# MECHANISMS FOR UNBOUNDED, CONFLICT-ROBUST HARDWARE TRANSACTIONAL MEMORY

Colin Blundell

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2010

---

Milo M. K. Martin, Associate Professor of Computer and Information Science
Supervisor of Dissertation

---

Jianbo Shi, Associate Professor of Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Rajeev Alur, Professor of Computer and Information Science

André DeHon, Associate Professor of Electrical and System Engineering

Maurice Herlihy, Professor of Computer Science, Brown University

Amir Roth, Associate Professor of Computer and Information Science

Mechanisms for Unbounded, Conflict-Robust Hardware Transactional Memory

*This dissertation is dedicated to my wife, Angelina.*

*Without you, this would not have been possible.*

# Acknowledgements

This dissertation would not have been possible without the love and support of my family. My deepest thanks go to my wife, Angelina. The opportunity to meet her has been the greatest reward of my decision to go to graduate school. She is both the source of my success and the reason that this success has meaning. I also thank Jacob for the joy that he has brought to my life and Merlin for his constant good humor, support, and loyalty.

The support of my mother, father, and brother has been instrumental in me reaching this point. They have shared in the joy of my successes and have helped me weather the setbacks. The foundation of my later success was laid in my parents' teaching when I was young. My brother David has been there with me through the ups and downs of our entire lives; he is the best friend a brother could ever hope for.

I am also deeply grateful to my extended family: the Boyarins, the Blundells, and the Stelmachs. The Boyarins have encouraged a spirit of questioning that has served me well as a researcher. My mathematical bent, on the other hand, can be traced to the influence of the Blundells. The Stelmachs have welcomed me as a son. All of these family members have given me constant love and support, and I thank them all. I wish to especially thank my cousin Jonah and my uncle Ian for the extraordinary closeness that we have shared over the course of my life.

I have been fortunate to be advised by Milo Martin. The last five years have been an awesome ride, during which I have benefited greatly from Milo's degree of technical knowledge and ability to share that knowledge. In addition, he has been the best advisor that I can imagine: generous with his time and wisdom, supportive through good times and bad times, extremely fair, and most of all, fun. He is not only an advisor to whom I am deeply indebted but a close friend.

In addition to Milo, I thank the other professors of UPenn's Architecture and Compilers Group (ACG), Amir Roth and E Lewis. Amir and E have greatly contributed to my success and to my

enjoyment of graduate school. Learning about microarchitecture from Amir has been like drinking from a firehose. E has been a second advisor to me. I also thank the members of my committee. Amir, André DeHon, Maurice Herlihy, and Rajeev Alur all provided valuable insight on my dissertation research, for which I am grateful.

I am also grateful for the mentorship of many other professors and senior colleagues. I have greatly benefited from the opportunity to collaborate with Tom Wenisch, who has taught me a great deal about computer architecture as well as the value of hard work. I am grateful to Steve Zdancewic for helping to broaden the scope of my research through the opportunity to collaborate on the Hard-Bound project. I also thank Insup Lee and Sampath Kannan for their support and mentorship during my early years in graduate school, Corina Pasareanu and Dimitra Giannakopoulou for their support during an instructive internship at NASA-Ames, and Joseph Silverman, Kathi Fisler, and Shriram Krishnamurthi for introducing me to research during my undergraduate years. I especially thank Alan Bivens, Calin Cascaval, Maged Michael, Stefanie Chiras, and Trey Cain for helping make my summer at IBM Research so productive and enjoyable that I decided to return after graduating.

I am also greatly indebted to many fellow students. Joe Devietti and Arun Raghavan have been my closest collaborators in graduate school, deeply contributing to my work on unbounded hardware transactional memory and conflict-robust transactional memory respectively. These collaborations have been the source of my greatest enjoyment in graduate school. I also thank the other members of ACG whose support and knowledge I have benefited from: Anne Bracy, Drew Hilton, Marc Corliss, Santosh Nagarakatte, Tingting Sha, and Vlad Petric.

I have been fortunate to have had wonderful friends throughout my time at Penn. Aaron Bohannon and I have had an incredible amount of fun exploring Philadelphia and discussing computer science, indie music, and everything in between over excellent gin and tonics. T.J. Green and his wife Elisabeth have been wonderful friends to both Angelina and myself. Dimitris Vytiniotis and Nate Foster have been close and supportive friends from my first days in graduate school. I have also enjoyed spending time with Jeff Vaughan, Jenn Wortman Vaughan, Matt Jacobs, Micah Sherr, Nick Taylor, Pavol Cerny, and Peng Li. Thank you all.

Throughout this time, I have also relied on and benefited from the support of friends made before graduate school. Ben Finkel has influenced my life more deeply than anyone else outside of my family; I cannot thank him enough. My friendships with Andrew McClain, Nick and Lela Beem, Sarovar Banka, Stephan Marguet, and Thea Brennan-Krohn mean more to me than I can

express. Diana Gross has been a close and supportive friend through many ups and downs over the last fifteen years.

I am grateful to the administrative and technical support staff of the CIS Department. Mike Felker is the cornerstone without which the department would surely collapse; I deeply appreciate his competence, friendliness, caring, and hard work over the past seven years. Rita Powell has been a pleasure to interact with. My research would not have been possible without the support of Penn Engineering's Computing and Educational Technology Services (CETS). I particularly thank Dan Widyono, who has been consistently terrific throughout my entire time at Penn. The professionalism of the Moore Business Office has eased many tasks. I especially acknowledge Amy Deitz, Gail Shannon, Mark West, and Towanda Marner.

My graduate work has been financially supported by an IBM Graduate Fellowship, National Science Foundation Grant CCF-0644197, and donations from Intel Corporation.

ABSTRACT

MECHANISMS FOR UNBOUNDED, CONFLICT-ROBUST HARDWARE TRANSACTIONAL MEMORY

Colin Blundell

Supervisor: Milo M. K. Martin

Conventional lock implementations serialize access to critical sections guarded by the same lock, presenting programmers with a difficult tradeoff between granularity of synchronization and amount of parallelism realized. Recently, researchers have been investigating an emerging synchronization mechanism called transactional memory as an alternative to such conventional lock-based synchronization. Memory transactions have the semantics of executing in isolation from one another while in reality executing speculatively in parallel, aborting when necessary to maintain the appearance of isolation. This combination of coarse-grained isolation and optimistic parallelism has the potential to ease the tradeoff presented by lock-based programming.

This dissertation studies the hardware implementation of transactional memory, making three main contributions. First, we propose the permissions-only cache, a mechanism that efficiently increases the size of transactions that can be handled in the local cache hierarchy to optimize performance. Second, we propose OneTM, an unbounded hardware transactional memory system that serializes transactions that escape the local cache hierarchy. Finally, we propose RetCon, a novel mechanism for detecting conflicts that reduces conflicts by allowing transactions to commit with different values than those with which they executed as long as dataflow and control-flow constraints are maintained.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With shared-memory multiprocessing becoming the norm in contexts ranging from webservers to mobile devices, the task of developing high-performance parallel programs is being faced by more programmers than ever before. One key challenge in developing such programs is the need to synchronize accesses to shared memory made by different threads. Implementing synchronization that is both (1) correct and (2) not a performance bottleneck has historically been a challenging task. The focus of this dissertation is ameliorating the challenge of high-performance synchronization in shared-memory parallel programs.

Today's dominant synchronization mechanism is *locks*. Programmers associate locks with pieces of data and use locks to serialize access to their associated data. Locks present a well-known correctness/performance tradeoff: for ease of reasoning, it is desirable to associate locks with data at a *coarse granularity*, but to avoid serialization, it is typically necessary to associate locks with data at a *fine granularity*.

Partly in response to the challenges of programming with locks, Herlihy and Moss [47] proposed an alternative synchronization mechanism, *transactional memory*. *Memory transactions* are segments of code that have the semantics of executing serially with respect to each other. In reality, however, the system executes them speculatively in parallel, detecting cases where transactions access the same data in a conflicting way and rolling back to preserve the appearance of serial execution. This combination of an interface of serialization with an implementation of speculative parallelism has potential to ease the correctness/performance tension of locks.

Current multiprocessors can be extended to support transactional memory in hardware with high concurrency and low overheads as long as transactions are small (*e.g.*, fit in the L1 cache) and exhibit little-to-no data contention [21]. Unfortunately, ensuring that transactions have these properties is likely to be nearly as challenging as achieving high performance using locks for non-expert developers. To help increase the utility of transactional memory as a general-purpose synchronization primitive, this dissertation has two goals. First, we seek to support unbounded transactions in hardware with high performance and low complexity. Second, we seek to increase the performance robustness of hardware transactional memory (HTM) to data conflicts, which we find to be the primary limitation to high performance. In particular, we seek to eliminate the performance impact of a commonly-occurring pattern of conflicts on auxiliary data, *i.e.* conflicts on data that is peripheral to a transaction's main computation. These conflicts can significantly degrade performance by inducing serialization into otherwise-parallel operations.

In the next section we outline the challenge of synchronization in shared-memory parallel programs, including the difficulties of programming with locks. Section 1.2 describes the promise of transactional memory and the challenges that this dissertation addresses. Section 1.3 presents our proposals for supporting unbounded transactions in hardware via the *permissions-only cache* and OneTM. Section 1.4 describes RetCon, our mechanism that increases the robustness of transactional memory to conflicts on auxiliary data. Finally, we detail the main contributions of the dissertation (Section 1.5), present the dissertation's structure (Section 1.6), and outline the differences between this dissertation and previously published versions of this work [9, 11, 12, 13] (Section 1.7).

## 1.1 The Problem of Synchronization in Shared-Memory Parallel Programs

A shared-memory parallel program is one in which multiple threads of execution operate concurrently in a single shared address space such as that provided by a shared-memory multiprocessor with the goal of accelerating performance over a sequential implementation. By default, the operations of threads are allowed to be interleaved at the granularity of individual memory accesses. However, program semantics often require that a given set of memory accesses by one thread be performed serially with respect to other threads (for example, if a thread in a banking application

that moves money from one account to another, a different thread should not be able to observe the intermediate state where the money is in neither bank account). Enforcing such serialization is the role of a synchronization primitive.

For programs that access shared data in a regular fashion (*e.g.*, a scientific workload where execution is divided into phases of private computation and phases of merging results), it is often sufficient to be able to ensure that all threads have reached a certain point in the program before any thread proceeds past that point. Barriers are a mechanism that enforce this property, having the semantics that all threads must reach the barrier before any thread can proceed past the barrier.

For programs such as the banking application described above that access shared data in an *irregular fashion*, however, barriers alone are insufficient. The current dominant synchronization primitive for handling such irregular synchronization is locks. By convention, a lock is associated with a piece of data. To synchronize accesses on this data, a thread acquires the lock associated with that data before accessing the data and releases the lock only when it is finished accessing the data. In this fashion, the thread can ensure that no other thread can access the data during this time.

Locks present several programming and performance challenges. First and foremost, locks present a difficult performance/correctness tradeoff: to avoid serialization it is desirable to associate data with locks at a fine granularity, but such fine-grained locking complicates reasoning about the program and increases the likelihood of bugs. Locks also make it difficult to separate interface from implementation, as to synchronize a given object the programmer must often be aware of the internal locking used in the object's implementation. Finally, programs using locks can deadlock if two threads acquire locks in an inconsistent order. We provide more background on shared-memory synchronization and the challenges of locks in Section 2.1.

