copy of the pointer is destructed, the shared reference count

is atomically decreased by one, and the rest of the object destroyed. If there is only one reference left, then the memory is automatically freed. This allows the application to function without any risk of incorrect accesses, dangling pointers or memory leaks.

Our goal is to eliminate the atomic operations on the shared reference count, while keeping the reference protected from memory hazards. In order to do that, we define a constructor that initializes the reference count with 0 and tries to start a hardware transaction. Inside the transaction, we read and update a field of the transactional pointer called `state`, shared between all transactional and non-transactional copies of the pointer. The `state` field is added to the read-set of the transaction and automatically monitored in hardware. Any other thread that tries to modify this field causes an abort. If there is no abort, the application continues its execution, using the transactional pointer protected by the transaction. If a conflict or another event causes the currently running transaction to abort, the transactional pointer follows a fallback path corresponding to the original implementation of the smart pointers, i.e., the reference count is initialized with the number of references of the smart pointer that we are copying and atomically incremented, or with 1 if it is a new smart pointer. When the transactional pointer is destroyed by the application, we check if there is a transaction running: if yes, the transaction commits. Otherwise, the object is destroyed in the same way as a normal smart pointer.

Further on, we modify the algorithm to support batching. More specifically, multiple transactional pointers can be added to an already started transaction, without having their reference count modified and without starting a transaction on their own. This is particularly convenient for applications using well-delimited groups of pointers, such as some operations on classical data structures. In order to allow batching, the constructor exploits an additional parameter indicating whether the transactional pointer in question is the first one in the batch (or a single pointer that needs to be protected) or it needs to be added to an already existing transaction. In the former case, the algorithm follows the steps described above. In the latter, we take advantage of the fact that all initializations happen inside the transaction. Thus, we do not need to specifically read the `state` field anymore. Finally, we add a supplementary check in the destructor of the transactional pointer: we only try to commit the transaction when the pointer that started it is destroyed. This design assumes a scenario in which:

- either all pointers die at once (e.g., at the end of a function in which they have been created), in which case they are destroyed in the reverse order of creation, thus making the first pointer to be destroyed last and keeping the transaction that protects all pointers running until it is safe to commit; or,
- the pointers added to the transaction are explicitly destroyed before the pointer that started the transaction.

The above steps are summarized in Algorithm 3. We call `tx_ptr` the data type that enhances C++ smart pointers with hardware transactions. The constructor creates a transactional pointer from an already existing smart pointer, which is passed as a parameter. We use `this` to designate the current transactional pointer being created. We cover in this pseudocode the extended algorithm suitable both for batching pointers as well as for single transactional pointers. Therefore, the constructor features a boolean parameter `add_to_tx` that indicates whether the current pointer has to be added to a running transaction or start a new one by itself. If it is the first pointer (line 9), it tries to start a transaction. All subsequent transactional pointers will be monitored by the same transaction and will not attempt to start a new one. If the transaction

Algorithm 3 Transactional pointer implementation.

```
1: function tx_ptr::Init(smart_ptr ptr, bool add_to_tx)
2:     this.rawptr ← ptr.rawptr
3:     this.refcount ← 0
4:     this.state ← ptr.state
5:     this.add_to_tx ← add_to_tx
6:     if add_to_tx ∧ is_fallback then
7:         Fallback(ptr)
8:     end if
9:     if ¬this.add_to_tx then
10:        if Tx_Start() then
11:            Update(this.state)
12:            is_fallback ← false
13:        else
14:            is_fallback ← true
15:            Fallback(ptr)
16:        end if
17:    end if
18: end function
19: function tx_ptr::Destroy()
20:    Write(this.state)
21:    if ¬this.add_to_tx ∧ Tx_Test() then
22:        Tx_End()
23:    end if
24: end function
```

starts, we update the shared `state` field, as mentioned; otherwise, the algorithm takes the fallback path. The call to a generic function `UPDATE()` (line 11) stresses the idea of accessing the field inside the transaction, without entering into implementation details. The variable `is_fallback` is a thread-local variable, set when the first pointer takes the fallback path. When a transaction aborts, all changes are rolled back and the execution is restarted. This means that all the added pointers will run their constructor from the beginning, following the path in line 6. In other words, all transactional pointers will take a non-transactional path, similar to a classical C++ smart pointer. While the transaction is running correctly, `is_fallback` remains false. The destructor (starting at line 19) first modifies the `state` field, in order to notify other threads that are handling the same pointer to abort their transactions since it may try to release memory (e.g., if the pointer is on the fallback path). Then, it commits the running transaction, in case there is one and it was started by the current pointer. A transactional pointer does not free the memory it points to in the destructor, leaving this task for the underlying smart pointer. Any transactional pointer is either backed up by at least one classical smart pointer that can correctly free the memory, or it transforms itself into a classical smart pointer (in case of abort). The presented algorithm implements a viable smart pointer structure, while aiming to reduce the overhead of atomic operations.

11.4 Implementation details

We build our transactional pointers on top of the `std::shared_ptr` structure in C++. In particular, we overload the constructor in the `std::shared_ptr` class in GCC 5.3.0 and modify several internal methods in order to accommodate the transactional logic. As such, a `tx_ptr`

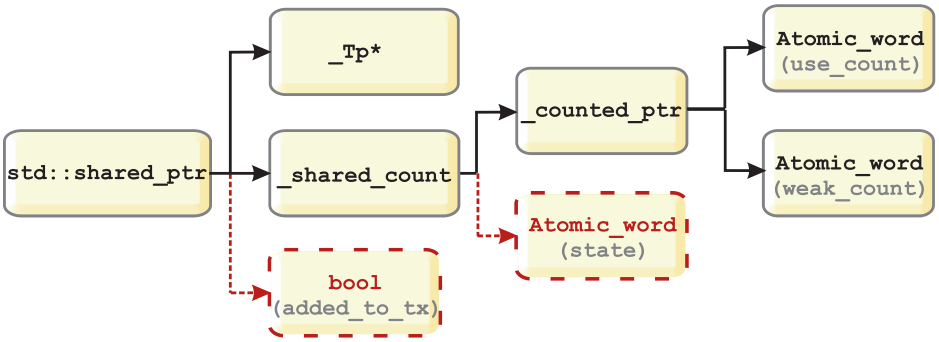


Figure 11.1 – Class structure of `std::shared_ptr`, with the additional fields for `tx_ptr` in dashed lines.

can simulate the normal behavior of a classical smart pointer and `tx_ptr`s can be created from `std::shared_ptr`s. The implementation follows closely the above algorithm.

The `std::shared_ptr` class is implemented as a C++ template, with the raw pointer (of generic type _Tp) and a variable of type `__shared_count` as the main fields (Figure 11.1). The latter is the class that implements the shared reference count object. The reference count object contains a pointer to two counters: `use_count` and `weak_count`. The latter is related to `std::weak_ptr`s (see Section 4.2.1) and its role is not in the scope of this work. The former contains the number of references a pointer has throughout the execution. We add in Figure 11.1 with dashed lines the necessary fields for implementing `tx_ptr`:

- A boolean variable in the main class, with the aim of indicating which pointers are to be added to the existing transactions or start a new one. This information is critical in the destructor when using batching, since the transaction must be committed only when all pointers in the group have been destroyed;
- The `state` field in the reference count class. This field is shared between classical smart pointers and their transactional copies. It is initialized in the constructor and accessed inside the transaction started by transactional pointers, in order to be monitored in hardware. It is further modified in the destructors of both classical and transactional smart pointers. Thus, if any other copy of the same pointer tries to destroy the pointer and deallocate the memory, writing the `state` field forces the transaction to abort and the `tx_ptr` to restart as a normal smart pointer. If the last underlying smart pointer is destroyed before its transactional copy, it does not free the memory, but passes the ownership to the aborted transactional pointer.

We implement the transactional memory operations on two different architectures: Intel Haswell and IBM POWER8 [47]. While there are several subtle differences in the APIs and the underlying HTM implementations, most of our code is common to both architectures. The goal is to compare the behavior of our transactional smart pointers for different HTM implementations. We want to determine if the architecture makes a difference or the performance gains remain constant in all cases.

Algorithm 4 Scenarios for the single-threaded micro-benchmark.

```
   SCENARIO 1
 1: for i ← 1, num_iter do
 2:     p ← new shared pointer
 3:     delete p
 4: end for

   SCENARIO 2
 5: for i ← 1, num_iter do
 6:     BEGIN-TX
 7:     p ← new tx shared pointer
 8:     delete p
 9:     COMMIT-TX
10: end for

   SCENARIO 3
11: for i ← 1, num_iter/m do
12:     BEGIN-TX
13:     for i ← 1, m do
14:         p ← new tx shared pointer
15:         delete p
16:     end for
17:     COMMIT-TX
18: end for
```

11.5 Use cases

In order to have a preliminary idea on the benefits of our transactional pointer implementation over the original C++ smart pointers, we devise two micro-benchmarks. This enables us to test both implementations in single-threaded and multi-threaded scenarios, with or without batching, on two different architectures.

11.5.1 Baseline: single-threaded micro-benchmark

We want to evaluate the possible gains of replacing atomic operations with hardware transactions. We develop a single-threaded micro-benchmark for studying how many transactional pointers have to be packed in a transaction in order to improve the performance over a pair of atomic operations, when the application runs on a single thread. The micro-benchmark consists of the scenarios presented in Algorithm 4. By *tx shared pointer* we refer to a `tx_ptr` implementation. In the first scenario, starting at line 1, we measure the time it takes to repeatedly create and destroy a normal C++ shared pointer, for a fixed number of iterations. As previously mentioned, when the pointer is created, an atomic increment is performed on the shared reference count; likewise, an atomic decrement is performed when the pointer is destroyed. This strategy reflects the performance when using a pair of increment/decrement atomic operations for `num_iter` iterations. The second scenario, starting at line 5 replaces the pair of atomic operations in each iteration with a hardware transaction. The third scenario (line 11) groups multiple create/destroy operations (i.e., multiple iterations) in a transaction. It behaves identically to the second scenario when $m = 1$.

Our intuition is that one hardware transaction should have at least the same performance as a pair of atomic operations. The goal of this micro-benchmark is to validate this assumption on the simplest possible scenario, without any conflicts or contention. If our intuition proves wrong in the most basic case, there is no point in pursuing this direction. Thus, we consider this scenario as the baseline for our evaluation.

Algorithm 5 Scenarios for the short-lived pointers micro-benchmark.

```
    SCENARIO 1
 1: loop
 2:     pick shared_ptr p
 3:     SOME_COMPUTATION(p)
 4: end loop

 5: function SOME_COMPUTATION(p)
 6:     add_to_tx ← True
 7:     q ← new tx_ptr(¬add_to_tx,p)
 8:     constant ops
 9:     delete q
10: end function
```

```
    SCENARIO 2
11: loop
12:     pick shared_ptr p[n]
13:     SOME_COMPUTATION(p[n])
14: end loop

15: function SOME_COMPUTATION(p[n])
16:     add_to_tx ← True
17:     q[0] ← new tx_ptr(¬add_to_tx,p[0])
18:     q[1 : n − 1] ← new tx_ptr(add_to_tx,
        p[1 : n − 1])
19:     constant ops
20:     delete q[n − 1, 0]
21: end function
```

11.5.2 Short-lived pointers micro-benchmark

We next look at a common real-life scenario where a pointer is copied to a local variable inside a function. The copy of that pointer has the lifespan of the function. We call the copy *short-lived pointer*. If a smart pointer is used, when creating such a copy, the reference counter is atomically incremented, while at the end of the function there is an atomic decrement. If the pointer is not accessed concurrently in the meantime, then the increment/decrement operations are redundant. We aim to replace this pair of atomic operations with one transaction spanning the entire function. In order to obtain this behavior, we use the `tx_ptr` pointer defined in Section 11.3. If two threads access the same pointer, the transaction will abort and fallback to a classical smart pointer; otherwise, the threads will continue together concurrently without affecting the reference counter in any way.

