Concurrent Copying Garbage Collection with Hardware Transactional Memory

Zixian Cai
蔡子弦

A thesis submitted in partial fulfilment of the degree of
Bachelor of Philosophy (Honours) at
The Australian National University

November 2020

Typeset in TeX Gyre Pagella, URW Classico, and DejaVu Sans Mono by XeTeX and XeLaTeX.

Except where otherwise indicated, this thesis is my own original work.

Zixian Cai
12 November 2020

To 2020, what does not kill you makes you stronger.

Acknowledgments

First and foremost, I thank my *shifu*[1], Steve Blackburn. When I asked you how to learn to do research, you said that it often takes the form of an apprenticeship. Indeed, you taught me the craft of research by example, demonstrating how to be a good teacher, a good researcher, and a good community leader. Apart from the vast technical expertise, you have also been a constant source of advice, support, and encouragement throughout my undergraduate career. I could not ask for a better mentor.

I thank Mike Bond from The Ohio State University, who co-supervises me. Meetings with you and Steve are always enjoyable for me. I am sorry for the meetings that went over time, often around the dinner time for you, lowering your glucose level. You help me turn complicated ideas into implementation with your experiences in hardware transactional memory. I am thankful for your patient guidance and inspiration.

Many people have helped me with this thesis. I thank Adrian Herrera, Kunal Sareen, and Brenda Wang for subjecting themselves to the draft of this document, and providing valuable feedback. For technical work, I thank Xi Yang for the help in using the underdocumented performance counters in Intel CPUs. I also thank Wenyu Zhao for the help in understanding the G1 collector built by him in MMTk. Finally, I thank The Australian National University for financially supporting me through ANU Honours Scholarship.

I am lucky to have worked with great people in the past four years. At ANU, Steve Blackburn and Tony Hosking took me on when I was only one month into my degree. Thank you for leading me into the fascinating world of computer systems and programming languages research. I am fortunate to have Steve Gould, Josh Milthorpe, Michael Norrish, and Ben Swift as supervisors for various projects. You all have taught me how to appreciate different disciplines within computer science. I also thank the PhB convenor, Ulrike Mathesius, and the student administrator, Janet Street, for helping me with the degree transfer, which made the fruitful exploration I had possible. At Microsoft Research, I thank Todd Mytkowicz for accepting me as a research intern, and three months of wonderful time in Redmond. I also thank my collaborators in the SCCL project for technical insights and for co-authoring the paper [Cai et al., 2020].

The honours year is often the most stressful period in a degree, and this year has been even more challenging due to the unprecedented events all over the world, including the COVID-19 pandemic. I thank my friends and fellow students who make my life brighter. There are too many of you to list here, but thanks especially go to Yuhui Chen, Zoey (Zhuo) Chen, Aditya Chilukuri, Adrian Herrera, Anh Phuong Le, Kunal Sareen, Yiyi Shao, Yichen Wang, Yutong Wang, Yueyue Xu, Bingyi Ye, and Wenyu

[1] 師父, see https://en.wikipedia.org/wiki/Shifu.

Zhao. Ben Titzer had a teaching stint at ANU for the long-awaited compilers course, and I am fortunate to be his TA. Ben, thank you for the "cultural education", especially the great movies from the 80s. It was a success[2], and I am now a Star Wars fan. Last but not least, a special shout-out goes to Brenda Wang. Brenda, the past four years is an important and transitional stage of my life. Your company has made tough moments more bearable and fun things even more enjoyable. Most importantly, you have helped me understand myself better. Thank you for always being there for me, and I hope I can also support you in your future endeavours.

Finally, I thank my family for their support, without which my overseas study would not be possible.

[2]Unfortunately, despite all the teaching, your TA still has not learnt how to eat burritos properly.

Abstract

Many applications, such as video-based or transaction-based ones, are latency-critical. Any additional latency may greatly degrade the user experience, inflicting significant financial loss on the vendor. Recently, an increasing number of these applications are written in managed languages, such as C♯, Java, JavaScript, and PHP, for productivity and reliability. Garbage collection (GC) provides automatic memory management to managed languages. However, GC can also induce pauses in the application, greatly affecting the user experience. This thesis explores the challenges of minimizing GC pauses.

Concurrent GC reduces pauses by working concurrently with the application (the mutator). Copying GC improves the mutator locality and reduces the heap fragmentation. Concurrent copying GC achieves both, but requires heavyweight synchronization to ensure that the concurrently executing mutator has a consistent view of the heap while the collector changes it. Existing implementations of concurrent copying GC use read barriers or page protections to prevent the mutator from using stale references. Unfortunately, these synchronization mechanisms introduce high overhead to the mutator.

My thesis is that, by using hardware transactional memory (HTM), mutators can execute transactionally during concurrent copying, achieving a consistent view of the heap, but with lower overhead than read barriers or page protection.

The contributions of this thesis are twofold. (1) I implement and evaluate a novel algorithm of using HTM to reduce the mutator overhead of concurrent copying GC. (2) I conduct a detailed analysis of HTM capacity, filling a significant gap in the literature, and informing the design of our HTM-based algorithm. I then use the insights on HTM capacity to implement several optimizations to improve the algorithm.

Using the Intel Transactional Synchronization Extension (TSX) as a case study, I measure the transaction capacity on this popular HTM implementation, and cross-validate the results with the literature and fill a gap in the literature, resolving ostensibly contradictory results. I have also explored different factors that may affect the effective capacity of transactions, which have not yet been reported in the literature (to the best of my knowledge). I implement the algorithm in MMTk, a framework for the design and implementation of GC. The implementation is evaluated on Intel TSX using several test programs. The results suggest that performing concurrent copying GC using HTM is viable.

This work deepens the understanding of HTM, its strengths and weaknesses, in the research community. Strategies using this work to fully exploit the capabilities of HTM can be generalized and applied to other applications of HTM. Finally, this work

enables the design and implementation of concurrent copying GC with lower mutator overhead with similar hardware support.

Contents

List of Figures

List of Tables

Part I

Prologue

Chapter 1

Introduction

Latency, defined by the time between the start and the completion of an event, is critical in many applications, including video-based ones and transaction-based ones. These applications are ubiquitous in modern life, and any additional latency of these applications can greatly affect the user experience. For example, search companies and e-commerce companies have measured the financial impact of latencies in their services [Linden, 2006]. However, decreasing the latency of these applications is no easy task, often requiring optimizations across the stack, from the design of programming languages to custom hardware chips.

Historically, many latency-critical applications are written in unmanaged programming languages, such as C and C++. These languages are notorious for causing numerous memory-related vulnerabilities, and the reliability of software suffers. Recently, an increasing number of these applications are instead written in managed languages, such as C♯, Java, JavaScript, and PHP. An important feature of managed languages is automatic memory management, also known as garbage collection (GC). GC frees programmers from manually allocating and reclaiming memory resources, increasing both the productivity of programmers and the reliability of software. However, GC often needs to pause application activities during a collection cycle, for example, to scan references on the application stacks. The pause time contributes to the latency of any application written in managed languages. Even worse, the pause time is often proportional to the heap size, which is unacceptable in modern applications, especially when running with several hundred GBs or even TBs of heap. Therefore, it is crucial that we reduce the pause time of GC to meet the latency requirements of latency-critical applications.

One important technique to reduce the pause time of GC is to run GC concurrently with application activities (mutator activities). This is known as concurrent GC. One class of concurrent GC algorithms is concurrent copying GC, where GC relocates objects to reclaim memory. Concurrent copying GC is widely used, for its ability to both reduce the pause time and reduce the heap fragmentation. The challenge of implementing concurrent copying GC is to make sure that the update of all pointers to moved objects is logically atomic, and stale references are not used by mutators.

Currently, read barriers and page protections are often used to provide the required atomicity for concurrent copying GC. These mechanisms, however, introduce large overhead to mutator activities. For example, read barriers mediate all mutator heap

reads, which happen very frequently.

1.1 Problem Statement

Concurrent copying GC reduces GC induced pauses in applications written in managed languages, which help latency-critical applications written in these languages meet the latency requirements. However, currently used synchronization mechanisms, read barriers and page protection, achieve heap consistency by introducing large overhead to the mutator activities. The challenge is how we can implement concurrent copying GC with lower mutator overhead.

1.2 Contributions

The contributions of this thesis have two parts.

In the first part of the thesis, I consolidate and contrast findings in the literature regarding HTM capacity. I then design and carry out experiments to explore factors that might affect HTM capacity, resolving apparent contradictions in the literature.

In the second part of the thesis, I implement and evaluate a novel concurrent copying GC algorithm that uses hardware transactional memory (HTM), co-developed with Bond and Blackburn. Then, armed with the insights of HTM capacity described in the first part, I devise optimizations for the algorithm, improving its practicality. Finally, I demonstrate the viability of the algorithm through a suite of test programs.

1.3 Thesis Structure

The goal of my honours project is to investigate a novel concurrent copying GC algorithm that uses HTM to reduce the mutator overhead compared with read barriers and page protection. While implementing the algorithm, I discovered that that practicality of the algorithm is predicated on the understanding of HTM capacity, which is now detailed in the first part of the thesis. The second part of the thesis is about the algorithm, its design, implementation and evaluation. In particular, I concretely demonstrate how the insights of HTM capacity can be used to guide the optimization of the algorithm.

Part II

Understanding HTM Capacity

Chapter 2

Background

Parallel programming is becoming more and more necessary. Historically, the clock rate, and hence the performance, of CPUs kept increasing while utilizing the same power budget by scaling down the feature size of transistors. This observation of scaling is known as Dennard scaling [Dennard et al., 1974]. Dennard scaling brought "free lunch" to software development, as the performance of software improved due to the advancement in the CPU performance rather than active optimization. Unfortunately, Dennard scaling came to an end in the early 2000s [Agarwal et al., 2000], and clock rates of CPU essentially stopped increasing. The total processing power (measured in total FLOPS) of CPUs is still increasing, however, in the form of massive parallelism. Common forms of this parallelism are CPUs with multiple cores, also known as multiprocessors. Multiprocessors are ubiquitous in the modern computing landscape, and they can be found everywhere from smart watches to supercomputers. Despite this ubiquity, automatically parallelizing serial code remains an active area of compilers research. For now, to fully utilize the available parallelism, explicit parallel programming is required.

However, parallel programming is difficult. It is up to the programmers to identify parts of the program amenable to parallelism. Except for tasks that are embarrassingly parallel, careful division of tasks and synchronization are often required to ensure the performance and correctness of parallel programs. Good parallel programming paradigms can aid programmers in expressing the parallelism.

While general-purpose parallel programming paradigms exist (*e.g.*, OpenMP and MPI), programmers still resort to low-level primitives (*e.g.*, compare-and-swap (CAS) and mutexes) to implement concurrent data structures. These low-level primitives are especially prevalent in the implementation of the runtimes of programming languages, where the use of concurrent data structures is frequent and performance critical. However, these low level primitives are notoriously error-prone, not productive to program, and hard to debug.

Transactional memory (TM) provides an alternative paradigm of parallel programming [Herlihy and Moss, 1993]. The idea stems from transactions found in database systems (see Section 2.1). First, Section 2.2 introduces the semantics of TM and its two incarnations, software transactional memory (STM) and hardware transactional memory (HTM). Then, Section 2.3 discusses Intel Transactional Synchronization Ex-

tension (TSX), the only available implementation of HTM on commodity hardware as of writing. Intel TSX is used as a case study in this work. Finally, Section 2.4 covers important principles of cache in modern CPUs, which help in understanding the behaviour of HTM.

2.1 Transactions

To understand TM, we first need to understand transactions. Transactions are common in the business context. For example, when processing a bank transaction, it is crucial that the changes to the account balances occur atomically.

Transactions are also no stranger to the computing world, especially in the case of database transactions [Bernstein, 1990; Gray and Reuter, 1992; Ramakrishnan and Gehrke, 2000]. The semantics of transactions can be captured using the notion of linearizability [Herlihy and Wing, 1990]. That is, each completed transaction appears to take effect instantaneously in some sequential order. We call a transaction succeeding *committing*. A transaction might also fail, and we say that the transaction *aborts*.

The four important properties of database transactions are known as ACID: atomicity, consistency, isolation, and durability. *Atomicity* focuses on the operations within a transaction; either all or none happen. For example, a transaction that transfers $100 from account A to account B must ensure that the balance of A is decremented by $100 if and only if the balance of B is incremented by $100. *Consistency*, however, depends on the data structures used and their invariant. For example, a transaction that adds a node to a linked list must make sure that if a node is successfully added, the length of the list is consistent with the total number of nodes. *Isolation* requires that transactions do not seem to affect each other, even though they might be executed in parallel. *Durability* requires that when a transaction commits, its effect is visible from the point of committing.

In Section 2.2, we will discuss transactional memory and to what extent ACID properties apply.

2.2 Transactional Memory

Lomet [1977] observed that transactions can be used as a synchronization mechanism for general purpose programming. However, no implementation detail was provided. Later, Herlihy and Moss coined the term *transactional memory* (TM) and proposed a hardware implementation in their seminal work [Herlihy and Moss, 1993]. Around the same time, Stone et al. [1993] proposed *Oklahoma Update*, which is similar to CAS, except that it can operate on multiple words.

The basic semantics of TM is very similar to database transactions (see Section 2.1), except that the durability is often ignored, as the data will not survive a power cycle due to DRAM being volatile.

It is worth noting that TM does not magically solve all parallel programming problems. It is still up to programmers to identify parts of the program amenable to paral-

lelism. If a critical section is broken into multiple transactions, TM cannot ensure correctness. Conversely, if a transaction includes extra work that can be executed safely without synchronization, unnecessary overhead will still incur.

2.2.1 Design Space

TM systems can differ in many ways, but two main mechanisms that any TM implementation must provide are work tracking and conflict detection [Harris et al., 2010].

Work tracking concerns speculative writes of an uncommitted transaction. In *eager versioning*, a TM system writes directly to the memory and records values it has overwritten. In case of a transactional abort, the TM system can undo or rollback the changes to the memory. In *lazy versioning*, a TM system instead holds speculative writes in a buffer, and only overwrites the memory when transactions commit. In either case, we call the maximum amount of work that can be tracked in a transaction the *transactional capacity*.

Conflict detection ensures isolation between transactions. TM can identify conflicts when transactions are running, known as *eager conflict detection*. TM can also identify conflicts only when transactions are attempting to commit, known as *lazy conflict detection*.

Apart from the two main mechanisms, TM systems can also differ in their programming models.

A TM system can be implicit or explicit in identifying transactional memory access. In *explicitly transactional* TM, all transactionally accessed memory locations have to be specified, such as in the case of the original TM proposal [Herlihy and Moss, 1993]. An *implicitly transactional* TM, however, only requires the programmer to mark the boundary of transactions, within which all memory accesses are treated as transactional.

A TM system can provide *strong isolation* or *weak isolation* for transactions. Strong isolation protects a transaction from both other transactions and non-transactional access. In contrast, weak isolation only guarantees transactional semantics among transactions.