## 1.2 Transactional Memory: Promise and Challenges

Transactional memory [47, 57] (TM) has been proposed as an alternate synchronization primitive to locks. Memory transactions are segments of code that have the semantics of executing serially with respect to each other. In reality, the system speculatively executes transactions in parallel, rolling back when two transactions conflict to preserve transactional semantics.

Transactional memory has potential to ease the challenges of lock-based programming. By combining an interface of isolation with an implementation of parallelism, transactions can ease the

performance/correctness tension of locks. Placing code within a transaction also ensures that that code will execute in isolation regardless of how the objects being accessed are internally synchronized. Because transactions can roll back, the problem of lock ordering deadlock is eliminated.

The initial transactional memory design [47] implements transactional memory in hardware via extensions to existing on-chip structures, utilizing a multiprocessor's cache coherence protocol to detect conflicts between transactions and its on-chip memory hierarchy to buffer speculative state. This *hardware transactional memory* (HTM) design has low performance overheads and is highly concurrent in the absence of data conflicts. However, the intial HTM proposal was restricted to transactions that are *bounded in size* (do not overflow the on-chip structures) and *bounded in time* (do not execute for longer than a scheduling quantum). For many common programming idioms (*e.g.*, using transactions to synchronize tree traversals in a library data structure), bounded transactions are insufficient. **Extending hardware transactional memory to provide support for unbounded transactions is the first goal of this dissertation.**

Supporting transactions of unbounded size is not the only challenge facing transactional memory, however. Our study of naively-written transactional workloads reveals that once size is eliminated as a constraint, conflicts form the dominant remaining performance bottleneck. In particular, we find a common pattern of *conflicts on auxiliary data*. Auxiliary data is simply data that is peripheral to a transaction's main computation, such as reference counts of shared objects, occupancy fields of hashtables, or simple performance counters. Conflicts on such data can cause significant performance loss. These conflicts are especially damaging because they induce serialization of operations that are *conceptually non-conflicting*, *e.g.*, simultaneous reads of a reference-counted shared object. **Increasing the robustness of hardware transactional memory to auxiliary data conflicts is the second goal of this dissertation.**

In the next two sections we outline our proposals for supporting unbounded transactions in hardware and increasing the robustness of transactional memory to auxiliary data conflicts. In Chapter 2 we provide a more thorough overview of transactional memory, including the potential of transactional memory (Section 2.1), transactional memory semantics (Section 2.2), high-level implementation approaches (Section 2.3 and Section 2.4), and bounded hardware transactional memory (Section 2.6).

## 1.3 The Permissions-Only Cache and ONETM

The primary challenge in designing unbounded hardware transactional memory is that because transactional workloads largely do not yet exist, it is unknown how large transactions will become. If overflows of the bounded HTM are rare, it would potentially be sufficient to provide a simple, low-performance mechanism for handling them. However, if overflows are frequent, such a mechanism would cause overall performance degradation. Unfortunately, supporting unbounded transactions with the same properties of high concurrency and low overheads provided by the bounded HTM is a challenging task (Chapter 4).

Instead, we propose a decoupled approach to the problem of supporting unbounded transactions in hardware. Our first objective is to ensure that overflows of the bounded HTM are **rare**. To do so, we introduce the *permissions-only cache*, a mechanism that efficiently extends the range of the bounded HTM from kilobytes to megabytes (Chapter 5). Our second objective is to handle overflows **simply**. To do so, we introduce ONETM, a mechanism that supports overflowed transactions via selective serialization (Chapter 6). In ONETM only one overflowed transaction is allowed to execute at a time on a per-application basis, eliminating prior proposals' need to detect conflicts between an unbounded number of unbounded transactions.

The permissions-only cache seeks to reduce the rate at which transactions overflow the bounded hardware transactional memory. To do so, it exploits the observation that to detect conflicts for a given block, the bounded HTM does not need to have the data for the block but rather only needs coherence permissions to the block and the knowledge of whether the block has been read and/or written by the transaction. The permissions-only cache thus maintains only coherence permissions for transactionally-accessed blocks. This size reduction allows it to achieve a 256:1 compression ratio; *e.g.*, a 4-KB permissions-only cache can track up to a megabyte of transactionally-accessed data.

With the knowledge that the permissions-only cache will likely make overflows rare we propose ONETM, a hardware-based approach for handling overflows by bounding concurrency among overflowed transactions. We explore two implementations. ONETM-Serialized stalls all other threads in an application when one transactions overflows. ONETM-Concurrent, by contrast, provides more concurrency than ONETM-Serialized by allowing bounded transactions and non-transactional code to execute concurrently with a single overflowed transaction. Both ONETM-Serialized and

ONETM-Concurrent avoid the complex structures required by prior proposals to track an unbounded amount of state per memory block.

## 1.4 RETCON

As described above, we find that once transaction size is eliminated as a constraint, conflicts form the primary limitation to performance on the workloads that we study. Moreover, we find a common pattern of conflicts on *auxiliary data*, *i.e.*, data that is peripheral to a transaction's main computation such as reference counts or hashtable occupancy fields. In the final part of this dissertation we aim to provide hardware support for minimizing the performance impact of auxiliary data conflicts.

To eliminate the performance impact of conflicts on auxiliary data, we exploit the facts that (1) transactions' control-flow and dataflow is generally insensitive to the exact values of auxiliary data and (2) the computation performed on auxiliary data is usually simple. We propose RETCON[1], a hardware mechanism that tracks the relationship between input and output values symbolically and uses this symbolic information to transparently repair the output state of a transaction at commit (Chapter 8). Conditionals form constraints on the acceptable range of values that an input can take when reacquired at commit. At commit time, all inputs that have been lost are reacquired, constraints are checked, and outputs are recomputed.

We tailor RETCON to fit the needs of the auxiliary data present in the workloads that we evaluate. RETCON tracks an input symbolically through a sequence of loads, simple arithmetic operations, branches, and stores, with more complex computation creating a constraint that the input value be the same at commit. To track symbolic information, RETCON adds a buffer to hold the initial values of symbolically-tracked blocks, a buffer to hold constraints, and a buffer to hold symbolically-tracked stores.

## 1.5 Contributions of this Dissertation

In our view, the most important contributions of this dissertation are as follows:

[1] Retcon, short for *retroactive continuity*, refers to soap operas' and comic books' practice of revising past events as necessary to match current reality.

- **Proposes a mechanism that extends the range of bounded hardware transactional memory.** The permissions-only cache exploits the fact that the information necessary for performing conflict detection can be encoded in the coherence permissions of transactionally-accessed cache blocks; the data is not necessary. By maintaining only coherence permissions for transactionally-accessed blocks, the permissions-only cache efficiently extends the range of bounded hardware transactional memory from kilobytes to megabytes.

- **Proposes a mechanism for supporting unbounded transactions in hardware with the goal of low design complexity.** ONETM bounds concurrency among unbounded transactions as a means of simplifying the implementation of the uncommon case and reducing the overhead that unbounded transactions impose on the rest of the system. By bounding concurrency among unbounded transactions, ONETM eliminates prior proposals' requirements of performing conflict detection between an unbounded number of unbounded transactions. This implementation works synergistically with the permissions-only cache to create a system in which the overall performance impact of serialization on overflow is low.

- **Develops an approach to the problem of conflicts on auxiliary data that allows transactions to resolve such conflicts without rollbacks.** We propose a repair-based approach to eliminating the performance impact of conflicts on auxiliary data. RETCON symbolically tracks the relationship between inputs to a transaction and outputs produced by that transaction, using this symbolic information to recover from conflicts without rollback before committing a transaction.

- **Quantitatively evaluates the above proposals.** We evaluate our proposed mechanisms using full-system simulation. This evaluation indicates that on a set of workloads (1) the combination of the permissions-only cache and ONETM provides the performance of an idealized, fully-concurrent unbounded hardware transactional memory, and (2) RETCON significantly increases the performance of workloads exhibiting conflicts on auxiliary data.

## 1.6 Dissertation Structure

We organize this dissertation into three parts: first, background on transactional memory and the challenges that we seek to address; second, our proposals for supporting unbounded transactions in

hardware; and third, our proposal for increasing the robustness of transactional memory to auxiliary data conflicts. We describe the structure of each part of the dissertation below.

- **Background and related work (Chapters 2, 3, and 4).** We first give an overview of transactional memory in Chapter 2. We describe our experimental infrastructure and characterize the transactional behavior of the workloads that we use in Chapter 3, illustrating the impact of auxiliary data conflicts on these workloads and examining the sizes of the transactions occurring in the workloads. We present an overview of previous proposals for unbounded hardware transactional memory in Chapter 4, concluding this chapter with a discussion of the challenges inherent in supporting an unbounded number of concurrently-executing unbounded transactions.

- **Supporting unbounded transactions in hardware (Chapters 5, 6, and 7).** We propose the permissions-only cache as a mechanism for reducing overflows of bounded hardware transactional memory in Chapter 5. Chapter 6 presents ONETM, our proposal for unbounded hardware transactional memory that limits the number of unbounded transactions that can be executing at a time to one. We experimentally evaluate our proposals for unbounded HTM in Chapter 7, finding that the combination of the permissions-only cache and ONETM can provide the performance of an idealized, fully-concurrent unbounded hardware transactional memory on our workloads.

- **Mitigating the performance impact of auxiliary data conflicts (Chapters 8 and 9).** Chapter 8 describes RETCON, our proposal to mitigate the performance impact of conflicts on auxiliary data. Chapter 9 experimentally evaluates the performance impact of RETCON on our workloads.

Finally, we conclude the dissertation by summarizing our proposals and presenting opinions on future opportunities and challenges in transactional memory in Chapter 10.

## 1.7 Differences from Previously Published Versions of this Work

This dissertation builds on material previously published by Blundell *et al.* [9, 13]. In addition, Figure 2.13 on page 37 and the text describing the implementation of the flash-clear and flash-invalidate operations in Section 2.6 are taken from Blundell *et al.* [12].

The presentation of the permissions-only cache and ONETM extends earlier work [9] by discussing the option of cleaning as a version management mechanism in addition to a log, discussing both a read-only permissions-only cache and a read-write permissions-only cache, discussing the impact of weak atomicity on the design of ONETM, and presenting pseudocode-based algorithms of our proposals. In addition, the quantitative evaluation of these proposals in this dissertation is significantly more thorough than in that earlier work, including evaluating a broader set of workloads, evaluating the impact of weak atomicity on ONETM performance, evaluating the impact of lazy clearing on ONETM-Concurrent performance, evaluating the impact of a read-only as well as a read-write permissions-only cache, evaluating the impact of the sector cache organization of the permissions-only cache, and evaluating a permissions-only cache of various sizes.

The presentation of RETCON extends earlier work [13] by presenting code examples of conflicts that RETCON can and cannot repair, evaluating the impact of imprecise constraint representation on RETCON performance, presenting evaluation data for RETCON configured to reacquire blocks serially at transaction commit, presenting evaluation data for RETCON configured with several different structure sizes, and analyzing the sensitivity of RETCON to predictor configuration.