We create a micro-benchmark that starts multiple threads which share an array of smart pointers. Each thread picks a random element from the shared array and calls a function. In the function, the thread creates a `tx_ptr` copy of the element. Then, it executes several constant-time operations. These operations are meant to simulate a computational workload that accesses the pointer value. If transactional pointers are used, these operations will be executed inside a transaction. Finally, the thread exits the function (which calls the destructor of the transactional pointer, thus committing the transaction). These steps are illustrated in Algorithm 5, lines 1 – 10.

As an optimization, we enable batching in the micro-benchmark, i.e., the creation of multiple pointers in a single transaction. The idea is that, if our initial intuition holds and a pair of atomic operations has almost the same overhead as a transaction, then replacing multiple pairs of atomic increment/decrement with a single transaction should improve the performance. Lines 11 – 21 in Algorithm 5 reflect these changes. We emphasize in the pseudocode the most significant alterations to the initial scenario (lines 18 and 20). We modify the original micro-benchmark algorithm as follows: instead of a single random element, each thread now picks several random

elements from the shared array (number defined at runtime). It creates a new array with these elements and calls the specific function having this array as a parameter. The function makes `tx_ptr` copies of all pointers, using the additional boolean parameter in the constructor in order to indicate which pointers will be added to the current batch. Basically, in line 17 we create the first transactional pointer, which can be stand-alone or the head of a batch. Then (line 18), all the other copies of the pointer receive the value `True` for the `add_to_tx` parameter, which indicates they are part of a batch and should be treated as such (they cannot be handled individually). A single transaction is started in line 17. All subsequent pointers in the batch are protected by this transaction. The constant computations take place inside the transaction. At the end of the function the copies of the pointer are destroyed in reverse order than they were created (line 20). Essentially, the pointers that are added to an existing batch are simply released, while the head of the batch is also responsible for committing the transaction.

We consider these scenarios as fairly common in application code. We generally expect few conflicts, the performance only depending on the overhead of using hardware transactions. In this case, the transactional overhead has two sources: the latency of starting and committing a transaction and the overhead of executing the code between the creation of a smart pointer and its destruction inside a transaction. In the case of short-lived pointers, the computations inside the function are also limited by the maximum size of a hardware transaction.

11.6 Summary

This chapter introduced the basics of *transactional smart pointers*, as a convenient combination between two important concepts in today's concurrent systems: transactional memory and C++ smart pointers. We presented the initial algorithm and an optimization, together with two micro-benchmarks that we considered suitable for their use. To sum up, the transactional algorithm defines a new type of C++ smart pointer with its typical features. In the original implementation, it was impossible for any of the threads sharing a smart pointer to deallocate memory that was still in use, due to the shared reference counter. In the transactional version, the counter stays at 0 for any number of shared copies. However, any illegal memory access is detected inside the hardware transaction that encapsulates the pointer. Thus, multiple threads sharing the same pointer will be able to safely use it without any contention on a shared counter. The evaluation on the described micro-benchmarks and the associated conclusions are further addressed in the next chapter.

Chapter 12

Evaluation on Micro-benchmarks

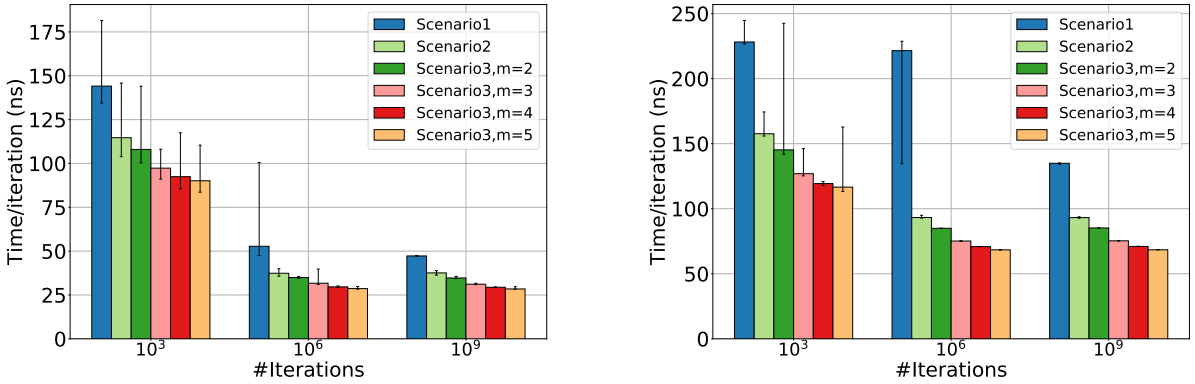
Contents

Based on the micro-benchmarks described in the previous chapter, we set up a preliminary batch of experiments to check the performance of our transactional pointers implementation in the most basic cases. We also verify if the discovered behavior is consistent across different architectures. We first test our strategy on a single thread, then in simple multi-threaded scenario involving short-lived pointers. We use these results to determine what situations benefit most from a transactional smart pointer implementation.

12.1 Experimental setup

We implement and test the micro-benchmarks on two different platforms: an 8-core (with 2 threads per core) Intel i7-5960X CPU @ 3.00 GHz (Haswell) machine with 32 GB RAM and a 10-core (8 threads per core) IBM POWER8 @ 3.42 GHz with 30 GB RAM, both with fully integrated HTM support. Settings that are specific to each micro-benchmark are further detailed.

Single-threaded micro-benchmark. For the third scenario of Algorithm 4 in Chapter 11 (line 11), we vary m from 2 to 5, since for values greater than 5 the performance is visibly better than the original implementation of shared pointers. We observe that the measured time varies during the first executions of the benchmark. Therefore, in order to have accurate results, we pin the thread to a core and we run 1000 warm-up iterations. Then we start measuring for the required number of iterations. Subsequently, we take the mean value per iteration over 50 runs for each version of the benchmark. We test for 10^3, 10^6 and 10^9 iterations.



(a) Intel Haswell

(b) IBM POWER8

Figure 12.1 – Execution time/iteration for single-threaded scenarios in Algorithm 4 (Section 11.5) with $m = 2, 3, 4, 5$ in Scenario3.

The total execution time of all iterations is measured with the system's high resolution clock, by calling `std::chrono::high_resolution_clock::now()` before and after the loop. The result is manipulated as a `std::chrono::duration<double, std::nano>`, i.e., with nanosecond accuracy. We obtain the time per iteration by dividing this total time by the predefined number of iterations.

Short-lived pointers micro-benchmark. In the case of Algorithm 5 in Chapter 11, we measure how many iterations of the short-lived function are done by each thread in a certain amount of time (customizable by the user). We test on up to 8 threads, pinned to the cores. We compare the total number of iterations (i.e., the sum of iterations over all threads) of our `tx_ptr` implementation with the original implementation of smart pointers. We configure the short-lived pointer experiments as follows: shared array of 1000 smart pointers, run time of 5 seconds, 100 constant-time operations. The experiments consist of running the micro-benchmark 50 times for an increasing number of threads on both platforms and taking the average over the total number of iterations in each case.

12.2 Results for single-threaded experiments

Figure 12.1 illustrates the performance in all previously described single-threaded scenarios for the mentioned values of m and number of iterations, both on the Intel Haswell machine and on IBM POWER8. Thus, for each number of iterations, the graph shows: one bar for the time it takes to create and destroy a classical C++ smart pointer (marked as Scenario1 in the legend); one bar for the latency of creating and destroying a transactional pointer (i.e., Scenario2); the remaining four bars for the time it takes to create and destroy transactional pointers that are grouped together in the same transaction in batches of 2, 3, 4 and, respectively, 5 pointers (i.e., Scenario3, having m = 2,3,4,5). The number of iterations is represented on the X axis. The Y axis shows the time per iteration in nanoseconds. The error bars in the graphs represent the minimum and maximum values for the measured times.

A first observation is that for a small number of iterations (e.g., 10^3), the measurements vary visibly, regardless of the machine. Such a short measurement is easily impacted by otherwise negligible delays, like the latency of the timing function itself. In consequence, the experiments

on a greater number of iterations vary less. Another interesting remark is that the general shape of the graphs is maintained across all architectures and experiment sizes. This suggests that our results are consistent and we can extract some general conclusions about the behavior of transactional smart pointers on a single thread. We also note that the execution times measured on Haswell and POWER8 differ. The separate library implementations on each architecture (e.g., HTM implementation) account for this difference.

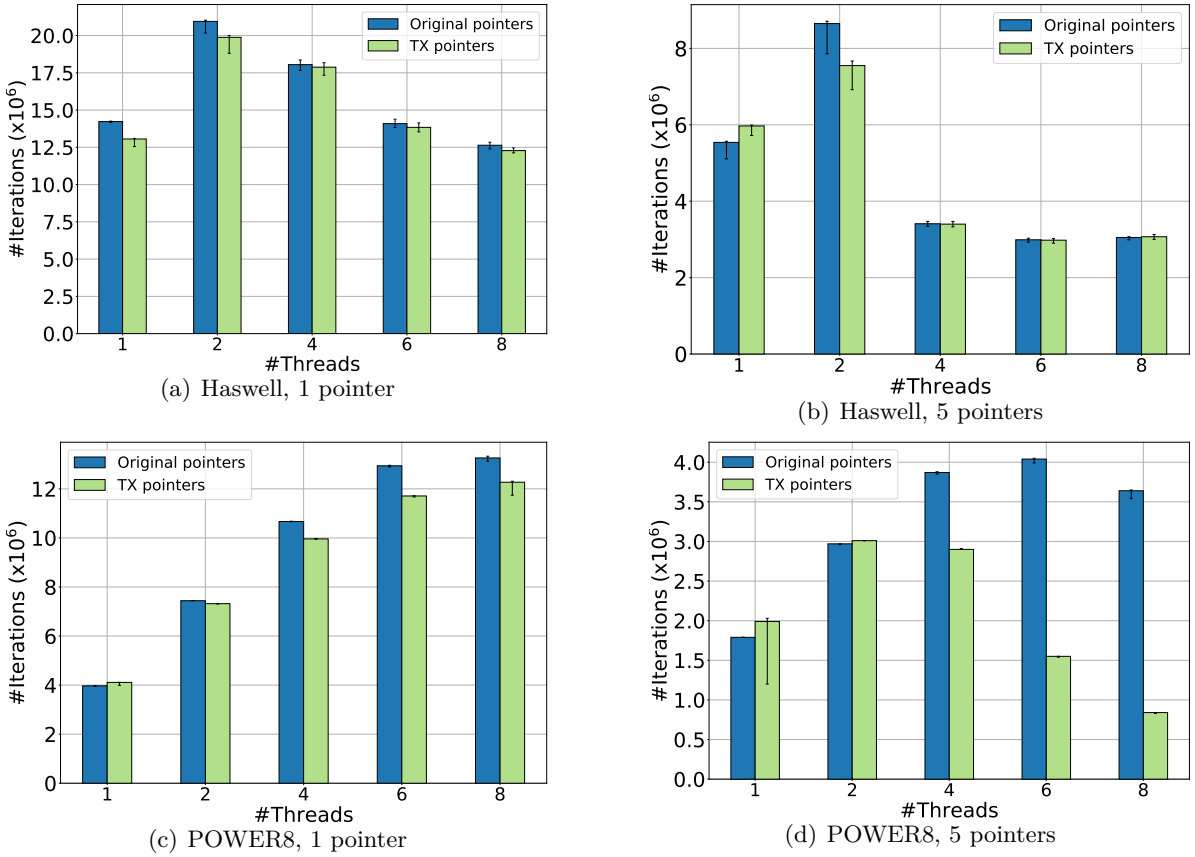
Focusing further on the results, we observe the following: when running on one thread, using a single hardware transaction per iteration always results in better performance than a pair of atomic operations. In other words, the second scenario performs better than the first for any number of iterations on both platforms. Generally, the improvement is consistently over 20 %. The performance further improves when m increases, i.e., when grouping multiple pointers inside a transaction. As expected, on a single thread with no interference, the gains in the batching scenario are gradually higher. For groups of 5 pointers, the execution time per iteration is around 37 % to 45 % lower on the Haswell machine and 48 % to 69 % lower on IBM POWER8. However, this improvement strongly depends on the number of pointers in the group. After a certain threshold (when the group of instructions inside the transaction becomes too large) the transaction overflows. In this case, the execution falls back to the classical implementation, losing all the benefits of batching. Thus, care must be taken as to how many pointers we pack in a hardware transaction in order to maximize the gains.