2.2.2 Software Transactional Memory

Despite being proposed in 1993, TM did not materialize on hardware until the Rock processor from Sun in 2009. Meanwhile, the research community explored software transactional memory (STM), first proposed by Shavit and Touitou [1995].

To achieve work tracking and conflict detection, STMs rely on software instrumentation of memory operations (at object- or word-granularity). Unfortunately, this instrumentation introduces both space and time overheads. To achieve good performance, the instrumentation often needs to be integrated with the runtime system, such as JIT compilers or GC [Harris et al., 2010], making STMs a poor choice for languages like C and C++ or for implementing the runtime itself. In addition, STMs do not provide strong isolation (see Section 2.2.1), making it hard to compose with third-party code, often in the form of shared libraries.

STMs do have the benefit of not being subject to hardware limitations, such as cache sizes. Compared with HTM, where the transactional capacity is often small, STM capacity is theoretically only bounded by the size of the memory.

Diegues et al. [2014] investigated some state-of-the-art STMs, in chronological order: TL2 [Dice et al., 2006], TinySTM [Felber et al., 2008], SwissTM [Dragojević et al., 2009], and NOrec [Dalessandro et al., 2010]. They showed that these STMs can provide practical, and sometimes competitive, performance. However, the energy efficiency of STMs is generally poor. Their work also showed that STMs are better than HTMs if the workload leads to heavy contention on limited hardware resources.

2.2.3 Hardware Transactional Memory

In recent years, hardware transactional memory (HTM) has been more widely studied and implemented by various vendors. They include the Rock processor from Sun [Chaudhry et al., 2009], Azul [Click, 2010], and various IBM processors like the Blue Gene/Q [Wang et al., 2012]. The Advanced Synchronization Facility (ASF) from AMD [Christie et al., 2010] is still at the proposal stage as of writing, leaving the Intel Transactional Synchronization Extensions (TSX) [Intel Corporation, 2020d] the only commodity HTM implementation.

For work tracking, HTM implementations often reuse and enhance existing hardware structures, such as caches or store buffers[1]. This has the advantage of minimal space and time overhead. However, the transaction capacity is subsequently bounded by the cache size which is several orders of magnitude more than the memory size.

For conflict detection, HTM can again reuse existing hardware features, such as cache coherency protocols. Cache coherence protocols are used to ensure each core of a multiprocessor has a coherent view of the memory. For example, the protocols can ensure a core will not read a stale value of a memory location from its cache if other processors have published writes of the same memory location. This ensures that a conflict abort of an ongoing transaction is raised when other processors write to a line that is read from/written to during the transaction.

With the hardware support, HTMs can generally achieve much lower overhead than STMs, enabling interesting applications, such as high performance game emulation [Yahfz, 2020].

2.3 Intel Transactional Synchronization Extensions

Using the taxonomy introduced in Section 2.2.1, Intel Transactional Synchronization Extensions (TSX) is a TM system that uses lazy versioning, uses eager conflict detection, is implicitly transactional, and provides strong isolation.

TSX has two different programming models: *Hardware Lock Elision* (HLE) and *Restricted Transactional Memory* (RTM). I focus on RTM in my thesis.

[1] This implies that work can only be tracked at cache-line granularity, and therefore, false sharing of cache lines can lead to false transactional conflicts.

HLE is a backward-compatible extension providing two instruction prefixes: `XACQUIRE` and `XRELEASE`. These two prefixes can be used on memory operations that acquire or release locks protecting critical sections, and thus identifies the transactional boundary. On hardware that does not support TSX, or if the transactional execution of the critical section fails, the CPU will execute the critical section non-transactionally, where the lock operations are not elided.

RTM, however, provides three new instructions. `XBEGIN` and `XEND` are used to mark the transactional boundary, while `XABORT` explicitly aborts a transaction with a supplied status code. `XBEGIN` specifies the address of an abort handler. In the event of an abort, all changes to the memory and architectural registers are reversed, and the control flow is transferred to the abort handler. The abort handler can examine the value of the `eax` register and find out the cause of the abort.

Under both models, the `XTEST` instruction can show whether it is executed in an ongoing transaction.

It is worth noting that, like many other TM implementations, TSX is only best-effort and there is no progress guarantee. Aborts can happen for many reasons other than conflicts, including the following reasons [Intel Corporation, 2020d, Volume 1: Chapter 16.3.5, 16.3.8]:

- Explicitly aborted via `XABORT`.
- Internal buffer overflowed.
- Synchronous exception events, such as a debug breakpoint.
- Asynchronous events, such as a timer interrupt.
- Unfriendly instructions, such as the ones causing privilege level transition (*e.g.* `SYSCALL`).
- Page faults.

Therefore, it requires careful construction of transactional code paths to avoid frequent aborts.

It is worth mentioning that since the inception of Intel TSX, it has suffered from many bugs across multiple generations of CPUs. Various microcode updates were issued, that either reduces the functionalities of TSX or even entirely renders TSX unusable. In 2014, TSX was disabled on Haswell and early Broadwell processors for causing "unpredictable system behavior" "[u]nder a complex set of internal timing conditions and system events" [Intel Corporation, 2020b, HSD136]. In 2019, a microarchitectural data sampling (MDS) vulnerability known as ZombieLoad 2 [Schwarz et al., 2019] targeted TSX. During an asynchronous abort of a transaction, such as a buffer overflow or conflict, microarchitectural side effects are not reversed, and the results of speculative read can be inferred later. Again in 2019, an erratum [Intel Corporation, 2020a, SKL179] was issued for Skylake and later processors. When RTM is enabled, PMU general purposes counter 3 "may contain unexpected values". A microcode update provides an option known as TSX Force Abort. A flag is provided to allow switching

between disabling counter 3 or disabling TSX entirely. It has been reported [RPCS3, 2020] that the microcode update results in increased transactional abort rates.

Readers should keep in mind that the changes to Intel TSX make the reproduction of results published in prior work much more difficult. Moreover, when conducting experiments, there can be unexpected results because of the changes.

2.4 Cache

As discussed in Section 2.2.3, conflict detection in HTM is often piggybacked on the memory hierarchy. Therefore, it is important to understand some basic principles of how a cache works.

Caches operate on blocks of memory, and the smallest unit of operations is called a *cache line*. A cache line can be either entirely in or not in a cache. Cache lines are placed and moved between different levels of cache. The placement and the movement are controlled by placement policies and replacement policies, are discussed below.

These policies can affect HTM capacity.

2.4.1 Placement

Cache placement policies refer to where to place cache lines in a cache. Below are multiple strategies, listed in the descending order of the level of associativity.

Fully associative: A cache line can be placed anywhere in the cache.

Set associative: A set associative cache is divided into multiple sets. A cache line is first mapped onto a set (usually using the modulo operation), and then the line can be placed anywhere in the cache.

Direct mapped: A cache line can only be placed at a fixed location in the cache.

For caches that are not fully associative, memory access patterns can affect the effective cache size. For example, if we access memory addresses in a certain stride, so that all the addresses map onto the same cache set, only a fraction of the total cache size is used in this case.

2.4.2 Replacement

When a cache line can be found in a cache, we call it a *cache hit*. Conversely, when a cache line cannot be found in a cache, we call it a *cache miss*. Since the memory hierarchy is built to reduce the latency of memory operations, we want to minimize the number of cache misses. When a cache miss occurs, the cache line in question will be placed into the cache. Cache replacement policies refer to when the above happens, which existing cache line is to be replaced.

It is trivial in the case of a direct mapped cache, as each cache line maps to a fixed location in the cache, so there is no choice to be made. However, in the case of fully associative or set associative cache, replacement is optimized so that future cache misses are

minimized. The cache replacement policies on commodity CPUs have been guarded as trade secrets, although there have been efforts in reverse engineering the policies [Abel and Reineke, 2014, 2020].

Two common cache replacement policies are as follows. For a cache that uses *first in, first out* (FIFO), the oldest cache line is replaced. For a cache that uses *least recently used* (LRU), the least recently used cache line is replaced. Note that it is difficult to keep track exactly when cache lines are accessed, especially given the limited silicon estate. Often, pseudo-LRU is used, such as the quad-age LRU (QLRU) algorithm used in Intel Ivy Bridge processors [Jahagirdar et al., 2012]. Pseudo-LRU only offers an approximation, and it is possible that a cache line, which is not the least recently used one, gets replaced.

2.4.3 Cache and HTM Capacity

As discussed in Section 2.2.3, conflict detection of HTM is often built on top of the memory hierarchy, and the transactional abort can occur when a cache is evicted.

Since memory access patterns can affect the effective cache size (see Section 2.4.1), it is reasonable to deduce that memory access patterns can also affect the effective HTM capacity for transactions. In fact, this relationship has been shown by Hasenplaugh et al. [2015].

We will discuss how the cache status and cache replacement policies can affect the effective transaction capacity in Chapter 5.

Chapter 3

Related Work

In this chapter, I will survey related work on the HTM capacity on Intel TSX. By contrasting the results, I will point out the apparent contradiction, with different work citing different sizes for maximum HTM capacity. I will also compare the methodologies used in prior work for measuring HTM capacity which I will attempt to resolve in Chapter 5.

3.1 Contradictory Reported HTM Capacity

In this section, I will list the HTM capacity of Intel TSX reported in the literature I have surveyed in chronological order. All CPUs used in the literature have 32KB L1 data cache and 256KB L2 cache per core.

- On Core i7-4770 (Haswell), Ritson and Barnes [2013] found that only transactions of sizes smaller than 16KB can consistently commit, and reported that no transactions with sizes larger than 26KB can commit. They speculate that the transactional work tracking is done in the L1 data cache.
- On a Haswell CPU, Yoo et al. [2013] from Intel stated that Haswell uses L1 data cache to track transactional states. However, the eviction of cache lines in the read-set does not cause an abort. Instead, they are moved to a secondary structure for tracking. In contrast, the eviction of cache lines in the write-set always causes an abort.
- On Xeon E3-1275 v3 (Haswell), Diegues et al. [2014] reported that the performance of transactional memory benchmarks is strongly dependent on the access patterns to L1 cache.
- On Core i7-4770 (Haswell), Goel et al. [2014] reported that the abort rate saturates at 128K cache lines (the size of L3 cache) for read-only transactions, and 512 cache lines (the size of L1 cache) for write-only transactions.
- On a Xeon E3-1200 v3 family CPU (Haswell), Pereira et al. [2014] found that the speedup of transactional data processing dropped dramatically at 256KB, which is the size of the L2 cache.

- On Core i7-4770 (Haswell), Wang et al. [2014] found that the biggest read-only transaction is 4MB (half the size of L3 cache) and the biggest write-only transaction is 31KB.
- On Core i7-4770 (Haswell), Dice et al. [2015] stated that the CPU tracks the read-set in all levels of caches, but the write-set is tracked in L1 cache. The size of the largest read-only transaction seen committing is 7.5MB, which is slightly smaller than 8 MB—the size of the L3 cache.
- On Core i7-4770 (Haswell), Hasenplaugh et al. [2015] found that the maximum size for read-only transactions is 75K cache lines (slightly more than half of the L3 cache size), and the maximum size of write-only transactions is 400 cache lines (slightly smaller than the L1 cache size).
- On Core i7-4770 (Haswell), Nakaike et al. [2015], found that the maximum sizes for read-only transactions and write-only transactions are 4MB and 22KB respectively.
- On Core i7-6600U (Skylake), Gruss et al. [2017] found that no read-only transaction exceeded the capacity of the L3 cache. Similarly, they found that the write-set size is constrained by the size of L1 data cache. They speculate that the read-set tracking is done using a probabilistic structure, such as a Bloom filter.

I observe that there does not seem to be a definitive answer of the capacity of Intel TSX. The reported write-set capacities range from 22KB [Nakaike et al., 2015] to 31KB [Wang et al., 2014]. The only thing that all work agrees on is that the write-set capacity is bounded by the L1 data cache size. The reported read-set capacities range from 22KB [Nakaike et al., 2015] to 7.5MB[Dice et al., 2015].

3.2 Mixed Methodologies

In Section 3.1, I enumerated the read-set capacities and the write-set capacities of Intel TSX reported in the literature. The write-set capacities range mildly but there is a huge discrepancy in the reported read-set capacities, ranging from 22KB to 7.5MB. The difference here might be due to how the capacities are measured. However, the methodologies used in the literature are mixed and sometimes unclear.

Some prior work [Dice et al., 2015] focuses on the HTM application, and therefore, it is understandable that the details of the experimental design were omitted. However, a performance-focused work Goel et al. [2014] does not include any detail either.

Some prior work evaluates the performance of TM systems using benchmark suites. For example, Diegues et al. [2014] used the STAMP suite [Chí et al., 2008] and Pereira et al. [2014] used Eigenbench [Hong et al., 2010]. Unfortunately, these benchmarks mix reads and writes in the same transaction. Although good to represent more realistic workload, these benchmarks are unhelpful in determining the read-set capacity and the write-set capacity separately. In addition, Eigenbench assumes an explicit TM

system, where all transactional memory accesses are explicitly identified, referred to as "transaction length". This characterisation of the workload is inapplicable in the context of an implicit TM system, such as Intel TSX.

A majority of the prior work reuse the same memory region for testing transactions of different sizes [Wang et al., 2014; Hasenplaugh et al., 2015; Nakaike et al., 2015; Gruss et al., 2017]. However, Ritson and Barnes [2013] explicitly chose distinct memory areas (aligned to a cache line boundary) for different transactions, aiming to minimize the effects of L2 and L3 processor caches.

Chapter 4

Methodology

Empirical evaluation is fundamental to this part of the thesis, and the following sections document the software and hardware setup used in the evaluation. I will also discuss the metric used by prior work to describe HTM capacity, and why the metric is problematic. Finally, I will discuss the methodology used in my experiments.

4.1 Software Platform

All machines used in the evaluation are running the identical image. The Linux distribution used is Ubuntu 18.04.5 with `5.4.0-47-generic` kernel. GCC version 7.5.0 is used to compile C programs with flag `-O2`. Apart from the programs being benchmarked, the system is otherwise idle, with as many background daemon turned off as possible.

4.2 Hardware Platform

The machines used in the evaluation are listed in Table 4.1. I choose a diverse set of microarchitectures in the evaluation, since the implementation of TSX is likely to improve over time. All machines used have simultaneous multithreading (SMT) turned on but with frequency scaling turned off in my evaluations. Since all processors used are Intel x86 processors, the above implies that Intel® Hyper-Threading Technology is turned on but Intel® Turbo Boost Technology is turned off.

4.3 Success Rate Curves

In Section 3.2, I contrasted the methodologies used in prior work. A common way to describe the HTM capacity in the literature is by using success rate curves [Ritson and Barnes, 2013; Goel et al., 2014; Wang et al., 2014; Hasenplaugh et al., 2015; Gruss et al., 2017]. The success rate is the number of transactions committed divided by the number of transactions attempted. Note that all the attempted transactions have the same characteristics other than the size, including the memory access pattern. The curve shows how the success rate varies with the sizes of the transactions.