# Chapter 2

# Overview of Transactional Memory

This chapter gives an overview of the basic interface and implementation space of transactional memory and presents the bounded hardware transactional memory that this dissertation employs as a foundation. The intent of this chapter is to give a framework, background, and terminology for the rest of this dissertation, not provide a complete tutorial on transactional memory. As dozens of papers on transactional memory have been published in the last several years, we refer the reader to the book by Larus and Rajwar [57] for a general introduction to transactional memory.

We first outline our problem context of synchronization in shared-memory parallel programs. We then detail the transactional memory interface in Section 2.2. In Section 2.3, we outline the basic implementation tasks required to execute memory transactions speculatively in parallel. We present three high-level algorithms for transactional memory systems and describe the challenges in implementing these algorithms entirely in software with low overheads (Section 2.4). The remainder of the chapter provides an overview of bounded hardware transactional memory. We first review multiprocessor memory systems in Section 2.5 before describing how to layer support for executing bounded transactions speculatively in parallel on top of such a memory system in Section 2.6. Finally, we present performance and semantic challenges of this bounded hardware transactional memory in Section 2.7 and close the chapter with a brief summary.

```
        int balance = 42;

proc1(){          proc2(){
  r1 = balance;     r2 = balance;
  r1 += 12;         r2 -= 10;
  balance = r1;     balance = r2;
}                 }

        P = proc1() || proc2()
```

Figure 2.1: **A program that requires synchronization for correct behavior.** `balance` is a shared variable, and `r1` and `r2` are registers. In order to ensure that the updates of both `proc1` and `proc2` are reflected in the final value of `balance`, the programmer must ensure that `proc1` executes entirely before `proc2` or vice versa. In the above example `proc1` and `proc2` may both perform their reads of `balance` before either performs its update, resulting in the final value reflecting only one of the updates (the one that occurs second).

## 2.1 Synchronization in Shared-Memory Parallel Programs

This dissertation considers *shared-memory parallel programs*, *i.e.*, programs in which (a) multiple threads of execution are created and (b) these threads communicate with each other via reads and writes in a single shared memory space. As shared-memory multiprocessors (described in Section 2.5) are appearing in a broader range of computers than ever before, the task of creating shared-memory parallel programs to run on these multiprocessors is likewise becoming more common.

By default, the system interleaves different threads' memory accesses at the granularity of individual reads and writes (as described in Section 2.5). In some cases, however, a thread must make a series of accesses to memory in isolation from other processors in order to guarantee correctness. Figure 2.1 on page 11 illustrates a program in which certain interleavings of memory accesses will result in incorrect behavior and must therefore be disallowed. Enabling isolation at a granularity coarser than a single memory access is the role of a *synchronization primitive*.

Below we first describe the current dominant synchronization primitive of locks. After outlining several challenges with using locks, we then present transactional memory and outline its potential to ease these challenges.

```
int balance = 42;
Lock lock;

proc1(){              proc2(){
  acquire(lock);        acquire(lock);
  r1 = balance;         r2 = balance;
  r1 += 12;             r2 -= 10;
  release(lock);        release(lock);
}                     }

P = proc1() || proc2()
```

Figure 2.2: **Synchronization via locking.** In order to ensure that the updates of both `proc1` and `proc2` are reflected in the final value of `balance`, the programmer uses a lock. Each thread acquires the lock before doing its computation, releasing the lock only when its computation is complete. As the semantics of the lock dictate that only thread can acquire it at a time, the computations performed by the two threads are executed sequentially.

### 2.1.1 Synchronization via Locks

A lock is an object that only one thread can hold at a time. A lock is typically associated with a piece (or several pieces) of data. By following the convention that a thread always *acquires* the lock associated with given data before manipulating that data and *releases* the lock only when finished manipulating the data, the programmer can ensure that different threads' accesses to that data are serialized at the desired granularity. The code between a lock acquire and its matching lock release is called a *critical section*. Figure 2.2 on page 12 shows how the programmer can prevent the undesired interleavings of Figure 2.1 on page 11 by employing a lock.

Developing parallel programs that are both correct and high-performance using locks is a challenging task. The primary reasons are that (1) locks synchronize *conservatively* and (2) locks are *associated with data*. We outline the reasons that these properties make using locks challenging below.

**The granularity problem.** First, the fact that locks synchronize conservatively introduces a difficult performance/correctness tension. Associating locks with data at a coarse granularity (*coarse-grained locking*) eases reasoning about program correctness by reducing the number of possible interleavings between threads. It also, however, can induce unnecessary serialization: all critical sections guarded by the same coarse-grained lock are forced to execute sequentially. Conversely,

```
Hashtable::insert(k,v){
  acquire(lock);
  Elem e = Elem(k,v);
  index = hash(k);
  bucket = buckets[index];
  bucket->insert(e);
  release(lock);
}
```

(a)

```
Hashtable::insert(k,v){
  Elem e = Elem(k,v);
  index = hash(k);
  bucket = buckets[index];
  acquire(bucket->lock);
  bucket->insert(e);
  release(bucket->lock);
}
```

(b)

Figure 2.3: **Two ways to synchronize a hashtable with locks.** In (a), a single lock guards the entire hashtable. This option enables easy correctness reasoning and facilitates the synchronization of operations on the table as a whole (*e.g.*, resize). However, it results in all inserts being serialized. In (b), a lock guards each bucket. This option enables inserts to different buckets to proceed in parallel. However, it makes correctness reasoning more difficult and makes operations on the table difficult to synchronize.

associating locks with data at a fine granularity (*fine-grained locking*) can increase performance at the cost of making programs more bug-prone, difficult to reason about, and difficult to maintain. Moreover, performing locking at *too* fine a granularity can also reduce performance by inducing overheads related to the lock acquires and releases themselves.

Figure 2.3 on page 13 provides an illustration of this granularity problem. The programmer desires to create a function that inserts an element into a hashtable in a thread-safe manner. The most natural way to implement this functionality is to associate a lock with the hashtable that is acquired before any operations are performed on the hashtable (part (a) of Figure 2.3 on page 13). Unfortunately, this synchronization policy serializes all inserts to the hashtable, even if they are inserts of distinct buckets.

An alternative way to accomplish this synchronization is to associate a lock with each bucket (part (b) of Figure 2.3 on page 13). This synchronization policy enables inserts into distinct buckets to proceed in parallel. Unfortunately, it also complicates the tasks of reasoning about the correctness of the program and maintaining the program. Furthermore, it makes operations on the entire hashtable (such as resizing the hashtable) difficult to synchronize.

**Lack of encapsulation.** A second problem with locks is lack of encapsulation, which arises due to the association of locks with data. If the programmer wishes to create a critical section protect-

ing access to multiple pieces of data guarded by different locks, she must be able to acquire the locks that guard each piece of data. Achieving this property requires exposing an object's internal synchronization to the outside world.

Figure 2.4 on page 15 provides an illustration of this problem. The programmer wishes to create a function that atomically moves an element from one hashtable to another. To implement this functionality, the programmer must be able to ensure that no other thread can access *either hashtable* for the duration of the computation. Providing a thread-safe implementation of `Hashtable::insert` is insufficient for this purpose. Instead, the programmer must be able to lock both hashtables simultaneously. To support this behavior, the hashtables must expose their internal locks, a textbook violation of separation between interface and implementation.

**Deadlock.** Finally, another problem with locks is the possibility of deadlock due to the need to acquire locks in a specific order. Consider again Figure 2.4 on page 15. The programmer has naively acquired the lock of `h1` first, followed by the lock of `h2`. Consider the case where one thread calls the function with pointers to two hashtables while another thread simultaneously calls the function with pointers to the same two hashtables in reverse order. In this case, deadlock can ensue as the first thread acquires the lock of the first hashtable, the second thread concurrently acquires the lock of the second hashtable, and each then waits (forever) for the other to release its lock. To solve this problem, the programmer has to impose an ordering on lock acquires (*e.g.*, sort lock acquire by ascending address). This order then has to be followed globally throughout the program.

### 2.1.2 Synchronization via Transactional Memory

Herlihy and Moss [47] introduced a novel synchronization primitive called *transactional memory* (TM). Memory transactions are segments of code with the semantics of executing atomically and in isolation from one another, while in reality the system executes them speculatively in parallel and rolls back when two transactions simultaneously execute the same piece of data. In the next section, we will give a thorough overview of the semantics of transactional memory. Here, we outline how TM has potential to ease the challenges of lock-based programming.

Figure 2.5 on page 15 shows an implementation of `Hashtable::insert` synchronized via transactions. Because transactions have the semantics of executing in isolation from each other,

```
move_elem(int key, Hashtable h1, Hashtable h2){
  acquire(h1->lock);
  acquire(h2->lock);
  value = h1->lookup(key);
  h1->remove(key);
  h2->insert(key, value);
  release(h1->lock);
  release(h2->lock);
}
```

Figure 2.4: **Moving an element from one hashtable to another using locking.** The programmer intends the removal from `h1` and insertion into `h2` to occur atomically with respect to other threads. It is thus not sufficient that `Hashtable::insert` be internally synchronized; the programmer must be able to ensure that other threads cannot access `h1` or `h2` for the duration of the computation. If `Hashtable` uses a single lock internally, this synchronization may be accomplished by exposing that lock (as above). This breaks encapsulation. In addition, the naive code example above can suffer from deadlock. To avoid this possibility, an ordering on lock acquire would have to be imposed. Finally, note that it is not immediately obvious how to implement this functionality if the hashtable uses per-bucket locks (as in part (b) of Figure 2.3 on page 13).

```
Hashtable::insert(k,v){
  transaction{
    Elem e = Elem(k,v);
    index = hash(k);
    bucket = buckets[index];
    bucket->insert(e);
  }
}
```

Figure 2.5: **Synchronizing a hashtable using transactions.** By wrapping the hashtable insert in a transaction, the programmer can reason about inserts made by different threads as if they were occurring sequentially. However, the system actually executes these inserts in parallel as long as the inserts are being made into distinct buckets. Synchronization via transactions thus achieves the ease of reasoning of coarse-grained locking (part (a) of Figure 2.3 on page 13) together with the high performance of fine-grained locking (part (b) of Figure 2.3 on page 13).

```
move_elem(int key, Hashtable h1, Hashtable h2){
  transaction{
    value = h1->lookup(key);
    h1->remove(key);
    h2->insert(key, value);
  }
}
```

Figure 2.6: **Moving an element from one hashtable to another using transactions.** As in Figure 2.4 on page 15, the programmer intends the removal from `h1` and insertion into `h2` to occur atomically with respect to other threads. By making the computation into a transaction, the programmer can guarantee this property irrespective of how (or even whether) `Hashtable` is internally synchronized. Moreover, there is no possibility of deadlock. Finally, concurrent calls to `move_elem` by different threads can execute in parallel as long as the elements being moved are distinct, whereas Figure 2.4 on page 15 requires the *hashtables* being manipulated by the different threads to be distinct in order to avoid serialization.

the programmer can reason about this code as if inserts from different threads are being performed sequentially — similar to the coarse-grained locking presented in part (a) of Figure 2.3 on page 13. However, because the system speculatively executes transactions in parallel, these inserts will actually proceed concurrently as long as they are being performed into distinct buckets — similar to the fine-grained locking presented in part (b) of Figure 2.3 on page 13 (note that if two simultaneous inserts access the same bucket, a conflict will be detected and one insert rolled back). Transactions thus have the potential to achieve the ease of reasoning of coarse-grained locks together with the performance of fine-grained locks.