In conclusion, according to the presented single-threaded benchmark, a hardware transaction is able to replace a single pair of atomic operations without affecting the performance of the application. The application performance is significantly improved if multiple pairs of atomic operations are replaced by a single hardware transaction.

12.3 Results for short-lived pointers experiments

Figure 12.2 compares the performance of transactional smart pointers with the original C++ pointers in this scenario. It illustrates our results for the Intel Haswell architecture in Figures 12.2(a) and 12.2(b), as well as for POWER8 in Figures 12.2(c) and 12.2(d). For both systems we test for a 1-pointer transaction and for batching multiple pointers. On the X axis we show the number of threads, while on the Y axis we have the number of iterations performed divided by 10^6 (higher values are better). We observe that overall the two implementations have similar performance. However, there are a few interesting observations worth mentioning. First, we notice that the same benchmark implementation scales differently on the Haswell machine compared to POWER8. Then, despite having a comparable performance, the transactional implementation is most frequently worse than the original. The exception is represented by most of the runs on one thread and a couple of batching cases (on 8 threads on the Haswell server, and on 2 threads on POWER8), where the transactional version does slightly better.

Considering each architecture alone, for Intel Haswell we make the following observations: generally, the difference in performance of the two implementations is small, becoming negligible on 4 or more threads. This applies to both presented cases: transactions encapsulating one pointer or a group of pointers. In the former case, the transactional version performs with at most 8 % worse than the original, the difference being between 1 % and 3 % when running on more than 4 threads. Similarly for the latter: batching results in having only one configuration with an average overhead over 0.5 %. More precisely, we find that when executing the benchmark on two threads, the transactional version ends up having around 12 % fewer iterations than the



(a) Haswell, 1 pointer

(b) Haswell, 5 pointers

(c) POWER8, 1 pointer

(d) POWER8, 5 pointers

Figure 12.2 – Number of iterations for one or multiple short-lived `tx_ptr` pointer copies (TX) and smart pointer copies (Original) in a function.

original implementation. In conclusion, grouping multiple pointers in a transaction seems to slightly improve the performance of transactional smart pointers, but not well enough to clearly outperform the current implementation of C++ smart pointers.

On the POWER8 server the difference between the original and the transactional pointers increases with the increasing number of threads. Basically, while the benchmark scales well with the number of threads in both scenarios (copy of 1 random pointer in a function, and, respectively, copies of multiple pointers), the transactional version scales poorly in the former scenario (Figure 12.2(c)) and not at all in the latter (Figure 12.2(d)). This indicates that the transactional implementation on this architecture suffers more from contention than the atomic operations. Thus, by trying to optimize the first results with batching, we also increased the overhead of the transaction with extra operations. In this case we observe that the behavior of the transactional execution is more similar to the runs on the Haswell machine, where it failed to scale as well.

This result led us to the conclusion that, in a multi-threaded environment where many operations are involved, the creation of a single `tx_ptr` does not bring an improvement over a pair of atomic operations. Given this negligible performance gain, we deduce that in this scenario using transactional pointers does not have a significant advantage over the original C++ smart pointers. We believe that the simulated computational workload in the function that

creates and destroys the short-lived pointers also played an important role in the unsatisfactory performance. The overhead is partly due to the increased latency of executing any of these operations inside a hardware transaction. Thus, compared to the baseline micro-benchmark, this scenario does not simply compare the overhead of a transaction with that of a pair of atomic operations, but rather the aforementioned overhead added to the sum of latencies of all computational operations executed transactionally with the overhead of a pair of atomic operations. Even so, the difference is negligible in most cases.

12.4 Summary

We evaluated transactional smart pointers on two micro-benchmarks in order to have a preliminary idea on their performance and potential uses. The results led us to the following two important conclusions:

1. **Not all scenarios benefit from transactional smart pointers.** The fact that a computational workload inside a transaction decreased significantly the performance of transactional pointers suggests that they cannot replace smart pointers in their entirety in any context. However, in well-thought scenarios, in which the latency of transactional operations is amortized by the lack of contention and increased parallelism, they would show a clear advantage over the current smart-pointer implementation. We further look at transactional pointers specialized on data structures traversal as an example of such a scenario;

2. **Transactional pointers are comparable to classical smart pointers in terms of performance.** On the single-threaded baseline, transactional pointers showed up to 69 % improvement over classical pointers. Despite the performance drops in some of the cases presented in Section 12.3, we showed that most of the times the difference between the executions with the original smart pointers and transactional pointers was negligible. We expect that specific operations that are better tailored for working with hardware transactions can be significantly improved with transactional pointers, while for other scenarios, less suited for HTM, they will not impact the performance in a serious way.

Chapter 13

Transactional Pointers for Concurrent Data Structures Traversal

Contents

The next step after experimenting with transactional smart pointers in the context of micro-benchmarks is the evaluation of their performance on data structures. We motivate and develop this choice throughout this chapter. We look at concurrent queue and concurrent linked list implementations. We show how to change the data structures implementation to use transactional pointers. Finally, we describe the necessary additions to the initial algorithm of transactional pointers in order to accomodate the new specifications.

13.1 Motivation

Initial experiments with transactional smart pointers on micro-benchmarks returned conflicting results (see Chapter 12). The single-threaded baseline experiment indicated that a hardware transaction should be able to successfully replace a pair of atomic operations and batching multiple pointers in a transaction should greatly improve the performance compared to the original implementation of the smart pointers. At the opposite end, in the short-lived pointers scenario the transactional implementation fails to perform adequately. While the performance is comparable for `std::shared_ptr`s and `tx_ptr`s, the former always have slightly better results. We believe that this discrepancy is partly due to the computational workload that is executed inside the transaction in the latter case. This increases the latency of each iteration that uses transactional pointers. As such, we concluded that HTM-based smart pointers have a considerable potential to improve classical `std::shared_ptr`s, but the context in which they are used significantly influences the extent of the improvement.

With these considerations in mind, we further focus on concurrent data structures. More specifically, we look at a singly linked list implementation. The main characteristics of interest in this context are as follows:

- **Natural occurrence of pointers with a reference count $\geq$ 1.** List's nodes will always be referenced by at least one other pointer until they are removed or the list is destroyed. Thus, traversing a linked list inherently matches our concept of transactional pointer: at every step a hardware transaction will replace a pair of atomic operations and protect the node at the same time from being deallocated prematurely. Moreover, batching also fits perfectly in this context, since most of the times more than a few nodes have to be processed before reaching the targeted one;
- **Fixed known transactional workload.** Transactions will only contain a check for the node's value, or a user-defined number of these checks, in case of batching. The most appropriate number of pointers in a group can be determined experimentally. As in the single-threaded micro-benchmark, we expect the performance to be improved with the increasing number of batched pointers up to a threshold. The threshold can be determined when the hardware transactions start overflowing due to an excessive workload. The number of pointers in a batch could also be restricted based on the latency in case of a conflict;
- **Practical application in real-life scenarios.** An efficient linked list implementation can be further incorporated in more complex data structures, such as hash tables. Since we are mostly looking at data structures' *traversal*, we expect to improve lookup operations, which represent the majority on this kind of data structures.

These aspects motivated our choice of scenario for experimenting with transactional pointers. This required reassessing the first implementation of `tx_ptr`s and adapting it to the particular demands of the new context.

13.2 Concurrent queue

For simplicity and reproducibility of our tests, we start experimenting with a concurrent queue. We design it as a linked list structure only based on shared pointers and *compare and swap*

Listing 13.1 – Concurrent queue using C++ smart pointers: list node.

```
1  struct Node{
2    T data; // generic data
3    std::shared_ptr<Node> next;

5    Node(T val) { // node constructor
6      data = val;
7      next = nullptr; // the next pointer is initially set to NULL
8    }
9  };
```

(CAS) operations. We only insert elements at the end of the linked list structure and remove the first element. We also provide a list traversal operation, which benefits most from transactional pointers. The goal is to first experiment with `tx_ptr`s in a simpler context, where the contention and most changes are only present at the extremities of the data structure.

13.2.1 Algorithm overview

For our concurrent queue implementation we used a classical algorithm, namely the *unbounded lock-free queue* algorithm proposed by Herlihy and Shavit [52, Chapter 10]. In a nutshell, the queue is implemented as a list of nodes, each containing a value and a pointer to the next element in the list. The queue has a sentinel node, called the *head*. The last element in the queue is called the *tail*. When the queue is empty, the head and the tail point to the same node.

The concurrent queue typically has the following operations:

- **enq(T value):** a new element is added to the queue. For this, a new node is created, the `tail` is located and the new node is appended at the end of the queue. Two CAS instructions are employed for this operation: one for appending the node and another one for setting the `tail` (i.e., changing its prior position to the current last node);
- **deq():** the first element is removed from the queue. If the queue is not empty, the `head` is changed to point to its successor, which thus becomes the new sentinel node. This is safely achieved with a CAS operation.

In addition to these two operations, we implement a simple **lookup()** operation, which iterates over the queue. It stops if the desired element is found or if the end of the queue is reached. We use this extra operation to simulate concurrent traversal of the data structure. We further present two concurrent queue implementations based on this algorithm: one using smart pointers, the other one using transactional pointers.

13.2.2 Implementation with smart pointers

Instead of having a simple pointer to the next element in the list, each node holds a smart pointer. The `head` and `tail` nodes are defined as `std::shared_ptr`, as well as the new nodes that are appended. When iterating over the list, the iterator is also a smart pointer. We implement the concurrent queue as a C++ template, allowing generic data types `T` for the elements of the queue.

Listing 13.2 – Concurrent queue using C++ smart pointers: lookup() operation.

```
1  std::shared_ptr<Node> lookup(T val){
2    std::shared_ptr<Node> p = atomic_load(&head);
3    while(1){
4      if (p == nullptr) // end of list
5        return nullptr;
6      if (p->data == val) // element found
7        return p;
8      p = p->next;
9    }
10 }
```

We use CAS operations specifically defined for shared pointers in the C++ standard library `libstdc++-v3`, included in the GCC5 release.[1] The result of a CAS operation is repeatedly checked in a loop, until it confirms that the desired operation took place. The operations that insert and delete elements are easily implemented with shared pointers and CAS, by changing atomically the first element to the new node, respectively the last element to the next node in the queue.