Table 4.1: Machines used in the evaluation.

Architecture	Haswell	Broadwell	Skylake	Coffee Lake
Model	Core i7-4770	Xeon D-1540	Core i7-6700K	Core i9-9900K
Technology	22nm	14nm	14nm	14nm
Clock	3.4GHz	2.0GHz	4.0GHz	3.6GHz
Cores × SMT	4 × 2	8 × 2	4 × 2	8 × 2
L1 Data Cache	32KB × 4	32KB × 8	32KB × 4	32KB × 8
L2 Cache	256KB × 4	256KB × 8	256KB × 4	256KB × 8
L3 Cache	8MB	12MB	8MB	16MB
Memory Size	16GB	16GB	16GB	32GB
Memory Type	DDR3-1600	DDR4-2133	DDR3-1600	DDR4-2133

I contend that the success rate curve is a problematic way to describe a TM system for the following two reasons.

Firstly, the success rate is misleading and cannot be meaningfully compared without extra information. When people see "success rate", they think that if they attempt a transaction of certain parameters, the success rate describes the likelihood/probability that the transaction will commit. However, from statistics knowledge, the law of large numbers tells us that we can only estimate the probability of events by repeated trials if the events satisfy certain conditions. For example, to estimate the probability that rolling a die produces six, we can roll the die for a large number of times and the ratio of results in which six appears should be close to the probability. The assumption that each outcome of rolling a dice is independent and follows the same distribution (i.i.d) is critical here. However, the outcome of a transaction depends on the previous transaction due to the nature of the hardware-based TSX implementation. Therefore, multiple attempts of the transaction of the same parameters are not i.i.d.

Secondly, in real workload, we will stop retrying a transaction once it successfully commits. Therefore, a more meaningful metric will be how many retries it takes for a transaction with certain parameters to commit.

4.4 Measuring HTM Capacities

The methodology I will adopt in this thesis is largely based on the `rtm-bench` benchmark due to Ritson and Barnes [2013]. The benchmark has the following advantages.

(1) The benchmark is publicly available on GitHub[1].

(2) The benchmark sets the affinity of each of OS threads it spawns, preventing an OS thread from being migrated to a different hardware thread.

[1] https://github.com/perlfu/rtm-bench

(3) The results on Core i7-4770 were published [Ritson and Barnes, 2013], which can act as a sanity check on my setup.

(4) By default, distinct memory areas (aligned to a cache line boundary) are used for different transactions to minimize the effect of caches. This enables me to perform experiments on how the cache status affects the HTM capacity.

Although the benchmark reports the experimental results using success rate curves, which I argue is not ideal (Section 4.3), devising a better metric is beyond the scope of this work. However, one needs to be mindful when interpreting the results.

Chapter 5

Experiments and Results

In the previous section, I discussed the relevant work and pointed out the apparent contradiction regarding the capacities of read-only and write-only transactions. In this section, I will present experimental designs and results that attempt to resolve the contradiction. Note that I will use Intel TSX (see Section 2.3) as a case study in this section. The techniques and methodologies discussed, however, should be generalizable to other similar HTM systems.

5.1 Baseline

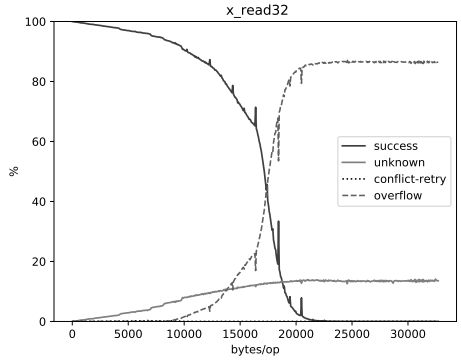
First, I attempt to reproduce the results of `rtm-bench` on Haswell. As shown in Fig. 5.1, the success rates of both read transactions and write transactions drop significantly around 16KB. The largest read transaction is about 22KB, and the largest write transaction is about 25KB. The results are very close to what was reported by Ritson and Barnes [2013].

In addition, I also perform the same experiment on newer platforms, including Broadwell, Skylake, and Coffee Lake (in chronological order), since the newer microarchitectures are expected to have an improved second-level structure that tracks evicted read-set addresses [Intel Corporation, 2020c, Chapter 16.2.4.2]. The results for the above three microarchitectures are shown in Fig. 5.2, Fig. 5.3, and Fig. 5.4 respectively.

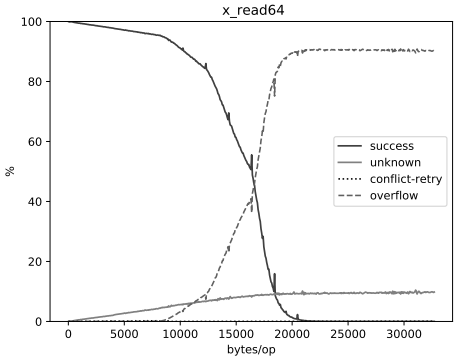
As shown in the plots, newer platforms do have larger read-only capacity. However, the write-only capacity is consistently around 20-25, which does not differ much from the prior work. In addition, the size of largest successfully read-only transactions is nowhere close to the size of the L3 cache, which is shown to be possible on prior work.

5.2 Reusing Memory Areas

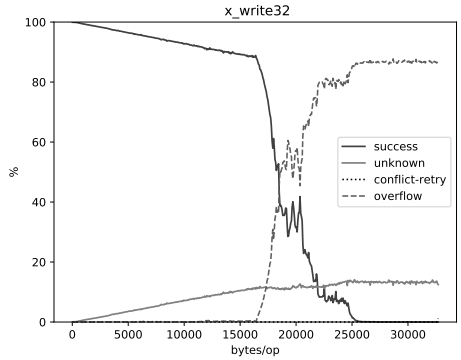
As discussed in Section 3.2, a majority of the prior work reuse the same memory region for testing transactions of different sizes. In this section, I modify the baseline so that transactions of the same size use the same memory area. The results are shown in



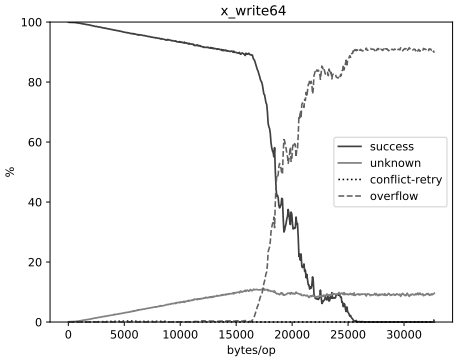
(a) Transactional 32-bit read



(b) Transactional 64-bit read

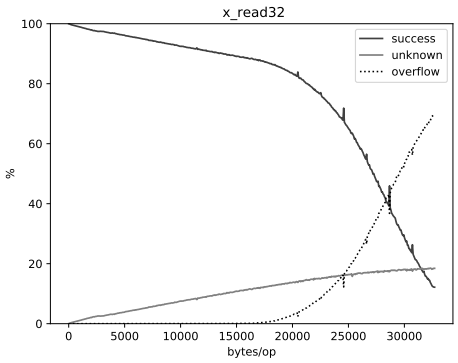


(c) Transactional 32-bit read

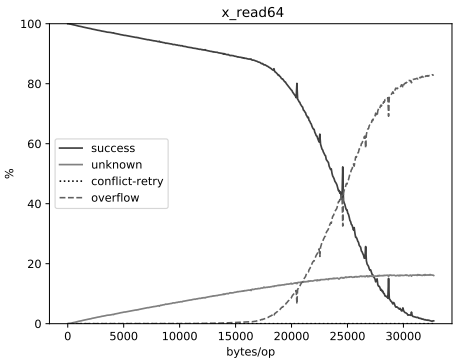


(d) Transactional 64-bit read

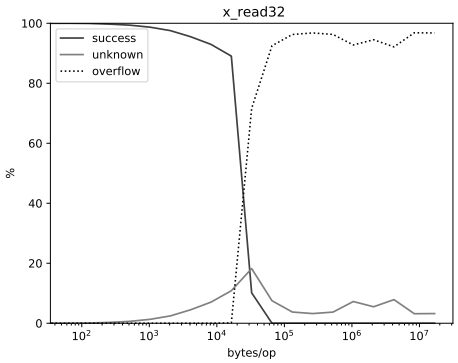
Figure 5.1: Success rate curves on Haswell.



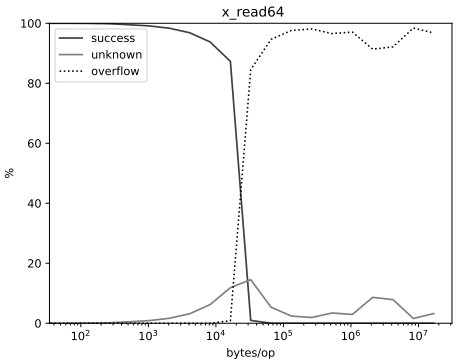
(a) Transactional 32-bit read



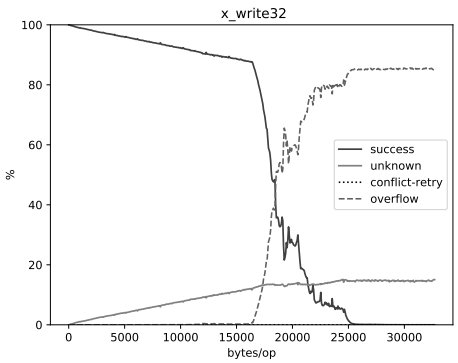
(b) Transactional 64-bit read



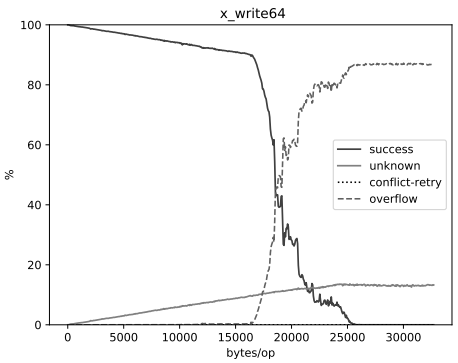
(c) Transactional 32-bit read in log scale



(d) Transactional 64-bit read in log scale

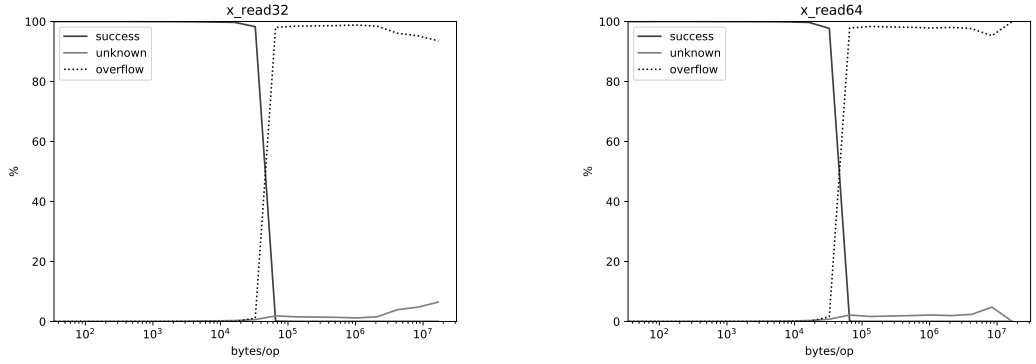


(e) Transactional 32-bit write



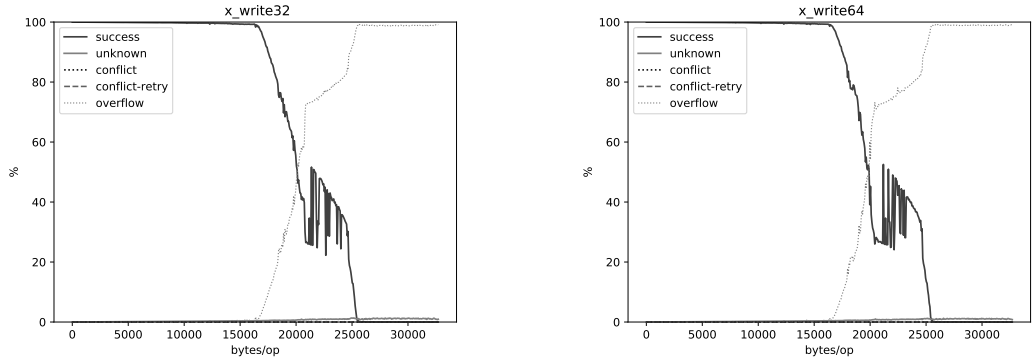
(f) Transactional 64-bit write

Figure 5.2: Success rate curves on Broadwell.



(a) Transactional 32-bit read in log scale

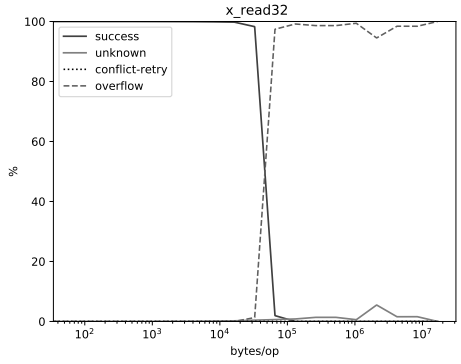
(b) Transactional 64-bit read in log scale



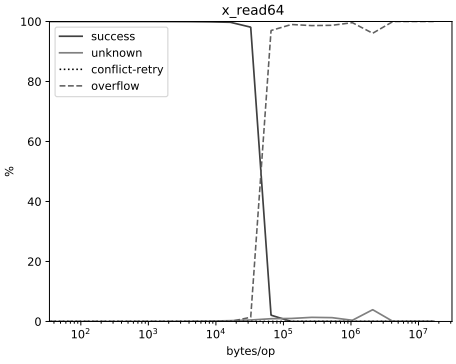
(c) Transactional 32-bit write

(d) Transactional 64-bit write

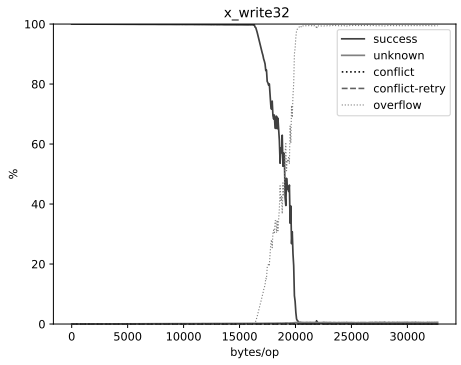
Figure 5.3: Success rate curves on Skylake.