Figure 2.6 on page 16 shows how a programmer can implement the `move_elem` function from Figure 2.4 on page 15 using transactions. Because the transaction wrapping the computation is guaranteed to occur in isolation, the programmer can achieve the desired semantics regardless of how (or even *whether*) the hashtables are internally synchronized. Locks' potential for deadlock by acquiring the locks of distinct pieces of data in inconsistent orders is also removed — if two transactions conflict, one is simply rolled back.

The remainder of this chapter provides an overview of the semantics and implementation of transactional memory.

## 2.2 Transactional Memory Semantics

In this section we first outline a basic transactional semantics that is common to nearly every transactional memory proposal and will be assumed throughout this dissertation. We then detail several advanced semantic issues that will be relevant in later parts of this dissertation.

### 2.2.1 Basic Semantics

A *memory transaction* (or more simply a *transaction*) is a segment of code that is guaranteed to execute atomically and in isolation from all other threads. This semantics (and name choice) is inspired by database transactions' guarantee of atomicity, consistency, isolation, and durability (the ACID properties) [37]. Transactional memory, however, does not guarantee the more heavyweight properties of durability (resilience against system failure) or consistency (validity against a user-specified schema). Below, we discuss the properties of atomicity and isolation in more detail as well as describe the basic transactional nesting model.

**Atomicity.** Atomicity means that either none or all of a transaction's updates occur. For example, in Figure 2.6 on page 16, if the removal of the element from `h1` occurs, then the insert of the element into `h2` is guaranteed to also occur.

**Isolation.** Isolation means that no transaction observes the intermediate state of another transaction. A transaction's updates are not visible to other transactions until it *commits*, at which time all updates instantaneously become visible. For example, in Figure 2.6 on page 16, the removal of the element from `h1` and insertion into `h2` are not made visible to other threads until the commit of the transaction, at which time they are simultaneously made visible. To be legal, an execution of a parallel composition of transactions must be *serializable*: the outcome of the execution must be equivalent to one in which the transactions are executed sequentially.

**Nesting model.** Transactions may be nested inside other transactions. The nesting model that we assume throughout this dissertation is one of *subsumption*: nested transactions release isolation only when the outermost containing transaction commits. More advanced models are possible [75, 78, 80], but not discussed further in this dissertation[1].

[1] Extending our proposals to support such more advanced nesting models is open research.

The property of serializability means that the programmer can reason about transactions from different threads as if they were always executed in some sequential order. Importantly, however, serializability does not imply that transactions must *actually* be executed sequentially. In the following sections, we will discuss how transactions may be executed speculatively in parallel while preserving these properties. First, however, we discuss more advanced semantic issues of transactional memory.

### 2.2.2 Advanced Semantic Issues

Here, we discuss more advanced semantic issues that are not addressed by the basic semantics presented above. These issues include the interaction of transactions and non-transactional code, whether an abort operation is part of the transactional interface, support for non-abortable actions, and starvation avoidance.

**Interaction between transactions and non-transactional code.** Transactions must be isolated with respect to each other, but their relationship to non-transactional code is not uniformly defined. *Strong atomicity* is defined as a semantics in which transactions are isolated from non-transactional accesses, while *weak atomicity* is defined as a semantics in which transactions are not guaranteed to be isolated from non-transactional accesses [10].

By default, this dissertation assumes strong atomicity as part of the transactional semantics. However, we will also discuss situations in which supporting only weak atomicity can increase performance and/or reduce complexity in our proposals.

**Explicit abort.** As we will describe below, transactional memory systems commonly execute transactions speculatively in parallel, rolling back in cases where serializability would be violated. This speculative execution may optionally be exposed to the programmer by providing an explicit abort operation as part of the transactional interface. Adding support for such an operation implies that the system must always be able to roll back a transaction. Our basic semantics does not include an explicit abort operation, and our proposals do not by default support it; where relevant, we will discuss the extensions to our proposals that would be necessary to support it.

**Supporting non-abortable actions.** Some effects can not easily be rolled back (*e.g.*, IO such as a network send). How to incorporate non-abortable actions into a transactional model is not immediately obvious. This issue is especially challenging if the transactional memory interface

includes an explicit abort operation. This dissertation proposes a mechanism for support for non-abortable actions within transactions in Section 6.3.

**Starvation avoidance.** A final issue not discussed above is whether the system provides a guarantee against transaction starvation, *i.e.*, whether the possibility that a given transaction will be continually aborted by others is disallowed. Not all transactional memory systems provide such a guarantee. However, we assume starvation avoidance as part of our transactional memory semantics as we consider it important for programmability. We discuss a basic mechanism for providing this guarantee in Section 2.3.

## 2.3 Transactional Memory Implementation Tasks and Terminology

As mentioned above, the goal of a transactional memory system is to execute transactions in parallel while preserving transactions' semantic properties. The basic framework for accomplishing this goal is to execute transactions *speculatively* in parallel, detect *conflicts* that would violate serializability, and *resolve* these conflicts by stalling or rolling back transactions to preserve serializability. As a result, transactional memory systems have three main tasks: conflict detection, conflict resolution, and version management (*i.e.*, supporting the ability to roll back to a pre-speculative state). Below we discuss ideas and terminology associated with accomplishing these tasks. In the next section we will present several high-level transactional memory algorithms that systems can seek to implement.

### 2.3.1 Conflict Detection

Because transactions are executing in parallel, they may access the same data simultaneously. If at least one of the accesses is a write, a potential violation of isolation exists: a transaction's update is potentially being observed by other threads before it commits. The role of conflict detection is to detect such potential violations (referred to as *conflicts*).

The basic way to detect conflicts is to monitor the addresses read and written by transactions, detecting cases where transactions simultaneously access the same address. We refer to this method of performing conflict detection as *address-based conflict detection*. The set of addresses that a transaction reads is called its *read set*, and the set of addresses that a transaction writes is called its *write set*. The task of address-based conflict detection, therefore, is to detect cases where one trans-

action's write set intersects with the read or write set of a simultaneously-executing transaction. A system may maintain the addresses in read and write sets at a precise byte granularity or at a coarser granularity (*e.g.*, hardware transactional memory systems generally perform conflict detection at a memory block granularity, as discussed in Section 2.6).

Address-based conflict detection comes in two basic forms: eager and lazy [77]. In eager conflict detection, the system checks each memory access made by a transaction for conflicts with all other concurrently-executing transactions. In lazy conflict detection, by contrast, the system performs conflict detection for a given transaction at the time that that transaction seeks to commit. In either case, once a conflict is detected, the system must resolve the conflict in a way that preserves serializability. We discuss a variety of ways in which this task can be accomplished in the next subsection.

Prior work has noted that eager and lazy conflict detection have different performance characteristics [15]. By detecting conflicts as soon as possible, eager conflict detection can reduce wasted work. Conversely, lazy conflict detection may avoid the need to resolve some conflicts altogther. Consider an execution where one transaction reads address $A$ and a second transaction writes $A$, with the reader seeking to commit first. Eager conflict detection would detect the conflict and have to resolve it. Under lazy conflict detection, by contrast, both the reader and the writer can commit. We will examine the performance impact of these differences in Section 3.3.

To achieve the best performance characteristics of both eager and lazy conflict detection, recent work has proposed mixed eager/lazy policies [98, 112]. As discussed in Section 8.4, RETCON will also realize the benefits of a mixed eager/lazy conflict detection policy (in addition to other benefits).

We finally note that a different mechanism of detecting conflicts is to check whether any values read by a transaction have been changed by another transaction [82, 109]. We refer to this mechanism for conflict detection as *value-based conflict detection*. For much of this dissertation we will strictly be concerned with address-based conflict detection. However, in Section 3.3 we will examine the benefits of incorporating value-based conflict detection. As discussed in Section 8.4, RETCON realizes these benefits as well.

### 2.3.2 Conflict Resolution

Once a conflict has been detected, the system needs to *resolve* the conflict in order to ensure that serializability is preserved. The most basic way to resolve a conflict is to roll back one of the transactions involved in the conflict (implemented as described below). This transaction can either be the transaction making the conflicting access or the transaction with whom the access conflicts. These basic policies, however, do not guarantee starvation avoidance.

A more sophisticated way to resolve conflicts is to use the age of the transactions involved [56, 89]. In *age-based conflict resolution*, each transaction is assigned a *timestamp* when it first begins. A transaction retains its timestamp until it commits (*i.e.*, if it aborts and restarts, it retains the same timestamp). When a conflict occurs, these timestamps are used to resolve it as follows. If the requestee is older than the requester, the requester is stalled until the requestee releases isolation on the conflicting address (*i.e.*, commits or aborts). Otherwise, the requestee aborts. In other words, a transaction wins conflicts with younger transactions and loses conflicts with older transactions. This policy provides a starvation avoidance guarantee: once a transaction becomes the oldest in the system, it is guaranteed not to be aborted by any other transaction [89].

Other conflict resolution policies have been explored in the literature. For example, Moore et al. [77] propose a policy in which the requestee stalls the requester unless it (the requestee) is already being stalled. We refer the reader to Larus and Rajwar [57] for other proposals. As prior work [15, 98] has shown that age-based conflict resolution generally has robust performance, we employ this policy throughout this dissertation.

We finally note that a different mechanism for resolving conflicts that is often employed by systems using lazy conflict detection is to employ a *commit token*. In systems employing a commit token, each transaction arbitrates for permission to commit by seeking to acquire the commit token. Once a transaction acquires the commit token, the system performs conflict detection for the transaction by checking whether any of the addresses in the transaction's write set have been read by other, as-yet uncommitted transactions. If so, the conflict is resolved by rolling back the uncommitted transaction — as the other transaction has in effect already committed, there are no other options. To avoid starvation in such a system, a transaction that is continually aborted may obtain the commit token early and hold it until it commits [41].

### 2.3.3 Version Management

Version management is the task of enabling a transaction to roll back its state. This task involves undoing any updates that the transaction has made to restore all updated blocks to their pretransactional state.

Like conflict detection, version management can be eager or lazy [77]. In eager version management, transactional writes are performed in-place. Before a transaction writes a location for the first time, it logs the pretransactional value of the location. At transaction commit, written memory locations already contain their correct values. To roll back, the transaction restores this pretransactional value.

In lazy version management, transactions perform writes into a private buffer. Transactional reads must check this buffer for forwarding before reading from memory. To commit, the transaction performs all its writes from the buffer into memory. Rollback can be accomplished simply by throwing away the buffer.

Again similar to conflict detection, eager and lazy version management have performance tradeoffs [77]. In eager version management, commits can be fast but rollback involves restoring prespeculative state. Conversely, lazy version management enables fast aborts at the expense of commits. Additionally, lazy version management introduces an indirection into transactional reads that is not present in eager version management.