Listing 13.1 illustrates the implementation of the list nodes. The `head` and `tail` nodes are declared as follows:

```
std::shared_ptr<Node> head;
std::shared_ptr<Node> tail;
```

They are initialized to the same empty node in the constructor of the queue class (not shown in the code). The implementation of the main two operations, i.e., `enq()` and `deq()`, follows closely the algorithm of the lock-free concurrent queue. The interface of the C++ smart pointers provides all functionality needed for directly adapting the algorithm to use their logic. The atomic operations (e.g., `atomic_load`, `atomic_compare_exchange_strong`) enforce safety and ensure that no intermediate results of the operations are seen during the execution.

The `lookup()` function (Listing 13.2) iterates over the list sequentially until it finds the requested value or reaches the end of the list. It starts with the current `head`. While this is not a typical operation for a queue, we add it to our implementation in order to further experiment with transactional pointers in this context. We consider that the list traversal could benefit the most from our implementation of `tx_ptr`s.

13.2.3 Implementation with transactional pointers

The next step was to change the above implementation to use transactional pointers. As mentioned before, since the `enq()` and `deq()` operations only need to swing a pointer to add or remove an element, they do not fit our target scenarios that can be improved by transactional pointers. Hence, we focus on the `lookup()` operation that iterates over the list. This operation also offers a suitable medium for experimenting with our batching optimization.

[1] In C++11 the operation `atomic_compare_exchange_strong(p, expected, desired)` checks if `p` has the same value as `expected`: if so, the value `desired` is atomically assigned to `p`; otherwise, `expected` becomes equal to `p`.

Listing 13.3 – Concurrent queue using C++ transactional pointers: lookup() operation.

```
1  std::shared_ptr<Node> lookup(T val, int batch_size){
2    bool add_to_tx = false; // first pointer should start a transaction

4    std::shared_ptr<Node> p(add_to_tx,batch_size, head); // start transaction
5    while(1){
6      if (p == nullptr)
7        return nullptr;
8      if (p->data == val)
9        return p;
10     p = p->next; // commit transaction and start a new one
11   }
12 }
```

Transactional pointers can be directly constructed from classical smart pointers. Therefore, instead of iterating with a pointer of type `std::shared_ptr` over the list of smart pointers, we use a pointer of type `tx_ptr` for this task. However, instead of incrementing and decrementing the reference count each time it passes over a smart pointer, the `tx_ptr` only starts a transaction and commits when stepping to the next node.

The only modification needed in the code is replacing the constructor of the pointer that will iterate over the list with the customized constructor for transactional pointers. Listing 13.3 shows the implementation of the `lookup()` function that uses transactional pointers. The only line that needs to be changed is emphasized (i.e., line 4). The transactional constructor starts the first transaction, which commits when moving to the next node. Then, a new transaction is started automatically and committed at the next step. This continues until the loop ends. In order to take advantage of batching, we also add another parameter to the signature of the function, called `batch_size`. It serves to set the size of the group of pointers we want to be encapsulated in a single transaction. Typically, this parameter is set to -1 for no batching or any value greater than 1 otherwise. Later, this parameter is passed to the transactional constructor. The customized constructor only needs the following additional information: if it must start a new transaction or the pointer it creates is added to an already existing transaction (parameter `add_to_tx`), the size of the batch and the smart pointer to be referenced. In this case, we start iterating from the `head`, i.e., a new reference to the first node is created, then it advances to the next nodes.

13.3 Concurrent linked list

We further extend our linked list data structure to execute insert and delete operations on any element, rather than just the ends. We consider this necessary for evaluating transactional pointers in more contended scenarios as well. In this case, the list traversal can conflict with update operations anywhere in the list.

13.3.1 Algorithm overview

Similar to the queue scenario, we rely on a classical concurrent linked list algorithm, more precisely the *lock-free list* algorithm proposed by Herlihy and Shavit [52, Chapter 9]. The list structure stays the same as for the queue, having two sentinel nodes and each node containing a value and a pointer to the next element in the list. It accepts the typical operations `add(T value)`, `remove(T value)` and `lookup(T value)`. The list is represented as a sorted set of unique values, as dictated by the lock-free list algorithm. The two sentinel nodes contain the minimum and maximum values permitted by the data type used. The list operations have the following particularities:

- **The lookup() operation** serves the same purpose as in the concurrent queue algorithm and keeps the same logic;
- **The remove() operation** finds the node with the requested value and *logically* removes it. More precisely, the node in question *is marked* as removed, but left in the list. This is done in order to avoid inconsistencies in the execution. The node containing the desired element (given as parameter to the `remove()` function) is found with the help of a function that iterates over the list and returns the nodes on either side of the element. A marked node is physically removed from the list in subsequent calls to a `find()` function, called either when removing or when inserting elements;
- **The add() operation** starts by finding the place of the new value in the sorted list. If the same value is already in the list, it returns. Otherwise, the `find()` function returns the nodes that will be the potential predecessor and successor of the new node. Then, the node is atomically inserted and linked with these nodes. If the atomic operation does not succeed (i.e., the list was modified by another thread in the meantime), new potential neighbor nodes are found and the operation retries.

In order for the algorithm to be correct, the list pointers have to be of a type that contains both the reference and a mark, and to be able to update them atomically, both at the same time or individually. Thus, all CAS operations needed when removing (logically or physically) or inserting an element, also compare the mark and update it atomically. This is the main difference in the list structure between the concurrent queue algorithm and the linked list.

13.3.2 Implementation details

We implement the concurrent lock-free list algorithm in C++ with raw pointers (as baseline), classical smart pointers and transactional pointers. Some modifications were necessary compared to the queue structure, in order to accommodate atomic marked pointers.

For the baseline implementation we rely on `std::atomic` library in C++, that allows us to use atomic operations on raw pointers. However, instead of a simple raw pointer for the `next` field, we use an aligned structure containing the pointer and the mark and a double-width CAS to update both of them atomically. The mark is an integer that accepts two values: 0 (the node is not marked for removal) and 1 (the node is marked for removal). We define a wrapper-function over the typical `std::atomic_compare_exchange_strong()` that handles marked-pointer structures and subsequently calls the standard operation. Besides this approach to marking node pointers, the implementation follows closely the theoretical algorithm.

For the implementation with std::shared_ptrs we adopt a different strategy. This is necessary because smart pointers are at least double in size than raw pointers, so a double-width CAS would only be enough for a std::shared_ptr object alone, leaving no room for the mark. Instead, we add a mark field in the control block of smart pointers. Thus, we do not modify the size of a smart pointer and each smart pointer has a mark directly associated. In addition, we rewrite the standard CAS operation for std::shared_ptrs to alter the internal mark as needed. With this strategy, the nodes of the linked list are defined as std::shared_ptr<Node> objects, as before. The benchmark that we use handles the list as an integer set; as such, for simplicity, we initialize the sentinel nodes with the minimum integer (INT_MIN) and maximum integer (INT_MAX), so that they always represent the extremities of the linked list. After these alterations, the rest of the operations are implemented exactly as dictated by the algorithm. The lookup() function has the same implementation as for the queue. It is important to note that, even though the reference counting in smart pointers helps solving some concurrency issues such as the ABA problem [52, Chapter 10], smart pointers are not thread-safe if read and modified at the same time. For example, the assignment of a smart pointer is not atomic. Therefore, synchronization through the atomic library was needed for a correct and safe execution when reading the value of the next pointer, the head, etc.

Finally, we provide an implementation featuring transactional pointers. Since we aim to directly compare the linked list with the queue, we kept the same logic: the transactional pointers are only used in the lookup() function with the goal of improving list traversal. This function has the same implementation for both data structures, thus there is no special modification to be done for adding transactional pointers other than what is described in Section 13.2.3.

13.4 Modifications to the initial tx_ptr implementation

The initial implementation of transactional pointers only started and committed transactions in the constructor and destructor of the transactional pointer. Moreover, batching was managed through the constructor as well, with the help of an extra parameter. This design was suitable for the scenarios presented in Section 11.5. However, our goal for concurrent data structures traversal is twofold: to encapsulate each iteration of the loop in the lookup() function in a hardware transaction; and to do so transparently for the user. In other words, we aim to make switching from classical smart pointers to transactional pointers as seamless as possible.

In order for the transactional pointers to work transparently, we modify the overloaded '=' (assignment) operator of C++ smart pointers to provide the required functionality. By default, this operator was responsible for assigning a raw pointer to the new std::shared_ptr and also share the reference count object. The reference count was altered to reflect the new state of the smart pointer, i.e., atomically incremented on both sides. In the case of transactional pointers, we must avoid touching the reference count, on the one hand, and we must handle transactions, on the other. In addition, we have to modify the assignment operator to handle batching. To this end, we implement an internal counter for tx_ptr and modify the '=' operator to commit and start a new transaction when the counter indicates the end of a batch. Whenever the transaction aborts due to a conflict, all the changes made to the group of pointers are rolled back and all pointers are recreated as common C++ smart pointers with reference counting.

These steps are illustrated in the pseudocode in Algorithm 6. On the left hand side we display the steps executed by the default implementation of C++ smart pointers, for comparison. On the right hand side, we show the necessary modifications for our strategy. First, we check if

Algorithm 6 Assignment operator: initial implementation (left) and extended implementation for transactional pointers logic (right).

```
1: function SMART_PTR::OP=(smart_ptr p)
2:     this.rawptr ← p.rawptr
3:     this.refcount ← p.refcount
4:     return this
5: end function
```

```
6: function TX_PTR::OP=(smart_ptr p)
7:     if is_fallback then
8:         FALLBACK(p)
9:         return this
10:    end if

11:    if curr_batch > 1 then
12:        this.rawptr ← p.rawptr
13:        this.state ← p.state
14:        curr_batch ← curr_batch − 1
15:    else
16:        TX_END()
17:        if curr_batch ≠ −1 then
18:            curr_batch ← batch_size
19:        end if
20:        if TX_START() then
21:            this.rawptr ← p.rawptr
22:            this.state ← p.state
23:        else
24:            FALLBACK(p)
25:        end if
26:    end if
27:    return this
28: end function
```

the fallback path has to be followed (line 7). This is especially important when batching is used and multiple pointers have to rollback and execute again the assignment method: if they are not protected by transactions anymore, they will find the variable `is_fallback` enabled. Thus, they will be aware they have to follow the path that turns them back to normal smart pointers. Otherwise, if no conflict or abort is detected, the execution further depends on the current state of the batch. If we are inside of a batch (that is, the group of pointers was not entirely processed yet), we just assign the raw pointer to the iterator and decrement the number of pointers left to process in the current batch (lines 11 – 14). This is done with the help of the aforementioned local counter, called `curr_batch`. Otherwise, if we reached the end of a batch (line 15), the current transaction commits (line 16), the counter is reinitialized with the batch size (line 18) and we try to start a new transaction for the next group of pointers (line 20). An important note is that we only decrement the current batch size and reinitialize it with the total size if batching was requested (i.e., if $batch\_size \neq -1$). Otherwise, a transaction is committed and started at every call of the assignment method. Finally, if the transaction starts successfully, we follow similar steps as in the constructor: we copy the raw pointer and the `state` field. If the transaction aborts, the fallback path described in Section 11.3 is followed.