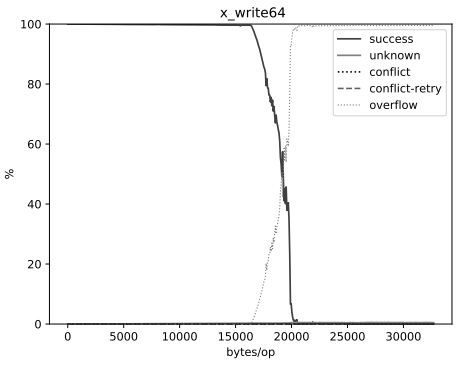
(a) Transactional 32-bit read in log scale



(b) Transactional 64-bit read in log scale



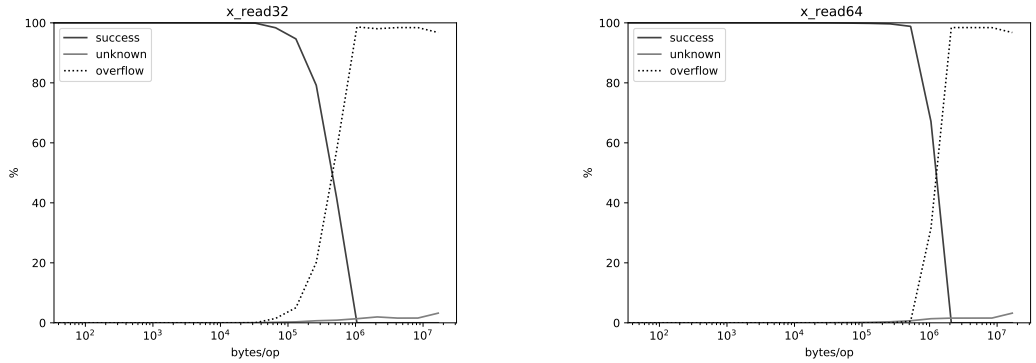
(c) Transactional 32-bit write



(d) Transactional 64-bit write

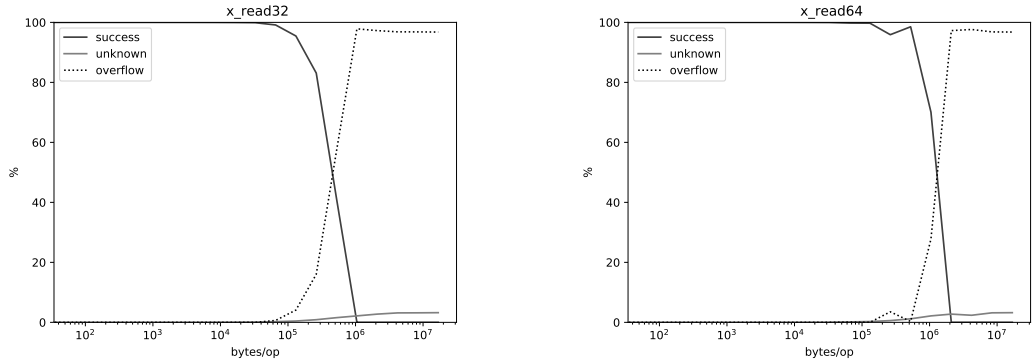
Figure 5.4: Success rate curves on Coffee Lake.

Fig. 5.5, Fig. 5.6, Fig. 5.7, and Fig. 5.8 for Haswell, Broadwell, Skylake and Coffee Lake respectively.



(a) Transactional 32-bit read (b) Transactional 64-bit read

Figure 5.5: Success rate curves on Haswell when reusing memory.

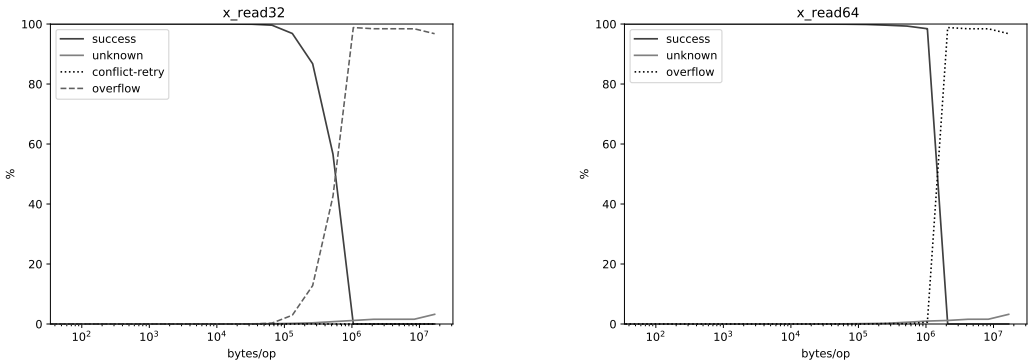


(a) Transactional 32-bit read (b) Transactional 64-bit read

Figure 5.6: Success rate curves on Broadwell when reusing memory.

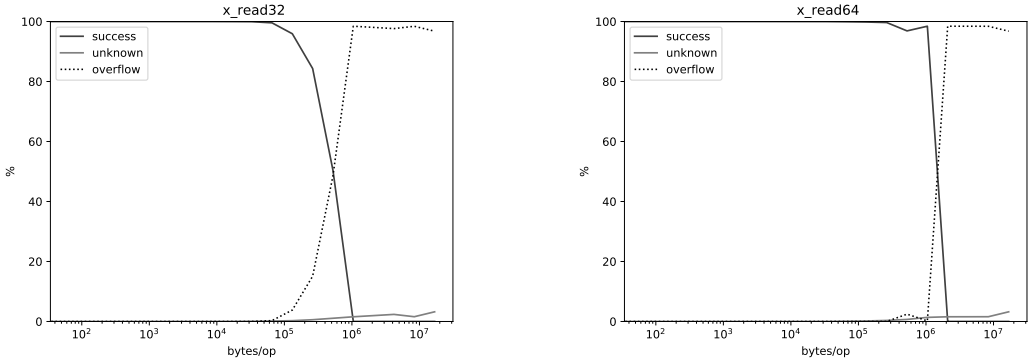
As shown in the plots, for each platform, reusing the memory area consistently increases the capacity of read-only transactions. Another interesting trend is that when transactions are performed using 64-bit reads instead of 32-bit reads, the capacity of read-only transactions is consistently larger across platforms.

I suspect that when a transaction is attempted, it can affect the cache content regardless whether the transaction commits or not. Thus, when a series of transactions are working on the same memory area, they are more likely to commit because of the interaction between HTM and the cache hierarchy. This could explain the contradiction in prior work.



(a) Transactional 32-bit read (b) Transactional 64-bit read

Figure 5.7: Success rate curves on Skylake when reusing memory.



(a) Transactional 32-bit read (b) Transactional 64-bit read

Figure 5.8: Success rate curves on Coffee Lake when reusing memory.

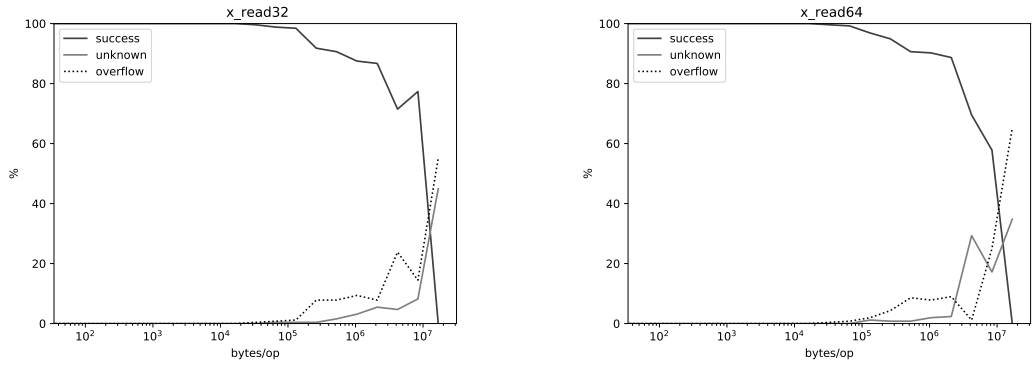
5.3 Invalidating and Warming Up Cache

As discussed in Section 2.4.3, cache placement policies can affect the effective HTM capacity. Based on the results in Section 5.2, I suspect that the content of the cache can also affect the effective HTM capacity. Therefore, I perform the following experiments. I only performed the experiments on Coffee Lake due to time constraint.

I investigate the following scenarios.

1. No-op: This amounts to the baseline.
2. Invalidation: Before each transaction, the `wbinvd` instruction[1] is used to invalidate all levels of cache.
3. Warmup: Before each transaction, the memory area that will be read in the transaction is read outside the transaction five times. On an otherwise idle system, the repeated read should warm up the cache for the upcoming transaction.

The results for the first scenario can be found in Section 5.1. The results for the latter two scenarios are shown in Fig. 5.9 and Fig. 5.10 respectively.



(a) Transactional 32-bit read

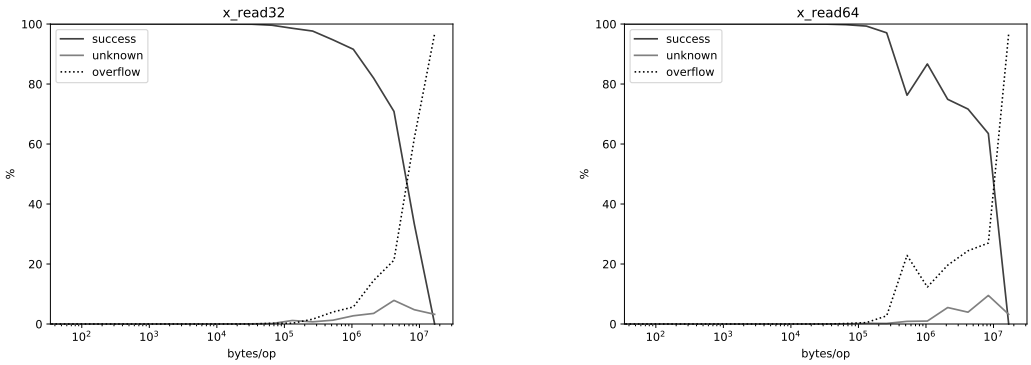
(b) Transactional 64-bit read

Figure 5.9: Success rate curves on Coffee Lake when invalidating caches.

As we can see from the plots, the largest transaction obtained in both scenarios on Coffee Lake is larger than the baseline on the same CPU (see Fig. 5.4). In addition, it is larger than the largest transaction obtained through reusing memory areas (see Fig. 5.8). In fact, it is a lot closer to the largest read-only transactions reported in prior work Section 3.1.

Normally, when caches are invalidated, the performance of programs will suffer, since all memory operations will result in cache misses. However, when caches are warmed up, the performance of programs will benefit from it, since more memory operations will result in cache hits. Therefore, the results here are interesting that both invalidating and warming up caches help large transactions commit.

[1]The `wbinvd` is privileged, and therefore this Linux kernel module is used. https://github.com/batmac/wbinvd



(a) Transactional 32-bit read

(b) Transactional 64-bit read

Figure 5.10: Success rate curves on Coffee Lake when warming up caches.

My hypothesis is that the results can be explained by the pseudo-LRU cache replacement policy (see Section 2.4.2). Under perfect LRU, cache lines accessed in a transaction, that is smaller than the cache size, should not be evicted by definition. However, under pseudo-LRU, when a cache set is full, a cache line read in the transaction might still get evicted despite having been recently used. When we invalidate the cache, all lines in a cache are invalid, and therefore, can accommodate cache lines of the transaction without invoking the displacement algorithm. Similarly, when the cache is warmed up, most lines used in the transaction are already in the cache, and the displacement algorithm will not be invoked if a memory operation hits the cache.

5.4 Summary

In this chapter, I explored how different factors can affect the capacity of transactions. In particular, reusing the memory area, invalidating the cache, and warming up the cache can all improve the read capacity of transactions across all platforms tested.

Part III

HTM GC

Chapter 6

Background

In the parlance of the garbage collection literature, the application is referred to as the *mutator* and the GC is referred to as the *collector*. Mutators are responsible for executing the application code, which mutates the heap in the process. Collectors provide automatic memory management, also known as *garbage collection* (GC), to facilitate the execution of mutators.

In this chapter, we survey the design space of GC algorithms, and look at how different design decisions lead to different performance characteristics of these algorithms. First, Section 6.1 introduces three fundamental elements of garbage collection algorithms: allocation, identification, and reclamation. Section 6.2 introduces tracing, which is the most widely used identification strategy. Next, Section 6.3 and Section 6.4 cover the two fundamental strategies for reclamation, non-copying GC and copying GC respectively. Copying GC leads to better locality and lower heap fragmentation. Then, Section 6.6 discusses concurrent GC, in which identification, reclamation, or both are executed concurrently with the mutators. Concurrent copying GC achieves the advantages of copying GC and concurrent GC, but requires heavyweight synchronization to ensure that the concurrently executing mutator has a consistent view of the heap while the collector changes it. Pursuing a concurrent copying GC algorithm with lower mutator overhead is at the heart of this work.

Also in this chapter, Section 6.5 and Section 6.7 introduce barriers and yieldpoints respectively. Both mechanisms play important roles in this work.

6.1 Organization of Garbage Collection Algorithms

Dynamic memory allocations in managed languages happen in a region of memory called the *heap*, which is managed by the GC. Different GC algorithms may divide the heap differently and impose different policies on different parts of the heap. These algorithms may also differ in how objects are placed, or in what actions to take upon the allocation of objects, etc. For many decades, many published GC algorithms conflate the above when describing how these algorithms work. Blackburn and McKinley [2008] introduced a taxonomy of GC algorithms, where an algorithm is defined by its heap organization and its heap operations. The heap organization prescribes how objects are mapped into different parts of the heap. The heap operations, however, can be

further divided into three parts, that were also identified by Jones et al. [2011]: (1) *allocation*: allocate new objects, (2) *identification*: identify garbage, naming the object to reclaim, and (3) *reclamation*: make memory occupied by garbage available for future allocation. This is worth noting that these three parts are not always independent. In particular, how space is reclaimed often affects how space is allocated.

To show how the taxonomy works, we can apply it on the classic *semispace* GC algorithm [Fenichel and Yochelson, 1969; Cheney, 1970]. The *heap organization* of semispace is that the heap is divided into two equally sized spaces, with only one space designated for allocation at any given time, called the *tospace*. The other space is called the *fromspace*. When a GC is triggered, the roles of the tospace and the fromspace are swapped, and the then-fromspace now-tospace is used for all allocation, including the relocation of objects. The three parts of the heap operations of semispace are as follows. Firstly,*allocation* into a semispace is via *bump-pointer allocation*. That is, a cursor is maintained for the tospace, and when allocating an object at the given size, the cursor is incremented by that size plus padding if any address alignment is required. Secondly, the *identification* of semispace is tracing (see Section 6.2). Finally, the *reclamation* of semispace is by copying (see Section 6.4), where all reachable objects in the then-tospace now-fromspace are copied into the now-tospace.

6.2 Tracing GC

An important part of GC is the identification of objects to reclaim. One identification strategy is by using reference counting, where each object has a counter for the number of references to that object. If an object has a reference count of zero, then it can be reclaimed.