## 2.4 Three High-Level Transactional Memory Algorithms

In this section we outline three high-level transactional memory algorithms: an algorithm employing eager conflict detection and eager version management, an eager conflict detection/lazy version management algorithm, and a lazy conflict detection/lazy version management algorithm. These algorithms are not the only possible transactional memory algorithms. However, many of the transactional memory systems that we will describe later in this dissertation implement one of these three basic algorithms. Our own proposals will largely seek to implement the eager conflict detection/eager version management algorithm, for reasons that we detail in Section 2.6.

**Eager Conflict Detection/Eager Version Management Transactional Memory Algorithm**

**begin**

```
in_transaction = true
timestamp = current time
read_set = {}
write_set = {}
old_values = {}
```

**read(A)**

```
foreach remote transaction T
  if A in T.write_set
    resolve_conflict(T)
if in_transaction
  read_set[A] = true
return Mem[A]
```

**write(A,v)**

```
foreach remote transaction T
  if (A in T.write_set) or
     (A in T.read_set)
    resolve_conflict(T)
if in_transaction
  if A not in write_set
    old_values[A] = Mem[A]
    write_set[A] = true
Mem[A] = v
```

**resolve_conflict(T)**

```
if (T.timestamp < timestamp)
  stall until T ends
else
  T.abort()
```

**abort**

```
foreach (A,v) in old_values
  Mem[A] = v
in_transaction = false
```

**commit**

```
in_transaction = false
```

Figure 2.7: **Algorithm for a transactional memory system employing eager conflict detection and eager version management.** The algorithm checks for conflicts on each memory access, utilizes age-based conflict resolution, and performs transactional writes in-place.

### 2.4.1 An Eager Conflict Detection/Eager Version Management Algorithm

Figure 2.7 on page 23 gives the pseudocode for a transactional memory algorithm implementing eager conflict detection and eager version management as well as age-based conflict resolution. Each transaction maintains its read set and write set, and additionally maintains a log of pretransactional values of written blocks (`old_values`). At transaction begin the transaction is assigned a timestamp for conflict resolution (if the transaction had previously been aborted, it would retain its previously-assigned timestamp). The read and write sets as well as the log of old values are set to be empty.

On transactional reads and writes, the algorithm checks the read and/or write sets of all other concurrently-executing transactions for conflicts. Assuming that no conflict is detected, the relevant address is added to the read or write set as appropriate. In the case of a write, the algorithm also

checks whether this is the first transactional write to the given location; if so, the pretransactional value of the location is added to the log of pretransactional values. The algorithm thus maintains the invariant that any address in the write set also has an entry in `old_values`.

If the algorithm instead detects a conflict, it initiates conflict resolution. Depending on the relative ages of the transactions involved in the conflict, the conflict will be resolved either by stalling the requester until the requestee releases isolation or by aborting the requestee.

To abort, a transaction restores the pretransactional value of each transactionally-written block. To commit, the transaction simply releases isolation by clearing its read and write sets.

### 2.4.2 An Eager Conflict Detection/Lazy Version Management Algorithm

Figure 2.8 on page 25 gives the pseudocode for an eager conflict detection/lazy version management transactional memory algorithm that employs age-based conflict resolution. This algorithm is similar to the eager conflict detection/eager version management algorithm described above. The difference is that instead of maintaining pretransactional values in a private buffer, this algorithm maintains *current* values in a private buffer (`curr_values`).

The change from eager version management to lazy version management affects several parts of the algorithm. First, a transactional write performs its write into the private buffer rather than into memory. As a consequence of this fact, transactional reads must check the buffer in order to obtain the correct value for locations that have been previously written by the transaction. Finally, the actions that the transaction takes on commit and abort are the inverse of the above algorithm. To commit, the transaction performs its writes from its private buffer into memory (stalling any other conflicting transactions during this process to maintain isolation). No special action has to be taken to restore pretransactional state on abort, as transactions do not modify memory until they commit.

### 2.4.3 A Lazy Conflict Detection/Lazy Version Management Algorithm

Figure 2.8 on page 25 gives the pseudocode for a lazy conflict detection/lazy version management transactional memory algorithm. The algorithm employs a commit token for conflict resolution as described in Section 2.3. This algorithm uses the buffer of current values (`curr_values`) in the same way as the above-described eager conflict detection/lazy version management algorithm.

**Eager Conflict Detection/Lazy Version Management Transactional Memory Algorithm**

**begin**

```
in_transaction = true
committing = false
timestamp = current time
read_set = {}
write_set = {}
curr_values = {}
```

**read(A)**

```
foreach remote transaction T
  if A in T.write_set
    resolve_conflict(T)
if in_transaction
  read_set[A] = true
  if A in curr_values
    return curr_values[A]
return Mem[A]
```

**write(A,v)**

```
foreach remote transaction T
  if (A in T.write_set) or
     (A in T.read_set)
    resolve_conflict(T)
if in_transaction
  write_set[A] = true
  curr_values[A] = v
else
  Mem[A] = v
```

**resolve_conflict(T)**

```
if (T.timestamp < timestamp) or
   (T.committing)
  stall until T ends
else
  T.abort
```

**abort**

```
in_transaction = false
```

**commit**

```
committing = true
foreach (A,v) in curr_values
  Mem[A] = v
in_transaction = false
```

Figure 2.8: **Algorithm for a transactional memory system employing eager conflict detection and lazy version management.** The algorithm checks for conflicts on each memory access, utilizes age-based conflict resolution, and performs transactional writes into a private buffer that is copied to memory at commit.

However, the choice of lazy rather than eager conflict detection and the utilization of the commit token result in significant differences.

Rather than detecting conflicts at each memory access, this algorithm performs conflict detection at commit. Note that in the pseudocode for reads and writes there is no longer any conflict detection. At commit, the transaction first arbitrates to receive a *commit token*, of which there is only one in the system[2]. Once a transaction has obtained the commit token, it performs its updates

[2]Prior work has proposed optimized implementations of the commit token, *e.g.*, implementations in which it is distributed [18, 20].

**Lazy Conflict Detection/Lazy Version Management Transactional Memory Algorithm**

**begin**

```
in_transaction = false
read_set = {}
write_set = {}
curr_values = {}
```

**read(A)**

```
if in_transaction
  read_set[A] = true
  if A in curr_values
    return curr_values[A]
return Mem[A]
```

**write(A,v)**

```
if in_transaction
  write_set[A] = true
  curr_values[A] = v
else
  Mem[A] = v
```

**abort**

```
in_transaction = false
```

**commit**

```
obtain commit token
foreach address A in write_set
  Mem[A] = curr_values[A]
  foreach remote transaction T
    if (A in T.read_set)
      T.abort()
release commit token
in_transaction = false
```

Figure 2.9: **Algorithm for a transactional memory system employing lazy conflict detection and lazy version management.** The algorithm performs transactional writes into a private buffer that is copied to memory at commit, at which time it checks for conflicts. It uses a commit token for conflict resolution.

into memory. During this process, it also checks for conflicts with all other concurrently-executing transactions, aborting any conflicting transactions that it finds.

### 2.4.4 Implementing These Algorithms

Implementing the above-described algorithms with high performance is challenging. In the case of eager conflict detection, the primary performance challenge is that conflict detection must be low-overhead since it is performed on every transactional memory access. In the case of lazy conflict detection, the primary performance challenge is engineering the system so that commit does not become a bottleneck.

One active body of research is devising software transactional memory (STM) implementations – *i.e.*, implementations of transactional memory that run on stock hardware (*e.g.*, [42, 43, 46, 68, 94, 97]). The predominant challenge faced by these implementations is that performing conflict detection in software can have high overheads [17].

The original transactional memory proposal observed that it is possible to support transactional memory in hardware with minor extensions to existing multiprocessors [47]. This design is highly concurrent and has low overheads, but bounds the size and duration of transactions. We present an overview of bounded hardware transactional memory in Section 2.6 after reviewing current multiprocessor memory systems in the next section. Extending this design to support transactions that are *unbounded* in size and duration is one of the major goals of this dissertation.

## 2.5 Review of Multiprocessor Memory Systems

The hardware context of this dissertation is *shared-memory multiprocessors*. A shared-memory multiprocessor is a computer with multiple processors and a single physical memory that is shared among the processors. This section reviews the memory systems of such multiprocessors, with an emphasis on the components that are relevant for implementing hardware transactional memory.

In a typical shared-memory multiprocessor, memory is divided into *blocks* of some fixed size. To exploit spatial locality, this fixed size is normally larger than a single word – *e.g.*, 64 bytes. The memory is distributed throughout the system. A *node* thus consists of a processor together with a slice of the physical memory as well as cache memory (described below). The node responsible for a given block is called the *home node* of the block. The nodes in a multiprocessor are connected via an *interconnection network* or *interconnect* along which they can send each other messages. To perform a memory access to a given block, a processor sends the request along the interconnect to the home node of the block, which responds with data and/or updates the value of the block as necessary. The unit that implements the necessary logic for a slice of physical memory (*e.g.*, responding to processors' requests) is called the *memory controller*.

To improve system performance, each processor typically additionally has one or more levels of private *cache*. A cache contains copies of recently-accessed memory blocks, allowing faster access to these blocks if the processor requests them again in the future. Below, we present a basic overview of caches followed by a discussion of cache coherence.

### 2.5.1 Caches

A cache is a structure that contains storage for a finite number of memory blocks. The cache is indexed by memory address. As the cache is smaller than memory, multiple memory blocks map to

the same entry in the cache, potentially causing ambiguity about which block is residing at a given entry. To resolve this problem, entries are tagged with the higher-order bits of the residing block's address. These bits are said to be the *tag* of the entry.

To reduce the performance impact of different memory blocks mapping to the same cache index, a cache may allow more than one block to reside at a given index. The number of blocks that reside at a given index is said to be the *associativity* of the cache. The entries residing at a given index are collectively called a *set*. When a given memory address is looked up for residency in the cache, each entry of the appropriate set must be checked for a match.

When a processor receives a data block from memory, it places the data into its cache. Subsequent requests to the same block can then be satisfied from this local cache rather than memory. As a processor's cache is typically much faster to access than memory (a handful of cycles as opposed to dozens or hundreds of cycles), caches have a large positive performance impact. Similar to memory, each cache has a corresponding *cache controller* that implements the necessary logic to manage the cache.

Caches can be *write-through* or *writeback*. In a write-through cache, writes are propogated immediately to the next lower level of the memory hierarchy (which may be the physical memory itself or a lower level of cache). In a writeback cache, processors can both read from and write to data in the cache. If a block in the cache has been written since it was last read from memory, the block is said to be *dirty* in the cache.

As caches are finite-sized, the cache controller may find that there is no free entry when it looks to insert a given block into the cache. In this case, the cache controller must *evict* a block from the cache, writing back the block being evicted to memory if it is dirty.