Altering the assignment operator represents the most relevant change that was required in order to adapt our transactional pointers' logic to the data structure traversal implementation. Other `tx_ptr`-specific methods were kept as they were. The transactional constructor did not need any modification either, except for the new argument (i.e., `batch_size`).

13.5 Summary

This chapter presented two new scenarios in which we experimented with transactional pointers. We focused on concurrent data structures. Based on the same linked list structure we implemented a concurrent queue and a concurrent ordered set, with classical smart pointers and with transactional pointers. We justified our choice and explained the differences between the implementations. Only a single line needs to be changed in the implementation of the data structure in order to pass from smart pointers to transactional pointers. We also discussed what changes were necessary inside the `tx_ptr` logic, to make this transformation transparent for the user.

Chapter 14

Evaluation on Concurrent Data Structures

Contents

The goal in creating transactional pointers is to improve the performance of classical smart pointers and to learn what scenarios are most convenient for their usage. We motivated our choice for concurrent data structures in Section 13.1. We are not directly comparing the implementation with smart pointers with the most efficient linked list implementations, since it is a well-known fact that smart pointers provide worse performance than raw pointers [65, Chapter 4]. However, smart pointers have many other advantages. The use of C++ smart pointers for the nodes enhances simplicity (manual memory management is not required) and safety (possible illegal accesses are handled or eliminated automatically). Recent proposals in C++ development [80] recommend always using smart pointers. In what concerns data structures, for example, the ABA problem can be easily eliminated by using smart pointers instead of raw pointers. The reason is that a reference inside a smart pointer is kept alive as long as some thread still points to it, making it impossible for another thread to allocate the same address for different data. Thus, we are not explicitly trying to create smart pointers that match the efficiency of raw pointers, but we are rather focusing on improving their performance to some degree, while still keeping the memory management advantage. However, in the evaluation

that follows we also show the performance of an implementation with raw pointers as baseline, in order to place our results in a broader context.

14.1 Experimental setup

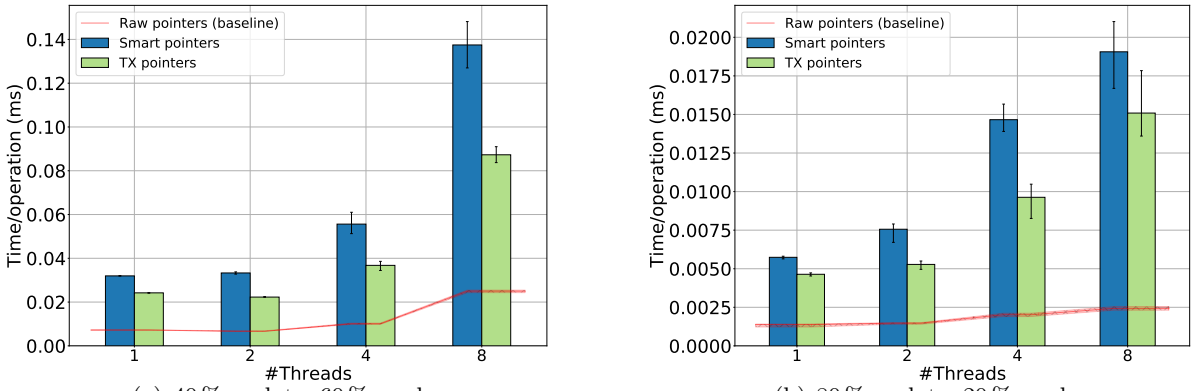
We experiment with the implementation based on the original C++ smart pointers and our transactional version, first without batching, then adding this feature as well. We test on two different architectures, as before: an 8-core (2 threads per core) Intel i7-5960X @ 3.00 GHz (codename Haswell) machine with 32 GB RAM and a 10-core (8 threads per core) IBM POWER8 @ 3.42 GHz with 30 GB RAM, both with fully integrated HTM support. We run the experiments on up to 8 threads pinned to the cores.

We develop a benchmark for comparing the performance of smart and transactional pointer implementations of the concurrent linked list structure (used as either a queue or an integer set). The benchmark works as follows. We initialize the list and populate it with elements. We start a number of threads that share the list. Each thread applies insert, remove and lookup operations on the shared list by a given ratio. We measure the time it takes each thread to finish the associated operations. In order for all threads to have comparable workloads, we generate the workloads before starting the threads. Specifically, we generate a random succession of operations according to the proportions given for each type of operation. Then, we generate a list of elements that will be inserted in the shared data structure, and a list of elements that will be looked up, based on the elements that are inserted. Given the dynamic character of the benchmark (a large number of concurrent insert and remove operations), not all the elements in the lookup-list will be found in the shared linked list at the moment when the operation is performed. When the list is used as an ordered set, specific elements must be passed to the `remove(T value)` function as well, as opposed to the concurrent queue which always dequeues the first element. For this, we use the same list as for the function that inserts elements. This way, we make sure that part of the elements are going to be already inserted in the list. Generally, around three quarters of the elements in the insert-list are found and removed.

We experiment with four main configurations, **three read-write workloads** and **one read-only workload**. The latter is a 100 % lookup workload, on a list initialized with 10^4 elements. For the queue implementation, each thread has to execute 10^6 operations on any workload. For the linked list implementation, each thread has to execute 10^5 operations on the shared list. The read-write workloads consist of the following combinations:

- 20 % insert, 20 % remove and 60 % lookup operations (balanced);
- 40 % insert, 40 % remove and 20 % lookup operations (update-intensive);
- 5 % insert, 5 % remove and 90 % lookup operations (read-intensive).

These three workloads are run on a list initially populated with 10^3 elements. At least half of the elements the threads search for are found in the list. We measure the time to execute the workload with the system's high resolution clock, `std::chrono::high_resolution_clock`. The result is manipulated as a `std::chrono::duration<double, std::milli>`, i.e., with millisecond accuracy. For each run we take the maximum between the times reported by each thread, then compute the mean over the 20 runs.



(a) 40 % update, 60 % read (b) 80 % update, 20 % read

Figure 14.1 – Execution time per operation on Intel Haswell for a concurrent queue implemented with smart pointers and transactional pointers, read-write workloads (no batching).

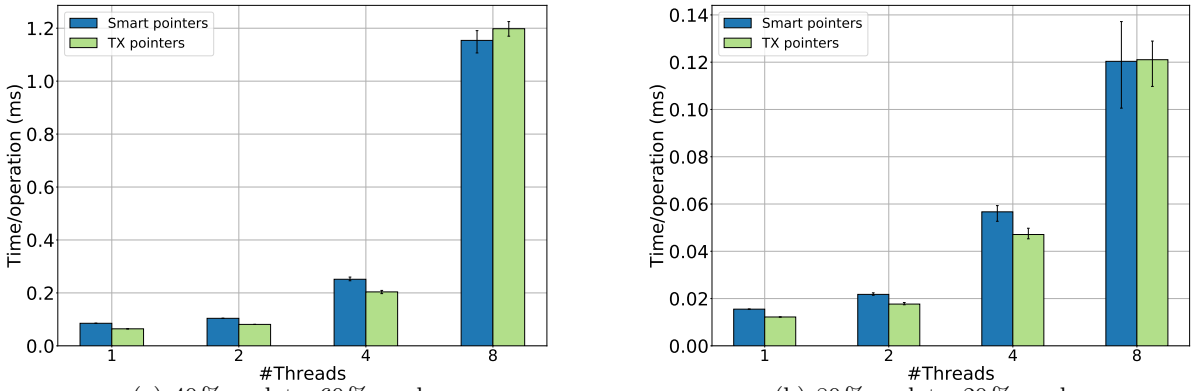
14.2 Read-write workloads

As a starting point, we evaluate the concurrent queue and the concurrent linked list on the read-write workloads. Both run the balanced workload, for comparison. Then, as a queue structure is usually update-intensive, we also test it on the second workload. Likewise, linked lists are search data structures, with a majority of lookup operations, hence we check its performance on the read-intensive workload. Generally, the measured times vary from one workload to the other and between architectures, depending on the contention level and proportion of lower-latency operations (e.g., update operations on the concurrent queue).

14.2.1 Concurrent queue evaluation

We first evaluate the simple transactional implementation, without batching, on the first two read-write workloads. The results on the Intel Haswell machine are shown in Figure 14.1. We show on the Y axis the time per operation in milliseconds. We also mark with a line the baseline performance, i.e., the implementation with raw pointers. The goal is to provide an intuition of how much closer to the best performance we get with our improvement over smart pointers.

In what concerns the balanced workload (Figure 14.1(a)), we observe that the transactional version performs generally better than the implementation using classical C++ smart pointers. Most of the times the improvement is good (24 % to 36 %). The execution time per operation worsens with the increasing number of threads, trend present on both implementations of smart pointers (and on the baseline). The performance gain with transactional pointers increases with the number of threads, reaching 36.5 % on 8 threads. Next, we analyze the results of the update-intensive workload (Figure 14.1(b)) and make the following observations. First, the results are similar to the outcome of the previous experiment. Specifically, we identify the same trend in what concerns scaling with the number of threads for both implementations independently: the performance worsens with the number of threads. However, the improvement with respect to classical smart pointers does not scale up to 8 threads anymore, being the most prominent on 4 threads (i.e., 34 %). This happens because the write-intensive workload involves increased contention with the growing number of threads. Thus, the performance gain starts dropping after 4 threads, reaching 20 % on 8 threads. The second important observation is



(a) 40 % update, 60 % read

(b) 80 % update, 20 % read

Figure 14.2 – Execution time per operation on IBM POWER8 for a concurrent queue implemented with smart pointers and transactional pointers, read-write workloads (no batching).

that the transactional pointers seem to suffer more from contention than the original atomic operations. The gain is somewhat smaller than for the other workload.

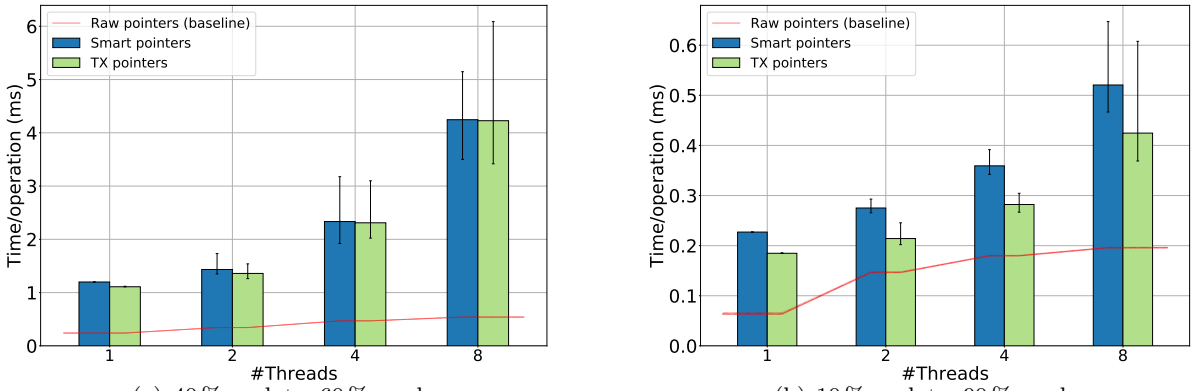
We run the same experiments on the IBM POWER8 machine (Figure 14.2). We do not show the baseline in this case since we expect it to be proportionally similar to the above results. We observe that the execution time per operation is generally longer on the POWER8 architecture for the same experiments on Intel Haswell. The improvement is less prominent as well (at most 24 % on any of the workloads). What is more, on this architecture the gain with transactional pointers constantly decreases with the increasing number of threads. On both workloads we find the smallest improvement on 4 threads: 18 % on the balanced workload and 16 % on the update-intensive workload. On 8 threads and both workloads, transactional pointers perform marginally worse than original smart pointers, showing an overhead of 3 % and 0.5 % on the balanced, respectively the update-intensive, workload. We believe this happens because of the increasing contention.