Another very widely used strategy is called tracing. Tracing is done by performing a transitive closure of the object graph. During the transitive closure, the reachability of objects are recorded, and garbage is indirectly identified as all the unreachable objects. Transitive closures start from a root set or simply *roots*. Roots are references that mutators can use without going through another object, which includes references on the stack or in registers. The idea is that for an object to be reachable by mutators, and therefore, should not be reclaimed, the object must be transitively reachable from roots.

6.3 Non-copying GC

In this section, we will discuss a class of GC algorithms that does not perform copying for reclamation. For these algorithms, the reclamation of unreachable objects is achieved by returning the block of memory occupied by such objects to a pool for future allocation, and thus increasing the total free space. The reachable objects, however, are left in place. As discussed in Section 6.1, the reclamation strategy often affects the allocation strategy. In the case of non-copying GC, free-list allocators are often used. These allocators fulfill an allocation request by choosing an available block of

memory no smaller than the requested size from a pool of memory, often arranged as lists of different sizes. If no suitable block can be found, the allocator requests more memory to replenish the pool. Two of the problems that arise from using non-copying GC are as follows.

Fragmentation It can be the case the heap still has free space through the blocks in the pool for allocation. However, there is no block of memory big enough available for an allocation, either through reusing one the blocks or requesting more memory. This phenomenon is known as *fragmentation* and it is common in long-running applications using a non-copying GC. As a result of fragmentation, GCs might be frequently triggered, despite the fact that the heap still has free space. Sometimes, more catastrophically, the mutator can be terminated with an out-of-memory error, when a large allocation cannot be satisfied even after GC. Fragmentation decreases the effective heap size when GC algorithms choose such reclamation strategy [Robson, 1971, 1974]. It's worth noting that in the case of severe fragmentation, GC can trigger an expensive in-place compaction of the heap [Printezis, 2001; Soman et al., 2004], where all reachable objects are moved towards one end of the heap[1].

Poor Locality Non-copying GC can also suffer from poor locality. It stems from the fact that reusing the memory blocks from the pool is the main form of allocation. However, the pool is not necessarily organized according to vicinity. Often, it is organized by the temporal order of when the memory blocks are returned to the pool. As a consequence, consecutive allocations requests might be satisfied with memory locations that are not adjacent. This can lead to bad cache locality [Blackburn et al., 2004a].

6.4 Copying GC

In Section 6.3, we discussed non-copying GCs and the problems they have. In this section, we will discuss a class of GC algorithms that relocate reachable objects during reclamation. Again, as discussed in Section 6.1, the reclamation strategy often affects the allocation strategy, and copying GC is often accompanied by bump-pointer allocation.

These algorithms work as follows. All reachable objects in one section of the heap, where GC is operating on, are copied into some other parts of the heap. All unreachable objects in that section are not copied, so that the section is freed en masse, and thus regains more usable memory. It is important to note that when we relocate an object, all references to that object need to be fixed to point to the new version of the object. Not only do we need to fix all references within the heap, we also need to fix from the roots (see Section 6.2), since they are also used by mutators.

Contrasting with their non-copying counterparts, copying GC has the following benefits.

[1] The relative positions of objects in the heap may or may not be preserved depending on the algorithm, and mutator's locality can be affected.

Defragmentation When relocating objects, the new versions of reachable objects are placed consecutively towards one end of the heap. This defragments the heap by reducing the number of small chunks of free memory that cannot be used to fulfill large allocation requests.

Good Locality Due to the nature of bump-pointer allocation, consecutive allocation made by mutators will be adjacent. Moreover, the relocation of objects not only reduces the fragmentation, but also places objects closer together, increasing the spatial locality.

6.5 Barriers

Barriers are code fragments that mediate heap access of mutators to facilitate the GC. They are used to make many GC algorithms efficient, including generational GC and concurrent GC. Different barriers can be triggered under different conditions. Read barriers are executed during a read of the heap, and write barriers are executed during a write of the heap. Reference barriers are executed for read/write of a reference field, and primitive barriers are executed for read/write of a primitive field.

An example is the *boundary barrier* used in a generational GC. A generational GC with two generations divides the heap into two contiguous partitions, the nursery space and the mature space. As discussed in Section 6.2, a transitive closure of the object graph is often needed for identification. However, in the case of a generational GC, we would like to be able to only perform GC on the nursery space, and avoid the full heap trace. The boundary barrier is a write barrier that checks all creations of references, and if a reference from the mature space into the nursery space is created (crossing the "boundary"), the location of the reference is recorded. During a nursery GC, we can simply use the references recorded by the barrier as a basis of tracing on the nursery space, knowing that we will not miss any reference and incorrectly reclaim objects. The boundary barrier is cheap because checking whether an object is a nursery object only requires two comparisons, since the nursery space is contiguous.

The costs of different barriers have been extensively studied in [Blackburn and Hosking, 2004; Yang et al., 2012]. The literature shows that barriers can introduce significant overhead to mutators due to their prevalence. In particular, frequently executed barriers in performance critical code paths, like read barriers, can have about 10% mutator overhead. In contrast, common write barriers have only about 1% mutator overhead.

In addition, to avoid the cost of making an out-of-line call, barriers are often inlined with the mutator code during compilation. As a consequence, barriers cannot be easily removed, unless all methods are compiled again.

In this work, barriers are used to record references to objects being moved, making it easier for the collector to continuously monitor whether any reference update has been missed.

6.6 Concurrent GC

Stop-the-world GC suffers from high latency as mutators are paused until one GC cycle has finished. The latency problem can be alleviated by interleaving the execution of the collectors and the mutators. The concurrency can be applied in different phases of GC activities, giving rise to *concurrent tracing* GC and *concurrent copying* GC.

6.6.1 Concurrent Tracing GC

Concurrent tracing GC performs the identification phase of the GC concurrently with the execution of mutators. However, performing the transitive closure of the object graph while mutators can potentially mutate the graph is racy.

A classic race condition is as follows. Let A be an object that is being inspected by GC. Meanwhile, the mutator sets a field of A to point to object B that has not been previously traced but only reachable from A. The collector finishes tracing through all references from A and moves to another object. Later, that field of A can be set to be B again, but the collector has finished A. As a result, object B will be missed by the collector, and thus be deemed unreachable. If the collector then reclaims the space of B, it will render the reference field of A invalid, which might result in a crash of the application.

The above example illustrates the importance of synchronization between collectors and mutators for concurrent tracing. The techniques for concurrent tracing are well understood [Steele, 1975; Dijkstra et al., 1978; Yuasa, 1990]. These techniques generally use write barriers (see Section 6.5), so that different invariants can be maintained during the execution of the mutators. For example, Dijkstra et al.'s barrier maintains the invariant that if an object has been marked as scanned and will not be visited again, it should not hold any reference to unmarked objects [Dijkstra et al., 1978]. The barrier works by checking the source object whenever a reference is created. If the creation of the reference will break the invariant, the referent will be marked by the barrier. Let's revisit the previous example. Upon the creation of the reference from A to B, Dijkstra et al.'s barrier will mark B, and therefore, B will not be reclaimed.

6.6.2 Concurrent Copying GC

In Section 6.6.1, we talked about concurrent tracing GC, where the identification phase runs concurrently with the mutators. In this section, we will discuss how the reclamation phase of copying GCs can execute concurrently with the mutators. To clarify the terminologies, we use *fromspace objects* to refer to objects identified for relocation, and *tospace objects* to refer to the corresponding copies of fromspace objects after relocation.

As discussed in Section 6.4, copying GC needs to ensure that all references to copied objects are updated to point to the new versions. In stop-the-world copying GC, fixing heap references and roots is relatively easy. Performing these updates concurrently, however, is considerably harder than concurrent tracing, where only liveness is of concern. At the core of concurrent copying algorithms, we need to make sure that copying and pointer updates are done in a logically atomic fashion with respect to the mutators.

If the copying of objects is not done atomically, two important problems will arise. Firstly, mutators might store values to fromspace objects, resulting in updates being lost. Secondly, different mutator threads might observe different values when reading from an objects, depending on whether they read from the fromspace object or the tospace object. These two problems have to be addressed by any concurrent copying GC algorithm.

Like concurrent tracing, barriers can be used to implement concurrent copying. Two classic barriers are Baker's barrier [Baker, 1978] and Brooks' barrier [Brooks, 1984].

Baker's barrier maintains the invariant that mutators only operate on tospace objects. For each reference read, the barrier checks whether the referent is a fromspace object, and if so, the barrier performs one the following three actions. If the referent has been copied, the barrier will return the tospace version. If the referent is being copied by a collector, the barrier will wait until the copying is finished, and then return the tospace version. If the referent has not been copied, the barrier copies the object itself, and then returns the tospace version.

Brooks' barrier in contrast allows mutators to operate on fromspace objects. However, upon every reference read, the barrier does an unconditional indirection on the forwarding pointer in the object header, so that if a tospace version of the object is available, the tospace version is used instead. It is worth noting that Brooks' barrier was originally designed for incremental GC on CPUs with a single hardware thread, where mutators do not run in *parallel* with collectors. Therefore, Brooks' barrier, as described in the original paper, only has a read barrier. If mutators run in parallel with collectors, the read barrier on its own does not prevent lost updates. A stronger variation of the Brooks' barrier is required for concurrent copying GC. There are a number of ways this could be addressed. For example, in Shenandoah [Flood et al., 2016], a write barrier is used to forward any fromspace object that is written to.

As discussed in Section 6.5, these read barriers introduce high mutator overhead. The overhead is incurred not only during GC time, but also when GC is not underway, because they cannot be easily inserted or removed.

Alternatively, page protection can be used to implement concurrent copying GC [Appel et al., 1988; Click et al., 2005; Kermany and Petrank, 2006]. The idea is that a page that potentially contains stale references is protected, and every access to a protected page will generate a page fault. The trap handler can then fix up the references in that page. Like read barriers, page protection is also quite expensive, measured to be around 20% in a state-of-the-art collector [Pizlo et al., 2008] that uses page protection.

The high mutator overhead of these concurrent copying algorithms motivates this work, which is whether we can perform concurrent copying with HTM (see Section 2.2.3) with lower cost.

Sapphire

Sapphire [Hudson and Moss, 2003] is a concurrent copying GC algorithm. Like Brooks' barrier, Sapphire allows mutators to operate on fromspace objects, but interestingly

achieves so without requiring any read barrier. The key idea is that it uses a write barrier to mirror all object writes, so that updates are reflected in both fromspace objects and tospace objects when copying is underway. However, the write barriers used by Sapphire are expensive, because of the duplication of writes of both reference and primitive fields, and also the use of compare-and-swap (CAS) for writes. The copying of objects is also "semantic"; tospace objects point only at tospace of objects. The copying phase terminates when all objects have been copied, and then the collector "flips" all global variables and references on the stack to tospace objects at once.

Some related work of concurrent copying GC with HTM builds upon the Sapphire collector. Sapphire has the advantage that objects can be copied individually, and when copying an object, references to that object do not need to be identified. Applying HTM transactions in this context is a lot easier, that the CAS can be replaced by transactions.

6.7 Yieldpoints

A yieldpoint is a commonly used mechanism in the runtime system of managed languages. A yieldpoint allows mutators to be interrupted at well-defined points in its execution, including loop back edges, method prologues, and method epilogues [Lin et al., 2015]. Various actions can be taken inside the yieldpoint, such as polling for pending requests from the GC to suspend the mutator, biased-locking or profiling for feedback-driven compilation.

In this work, yieldpoints are used to let mutators handshake with collectors, and enter transactions if necessary (see Section 8.4).

Chapter 7

Related Work

In this chapter, I will discuss prior work on applying transactional memory (TM) to GC. In Section 7.1, I will show how TM can be used in parallel GC where there are race conditions among collector threads. In Section 7.2, I will show how TM can be applied in concurrent copying GC (see Section 6.6.2). For each scenario, I will identify the race conditions that need to be resolved for the soundness of GC, and then summarize how each of the algorithms applies TM to address the race conditions.

7.1 Parallel GC

In this section, I will examine related work on using TM in parallel GC, where multiple collector threads execute in parallel, hopefully improving the GC throughput. First, I will talk about parallel copying GC, where it is important that there does not exist multiple copies of the same object. Then, I will discuss parallel bitmap marking, which is an important identification mechanism used by some implications of algorithms like mark-sweep.

7.1.1 Parallel Copying GC

A parallel copying GC algorithm needs to discern the copying state of an object: uncopied, being copied, and copied. A race condition is that multiple collector threads try to copy the same object simultaneously, and update the copying state of the object from uncopied to being copied. This race condition leads to multiple distinct copies of the same original object being made, which makes the heap inconsistent. Therefore, it is important that one and only one copy of an object is made when relocating the object. Traditionally, this is achieved in the following way. When a collector thread attempts to copy an uncopied object, it atomically sets the state to being copied (*e.g.*, using compare-and-swap), informing other collector threads attempting to forward the same object.

Ritson et al. [2014] identified that transactional memory can be used to eliminate the intermediate being copied state, and an object changes atomically from uncopied to copied. A collector thread checks that the copying state of an object is uncopied in a transaction. In the same transaction, the thread sets the copying state to copied and

installs the forwarding pointer, and thus attempts to publish the copied object. The atomicity of transactional memory guarantees that, if any thread commits the transaction of copying an object, any other transaction on the same object will abort due to transactional conflict, ensuring only one copy of the object is ever made visible. An optimization is that multiple transactions are combined, amortizing the overhead of starting transactions.

7.1.2 Parallel Bitmap Marking

Many GC algorithms, such as mark-sweep, use marking in the identification phase of GC to record the liveness information of objects. Broadly speaking, the bits used for marking can either be stored in the object header or in bitmaps on the side.

We know that the load and store of machine words can be done atomically in many architectures. Therefore, when the object header is used for marking bits, the race, that multiple collectors are marking the same object, is benign.

However, when bitmap is used for marking, the mark bits of multiple objects might be stored in the same machine word. The race condition is that when multiple collect threads contend for the same machine word for marking different objects, the marking might get lost in the process.

Similar to parallel copying GC, the race condition is often resolved by using CAS, which hinders the marking throughput of collectors. Ritson et al. explored using HTM to solve the above problem. When marking is done in transactions, plain loads and stores can be used, and the atomicity of transactions guarantees that if transactions commit, no marking is lost.

7.2 Concurrent Copying GC

Previously, I have discussed prior work on applying TM on parallel GC, where multiple collector threads run in parallel. In this section, I will discuss attempts in using TM for concurrent copying GC, which is at the core of this work. Recall that two important problems in concurrent copying GC is the lost update problem and the atomicity of the movement of objects with respect to mutators (see Section 6.6.2).

I will discuss four different algorithms. For each of the algorithms, I will point out its shortcomings, motivating the novel algorithm I am introducing in Chapter 8.

7.2.1 STM Sapphire

McGachey et al. [2008] applied software transactional memory (STM) on Sapphire, referred to as STM Sapphire here. To the best of my knowledge, STM Sapphire is the first GC algorithm that uses TM. The STM Sapphire algorithm differs from the original Sapphire algorithm (see Section 6.6.2) in the following aspects.