### 2.5.2 Cache Coherence

In a multiprocessor, a block may reside in multiple processors' caches at the same time. This can cause problems if one of the processors writes the block: the current value of the block will appear to be different depending on the processor reading this value. Preventing this problem from arising is the role of the *cache coherence protocol*. The coherence protocol is a distributed algorithm that allows a multiprocessor's caches to safely manipulate data. In this section, we present a high-level overview of cache coherence protocols. Our intent is that the overview be sufficient to enable

**Cache Coherence Algorithm**

**load(A)**

```
if Cache[A].state == I:
  obtain_permissions(A,read)
return Cache[A].data
```

**store(A,v)**

```
if Cache[A].state != M:
  obtain_permissions(A,write)
Cache[A].data = v
```

**obtain_permissions(A,perm)**

```
if perm == read:
  foreach remote cache C:
    send downgrade request to C
  wait for acknowledgements
  Cache[A].state = S
else:
  foreach remote cache C:
    send invalidate request to C
  wait for acknowledgements
  Cache[A].state = M
```

**handle_downgrade_request(A)**

```
if Cache[A].state == M:
  Cache[A].state = S
send acknowledgement
```

**handle_invalidate_request(A)**

```
Cache[A].state = I
send acknowledgement
```

Figure 2.10: **Coherence algorithm.** High-level operation of an invalidation-based cache coherence protocol in which processors can have write permissions ("M state"), read permissions ("S state"), or no permissions ("I state") to a given block. The protocol enforces the invariant that at most one processor can have write permissions to a block at a given time. This figure focuses on showing how the transfer of permissions occurs; transfer of data is not shown.

understanding of the role of a coherence protocol in supporting hardware transactional memory. We refer the reader to Chapters 5-8 of Culler and Singh [29] as well as Martin's dissertation [69] for a more thorough introduction.

This dissertation considers *invalidation-based coherence protocols*. Invalidation-based coherence protocols enforce the invariant that for a given memory block, at any time *either* one processor can write the block *or* any number of processors can read the block. Enforcing this invariant (the so-called "coherence invariant") ensures that at any given time the current value is unambiguous (subject to the multiprocessor's memory consistency model [3]).

As part of enforcing the coherence invariant, the coherence protocol introduces *coherence permissions*. For a given block, a processor can either have permission to both read and write the block, permission to read (but not write) the block, or no permissions to the block at all. These permissions are encoded in the *coherence state* of the block, maintained in a cache entry along with the data and

tags. There are three basic coherence states, corresponding to the three types of permissions: the *Modified* or *M* state corresponds to read/write permissions, the *Shared* or *S* state corresponds to read-only permissions, and the *Invalid* or *I* state corresponds to no permissions. Maintaining the coherence invariant thus equates to ensuring that (1) when a processor enters the M state, all other processors are in the I state, and (2) when a processor enters the S state, there is no other processor in the M state.

When a processor wants to write a block, it first *invalidates* all readers of the block (transitions them to I state). When a processor wants to read a block, it first *downgrades* any writer of the block (transitions them to S state). To invalidate or downgrade a remote processor, the requesting processor sends a *coherence request*. The remote processor processes this request and then sends an *acknowledgement* back to the requesting processor. A processor being invalidated or downgraded from the M state includes the data in its response. Figure 2.10 on page 29 gives the high-level operation of a basic three-state protocol, focusing on showing how the transfer of coherence permissions occurs (the data transfer is not shown).

Many variants of this basic protocol exist. For example, *directory protocols* have the property that coherence requests are sent only to sharers of a block, potentially reducing bandwidth requirements and increasing system scalability. Protocols may include additional coherence states to optimize performance. For example, adding an *Exclusive* state indicating that the processor has write permissions for the block but the block has not yet been written can enhance performance by reducing so-called "upgrade misses" (*i.e.*, cases where a processor has a block in $S$ state and wants to write the block). All invalidation-based protocols, however, maintain the same basic property that a reader of a given block is guaranteed to see any write request for that block and a writer of a given block is guaranteed to see any request for that block.

## 2.6 Bounded Hardware Transactional Memory

In this section we present a system that can support the speculative parallel execution of transactions in hardware. This system is inspired by the original hardware transactional memory proposal [47]. In particular, the key observation made in that work is that as long as a processor has coherence permissions to a transactionally-accessed block, that processor is guaranteed to observe all conflicting requests for the block. This fact can be exploited to build a hardware implementation of the eager

conflict detection/eager version management algorithm presented in Section 2.4. This implementation utilizes existing structures and has low overheads, but can support only transactions that are bounded in size and time.

The bounded HTM adds the ability to take a register checkpoint, a counter (called the transactional nesting depth or TND counter), and two bits per L1 cache line[3] (called the *read bit* and *written bit*) to a conventional multiprocessor. To initiate a transaction, the processor checkpoints the register state. Transactional loads and stores set the read bit and written bit respectively; the cache checks the state of these bits to determine conflicts on incoming coherence requests from other processors. Version management may be supported either by logging pretransactional values in a log in virtual memory or by buffering these values in lower levels of the memory hierarchy; we discuss both options below. To commit a transaction, the processor atomically clears the read and written bits in a few-cycle operation called a *flash-clear* operation (discussed in detail below). The specifics of abort depend on the version management mechanism used.

Below we discuss the components of the bounded HTM in more detail. We first discuss conflict detection via cache coherence, how age-based conflict resolution can be implemented in hardware, and options for version management. In Section 2.6.4 we present two complete bounded HTM algorithms. Finally, we discuss the reasons that this implementation places restrictions on the size and duration of transactions.

We note that hardware similar to that discussed in this section has been proposed in several contexts beyond transactional memory. Among other uses, researchers have proposed using such hardware for the speculative parallel execution of lock-based critical sections [73, 88, 89] (*i.e.*, speculative lock elision or SLE), the speculative parallel execution of sequential programs [25, 33, 54, 87, 101, 104] (*i.e.*, thread-level speculation or TLS), speculative implementations of memory consistency [12], speculative compiler optimizations [79], acceleration of software transactional memory [24, 30, 55, 63, 95, 99, 100], and speculative resource reclamation [72]. Some of our proposals could be applicable in these other contexts, a point to which we return in Chapter 10.

[3]For a 64KB cache with 64-byte blocks, this addition requires 2k bits (256 bytes), representing 0.4% overhead.

### 2.6.1 Conflict Detection via Cache Coherence

As mentioned above, the bounded HTM exploits the property of cache coherence that a cache with read permissions for a given block is guaranteed to see all write requests for the block (because it must be downgraded for the requester to obtain write permissions), and a cache with write permissions for that block is guaranteed to see any request for the block (because it must be downgraded for the requester to obtain read permissions and invalidated for the requester to obtain write permissions). When a processor makes a transactional read and obtains read permissions for the block, it sets the entry's read bit in the cache. Similarly, a transactional write results in write permissions being obtained and the entry's written bit being set.

To perform conflict detection, the cache checks the read and written bits of a given entry on incoming coherence requests from other processors. An external write request to a transactionally-read block (a block whose entry has its read bit set) is a conflict, while any external request to a transactionally-written block (a block whose entry has its written bit set) signals a conflict. As described above, an entry with the read bit set is guaranteed to have at least read permissions for the block and thus observe all external write requests; an entry with the written bit set is guaranteed to have write permissions for the block and thus observe all external read requests. Thus, once a processor sets the read or written bit for a given cache block, it is guaranteed to detect all conflicts on that block *as long as the block remains in the cache*.

### 2.6.2 Conflict Resolution via Timestamping

Timestamps can be implemented by a loosely synchronous timer. Conflicts between a transaction and a non-transactional request (*i.e.*, a request without a timestamp) may be resolved either by aborting the transaction or stalling the request. Alternatively, non-transactional requests may also be assigned timestamps. To handle wraparound, the system can prevent any new transaction (*i.e.*, one that would be assigned a timestamp after the wraparound) from starting until all existing transactions have committed.

### 2.6.3 Options for Eager Version Management

In this subsection, we discuss version management options assuming writeback caches. The primary challenge of version management is that a transaction may write a block that is already non-

transactionally dirty in the cache. Version management must support recovery to the pretransactional value of the block to enable rollback. Below we discuss two options for providing this support. In the first option, pretransactional values are *cleaned* to lower levels of the memory hierarchy. In the second option, these values are buffered in a log in virtual memory. As each of these options allows a transaction to write directly into the L1 cache, they are instantiations of eager version management.

**Cleaning.** In this option, when a transaction wishes to perform a write to a block with the dirty bit set in the cache but *without* the written bit set, the current value of the block is first written back ("cleaned") to the next lower level of the memory hierarchy. On transaction abort, the entries for transactionally-written blocks are invalidated from the L1 cache (via a single-cycle operation called *conditional flash-invalidate* that we describe in Section 2.6.5). The pretransactional values of these blocks will then be refetched on demand from the lower levels of the memory hierarchy. On a commit, transactionally-written values are committed atomically when the written bits for the corresponding entries are cleared in the L1 cache. After a commit, a stale value for a transactionally-written block resides lower in the memory hierarchy; however, all memory requests (whether local or remote) must access the processor's L1 cache, which contains the current value. The stale value will simply eventually be overwritten when the processor either writes back the current value or transfers it to another processor.

Due to the simplicity of this mechanism, we assume it as the default mechanism of version management for our proposals. However, one limitation of this mechanism is that it restricts the amount of data that can be transactionally written to the size of the L1 cache: a transactionally-written block cannot be evicted from the L1 cache because it would have to be written back to memory, where it would overwrite the pretransactional value of the block. We next present an alternative mechanism for version management that has no such size restriction.

**Log.** Moore et al. [77] proposed version management via thread-private *logs* that reside in virtual memory. Before performing a speculative write, LogTM first writes the address and old value of the block being written into the current thread's log. When a transaction commits, the log is discarded by restoring the log pointer to the beginning of the log buffer. When a transaction aborts, the system iterates over the log entries in hardware or software, restoring each block. To avoid logging the

same memory block multiple times, log updates are elided when the written bit associated with the block is already set, indicating that it has previously been logged.

This mechanism has tradeoffs with the cleaning mechanism described above. As the log resides in virtual memory, this mechanism has the advantage that it can buffer an unbounded amount of data. However, if the log is walked in software, then the organization of the log must be part of a multiprocessor's architectural interface. Conversely, if the log is walked in hardware, the logic for doing so must be added to the multiprocessor.

### 2.6.4 Bounded HTM Algorithms

This section presents algorithms for bounded HTM systems employing cleaning and a log for version management. For simplicity of presentation, the algorithms elide the manipulation of the transactional nesting depth (TND) counter.

Figure 2.11 on page 35 presents an algorithm for a bounded HTM system using cleaning for version management. As described above, this algorithm sets read and written bits on transactional accesses and augments the logic for handling external coherence requests to detect conflicts based on these bits. When a transaction writes a block for the first time, the old value of the block is first cleaned to the next lower level of memory. On abort, all transactionally-written blocks are invalidated from the cache. Finally, the clear and invalidation operations performed as part of transaction commit and abort occur atomically in a small handful of cycles due to the support for flash-clear and conditional flash-invalidate. We discuss this support below as well as support for register checkpointing.

Figure 2.12 on page 36 presents a corresponding algoritm for a bounded HTM using a log that is maintained in virtual memory for version management. On the first transactional write to a block, the address and old value of the block are written into the log. On abort, the log is walked by either software or hardware to restore these old values. At both abort and commit the log is thrown away by resetting the log pointer to the base address of the log. In all other respects this algorithm is identical to the one presented above.