Overall, these results indicate that, even if we replace a single pair of atomic operations with a hardware transaction, we already start gaining in performance.

14.2.2 Concurrent linked list evaluation

We further experiment with a linked list implementation in order to have a scenario in which the transactions can conflict with insert and remove operations for the entire length of the list, not only at the extremities. We aim to understand to what extent transactional pointers are impacted by contention. We already have an idea from the queue experiments on the update-intensive workload, but in the case of the linked list the potential conflicts are spread over all elements.

Figure 14.3 shows the results for the balanced workload and for read-intensive workload. As for the queue experiments, we also include the baseline to have an idea about how raw pointers would perform on this algorithm. The observations made on the queue structure mostly hold in this scenario as well: the time per operation does not scale with the number of threads. We find the smallest gain on 8 threads (5 % for the first workload, respectively 18 % on the read-intensive one). We compare directly the outcome for the list and for the queue on the balanced workload. It is obvious in this case that, while the performance of transactional pointers is still slightly



(a) 40 % update, 60 % read (b) 10 % update, 90 % read

Figure 14.3 – Execution time per operation on Intel Haswell for a concurrent linked list implemented with smart pointers and transactional pointers, read-write workloads (no batching).

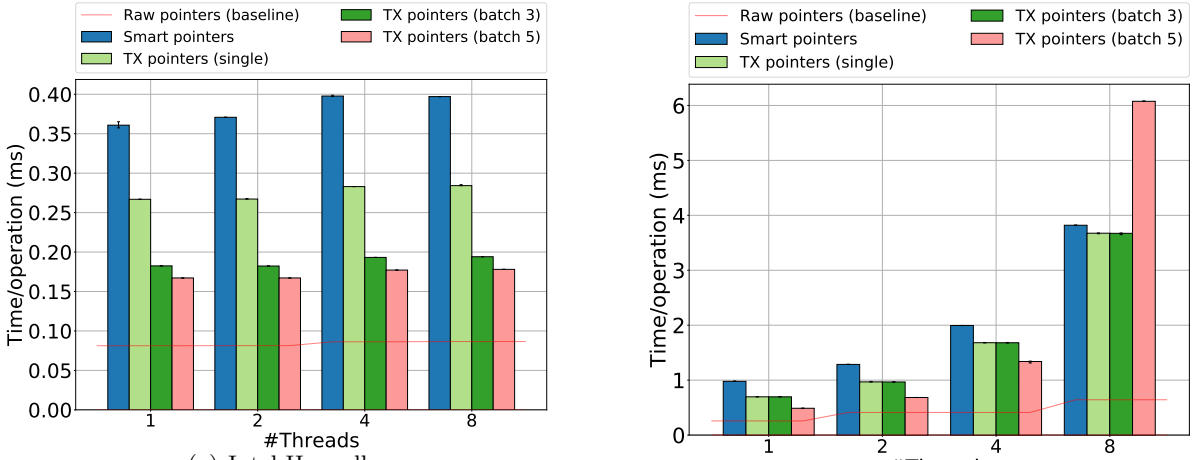
better, the gain is negligible (between 7 and 0.5 % improvement). The two types of pointers have comparable results, both far from the baseline. We conclude that both the type of workload as well as the extent of contention are important factors in the decision of using transactional pointers in a particular scenario.

On the other hand, we consider the distribution of operations for the read-intensive workload more appropriate for real-life workloads on search data structures. Figure 14.3(b) shows the performance of our transactional pointers on a linked list running this workload. In this favorable case for a less-contended list traversal, we find that transactional pointers improve smart pointers with up to 22 %. Generally, the improvement is more constant in this case, only varying between 18 and 22 %. These experiments confirm the findings in Chapter 12: if the workload is convenient for the case of transactional pointers, than the improvement is non-trivial; otherwise, they have comparable performance with smart pointers.

14.3 Read-only workload

For analyzing the performance on the read-only workload we can use either the queue or the list, since the implementation for the lookup operation is the same for the two. Moreover, for either of the two structures, the only difference between its two implementations (i.e., with `std::shared_ptr` and with `tx_ptr`) is in the way in which the lookup function works. As such, we focus on stressing and comparing strictly this operation. Figure 14.4 shows the results in this scenario for the implementation with the original C++ smart pointers, as well as transactional pointers with one transaction per pointer, one transaction for a group of 3 pointers, and one transaction for a group of 5 pointers.

First, looking at the results on the Intel Haswell architecture (Figure 14.4(a)), we observe that the average execution time per operation has the same order of magnitude as in the read-intensive workload on the linked list and one order of magnitude higher than the update-intensive workload on the concurrent queue. Normally the queue update operations have a smaller latency than the lookup, hence the shorter reported times for a combined workload. The results on the read-only workload are also more stable. Second, we note that the improvement for a single transactional pointer (i.e., without batching) stays at 26 % – 28 %, consistent with the



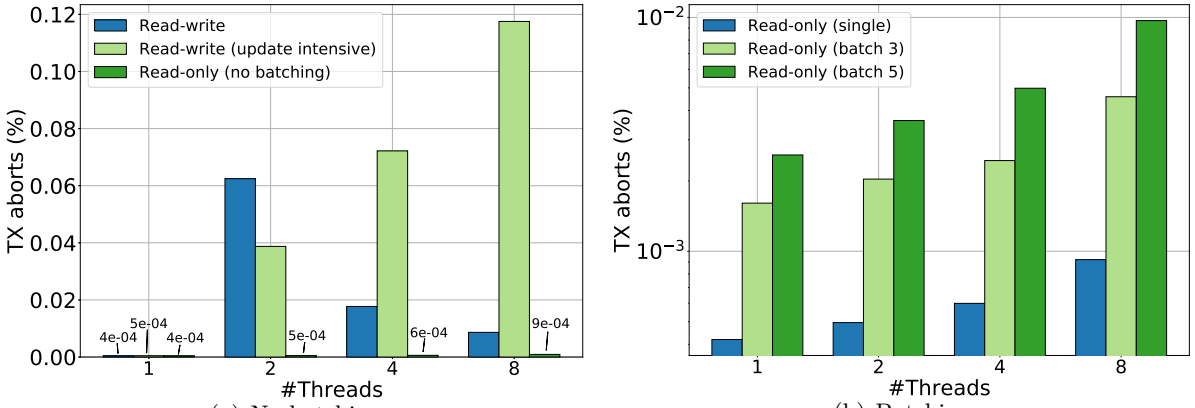
(a) Intel Haswell

(b) IBM POWER8

Figure 14.4 – Execution time per operation for a concurrent queue implemented with smart pointers and transactional pointers, with 100 % lookup operations (batching enabled).

first read-write experiment on the queue and read-intensive one on the linked list. Then, as we group more pointers inside a hardware transaction, the transactional implementation is around 50 % faster (batch of 3 pointers), respectively 55 % faster (batch of 5 pointers) than the original implementation. Thus, we observe that the performance increase is more spectacular when passing from no batching to a group of 3 pointers than from a batch of 3 to one of 5 pointers. While the batch size increases, the performance improvement reaches a plateau and starts degrading when the batch becomes too large for being handled properly by a hardware transaction. We also experimented with batches of 10 and 20 pointers inside a single transaction (not shown in the figure) and we noticed that the gain was negligible compared to the one on 5-pointer groups. Regarding scaling with the number of threads, we mostly observe the same trend as for the balanced workload on the queue: the improvement increases slightly with the number of threads.

When comparing the results on Haswell with the ones on POWER8, we make several interesting observations. The trends are similar for the two architectures: scaling with the batch size and with the number of threads, *in most cases*. A batch of 5 pointers results in up to 50 % improvement. But this is where the similarities stop. On POWER8, the scaling with the batch size is more notable from a batch of 3 pointers to a batch of 5, than from no batching to batch of 3 pointers. Also, the performance increase with the number of threads is much more visible than on Haswell. However, the actual improvement over smart pointers is decreasing with the number of threads: from 28 % down to 3 % for a single pointer and batch of 3 pointers and from 50 % improvement down to 50 % overhead for a batch of 5 pointers. The case where we use groups of 5 pointers on 8 threads is the only one in which we encounter such an overhead. We believe that this is explained by the contention and the high cost of rolling back a bigger group of pointers in case of abort.



(a) No batching (b) Batching

Figure 14.5 – Evolution of abort rate with respect to the number of concurrent threads, type of workload (left) and batch size for a read-only workload (right).

14.4 Discussion: abort rate

Finally, we analyze the impact of abort rate on the performance of transactional pointers. For this study, we employ the concurrent queue implementation, with the various workloads described. We run the experiments on the Intel Haswell server and measure transactional statistics with the `perf` tool. We look at the statistics for the following events: `tx-start`, `tx-abort` and `tx-commit`. They indicate the number of transactions that were started, how many out of these were aborted and, respectively, committed. The reported results represent the mean over 20 runs. We focus on two main cases: **how does the abort rate evolve with respect to contention** (Figure 14.5(a)), and **how it changes with respect to the batch size on a read-only workload** (Figure 14.5(b)). Both figures show the percentage of aborted transactions on the Y axis. The order of magnitude for the number of started transactions (for which we compute the abort rates) is between 9 and 11, depending on the number of threads and batch size.

For the former scenario, we compare the abort rate on the first two read-write workloads and the read-only workload without batching. We emphasize the values that are under 10^{-3} % with a label indicating the value, since the corresponding bars are not visible. We observe that the abort rate increases with the increasing number of threads for the update-intensive workload, while doing the opposite for the balanced scenario. We believe this happens because in the update-intensive scenario there are fewer transactions started (smaller proportion of lookup operations), but more aborts due to the increased contention with the number of threads. For the balanced workload we have many aborts as well, but the number of transactions started grows faster with the number of threads. As expected, the abort rate is also greater for the update-intensive workload than for the other two. Likewise, the read-only workload has the smallest rate: it never exceeds 10^{-3} % aborted transactions. We can say that, generally, the abort rates are negligible in all workloads, having less than 0.5 % aborts in the most contended scenario. However, if we correlate these results with the execution time per operation observed in Section 14.2.1, we can speculate that the number of aborts, among other issues, plays a role in the weaker performance of the transactional pointers. Thus, we conclude that it is important to study the potential abort rate when deciding what scenario would benefit most of a transactional smart pointer implementation.

In the latter scenario we studied the evolution of the abort rate with the batch size. Again, the trend is according to our expectations: the abort rate increases both with the growing batch size as well as with the increasing number of threads. However, the worst abort rate was under 10^{-2} % of the total number of started transactions. As long as the batch is small enough to avoid overflowing the transaction, the impact of aborts on the execution time in the read-only scenario is minimal. However, it is important to note that, if the transaction protecting a large batch aborts, then it is rolled back and all the transactional pointers are restarted as classical smart pointers. This may reduce the beneficial effect of batching. Therefore, for a good performance it is important to find a suitable batch size that balances gains and group proportions.