The core idea of the original Sapphire algorithm is to allow mutators to operate on both fromspace objects and tospace objects, through the use of a write barrier that mirrors all writes to objects being copied. The STM Sapphire algorithm uses a strong

invariant, only allowing mutators to operate on the tospace objects. This is achieved by installing forwarding pointers in the object headers, and redirecting mutator reads and writes on copied objects through read and writer barriers.

The TM system used in STM Sapphire relies on version numbers in object headers for conflict detection. The version numbers are also cleverly used by the collectors for avoiding the overhead of starting full transactions for object copying. Instead, a collector stores the version numbers of all objects it copies, and compares the version numbers in the end. If any of the version numbers of copied objects has changed, some updates have been lost and the copying operation is aborted. It is worth noting that because the transactional metadata is recorded in object-granularity, only one object can be copied in one transaction.

The main problem for STM Sapphire is that read and write barriers are required just to provide weak isolation (see Section 2.2.1) for transactions due to how the STM system works. Even though GC barriers can piggyback on transactional barriers, saving the cost of following forwarding pointers again, the combined barriers still impose a big overhead on mutators. In addition, in order to provide strong isolation (of transactions from non-transactional code), the write barrier puts an object into exclusive mode before writing into it, further limiting the efficacy of the algorithm.

7.2.2 HTM Sapphire

Hardware transactional memory (HTM) has also been applied on Sapphire [Ritson et al., 2014; Ugawa et al., 2018], referred to as HTM Sapphire here.

In contrast to STM Sapphire, HTM Sapphire is closer to the original Sapphire collector. HTM Sapphire uses the same write barrier to replicate mutators writes, allowing mutators to operate on fromspace object. However, HTM Sapphire optimizes over the original Sapphire collector by replacing CAS in object copying with transactions.

HTM Sapphire addresses many of the disadvantages of STM Sapphire. With the work tracking done in the hardware at the address level instead of in per-object metadata, HTM Sapphire can copy multiple objects in a single transaction, amortizing the overhead [Ritson and Barnes, 2013] of starting a transaction. More importantly, the HTM implementation—Intel TSX—used in HTM Sapphire provides strong isolation for free. This removes the use of a read barrier and "places no restrictions on mutators", eliminating a major source of overhead.

However, HTM Sapphire still has the same overhead of duplicating mutator writes as the original Sapphire collector.

7.2.3 Collie

To the best of my knowledge, Collie is the first GC algorithm that uses HTM [Iyengar et al., 2012]. The core idea of the algorithm is the notion of "individually transplantable" objects.

Individually transplantable objects can be copied by collectors optimistically without using any transaction. Collectors only use transactions to publish the copying,

making sure that all references to a copied object (*the referrer set*) are updated atomically. This greatly reduces the amount of memory operations in transactions, decreasing the chance of aborts due to exceeding the capacity limit. However, individually transplantable objects need to satisfy two conditions. First, they are not written to during copying. Otherwise, updates to these objects can get lost. Second, the referrer set of the objects has to be stable so that no new reference to these objects can be created. Otherwise, collectors might not update all references to these objects, resulting in an inconsistent heap.

Any object that does not satisfy the above two conditions are marked as "non-individually transplantable" objects. Non-individually transplantable objects are not copied by collector transactions. Instead, these objects are virtually "copied" by creating a tospace virtual memory mapping of the fromspace pages. The reference update is then done through the use of a read barrier that fixes fromspace references to point to tospace objects. The read barrier also prevents more fromspace references from being created[1]. On Azul hardware, the read barrier is implemented in the hardware as a special reference load instruction.

Non-individually transplantable objects can be identified in three different ways. First, a write barrier marks any object being written to as non-individually transplantable. Second, non-individually transplantable objects can be implicitly identified when the read barrier operates. Since the read barrier fixes fromspace references, the barrier has to write to the referrer sets of some objects, which causes aborts of collector transactions operating on the same referrer sets. An aborted collector transaction can then mark objects as being non-individually transplantable. Third, in the pre-compaction checkpoint, all objects directly reachable from stack roots and registers are marked as non-individually transplantable, avoiding the complicated scenarios of updating references in registers and stacks.

The problems with Collie are twofold. Firstly, it still requires the use of a read barrier. Without proprietary hardware, the read barrier will be costly to implement on commodity hardware as discussed in Section 6.5. Secondly, the algorithm effectively pins all non-individually transplantable objects, reducing the efficacy of heap defragmentation.

7.2.4 Chihuahua

Chihuahua [Anderson et al., 2015] is a concurrent copying GC that uses HTM. The algorithm uses the semispace collector (see Section 6.1) as a starting point. However, the overall algorithmic structure is much closer to STM Sapphire (see Section 7.2.1) than semispace.

Unlike other Sapphire-based collectors, which rely on the replication of writes to allow mutators operate on fromspace objects, Chihuahua is similar to Collie (see Section 7.2.3) that it requires a read barrier and a write barrier to ensure that mutators operate on tospace objects. Collectors copy objects transactionally, which solves the lost

[1] References can only be obtained through allocation or existing references. If no fromspace references can be read, then no fromspace references can be created.

update problem. The object copying transaction simply copies an object and installs a forwarding pointer in the object header. Note that other than the barriers, Chihuahua does not place other constraints on mutators, since the strong isolation provided by the HTM system protects the object copying transactions from non-transactional mutators.

The problem with Chihuahua is that it still relies on a read barrier, which involves high mutator overhead. In addition, Chihuahua uses one copying transaction per object, incurring the high overhead of starting transactions [Ritson and Barnes, 2013]. Chihuahua also does not implement necessary compiler intrinsics for HTM. Instead, it has to perform calls to C functions using the *syscalls* interface in JikesRVM for HTM operations, which is much slower than executing generated instructions directly.

Chapter 8

Design

In Section 7.2, I surveyed related work on using transactional memory (TM) on concurrent copying GC. These algorithms, despite using TM, still require the use of read barriers (or expensive write barriers in the case of HTM Sapphire), which result in high mutator overhead. In this chapter, I will introduce the novel concurrent copying GC algorithm co-developed with Bond and Blackburn. The algorithm only requires a cheap write barrier, and it achieves soundness by briefly running mutators transactionally when the collector is moving objects.

Recall that the two most fundamental problems in designing a concurrent copying GC algorithm are the lost update problem and the atomic movement of objects (copying and reference updates) with respect to the mutators. Instead of introducing the algorithm as a whole, I will show step-by-step how the algorithm can be built from some very basic mechanisms, addressing the above two problems. The techniques introduced in this chapter only require a HTM system that provides strong isolation. However, in the pseudocode, I assume the programming model of RTM introduced in Section 2.3.

8.1 Setup

To begin with, we just need a simple stop-the-world copying GC algorithm, such as the classic semispace algorithm. Such an algorithm collects the entire heap during each GC cycle. As discussed in Section 2.2.3, hardware transactional memory (HTM) only has limited capacity due to hardware limitations. Therefore, it is critical that we are able to collect a small subset of the heap individually, so that the transaction capacity is big enough for transactions that the collector might perform. However, picking arbitrary objects to form the subset makes the membership test of the subset very expensive. Therefore, we often require the subset to be a contiguous section of the heap, in terms of virtual address ranges. I refer to such sections of the heap as *regions* in the following section. The goal is to move all reachable objects in the region identified for collection (the fromspace) to another region (the tospace).

Being able to perform copying GC on a region individually implies that we need to be able to determine all references from outside the region into the region (*inter-region references*). This is required both for identification, so that reachable objects are not

reclaimed, but also for reclamation, so that all references to copied objects are updated. Of course, one can find all such references by performing a full heap trace. However, the time taken to perform a full heap trace is proportional to the heap size, which can be very big in some applications. In addition, in the case of concurrent copying GC, the set of inter-region references might change while the mutators are running concurrently with the collector. Thus, it would be beneficial to be able to dynamically maintain the set of incoming inter-region references for a region, also known as its *remembered set*.

The above can be achieved by using a reference write barrier (see Section 6.5), which is shown in Algorithm 1. This is the only barrier used in this algorithm. Whenever mutators create a reference from outside the region into the region, the memory location containing the reference (the slot) is added to the remembered set of the region[1].

Algorithm 1 The write barrier

```
1 void objectReferenceWrite(ObjectReference src, Address slot, ObjectReference tgt) {
2     if (isInSpace(fromSpace, tgt)) {
3         if (!isInSpace(fromSpace, src)) {
4             remset.insert(slot);
5         }
6     }
7 }
```

8.2 Avoiding lost updates

In Section 8.1, I introduced the barrier. The barrier dynamically maintains remembered sets while mutators are executing. Then, remembered sets can be used by the collector to update all inter-region references when collecting a region. This allows the collector to collect the region individually without performing a full heap trace.

However, we still need to solve the lost update problem when the collector moves objects concurrently with mutator writes. The transactional memory solution (see Algorithm 2) is simple: objects are copied inside a transaction, and all references to these objects (found in the remembered set) are updated in the same transaction. When a transaction commits, all objects are atomically moved with respect to the mutators: no update to objects are lost during copying, and all fromspace references are updated.

8.3 The Problematic Gap

In Section 8.2, I introduced a collector transaction for atomically copying reachable fromspace objects into tospace and updating all fromspace references within the heap.

However, the transaction has a critical flaw. Remember that references not only reside within the heap, they can also be on the stacks of mutators. The reference write

[1]For readers that are familiar with GC algorithms, the write barrier is similar to the barrier used in a generational GC, or the barrier used in a region-based GC.

Algorithm 2 The collector transaction

```
1  void copyRegion() {
2      for (ObjectReference object: fromSpace) {
3          newObject = copy(object);
4          // Set forwarding pointer
5          forwardingAddress(object).store(newObject);
6      }
7      for (Address slot: remset) {
8          // Update fromspace reference
9          object = slot.load();
10         if (isInSpace(fromSpace, object)) {
11             // extract forwarding pointer
12             newObject = forwardingAddress(object).load();
13             slot.store(newObject);
14         }
15     }
16 }
```

barrier used in this algorithm does not mediate stack operations[2]. Otherwise, it will be prohibitively expensive, since stack operations are very frequent. Therefore, the collector will not be able to update stack references by using the remembered set.

The collector cannot scan the stacks of threads inside the transaction either. To scan the references on the stack, the collector looks up the reference map, which encodes the locations in a stack frame that are references, for a given method at a particular point of execution. Therefore, the collector has to be able to unwind the stack of each mutator, and extract the execution status (such as the saved instruction pointer) from each of the stack frames. This is impossible to do without stopping the mutator.

It is also impossible for the collector to notify mutators via shared memory in a transaction, which will abort the transaction due to conflicts.

As a result, there exists a problematic *gap* between the time that the collector publishes the copying results by committing the transaction and that the mutators are informed that the stacks need to be fixed. During this gap, albeit small, mutators might use fromspace references on the stack, resulting in the read of stale values or values written get lost. Therefore, the above algorithm is not a sound concurrent copying GC algorithm.

8.4 Covering the Gap

In Section 8.3, I pointed out a gap between the completion of the collector transaction and the mutator acknowledging that. During the gap, mutators might use fromspace references on the stack, which makes the concurrent copying algorithm unsound.

Before discussing how the gap is covered in the algorithm, I will first discuss two non-solutions to the gap. One option is to stop the mutator at the start of the collector

[2]It is possible to implement barriers that mediate stack operations. However, such barriers will be prohibitively expensive.

transaction. However, this amounts to stop-the-world GC, thus unhelpful. Another option is to add a barrier conditioned on whether a collector transaction has committed, so that mutators have the opportunity to update their stacks before using a stale reference. However, the barrier has to be on all object uses, which directly contradicts with the goal of this work of reducing the mutator overhead for concurrent copying GC.

The following parts introduce the novelty of this algorithm, that the gap can be covered by running mutators briefly in transactions.

Signal an Occurred Gap To identify whether a gap has occurred, a global monotonic counter `CollectorCopyingState` is added to indicate the copying state of the collector. The collector protocol, which is shown in Algorithm 3, ensures that the counter is odd before the collector starts a transaction (line 2). The counter is only even by incrementing an odd counter, either when a transaction commits (line 6), or after a transaction aborts (line 9). The gap between the completion of the collector transaction and the mutator acknowledging the completion now becomes the gap between the collector setting the counter to even and the mutator seeing the new counter value. To summarize, the counter changing from odd to even signals to mutators that a gap has possibly[3] occurred during the mutator execution.

Algorithm 3 The collector protocol

```
1  void collector() {
2      CollectorCopyingState++;
3      collectorHandshake();
4      if (XBEGIN() != aborted) {
5          copyRegion();
6          CollectorCopyingState++;
7          XEND();
8      } else {
9          CollectorCopyingState++;
10     }
11 }
```

Handshaking Between the Collector and Mutators When the collector sets the counter to odd (Algorithm 3, line 2), it requests (Algorithm 4, line 2) mutators to take yieldpoints (see Section 6.7) to acknowledge the new counter. Before starting the collector transaction, the collector waits (Algorithm 4, line 5) until all mutators acknowledge the new counter value and follow the mutator protocol to avoid unsound execution.

When the mutators take the yieldpoint (see Algorithm 5), they acknowledge the handshake by setting a thread local snapshot of the counter value (line 9). In addition, if one or more collector transactions might have completed since last time the counter was read (line 5), the mutators update their own stack, fixing any fromspace references if found. It is worth noting that when the collector transaction aborts, the counter is

[3] The counter changing from odd to even can also be due to the collector aborting a transaction.

Algorithm 4 The collector handshake

```
1  void collectorHandshake() {
2      for (Thread t: mutatorThreads) {
3          t.takeYieldPoint = true;
4      }
5      for (Thread t: mutatorThreads) {
6          while (t.copyingState == CollectorCopyingState) {
7          }
8      }
9  }
```

still updated, but no visible change has been made to the heap, which is benign. In this case, the mutators still need to walk the stacks, which is unnecessary.

Algorithm 5 The mutator handshake

```
1  void mutatorHandshake() {
2      int csSnapshot = CollectorCopyingState;
3      if (csSnapshot > t.copyingState) {
4          // (t.copyingState, csSnapshot] contains even number
5          if (interval(t.copyingState, csSnapshot).hasEven()) {
6              // one or more collector transactions have completed
7              fixStack();
8          }
9          t.copyingState = csSnapshot;
10     }
11 }
```

Undo Execution During the Gap Finally, I will introduce the mutator protocol, which makes mutators execute in a series of transactions if the collector is attempting a collection. As shown in Algorithm 5, mutators are notified in the yieldpoints that the collector is attempting a collection. Recall that the execution of the application code is interrupted in yieldpoints, and therefore, the handshaking happens before any potentially erroneous execution of the application code.

The mutator protocol, shown in Algorithm 6, is that when mutators read an odd counter value, the application code is executed in a transaction until the next yieldpoint. In addition, prior to committing the transaction, the current counter value is compared with the last value read by the mutator.