**Bounded HTM Algorithm Using Cleaning for Version Management**

**begin**

```
in_transaction = true
timestamp = clock
```

**load(A)**

```
if Cache[A].state == I:
  obtain_permissions(A,read)
if in_transaction:
  Cache[A].read_bit = true
return Cache[A].data
```

**store(A,v)**

```
if Cache[A].state != M:
  obtain_permissions(A,write)
if (in_transaction) and
   (not Cache[A].written_bit):
  write back A
  Cache[A].written_bit = true
Cache[A].data = v
```

**abort**

```
flash-invalidate writes
flash-clear written bits
flash-clear read bits
in_transaction = false
```

**commit**

```
flash-clear written bits
flash-clear read bits
in_transaction = false
```

**handle_downgrade_request(A,ts)**

```
if Cache[A].written_bit:
  resolve_conflict(A,ts)
if Cache[A].state == M:
  Cache[A].state = S
send acknowledgement
```

**handle_invalidate_request(A,ts)**

```
if Cache[A].read_bit or
   Cache[A].written_bit:
  resolve_conflict(A,ts)
Cache[A].state = I
send acknowledgement
```

**resolve_conflict(A,ts)**

```
if ts < timestamp:
  abort()
else
  defer until commit/abort
```

**evict(A)**

```
if (Cache[A].read_bit) or
   (Cache[A].written_bit)
  abort()
if Cache[A].dirty:
  write back A
Cache[A].valid = false
```

Figure 2.11: **Bounded HTM algorithm using cleaning for version management** The algorithm builds on basic cache coherence (presented in Figure 2.10 on page 29). To enable conflict detection, it adds a *read bit* and a *written bit* to each entry in the cache. Processors set these bits on transactional accesses and check them to determine conflicts when handling incoming coherence requests. Version management is implemented by *cleaning* the pretransactional value of a block to the next lower level of memory before the first transactional write of the block. The `obtain_permissions` function is unchanged from Figure 2.10 on page 29.

**Bounded HTM Algorithm Using a Log for Version Management**

**begin**

```
in_transaction = true
timestamp = clock
```

**load(A)**

```
if Cache[A].state == I:
  obtain_permissions(A,read)
if in_transaction:
  Cache[A].read_bit = true
return Cache[A].data
```

**store(A,v)**

```
if Cache[A].state != M:
  obtain_permissions(A,write)
if (in_transaction) and
   (not Cache[A].written_bit):
  store (A, Cache[A]) into log
  increment log pointer
  Cache[A].written_bit = true
Cache[A].data = v
```

**abort**

```
for (A,v) in log:
  Cache[A] = v
flash-clear written bits
flash-clear read bits
reset log pointer
in_transaction = false
```

**commit**

```
flash-clear written bits
flash-clear read bits
reset log pointer
in_transaction = false
```

**handle_downgrade_request(A,ts)**

```
if Cache[A].written_bit:
  resolve_conflict(A,ts)
if Cache[A].state == M:
  Cache[A].state = S
send acknowledgement
```

**handle_invalidate_request(A,ts)**

```
if Cache[A].read_bit or
   Cache[A].written_bit:
  resolve_conflict(A,ts)
Cache[A].state = I
send acknowledgement
```

**resolve_conflict(A,ts)**

```
if ts < timestamp:
  abort()
else
  defer until commit/abort
```

**evict(A)**

```
if (Cache[A].read_bit) or
   (Cache[A].written_bit)
  abort()
if Cache[A].dirty:
  write back A
Cache[A].valid = false
```

Figure 2.12: **Bounded HTM algorithm using a log for version management** The only difference between this algorithm and Figure 2.11 on page 35 is the use of a log rather than cleaning for version management. This log is maintained starting at a fixed location in virtual memory. When the transaction writes a block for the first time, hardware stores the address and old value of the block into the log. On abort, the log is walked (either in hardware or software) to restore the old values of all transactionally-written blocks. On both abort and commit the log is thrown away by resetting the log pointer to the base address of the log.

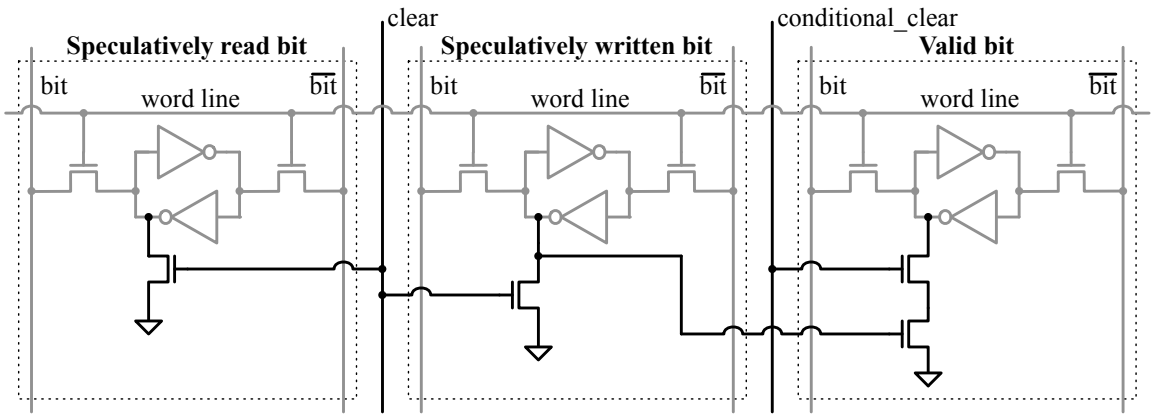


Figure 2.13: **Six-transistor SRAM cells (in gray) augmented with circuitry (in black) for flash-clear (left-most and middle cells) and conditional flash-clear (right-most cell).** When the `clear` signal is asserted, both the read and written bits are pulled down to zero. When the `conditional_clear` is asserted, the valid bit is pulled down to zero (invalid) if the speculatively written bit is one.

### 2.6.5 Implementation Details

Here we discuss the implementation details of register checkpointing and the flash-clear and flash-invalidate operations.

**Register checkpointing.** Several different approaches to checkpointing the register state exist, with the processor core microarchitecture playing a large role in determining which mechanism is the most appropriate. Some out-of-order processors implement register map table checkpointing for recovery from in-window speculation (*e.g.*, the MIPS R10000 [116] and the Alpha 21264 [53]); such processors can extend this support to beyond-the-window speculation by not freeing physical registers that are referenced in the checkpointed map table until speculation commit [4]. For processors employing in-order pipelines, the register file itself can be checkpointed [32].

**Transactional access bits added to the data cache tags.** As described above, the bounded HTM assumes that the read and written bits support two single-cycle or few-cycle flash-clear operations: first, a flash clear of all transactionally-read and transactionally-written bits, and second, a flash conditional-invalidation operation that clears the valid bit of any block that has the transactionally-written bit set. Figure 2.13 on page 37 illustrates standard 6T SRAM cells augmented to support these operations.

### 2.6.6 Restrictions on Transaction Size and Duration

The implementation described above supports transactions with both low overheads and high concurrency in the absence of data conflicts. However, it also limits both the volume of data that may be accessed within a transaction and the duration of a transaction.

The size of a transaction is limited for multiple reasons. First, the conflict detection mechanism fundamentally relies on transactionally-accessed blocks remaining in the cache. The reason is that if the cache evicts the block, it both gives up coherence permissions to the block and loses the state of the read and written bits for the block (they are now part of the state of the block currently in the entry formerly occupied by the evicted block). By losing coherence permissions, the cache is no longer guaranteed to observe conflicting requests for the block; by losing the read and written bits, it would have no way to determine what a conflict is even if it did receive these requests. Second, the eviction of a transactionally-written block would force a writeback of that block. If cleaning is used as the version management mechanism, writeback would overwrite the pre-speculative value residing lower in the memory hierarchy.

The bounded HTM also limits the duration of a transaction, as the in-cache read and write bits as well as transactionally-written state implicitly belong to the currently executing transaction. This implementation has no mechanism for transferring read and write bits or transactionally-written blocks from the cache to architected state, and so it must abort transactions on a context switch.

## 2.7 Semantic and Performance Challenges of Bounded HTM

The bounded hardware transactional memory supports speculative execution of transactions with low overheads and minor extensions to current multiprocessors. However, it also presents semantic and performance challenges, as we describe in this section.

The most basic challenge of bounded HTM is that it supports only transactions that are bounded in size and time, as detailed in Section 2.6.6. This semantic restriction forces programmers to either (1) be able to guarantee that their transactions fit within the time and space bounds of the HTM or (2) incorporate an alternative, "backup" means of synchronization (*e.g.*, locks) in addition to transactions. The first alternative is simply not possible in many real-world situations (*e.g.*, using a transaction to protect a tree traversal in a library data structure). The second alternative undermines transactional memory's promise of making high-performance synchronization easier to

achieve. Thus, for hardware transactional memory to be useful as a general-purpose synchronization primitive, it must be able to support transactions that are unbounded in size and time.

A related problem is that if overflows occur *frequently*, it will be necessary to handle them with high performance or else they will become a performance bottleneck. That is, the problem of overflows is potentially a performance problem as well as a semantic problem. Ideally, an unbounded HTM would be able to support unbounded transactions while maintaining the bounded HTM's properties of high concurrency, low overheads, and design simplicity. As we detail in Chapter 4, achieving all of these properties simultaneously is challenging.

Finally, like any transactional memory system, the bounded HTM achieves parallelism only in the absence of data conflicts. In addition to the *true conflicts* that can occur in any transactional memory system, the bounded HTM can potentially experience *false sharing conflicts* (conflicts due to two transactions accessing different words on the same cache block) due to its detection of conflicts at a cache-block granularity. If conflicts are frequent (whether due to true sharing or false sharing), performance degradation will occur due to stalls and/or aborts.

## 2.8 Summary

In this chapter, we presented a basic overview of transactional memory to provide context for the remainder of this dissertation. We first presented our basic problem context of synchronization in shared-memory parallel programs, outlined the challenges of synchronization via locks, and discussed the potential of transactional memory to ease the task of high-performance synchronization (Section 2.1). In Section 2.2 we described transactional semantics, including the basic transactional interface of isolation and atomicity as well as advanced semantic issues such as the strong/weak atomicity and support for explicit abort. We then presented the basic implementation tasks involved in executing transactions speculatively in parallel while maintaining isolation and discussed the terminology associated with those tasks (Section 2.3). We presented three high-level transactional memory algorithms in Section 2.4. After reviewing multiprocessor memory systems in Section 2.5 we described how to layer support for executing bounded transactions speculatively in parallel on top of these systems in Section 2.6. Finally, we discussed semantic and performance challenges of this bounded hardware transactional memory in Section 2.7.

In the next chapter we introduce the workloads that we use to drive and evaluate the proposals of our dissertation. We study the extent to which conflicts limit performance, the nature of these conflicts, and the sizes of the transactions in these workloads.