14.5 Summary

This chapter concludes the evaluation of HTM-based smart pointers. We experimented with two concurrent data structures on four different workloads. An important insight is that some scenarios proved to be more appropriate for using transactional pointers, case in which they could improve the performance of classical smart pointers by up to 55 %. Otherwise, transactional pointers did not degrade the performance, compared to smart pointers. We also studied the impact of aborts on the transactional pointers' performance for various workloads. We found that the number of aborts in this context is negligible and has a minimal effect on the outcome.

Part IV

Conclusions and Future Work

Chapter 15

Conclusions

While the computing world moves towards an ever increasing number of processors in commodity hardware and writing correct software becomes overly complex, programming paradigms tend to turn in the direction of *lock-free synchronization* and *automatic memory management*. The aim of this work was to combine the two concepts and **to integrate a novel synchronization method into memory management mechanisms**.

Even though hardware transactional memory (HTM) has already been offered for public use in the last few years as a convenient alternative to the classical locking systems, it is yet to be discovered which are its major advantages in practice. We believe that HTM represents the perfect answer to the various performance problems that still discredit automatic memory management in general. Since transactional memory is able to run pieces of code atomically without blocking, it was an important breakthrough in multi-core synchronization. However, the first implementations (in software) were less practical than desired. Efforts have been made for integrating TM in hardware and in 2013 the first implementations started to appear in commodity hardware. Since then, a lot of research has been done on finding scenarios and situations that benefit most from HTM-based synchronization. We complement this work with results and recommendations on how to improve the performance of memory management in two widely-used real-world frameworks with HTM, namely Java and C++.

More precisely, we focused on the following aspects:

Extensive study on the performance of Java GCs. We evaluated the extent to which garbage collectors affect the execution of an application, with respect to throughput and responsiveness. We found that the results obtained from testing the GC on a benchmark suite are contradicted to some degree by the real-life usage in a memory-intensive server application. We believe that one of the reasons for this was the small memory footprint of the benchmarks, which have been implemented in 2009 for smaller memories than on today's servers. However, by running the benchmarks we learned that the GCs are not always behaving as expected; contrary to our assumptions, they showed that:

- enabling the TLAB is not always beneficial, but can also decrease the performance of the application;
- the average pause time of the GC can increase with the decreasing young generation size, for the same heap size;
- for the benchmark suite, the concurrent GCs (ConcurrentMarkSweep and G1) were ranked last in terms of performance, while the default GC, ParallelOld, proved to be the most stable, with good performance in terms of GC pauses and total execution time.

However, our experiments on the Apache Cassandra Server, using a large heap and young generation, indicated the opposite: in this situation, ParallelOld resulted in huge pause times for the application (hundreds of seconds, up to 4 minutes long), becoming unacceptable in practice. Even G1 and ConcurrentMarkSweep collectors introduced pauses of a few seconds (up to 3.5 seconds for G1), which might affect the application in case of a real-time or a distributed system, where the nodes need to communicate without delay. We showed that the encountered garbage collections also harm the user experience: almost every peak in the client response time was associated to a collection on the server.

Algorithm for an HTM-based pauseless GC in Java. The most widely used collectors in OpenJDK8 – ParallelOld, ConcurrentMarkSweep, and G1 GC – are all parallel, but none of them is fully concurrent. In other words, all of them use a different thread for collecting but, at the same time, they all need to stop the application threads at different phases of their collection. Moreover, as previously mentioned, the pauses caused by the GCs are significant. These limitations led us to consider creating a concurrent GC algorithm that employs hardware transactional memory. Not only would this remove main stop-the-world pauses, but it would permit the GC to take full advantage of a multi-core system without draining its resources.

There are two different approaches for implementing a GC with HTM support: one could either instrument the GC or the application accesses. In our algorithm we followed the latter approach, and replaced the blocking access barriers of the mutators with transactions. We then devised a few optimizations for improving the GC implementation.

Transactional barriers: implementation and evaluation. We built the implementation on top of HotSpot's CMS collector. Before fully implementing a new GC, we first focused on assessing the overhead introduced by our transactional implementation with the goal of verifying whether our approach is feasible. We tested on a widely-used benchmark suite, DaCapo, and on a real-life client-server system with Apache Cassandra. We implemented the transactional barriers in the interpreter and in the JIT compiler. We found that in interpreter mode, after a first optimization of the algorithm, we obtained a reasonable overhead of less than 5 % for all benchmarks. However, by checking the equivalent overhead values for transactional barriers in the JIT compiler, we found that it was not within reasonable limits, having 5 out 6 DaCapo benchmarks and 6 out of 7 Cassandra workloads with overheads greater than 20 %, going as high as 195 % for one DaCapo benchmark.

We reassessed the algorithm and proposed a second optimization, *selective access barriers*. The barriers select the fastest way to proceed depending on the GC state. If a collection is in progress, they install transactional barriers and allow the application to continue concurrently with the GC, while all potentially dangerous accesses are monitored in

hardware; otherwise, they do not cause any overhead. We found a lower bound for the possible savings and we showed that even in these conditions the previous overhead was reduced by 30 % to 40 % for write-intensive workloads and by up to 93 % for read-intensive workloads. We further estimated the CPU time of a potential transactional GC based on the current data and observed that it was comparable to CMS in all cases. This is a good indication that the selective barriers make a full implementation of the pauseless transactional GC worthwhile, as opposed to the former transactional barriers. Therefore, we also made some considerations regarding the challenges encountered when implementing such a GC, and lined up the next steps towards its completion.

HTM-based smart pointers in C++. We designed a transactional pointer structure on top of the C++ `std::shared_ptr`, which uses reference counting for correctly managing the memory. Reference counting is a convenient form of memory management with interesting properties and synchronization features, where each object is protected from invalid accesses by keeping a shared reference counter. In some cases the atomic increment/decrement operations on the shared counter prove to be unnecessary and expensive. We considered this to be a promising opportunity for improvement with HTM. Our goal was to replace the atomic operations needed for the creation and destruction of the smart pointer with a hardware transaction. We experimented with micro-benchmarks, in single- and multi-threaded settings, on two different architectures and with the possibility of batching multiple pointers in a transaction. We also compared the performance of the original and transactional implementations on concurrent data structures: a queue and a linked list. We obtained up to 55 % improvement over smart pointers on read-intensive workloads, and no deterioration in contended scenarios. We believe that the results provide valuable insights into which scenarios would benefit most from using transactional pointers.

In conclusion, this work aimed to provide a few solutions for improving known real-world issues with a novel synchronization technique, hardware transactional memory. Throughout our experiments, we learned what are some of the downsides and advantages of HTM. Most importantly, we observed that HTM does not provide a straightfoward, easy to use and efficient solution in all cases. Rather, it usually needs a fair amount of tweaking to work in the desired way. This is partly due to the implementation, inherently limited in hardware. However, despite all these issues, once optimized to fit a particular problem, HTM provides considerable improvement over classical synchronization methods. We applied these findings to automatic memory management. The real-life systems we chose to improve (i.e., Java's GCs and C++ smart pointers) were already optimized and widely used. We showed there is still room for improvement and that HTM is a good candidate for tackling specific issues in this context. We believe our work is a stepping-stone towards implementing fully-concurrent Java GCs and lower-latency smart pointers for concurrent data structures traversal.

Chapter 16

Future Work

Contents

The subject addressed in this thesis is broad and popular in the research community. Our experiments and conclusions put the basis for larger, complete systems that can be practically used at a large scale. As such, this work can be extended on multiple directions. This chapter assembles a few pointers to potential future development that will advance the work towards this goal. We address separately garbage collection in managed systems and other automatic memory management mechanisms. We start with closely-related topics that directly extend our current work, going to entirely new research perspectives on the aforementioned domains, and comment what are their respective future prospects.

16.1 HTM-based pauseless GC

In our work we designed an algorithm for a fully-concurrent GC that employs HTM for synchronizing the collector threads with the application threads. We only partly implemented the algorithm, but sufficiently for running a feasibility study. Our results suggested that the full implementation of the algorithm would improve current GCs and we provided the next steps necessary for its completion. We considered the rest to be mostly an engineering effort. However, as future work, there is no denying that **the full implementation of the HTM-based Java GC algorithm** that we designed is an important and obvious direction. The main obstacle that needs to be overcome is removing the current blocking synchronization, which represents an intricate and complex part of the HotSpot JVM. As an example, there are over 10 locking constructs used only by the VM thread to handle VM operations in its main loop and associated functions. Certainly, only some of them are related to the GC activity, but their effects can be

traced across many classes and source files. Java HotSpot has a complex way of ensuring runtime safety, based on internal assumptions regarding the stack, when the mutators must rendez-vous, etc. These assumptions are enforced with asserts and guarantees, which can sometimes be high-level and easy to relax in order to change the JVM logic; other times, the same asserts can be vital for identifying errors. We give these few disparate examples with the aim of providing an intuition of what lies behind the full implementation of the HTM-based GC. They show that an in-depth knowledge of all JVM internals to the smallest detail is required for this endeavor. However, once completed, we expect it to offer further insight on the benefits of HTM-based synchronization. We believe that such a GC would be a valuable addition to the current state-of-the-art, opening the way for a new generation of pauseless GCs. Moreover, exactly the same transactional algorithm can be applied to the old generation collector, in order **to make CMS both concurrent and compacting**. The current implementation of ConcurrentMarkSweep GC suffers from memory fragmentation, employing lists of memory chunks to keep track of free memory. Transactional barriers can be used to safely move objects in the old generation, similar to the copying process in the young generation. One critical modification to the current implementation of CMS would be the addition of a forwarding pointer in the header of the promoted objects. We believe that this optimization would enhance even more the performance of a pauseless transactional GC.

There are two different ways in which a GC can be combined with HTM: on the application side (our approach) and on the GC side. Another research direction of interest would be **to design a GC algorithm that uses HTM on the GC side**. There is already literature exploiting this approach (the Collie collector [55]), but the solution has two important drawbacks: it only uses one core for collection (it is not parallel), and it needs two passes over the object graph. Both issues can lead to memory exhaustion during the collection. We believe that current HTM implementations could be successfully used to extend the logic of the Collie GC to support parallel collections on a reserved subset of the machine cores and to require only one pass over the object graph, by avoiding illegal accesses with the help of HTM.

Moving away from the platform on which this work was developed, we consider there are many opportunities of improving garbage collection on various managed systems. For example, Python and its PyPy compiler could benefit of a fully-concurrent GC. There were some attempts of improving Python's performance with HTM, but they were mostly focusing on its general synchronization mechanisms, not on the garbage collector. We find there is room for improvement and that the context is ideal for applying the insights gathered through our work, regarding both HTM and garbage collection in general. Therefore, another viable direction for future work is **enhancing Python with a concurrent HTM-based GC**. The same reasoning applies to other scripting languages, such as Ruby. There is not sufficient research done in this direction, leaving many opportunities for experimenting with HTM in the context of dynamically-typed languages.

16.2 HTM-based reference-counting

During our work we experimented with HTM in the context of the reference-counting memory management strategy. We focused on C++ smart pointers, as an interesting widely-used data type that employs this mechanism. We designed transactional smart pointers specialized on concurrent data structure traversal. We showed their efficiency and capabilities on a linked list data structure. As a future project, we aim to further implement **more complex data**

structures with transactional pointers, in order to verify whether our findings hold for tree structures or hash tables. We expect the latter in particular to greatly benefit from HTM-based pointers, since the lookup operations showed such a great improvement on a linked list structure. Usually the hash table buckets are implemented as singly linked lists and lookup operations are the majority on any hash table. As for the tree structures, they represent a good opportunity to further optimize and specialize transactional pointers in case they are not yet fit for improving lookup performance on trees. At the same time, we also plan to **extend the linked list implementation to use transactional pointers in all functions that iterate over the list**, not only in the lookup function. An example of such a function is the one that finds the neighbors of nodes that are removed or inserted. As a side goal, we would like to design a stronger evaluation. Up to this point, we experimented with transactional pointers on a simple benchmark, especially tailored to stress the data structure as much as possible. We aim to **extend the current evaluation to well-established benchmark suites** for concurrent data structures, such as the Synchrobench benchmarks [45]. The results could give more confidence and better insight on the benefits of HTM-based smart pointers.