The correctness can be demonstrated using a case-by-case analysis.

(1) The collector transaction finishes between line 2 and line 4. In this case, the application code is running entirely within the gap. However, the check at line 6 will explicitly abort the transaction.

(2) The collection transaction finishes after line 4 but before line 11. In this case, the application code is partially running during the gap. However, there will be a transactional abort due to the read at line 6.

Algorithm 6 The mutator protocol

```
1  void mutator(Thread t) {
2      mutatorHandshake();
3      if (t.copyingState.isOdd()) {
4          if (XBEGIN() != aborted) {
5              runUntilNextYieldpoint();
6              if (CollectorCopyingState > t.copyingState) {
7                  // copying state has changed since last we knew
8                  // execution may be wrong
9                  XABORT();
10             }
11             XEND();
12         } else {
13             mutatorHandshake();
14         }
15     }
16 }
```

(3) The mutator transaction commits. In this case, the semantics of HTM guarantees that no application code is executed during the gap.

To summarize, when covering a mutator's execution with a transaction, the transaction will abort, either explicitly or due to transactional conflict, if the collector publishes the results of copying. This achieves the effect of a barrier, conditioned on that collector publishing copying, but without paying any of instrumentation cost.

8.5 Summary

I introduced the novel concurrent copying GC algorithm using HTM in previous sections. I started with adding a write barrier to a simple stop-the-world copying GC to dynamically maintain the remembered set. This allows a region to be collected individually without requiring a full heap trace. Then, I identified the problematic gap between the completion of the collector transaction and the mutators acknowledging the completion. Mutators executing during the gap can potentially use the fromspace references on the stack, resulting in erroneous execution. Next, I described how the collector and mutators can coordinate around a monotonic counter. By running mutators inside transactions, any execution during the gap will result in the abort of the mutator transaction, ensuring all incorrect execution is rolled back. Compared with read barriers, which cannot be easily disabled and incurs overhead for each read, our algorithm only incurs the overhead when starting a transaction, and mutators are only running in transactions if the collector is attempting a collection.

In the next sections, I will describe some optimizations that I have devised and implemented based on the insights of HTM described in the first part of the thesis. Note that the mutator transaction described in Algorithm 6 almost entirely consists of the execution of the application code, and therefore, there is not much room for any optimization. As a result, I will focus on optimizing the collector transactions.

8.6 Optimization: Optimistic Copying

The first optimization is optimistic copying. The goal of the optimization is to reduce the amount of transactional writes in the collector transaction, since the transactional write capacity is much lower than the transactional read capacity as seen in Chapter 5. We observe that the copying of objects inside a transaction will generate a large amount of transactional writes. The idea is to optimistically perform the actual copying of objects outside the collector transaction, eliminating the main source of transactional writes.

This is achieved using an intermediate copying stage in a shadow region. For each object O identified for copying in fromspace, a word-by-word duplicate O_s is created in a shadow region, acting as a snapshot of O. Then, O_s is used as a surrogate for O to create the target copy O' in tospace. Finally, O and O_s are compared in a transaction. If O and O_s match, no write to O has been lost during the copying, and O' is correct. If O and O_s do not match, the copying is wrong, and the transaction is aborted. A further optimization is that we can repair O' by copying from O wherever O and O_s do not agree.

This method turns a large amount of transactional writes (copying O into O' directly) inside the collector transaction into transactional reads (comparing O and O_s), decreasing the chance of aborts due to exceeding the capacity of HTM.

8.7 Optimization: Cache Warmup

The second optimization is the cache warmup. In Section 5.3, I showed that the transactional read capacity depends on the cache status. In particular, if the cache is largely occupied by valid but unneeded cache lines, the cache lines from transactional work might be evicted despite being recently accessed due to the pseudo LRU replacement policy, causing superfluous capacity aborts.

As discussed previously, both invalidating the cache using `wbinvd` or warming up the cache can improve the chance of transactions successfully committing. However, the `wbinvd` instruction invalidates both data and instruction caches at all levels, causing the slowdown of all running threads. Therefore, it is preferable to use cache warmups. Prior to starting the collector transaction, the body of the transaction, especially the comparisons of shadow objects, is executed but the results are discarded. When executed multiple times, the majority of cache lines accessed in the transaction are likely in the cache, reducing the chance of getting evicted due to pseudo LRU.

Chapter 9

Implementation and Evaluation

Implementing a concurrent copying GC is no easy task, and often requires an enormous amount of engineering to make the GC stable enough to run real-world workloads, and to tune the GC to obtain optimal performance. This is unfortunately outside the scope of this thesis due to time constraint. In this section, I will focus on demonstrating that the algorithm described in Chapter 8 is viable. In particular, I will describe some handcrafted test programs I wrote to demonstrate that the important mechanisms of the algorithm are working.

9.1 Setup

The algorithm is implemented in MMTk [Blackburn et al., 2004a,b].

The implementation tries to follow the algorithm as faithfully as possible, but it has more pragmatic shortcuts in certain places. I start with the simplest region based collector implemented in MMTk [Zhao and Blackburn, 2020]. However, only one region can be eligible for concurrent copying collection at a time. This simplifies the implementation of the barrier (see Section 8.1).

I also simplify how concurrent GC is triggered. Instead of triggering a concurrent GC when the region is full or almost full, I changed the `System.gc()` handler so that a concurrent GC can be manually triggered.

From the application point of view, only objects, whose class has a special annotation, can be allocated into the concurrently collected region. This makes the debugging significantly easier, since I can control exactly which objects should be inside the concurrently collected region, and then check whether they have been correctly copied.

To verify whether an object has been correctly collected, I check that all references to the object have been updated to point to the copied version, and the fields of the object are preserved. The former can be done by using compiler intrinsics (known as `vmmagic`), and the latter can be done through normal Java code.

9.2 Test Programs

All test programs are around linked lists, and the canonical node definition of a singly linked list is used as shown in Algorithm 7. Note the special annotation added to the `Node` class.

Algorithm 7 A linked list node.

```
1  @ConcurrentCollection
2  class Node {
3    public volatile int value;
4    public volatile Node next;
5  }
```

The first program (see Algorithm 8) tests whether stack references are updated correctly. This requires the update of stack references by the owning mutator in a yieldpoint.

Algorithm 8 Read from an object directly reachable from the stack roots.

```
1  public static void main(String[] args) {
2    Node a = new Node();
3    a.value = 42;
4    VM.sysWriteln(ObjectReference.fromObject(a).toAddress());
5    VM.sysWriteln(ObjectReference.fromObject(a.next).toAddress());
6
7    System.gc();
8
9    // wait for the concurrent GC to finish
10   int wait = 1 << Integer.parseInt(args[0]);
11   for (int i = 0; i < wait; i++) {
12   }
13
14   VM.sysWriteln(ObjectReference.fromObject(a).toAddress());
15   VM.sysWriteln(ObjectReference.fromObject(a.next).toAddress());
16   assert a.value == 42;
17 }
```

The second program is similar to the first one, but it also tests whether any concurrent update is lost. This requires that the collector's copying of objects is atomic with respect to the mutators.

Finally, the third program checks whether objects not directly reachable from stack roots are correctly collected. This requires that all objects are correctly traced, and heap references are correctly updated in the collector transaction.

Although not comprehensive, the test programs shown here demonstrate that the most fundamental mechanisms of the algorithm are working.

Algorithm 9 Concurrently write to an object directly reachable from the stack roots.

```
public static void main(String[] args) {
  Node a = new Node();
  a.value = 42;
  VM.sysWriteln(ObjectReference.fromObject(a).toAddress());
  VM.sysWriteln(ObjectReference.fromObject(a.next).toAddress());

  System.gc();

  // write to the object when the GC is running
  int wait = 1 << Integer.parseInt(args[0]);
  int add = 0;
  for (int i = 0; i < count; i++) {
    if (i % 100 == 0) {
      add += 1;
      a.value += 1;
    }
  }

  VM.sysWriteln(ObjectReference.fromObject(a).toAddress());
  VM.sysWriteln(ObjectReference.fromObject(a.next).toAddress());
  assert a.value == 42;
}
```

Algorithm 10 Read from objects only transitively reachable from the stack roots.

```
public static void main(String[] args) {
  Node head = new Node();
  head.value = 0;
  Node tail = head;

  for (int i = 1; i <= 128; i++) {
    Node n = new Node();
    n.value = i;
    tail.next = n;
    VM.sysWriteln(ObjectReference.fromObject(tail).toAddress());
    VM.sysWriteln(ObjectReference.fromObject(tail.next).toAddress());
    tail = n;
  }

  System.gc();

  Node n = head;
  int i = 0;
  while (n.next != null) {
    VM.sysWriteln(ObjectReference.fromObject(n).toAddress());
    VM.sysWriteln(ObjectReference.fromObject(n.next).toAddress());
    assert n.value == i;
    i++;
  }
}
```

Part IV

Epilogue

Chapter 10

Conclusion

The goal of this thesis is to design a concurrent copying GC algorithm that has lower mutator overhead compared with using read barriers or page protections. By analysing existing work on using HTM to facilitate GC under the same framework, this thesis crisply identified how different HTM GC algorithms relate to each other, how they differ, and their mutator overhead characteristics. Then, the thesis introduced a novel concurrent copying GC algorithm, and shows stepwise how the algorithm can be constructed using simple primitives. Finally, a set of test programs were used to test that the fundamental mechanisms of the algorithm are working, and that the algorithm is viable.

The practicality of the algorithm is predicated on successfully exploiting the capacity of HTM transactions. To achieve this, this thesis deepened the understanding of factors that affect the HTM capacity. In particular, this thesis explored how the cache status affects the HTM capacity, and how both warming up the cache and invalidating the cache—two seemingly opposite operations—can help large read-only transactions commit. The results resolved the apparent contradiction in the capacity numbers cited by prior work.

10.1 Future Work

The following sections point out future research directions following this work.

10.1.1 Performance Evaluation of Optimizations

Section 8.6 and Section 8.7 presented two optimizations of the algorithm introduced in Chapter 8. Both optimizations have been implemented, and have been shown to work in informal tests. Future work can quantitatively measure how these optimizations affect the maximum number of objects that can be collected in a single transaction.

10.1.2 Improving the Implementation

As discussed in Section 9.1, the current implementation of the algorithm has some compromises for practicality reasons. However, to truly demonstrate the efficacy of

the algorithm, the implementation has to be improved and tuned, so that more realistic benchmarks can run with desired performance.

10.1.3 Concurrent Stack Processing

Finally, I discussed how the techniques introduced in this work can be applied to another area of GC.

With the prevalence of multiprocessors, GC can now take advantage of the available processing power to increase the collector throughput. However, applications written in managed languages start to utilize the hardware parallelism as well, often in the form of spawning many threads. As discussed in Section 6.2, tracing GC often needs to process roots, including roots on the stack of mutator threads. Stack roots gathering often requires the suspension of mutator threads, of which the pause time is proportional to the number of mutator threads. This poses a scalability problem.

Just like concurrent tracing GC (see Section 6.6.1) and concurrent copying GC (see Section 6.6.2), we could address the above problem by allowing concurrent stack processing. The synchronization problems need to be addressed in concurrent stack processing are twofold. Firstly, mutators may modify a stack frame that collectors are processing. Secondly, mutators may be executing on a stack frame containing stale references, in the case that collectors have relocated objects but have not had a chance to update the references on the stack.

We observe that the synchronization problems of concurrent stack scanning are very similar to what transactions described in Chapter 8 try to address. To solve the first problem, we can have collectors running in transactions when processing stack frames, and if a stack frame is changed during processing, the transactions will abort to transactional conflict. To solve the second problem, we can have mutators running in a series of transactions when concurrent stack processing is underway. A counter can be used to indicate stack status, and collectors can update the counter when references on the stack may become invalid. Before a mutator can use a stack frame, it reads the corresponding counter, and the transactions will abort if the counter has been updated by the collector.

Oŝterlund [2020] proposed a very similar idea, the stack watermark barrier. There are watermarks associated with each stack to indicate the current processing status. When collectors finish processing stack frames, it adjusts the watermark. When a mutator thread tries to use a stack frame, it checks the corresponding watermark to see whether the frame has been processed, and if not, the thread enters the slow path to fix the references. The invocation of the barrier, in other words, the check against the watermark, piggybacks on the yieldpoint. Since yieldpoints are inserted into operations that may pop a stack frame, such as returning from a function, they are natural places to perform such checks.

Bibliography

Abel, A. and Reineke, J., 2014. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 141–142. (cited on page 13)

Abel, A. and Reineke, J., 2020. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. (cited on page 13)

Agarwal, V.; Hrishikesh, M. S.; Keckler, S. W.; and Burger, D., 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00 (Vancouver, British Columbia, Canada, 2000), 248–259. Association for Computing Machinery, New York, NY, USA. doi:10.1145/339647.339691. https://doi.org/10.1145/339647.339691. (cited on page 7)

Anderson, T. A.; O'Neil, M.; and Sarracino, J., 2015. Chihuahua: A concurrent, moving, garbage collector using transactional memory. In *10th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT 2015 (Portland, Oregon, USA, 2015). http://transact2015.cse.lehigh.edu/anderson-transact-2015.pdf. (cited on page 46)

Appel, A. W.; Ellis, J. R.; and Li, K., 1988. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88 (Atlanta, Georgia, USA, 1988), 11–20. Association for Computing Machinery, New York, NY, USA. doi:10.1145/53990.53992. https://doi.org/10.1145/53990.53992. (cited on page 40)

Baker, H. G., 1978. List processing in real time on a serial computer. *Commun. ACM*, 21, 4 (Apr. 1978), 280–294. doi:10.1145/359460.359470. https://doi.org/10.1145/359460.359470. (cited on page 40)

Bernstein, P. A., 1990. Transaction processing monitors. *Commun. ACM*, 33, 11 (Nov. 1990), 75–86. doi:10.1145/92755.92767. https://doi.org/10.1145/92755.92767. (cited on page 8)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004a. Myths and realities: The performance impact of garbage collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04 (New York, NY, USA, 2004), 25–36. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1005686.1005693. https://doi.org/10.1145/1005686.1005693. (cited on pages 37 and 57)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004b. Oil and water? high performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, 137–146. IEEE Computer Society, USA. (cited on page 57)

Blackburn, S. M. and Hosking, A. L., 2004. Barriers: Friend or foe? In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04 (Vancouver, BC, Canada, 2004), 143–151. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1029873.1029891. https://doi.org/10.1145/1029873.1029891. (cited on page 38)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08 (Tucson, AZ, USA, 2008), 22–32. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1375581.1375586. https://doi.org/10.1145/1375581.1375586. (cited on page 35)

Brooks, R. A., 1984. 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 (Austin, Texas, USA, 1984), 256–262. Association for Computing Machinery, New York, NY, USA. doi:10.1145/800055.802042. https://doi.org/10.1145/800055.802042. (cited on page 40)