After experimenting with smart pointers, we drew the conclusion that HTM can improve the synchronization through reference counting in some scenarios. Garbage collectors based on this strategy are nowadays avoided because of the performance penalty. Another interesting research direction would be **to enhance a reference-counting GC with HTM**. Recently, many optimizations were proposed for reference-counting GCs and they are trying to make a comeback. However, they are still lagging behind, performance-wise, their improvement not offering enough incentive for the industry world to adopt them on a large-scale. We think that HTM would help increasing the concurrency and reduce latency overhead. According to our results, a hardware transaction should be able to successfully replace a pair of atomic operations on a shared reference counter and to shorten the delay of their execution. An extensive evaluation is needed to establish how they compare with real-life GCs. An extension to this idea would be to design a concurrent Java GC based on reference-counting and HTM and compare its performance with the full implementation of the HTM-based GC algorithm presented in this work.

Finally, there are other widely-used programming languages that rely on reference-counting. An exciting subject would be **to improve the reference-counting mechanism in Objective-C**, the main automatic memory management technique on both MacOS and iOS. Recently, a number of devices running the former already have hardware support for transactional memory. An in-depth study on how reference-counting performs in this context and how it could be further optimized, potentially with HTM, would be a new and interesting step for our research. It may allow us to extend our work to mobile devices, a popular research platform in the last few years. We consider garbage collection on embedded systems to be generally compelling as an extension of our current research.

Bibliography

[1] The Apache Cassandra project. http://cassandra.apache.org.

[2] CompressedOops. https://wiki.openjdk.java.net/display/HotSpot/CompressedOops.

[3] Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 55–64, New York, NY, USA, 2011. ACM.

[4] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: an automated transactional approach to concurrent memory reclamation. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 81–92, New York, NY, USA, 2003. ACM.

[6] David F. Bacon, Perry Cheng, and V. T. Rajan. The Metronome: a simpler approach to garbage collection in real-time systems. In *On the Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pages 466–478. Springer, 2003.

[7] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298, New York, NY, USA, 2003. ACM.

[8] David F. Bacon, Perry Cheng, and Sunil Shukla. And then there were none: a stall-free real-time garbage collector for reconfigurable hardware. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 23–34, New York, NY, USA, 2012. ACM.

[9] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[10] Alexandro Baldassin, Edson Borin, and Guido Araujo. Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 87–96, New York, NY, USA, 2015. ACM.

[11] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[12] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 344–358, New York, NY, USA, 2003. ACM.

[13] Colin Blundell, E. Christopher Lewis, and Milo Martin. Deconstructing transactional semantics: the subtleties of atomicity. In *Proceedings of the 4th Annual Workshop on Duplicating, Deconstructing, and Debunking*, WDDD '05, pages 48–55, 2005.

[14] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, New York, NY, USA, 1991. ACM.

[15] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[16] Lars Frydendal Bonnichsen, Christian Wilhelm Probst, and Sven Karlsson. Hardware transactional memory optimization guidelines, applied to ordered maps. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 3, pages 124–131. IEEE, 2015.

[17] Steven R. Brandt, Hari Krishnan, Gokarna Sharma, and Costas Busch. Concurrent, parallel garbage collection in linear time. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 47–58, New York, NY, USA, 2014. ACM.

[18] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 256–262, New York, NY, USA, 1984. ACM.

[19] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. A performance study of Java garbage collectors on multicore architectures. In *Proceedings of the 6th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 20–29, New York, NY, USA, 2015. ACM.

[20] 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? *Queue*, 6(5):40:46–40:58, September 2008.

[21] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed

storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008.

[22] Hyung-Kyu Choi, HyukWoo Park, and Soo-Mook Moon. Biased allocator for generational garbage collector. In *International Workshop on Dynamic Compilation Everywhere*, DCE '14, Viena, Austria, 2014.

[23] Cliff Click. HTM will not save the world. Presentation at Transactional Memory Workshop (TMW10), 2010.

[24] Nachshon Cohen and Erez Petrank. Data structure aware garbage collector. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 28–40, New York, NY, USA, 2015. ACM.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[26] Lawrence Crowl, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Integrating transactional memory into C++. In *Workshop on Transactional Computing*, TRANSACT '07, 2007.

[27] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *Workshop on Transactional Computing*, TRANSACT '07, 2007.

[28] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: the secret to scaling concurrent search data structures. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 631–644, New York, NY, USA, 2015. ACM.

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[30] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[31] David Detlefs and Tony Printezis. A generational mostly-concurrent garbage collector. Technical report, Mountain View, CA, USA, 2000.

[32] Dave Dice. Biased locking in HotSpot. https://blogs.oracle.com/dave/biased-locking-in-hotspot.

[33] Dave Dice, Tim Harris, Alex Kogan, Yossi Lev, and Victor Luchangco. The influence of malloc placement on TSX hardware transactional memory. In *Workshop on Transactional Computing*, TRANSACT '16, 2016.

[34] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '0Workshop on Transactional Computingtent9, pages 157–168, New York, NY, USA, 2009. ACM.

[35] David Dice, Mark S. Moir, and William N. Scherer III. Quickly reacquirable locks, 2010. US Patent 7,814,488.

[36] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[37] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, April 2011.

[38] Aleksandar Dragojevic, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 99–108, New York, NY, USA, 2011. ACM.

[39] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[40] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: an open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, pages 13:1–13:9, New York, NY, USA, 2016. ACM.

[41] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. *Best papers from PLOS'11, SIGOPS Operating System Review (OSR)*, 45(3):15–19, January 2011.

[42] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM.

[43] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle, USA, 2015.

[44] Justin E. Gottschlich and Hans-J. Boehm. Generic programming needs transactional memory. In *Workshop on Transactional Computing*, TRANSACT '13, 2013.

[45] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 1–10, New York, NY, USA, 2015. ACM.

[46] Vincent Gramoli, Petr Kuznetsov, and Srivatsan Ravi. In the search for optimal concurrency. In *Proceedings of the 23rd International Colloquium on Structural Information and Communication Complexity*, SIROCCO '16, pages 143–158, Cham, 2016. Springer International Publishing.

[47] Tom R. Halfhill. POWER8 hits the merchant market. Memory bandwidth helps IBM server processor ace big benchmarks. *Microprocessor report*, 2014.

[48] William Hasenplaugh, Andrew Nguyen, and Nir Shavit. Quantifying the capacity limitations of hardware transactional memory. In *Workshop on the Theory of Transactional Memory*, WTTM '15, 2015.

[49] Isuru Herath, Demian Rosas-Ham, Daniel Goodman, Mikel Luján, and Ian Watson. A case for exiting a transaction in the context of hardware transactional memory. In *Workshop on Transactional Computing*, TRANSACT '12, 2012.

[50] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[51] 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*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[52] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., USA, 2008.

[53] Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying GC without stopping the world. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pages 48–57, New York, NY, USA, 2001. ACM.

[54] Balaji Iyengar, Edward Gehringer, Michael Wolf, and Karthikeyan Manivannan. Scalable concurrent and parallel mark. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 61–72, New York, NY, USA, 2012. ACM.

[55] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. The Collie: a wait-free compacting collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, New York, NY, USA, 2012. ACM.

[56] Richard Jones and Rafael D. Lins. *Garbage Collection: algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[57] Tomas Kalibera, Filip Pizlo, Antony L. Hosking, and Jan Vitek. Scheduling hard real-time garbage collection. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 81–92. IEEE, 2009.

[58] Gabriel Kliot, Erez Petrank, and Bjarne Steensgaard. A lock-free, concurrent, and incremental stack scanning for garbage collectors. In *Proceedings of the 2009 International Conference on Virtual Execution Environement*, VEE '09, pages 11–20, New York, NY, USA, 2009. ACM.

[59] Manaschai Kunaseth, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, David F. Richards, and James N. Glosli. Performance characteristics of hardware transactional memory for molecular dynamics application on Blue Gene/Q: toward efficient multithreading strategies for large-scale scientific applications. In *Proceedings of the 27th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum*, IPDPSW '13, pages 1326–1335. IEEE, 2013.

[60] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the 30th International Conference on Data Engineering*, ICDE '14, pages 580–591. IEEE, 2014.

[61] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, Eliot Moss, Aashish Phansalkar, Darko Stefanovik, Thomas Vandrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51:83–89, August 2008.

[62] Christian Märtin. Multicore processors: challenges, opportunities, emerging trends. In *Proceedings of Embedded World Conference*, pages 25–27, 2014.

[63] Phil McGachey, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. Concurrent GC leveraging transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 217–226, New York, NY, USA, 2008. ACM.

[64] Remigius Meier, Armin Rigo, and Thomas Gross. A transactional memory system for parallel Python. https://bitbucket.org/pypy/extradoc/raw/12bda771698925b8296808959e7b830aef8b78b8/talk/dls2014/paper/paper.pdf, 2014.

[65] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[66] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: a high-performance Big-Data-friendly garbage collector. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 349–365. USENIX Association, 2016.

[67] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 195–212, New York, NY, USA, 2008. ACM.

[68] Rei Odaira, Jose G. Castanos, and Hisanobu Tomari. Eliminating global interpreter locks in Ruby through hardware transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 131–142, New York, NY, USA, 2014. ACM.

[69] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 159–172, New York, NY, USA, 2007. ACM.

[70] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 69–82, New York, NY, USA, 2010. ACM.

[71] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the 2010 International Conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

[72] Nicholas Riley and Craig Zilles. Hardware tansactional memory support for lightweight dynamic language evolution. In *Proceedings of the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 998–1008, New York, NY, USA, 2006. ACM.

[73] Carl G. Ritson, Tomoharu Ugawa, and Richard E. Jones. Exploring garbage collection with Haswell hardware transactional memory. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 105–115, New York, NY, USA, 2014. ACM.

[74] Konstantinos Sagonas and Jesper Wilhelmsson. Mark and split. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 29–39, New York, NY, USA, 2006. ACM.

[75] Arnold Schwaighofer. Tail call optimization in the Java HotSpot VM. Master thesis, Johannes Kepler Universität Linz, March 2009.

[76] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? Getting reference counting back in the ring. In *Proceedings of the 2012 International Symposium on Memory Management*, pages 73–84, New York, NY, USA, 2012. ACM.

[77] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 93–110, New York, NY, USA, 2013. ACM.

[78] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang Ni, and Adam Welc. Towards transactional memory semantics for C++. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 49–58, New York, NY, USA, 2009. ACM.

[79] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 11–20, New York, NY, USA, 2010. ACM.

[80] Herb Sutter. Atomic smart pointers. Technical Report N4058, 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4058.pdf.

[81] Fuad Tabba. Adding concurrency in Python using a commercial processor's hardware transactional memory support. *ACM SIGARCH Computer Architecture News*, 38(5):12–19, April 2010.

[82] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the 2011 International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

[83] András Vajda. *Programming Many-core Chips*. Springer Science & Business Media, 2011.

[84] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[85] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.