Cai, Z.; Liu, Z.; Maleki, S.; Musuvathi, M.; Mytkowicz, T.; Nelson, J.; and Saarikivi, O., 2020. Synthesizing optimal collective algorithms. *arXiv e-prints*, (Aug. 2020). https://arxiv.org/abs/2008.08708. (cited on page vii)

Chaudhry, S.; Cypher, R.; Ekman, M.; Karlsson, M.; Landin, A.; Yip, S.; Zeffer, H.; and Tremblay, M., 2009. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29, 2 (2009), 6–16. (cited on pages 9 and 10)

Cheney, C. J., 1970. A nonrecursive list compacting algorithm. *Commun. ACM*, 13, 11 (Nov. 1970), 677–678. doi:10.1145/362790.362798. https://doi.org/10.1145/362790.362798. (cited on page 36)

Chí, C. M.; Chung, J.; Kozyrakis, C.; and Olukotun, K., 2008. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, 35–46. doi:10.1109/IISWC.2008.4636089. (cited on page 16)

Christie, D.; Chung, J.-W.; Diestelhorst, S.; Hohmuth, M.; Pohlack, M.; Fetzer, C.; Nowack, M.; Riegel, T.; Felber, P.; Marlier, P.; and Rivière, E., 2010. Evaluation of AMD's Advanced Synchronization Facility within a complete transactional memory stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10 (Paris, France, 2010), 27–40. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1755913.1755918. https://doi.org/10.1145/1755913.1755918. (cited on page 10)

Click, C., 2010. HTM will not save the world. In *Transactional Memory Workshop 2010*, TMW 2010. https://sss.cs.purdue.edu/projects/tm/tmw2010//talks/Click-2010_TMW.pdf. (cited on page 10)

Click, C.; Tene, G.; and Wolf, M., 2005. The Pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05 (Chicago, IL, USA, 2005), 46–56. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1064979.1064988. https://doi.org/10.1145/1064979.1064988. (cited on page 40)

Dalessandro, L.; Spear, M. F.; and Scott, M. L., 2010. NOrec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10 (Bangalore, India, 2010), 67–78. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1693453.1693464. https://doi.org/10.1145/1693453.1693464. (cited on page 10)

Dennard, R. H.; Gaensslen, F. H.; Yu, H.; Rideout, V. L.; Bassous, E.; and LeBlanc, A. R., 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9, 5 (Oct 1974), 256–268. doi:10.1109/JSSC.1974.1050511. (cited on page 7)

Dice, D.; Harris, T.; Kogan, A.; and Lev, Y., 2015. The influence of malloc placement on TSX hardware transactional memory. *arXiv e-prints*, (Apr. 2015). https://arxiv.org/abs/1504.04640v2. (cited on page 16)

Dice, D.; Shalev, O.; and Shavit, N., 2006. Transactional locking ii. In *Distributed Computing*, 194–208. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 10)

Diegues, N.; Romano, P.; and Rodrigues, L., 2014. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14 (Edmonton, AB, Canada, 2014), 3–14. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2628071.2628080. https://doi.org/10.1145/2628071.2628080. (cited on pages 10, 15, and 16)

Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; and Steffens, E. F. M., 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21, 11 (Nov. 1978), 966–975. doi:10.1145/359642.359655. https://doi.org/10.1145/359642.359655. (cited on page 39)

Dragojević, A.; Guerraoui, R.; and Kapalka, M., 2009. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09 (Dublin, Ireland, 2009), 155–165. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1542476.1542494. https://doi.org/10.1145/1542476.1542494. (cited on page 10)

Felber, P.; Fetzer, C.; and Riegel, T., 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08 (Salt Lake City, UT, USA, 2008), 237–246. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1345206.1345241. https://doi.org/10.1145/1345206.1345241. (cited on page 10)

Fenichel, R. R. and Yochelson, J. C., 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12, 11 (Nov. 1969), 611–612. doi:10.1145/363269.363280. https://doi.org/10.1145/363269.363280. (cited on page 36)

Flood, C. H.; Kennke, R.; Dinn, A.; Haley, A.; and Westrelin, R., 2016. 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 (Lugano, Switzerland, 2016). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2972206.2972210. https://doi.org/10.1145/2972206.2972210. (cited on page 40)

Goel, B.; Titos-Gil, R.; Negi, A.; McKee, S. A.; and Stenstrom, P., 2014. Performance and energy analysis of the Restricted Transactional Memory implementation on Haswell. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 615–624. doi:10.1109/IPDPS.2014.70. (cited on pages 15, 16, and 19)

Gray, J. and Reuter, A., 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. ISBN 1558601902. (cited on page 8)

Gruss, D.; Lettner, J.; Schuster, F.; Ohrimenko, O.; Haller, I.; and Costa, M., 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, 217–233. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss. (cited on pages 16, 17, and 19)

Harris, T.; Larus, J.; and Rajwar, R., 2010. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edn. ISBN 1608452352. (cited on page 9)

Hasenplaugh, W.; Nguyen, A.; and Shavit, N., 2015. Quantifying the capacity limitations of hardware transactional memory. In *7th Workshop on the Theory of Transactional Memory*, WTTM 2015 (Donostia-San Sebastián, Spain, 2015). http://www.gsd.inesc-id.pt/~salaa/wttm2015/html/abstracts/Hasenplaugh.pdf. (cited on pages 13, 16, 17, and 19)

Herlihy, M. and Moss, J. E. B., 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on*

Computer Architecture, ISCA '93 (San Diego, California, USA, 1993), 289–300. Association for Computing Machinery, New York, NY, USA. doi:10.1145/165123.165164. https://doi.org/10.1145/165123.165164. (cited on pages 7, 8, and 9)

Herlihy, M. P. and Wing, J. M., 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12, 3 (Jul. 1990), 463–492. doi:10.1145/78969.78972. https://doi.org/10.1145/78969.78972. (cited on page 8)

Hong, S.; Oguntebi, T.; Casper, J.; Bronson, N.; Kozyrakis, C.; and Olukotun, K., 2010. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, 1–11. doi:10.1109/IISWC.2010.5648812. (cited on page 16)

Hudson, R. L. and Moss, J. E. B., 2003. Sapphire: copying garbage collection without stopping the world. *Concurrency and Computation: Practice and Experience*, 15, 3-5 (2003), 223–261. doi:10.1002/cpe.712. https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.712. (cited on page 40)

Intel Corporation, 2020a. 6th generation intel® processor family specification update. https://cdrdv2.intel.com/v1/dl/getContent/332689. Accessed: 2020-10-15. (cited on page 11)

Intel Corporation, 2020b. Desktop 4th generation Intel® Core™ processor family, desktop Intel® Pentium® processor family, and desktop Intel® Celeron® processor family specification update. https://www.intel.com.au/content/www/au/en/products/docs/processors/core/4th-gen-core-family-desktop-specification-update.html. Accessed: 2020-10-15. (cited on page 11)

Intel Corporation, 2020c. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. (cited on page 23)

Intel Corporation, 2020d. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. (cited on pages 10 and 11)

Iyengar, B.; Tene, G.; Wolf, M.; and Gehringer, E., 2012. The Collie: A wait-free compacting collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12 (Beijing, China, 2012), 85–96. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2258996.2259009. https://doi.org/10.1145/2258996.2259009. (cited on page 45)

Jahagirdar, S.; George, V.; Sodhi, I.; and Wells, R., 2012. Power management of the third generation Intel Core micro architecture formerly codenamed Ivy Bridge. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, 1–49. (cited on page 13)

Jones, R.; Hosking, A.; and Moss, E., 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edn. ISBN 1420082795. (cited on page 36)

Kermany, H. and Petrank, E., 2006. The Compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06 (Ottawa, Ontario, Canada, 2006), 354–363. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1133981.1134023. https://doi.org/10.1145/1133981.1134023. (cited on page 40)

Lin, Y.; Wang, K.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2015. Stop and go: Understanding yieldpoint behavior. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15 (Portland, OR, USA, 2015), 70–80. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2754169.2754187. https://doi.org/10.1145/2754169.2754187. (cited on page 41)

Linden, G., 2006. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. Accessed: 2020-11-04. (cited on page 3)

Lomet, D. B., 1977. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software* (Raleigh, North Carolina, 1977), 128–137. Association for Computing Machinery, New York, NY, USA. doi:10.1145/800022.808319. https://doi.org/10.1145/800022.808319. (cited on page 8)

McGachey, P.; Adl-Tabatabai, A.-R.; Hudson, R. L.; Menon, V.; Saha, B.; and Shpeisman, T., 2008. Concurrent GC leveraging transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08 (Salt Lake City, UT, USA, 2008), 217–226. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1345206.1345238. https://doi.org/10.1145/1345206.1345238. (cited on page 44)

Nakaike, T.; Odaira, R.; Gaudet, M.; Michael, M. M.; and Tomari, H., 2015. Quantitative comparison of hardware transactional memory for Blue Gene/Q, ZEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15 (Portland, Oregon, 2015), 144–157. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2749469.2750403. https://doi.org/10.1145/2749469.2750403. (cited on pages 16 and 17)

Österlund, E., 2020. JEP 376: ZGC: Concurrent thread-stack processing. https://openjdk.java.net/jeps/376. Accessed: 2020-10-10. (cited on page 64)

Pereira, M. M.; Gaudet, M.; Amaral, J. N.; and Araújo, G., 2014. Multi-dimensional evaluation of Haswell's transactional memory performance. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, 144–151. doi:10.1109/SBAC-PAD.2014.33. (cited on pages 15 and 16)

Pizlo, F.; Petrank, E.; and Steensgaard, B., 2008. A study of concurrent real-time garbage collectors. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08 (Tucson, AZ, USA, 2008), 33–44.

Association for Computing Machinery, New York, NY, USA. doi:10.1145/1375581.1375587. https://doi.org/10.1145/1375581.1375587. (cited on page 40)

Printezis, T., 2001. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. USENIX. http://www.usenix.org/publications/library/proceedings/jvm01/printezis.html. (cited on page 37)

Ramakrishnan, R. and Gehrke, J., 2000. *Database Management Systems*. McGraw-Hill, Inc., USA, 2nd edn. ISBN 0072440422. (cited on page 8)

Ritson, C. G. and Barnes, F. R., 2013. An evaluation of Intel's Restricted Transactional Memory for CPAs. In *Communicating Process Architectures 2013 Proceedings of the 35th WoTUG Technical Meeting* (Eds. P. H. Welch; F. R. Barnes; J. F. Broenink; K. Chalmers; J. B. Pedersen; and A. T. Sampson), 271–291. Open Channel Publishing. https://kar.kent.ac.uk/36939/. (cited on pages 15, 17, 19, 20, 21, 23, 45, and 47)

Ritson, C. G.; Ugawa, T.; and Jones, R. E., 2014. Exploring garbage collection with Haswell hardware transactional memory. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14 (Edinburgh, United Kingdom, 2014), 105–115. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2602988.2602992. https://doi.org/10.1145/2602988.2602992. (cited on pages 43, 44, and 45)

Robson, J. M., 1971. An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18, 3 (Jul. 1971), 416–423. doi:10.1145/321650.321658. https://doi.org/10.1145/321650.321658. (cited on page 37)

Robson, J. M., 1974. Bounds for some functions concerning dynamic storage allocation. *J. ACM*, 21, 3 (Jul. 1974), 491–499. doi:10.1145/321832.321846. https://doi.org/10.1145/321832.321846. (cited on page 37)

RPCS3, 2020. Major TSX regressions on cpus with TSX_Force_Abort present. https://github.com/RPCS3/rpcs3/issues/6028. Accessed: 2020-10-15. (cited on page 12)

Schwarz, M.; Lipp, M.; Moghimi, D.; Van Bulck, J.; Stecklina, J.; Prescher, T.; and Gruss, D., 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19 (London, United Kingdom, 2019), 753–768. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3319535.3354252. https://doi.org/10.1145/3319535.3354252. (cited on page 11)

Shavit, N. and Touitou, D., 1995. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95 (Ottowa, Ontario, Canada, 1995), 204–213. Association for Computing Machinery, New York, NY, USA. doi:10.1145/224964.224987. https://doi.org/10.1145/224964.224987. (cited on page 9)

Soman, S.; Krintz, C.; and Bacon, D. F., 2004. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04 (Vancouver, BC, Canada, 2004), 49–60. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1029873.1029880. https://doi.org/10.1145/1029873.1029880. (cited on page 37)

Steele, G. L., 1975. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18, 9 (Sep. 1975), 495–508. doi:10.1145/361002.361005. https://doi.org/10.1145/361002.361005. (cited on page 39)

Stone, J. M.; Stone, H. S.; Heidelberger, P.; and Turek, J., 1993. Multiple reservations and the Oklahoma update. *IEEE Parallel Distributed Technology: Systems Applications*, 1, 4 (1993), 58–71. (cited on page 8)

Ugawa, T.; Ritson, C. G.; and Jones, R. E., 2018. Transactional Sapphire: Lessons in high-performance, on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst.*, 40, 4 (Dec. 2018). doi:10.1145/3226225. https://doi.org/10.1145/3226225. (cited on page 45)

Wang, A.; Gaudet, M.; Wu, P.; Amaral, J. N.; Ohmacht, M.; Barton, C.; Silvera, R.; and Michael, M., 2012. 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 (Minneapolis, Minnesota, USA, 2012), 127–136. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2370816.2370836. https://doi.org/10.1145/2370816.2370836. (cited on page 10)

Wang, Z.; Qian, H.; Li, J.; and Chen, H., 2014. Using Restricted Transactional Memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14 (Amsterdam, The Netherlands, 2014). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2592798.2592815. https://doi.org/10.1145/2592798.2592815. (cited on pages 16, 17, and 19)

Yahfz, 2020. RPCS3 inside look: A deep-dive into hardware and performance scaling! https://rpcs3.net/blog/2020/08/21/hardware-performance-scaling/. Accessed: 2020-10-14. (cited on page 10)

Yang, X.; Blackburn, S. M.; Frampton, D.; and Hosking, A. L., 2012. Barriers reconsidered, friendlier still! In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12 (Beijing, China, 2012), 37–48. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2258996.2259004. https://doi.org/10.1145/2258996.2259004. (cited on page 38)

Yoo, R. M.; Hughes, C. J.; Lai, K.; and Rajwar, R., 2013. Performance evaluation of Intel® Transactional Synchronization Extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13 (Denver, Colorado, 2013). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2503210.2503232. https://doi.org/10.1145/2503210.2503232. (cited on page 15)

Yuasa, T., 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11, 3 (1990), 181–198. doi:https://doi.org/10.1016/0164-1212(90)90084-Y. http://www.sciencedirect.com/science/article/pii/016412129090084Y. (cited on page 39)

Zhao, W. and Blackburn, S. M., 2020. Deconstructing the Garbage-First collector. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20 (Lausanne, Switzerland, 2020), 15–29. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3381052.3381320. https://doi.org/10.1145/3381052.3381320. (cited on page 57)