# Chapter 3

# Characterization of Transactional Behavior

In this chapter we introduce the transactional workloads that we use as benchmarks to drive and evaluate our proposals. We examine three questions: (1) to what extent conflicts hinder performance in these workloads, (2) the nature of these conflicts, and (3) how large the transactions in these workloads become. The answer to the last question will help to determine the nature of our proposal for a hardware transactional memory that efficiently supports unbounded transactions (Chapter 5 and Chapter 6), whereas the answers to the first two questions will help focus our efforts on making hardware transactional memory robust to auxiliary data conflicts (Chapter 8).

In the next section, we describe the workloads that we use. In Section 3.2 we describe the infrastructure and methodology that we use for quantitative evaluation in this chapter and throughout the rest of this dissertation. We examine the performance of an idealized unbounded hardware transactional memory system on these workloads in Section 3.3, finding that performance is generally limited by conflicts. We analyze the extent to which straightforward software restructurings can reduce these conflicts and examine the nature of the conflicts that remain after these restructurings in Section 3.4. We then study the question of how large the transactions in these workloads become in Section 3.5. We close the chapter in Section 3.6 with a discussion of the implications of our findings.

| Workload | Description and Input Parameters |
|---|---|
| genome | From STAMP, gene sequencing, *g1024 s48 n65536* |
| genome-sz | Variant with resizable hashtable |
| intruder | From STAMP, network packet intrusion detection, *a10 l4 n2038 s1* |
| intruder_opt | Variant with fixed-size hashtable and thread-private queues |
| intruder_opt-sz | Variant with resizable hashtable and thread-private queues |
| kmeans | From STAMP, partition-based clustering, *m15 n15 t0.05 irandom-n2048-d16-c16.txt* |
| labyrinth | From STAMP, shortest-distance path routing, *irandom-x32-y32-z3-n96.txt* |
| ssca2 | From STAMP, graph kernels, *s13 i1.0 u1.0 l3 p3* |
| vacation | From STAMP, travel reservation system, *n4 q60 u90 r16384 t4096* |
| vacation_opt | Variant with fixed-size hashtable |
| vacation_opt-sz | Variant with resizable hashtable |
| yada | From STAMP, Delaunay mesh refinement, *a20 i633.2* |
| python | Python interpreter, *bm_threading.py -n32768 (from Google's Unladen-Swallow [2] suite)* |
| python_opt | Variant of python with interpreter optimizations |

Table 3.1: **Workloads used in this dissertation.**

## 3.1 Workloads

As transactional memory is an emerging synchronization primitive, no standard set of transactional benchmarks yet exists. We draw the bulk of the workloads that we use from the STAMP [76] benchmark suite, which has emerged as a *de facto* standard. We additionally run a transactionalized variant of the reference Python interpreter [113]. We describe these workloads in more detail below. Table 3.1 on page 42 summarizes the workloads (we discuss the _opt variants in Section 3.4). We analyze the performance characteristics of these workloads in detail in Section 3.3, Section 3.4, and Section 3.5. We note, however, that the performance characteristics of these workloads may or may not be representative of future workloads. This fact will particularly influence our proposal for unbounded hardware transactional memory (Chapter 5 and Chapter 6).

### 3.1.1 STAMP

The STAMP benchmark suite [76] is a set of transactional memory benchmarks that for the most part use coarse-grained transactions, with the intent of simulating the practices of naive programmers. The transactions in STAMP are often large and/or highly-conflicting.

STAMP includes a hashtable that defaults to be non-resizable. For all workloads that use this hashtable we also run a second variant in which the hashtable is configured via STAMP's compile-

time flags to automatically resize as needed. The resizable hashtable maintains an internal occupancy field that is incremented on every insert; as part of such an insert, it checks the value of this field to determine if it is necessary to resize the hashtable in order to maintain expected constant-time access. It is thus significantly more vulnerable to conflicts than the non-resizable hashtable, which does not maintain this occupancy field. The STAMP workload variants using the resizable hashtable reflect the performance of using standard library implementations of hashtables in Java and C++, which are generally resizable by default. These variants have "`-sz`" appended to their names.

We run all workloads in the suite except `bayes`, from which we could not extract useful conclusions due to high runtime variability. We give a brief overview of each of these workloads below; for more details, we refer the reader to Minh *et al.* [76].

**genome.** `genome` implements a genome reconstruction algorithm. There are several static transactions within the program. A transaction is employed to make a series of inserts into a shared hashtable; a transaction is employed to find a free entry in a shared array and occupy that entry; a transaction is used to protect an insert into a shared hashtable; and finally, a transaction is used to access a shared list. `genome` spends a large percentage of time in transactions, which themselves grow fairly large. The performance of `genome` is sensitive to whether the hashtable is resizable or not.

**intruder.** `intruder` implements an intrusion detection algorithm. Transactions protect accesses to shared lists. Additionally, the program uses a map implemented as a red-black tree that internally uses transactions. The transactions in `intruder` are generally small and highly-conflicting.

**kmeans.** `kmeans` implements K-means clustering. A transaction is used to protect accesses to a shared array. Additional transactions protect read-modify-write operations. The transactions in `kmeans` are small and rarely conflict.

**labyrinth.** `labyrinth` routes shortest paths between pairs of startpoints and endpoints in a two-dimensional grid (*i.e.*, wire routing). To route a pair of points, a thread copies the grid into a local copy, routes a path through the local copy, and attempts to claim this path in the global grid. A transaction is used to protect this entire operation. However, the transaction releases conflict detection on the shared grid via an *early release* operation [46] after copying it, as the algorithm itself detects conflicts on a semantic level (when trying to claim a path, threads check whether nodes

in the path have been claimed by other threads in the interim). To achieve the same behavior without early release (a mechanism that our systems do not provide), we modified the code to perform the grid copy before entering the transaction. Transactions are additionally used to protect accesses to a shared queue and inserts into a shared list. These transactions are generally small.

**ssca2.** `ssca2` is a set of graph kernels. Transactions are used to protect inserts of nodes into the shared graph. `ssca2` spends little time in transactions.

**vacation.** `vacation` implements a travel reservation system. The program maintains several in-memory database structures (*e.g.*, a database storing information for hotel reservations). These structures are implemented as red-black trees that use transactions internally for synchronization. Transactions are also used to make updates to multiple databases in isolation (similar to Figure 2.6 on page 16). The transactions in `vacation` grow fairly large and have a fair amount of conflicts.

**yada.** `yada` implements an algorithm for Delaunay mesh refinement. The mesh is a graph structure with edges represented as adjacency lists. Threads repeatedly add elements to the mesh, remove elements from the mesh, and modify the edges in the mesh. Transactions protect threads' accesses and modifications to the mesh. These transactions are large and highly-conflicting.

### 3.1.2 Python

We created a transactionalized version of the standard Python interpreter implementation [113], `python`. Although the interpreter supports threading, this threading is explicitly designed for responsive graphical user interfaces and I/O events—not for supporting parallel execution on multicores [113]. In fact, threads synchronize using a global interpreter lock (GIL). Although threads may perform selected system calls without holding the GIL, threads may interpret bytecodes only while holding the GIL. Thus, in the absence of speculation, bytecode interpretation is completely serialized by the GIL. We transactionalized the interpreter by replacing acquires and releases of the GIL with transactions. The resulting workload has frequent conflicts on shared data structures such as freelists and reference counts. We discuss the extent to which we were able to eliminate these conflicts via straightforward software restructuring in Section 3.4.

| Parameter | Value |
|---|---|
| Processor | 32 in-order x86 cores, 1 IPC |
| L1 cache | 64 KB, 4-way set associative, 64B blocks, 1-cycle hit latency |
| L2 cache | Private, 1MB, 4-way SA, 64B blocks, 10-cycle hit latency |
| Memory | 100-cycle DRAM lookup latency |
| Coherence | Directory-based protocol, 20-cycle hop latency |

Table 3.2: **Simulated machine configuration.**

## 3.2 Experimental Infrastructure and Methodology

In this section we outline the infrastructure that we use to perform all of the quantitative analysis in this dissertation. We also outline the transactional memory configuration that we evaluate to analyze transaction sizes and performance bottlenecks in the workloads that we use.

**Infrastructure.** We perform our quantitative analysis using a full-system, execution-driven simulator that we have developed internally. Our simulator simulates the x86-64 architecture. It uses PTLSim [118] to crack x86 instructions into sequences of micro-ops, which it then executes. In addition, it utilizes Simics [66, 67], a functional simulator, to handle rarely-occurring complex instructions (*e.g.*, memory-mapped IO). This simulator is thus an instance of *timing-first simulation* [74], in which a detailed timing simulator handles nearly all of a given instruction set but utilizes a complete functional simulator as a backend for certain rare instructions. The memory system implementation is based on an early version of the GEMS [71] memory system simulator, but with heavy modifications. In particular, we have extended the simulator to model all of the transactional memory systems described in this dissertation. A version of this simulator was forked by Neelakantam *et al.* and developed into the publicly-available $FeS_2$ simulator [1]; there is significant divergence between that codebase and that of the simulator that we employ.

**Simulated machine.** We configure our simulator to model a 32-processor multiprocessor. Table 3.2 on page 45 presents the details of our memory system configuration. Our simulated memory hierarchy models a directory-like MOESI protocol (Section 2.5.2). We augment this protocol with a migratory sharing mechanism that optimizes the transfer of memory blocks exhibiting a read-modify-write sharing pattern (*e.g.*, locks) [27, 106]. Throughout this dissertation, we assume single-cycle register checkpoint and restore operations. We model a one-instruction-per-cycle (IPC) processor. The simulator is capable of booting an unmodified operating system. To perform the ex-

periments of this dissertation, we run the workloads that we evaluate in the Fedora Core 5 operating system with version 2.6.15 of the Linux kernel running on the simulator.

**Evaluation methodology.** Our process of setting up the timing evaluations of a given application is as follows. Each workload has an initial serial "setup" phase, followed by a parallel phase that does the main work of the application, typically followed by a final serial "cleanup" phase. We place a barrier at the start of the application's parallel phase and another barrier at the end of the application's parallel phase. We perform an initial run of the application. In this initial run, we take a checkpoint of the application state once all threads have reached the "begin parallel phase" barrier (using Simics' ability to save and restore checkpoints of system state). We then perform our timing evaluations starting from this checkpoint. A timing run is complete once all threads have reached the "end parallel phase" barrier. As these workloads generally do not employ internal barriers, time spent in barriers in these runs is generally time spent in this "end parallel phase" barrier and thus an indicator of load imbalance.

Our focus in these evaluations is on parallelism rather than concurrency. To avoid the effects of context switching perturbing our application runs, we set up each run with the same number of threads as cores and use a Linux system call to pin each thread to a unique core.

**Transactional memory configuration evaluated in this chapter.** In this chapter we simulate an idealized hardware implementation of the eager conflict detection/eager version management transactional memory algorithm presented in Figure 2.7 on page 23. This system can support transactions of any size with full concurrency and no overheads. It uses logging (Section 2.6) to implement unbounded version management. Our purpose in evaluating this configuration is to perform a limit study that enables us to answer the questions of (1) to what extent conflicts present a performance bottleneck and (2) what are the sizes of transactions. In order to examine the impact of conflicts independent of the specific overheads of a particular version management mechanism, we assign zero cost to restoring a transaction's log on abort in this idealized system. We present this conflict analysis next.

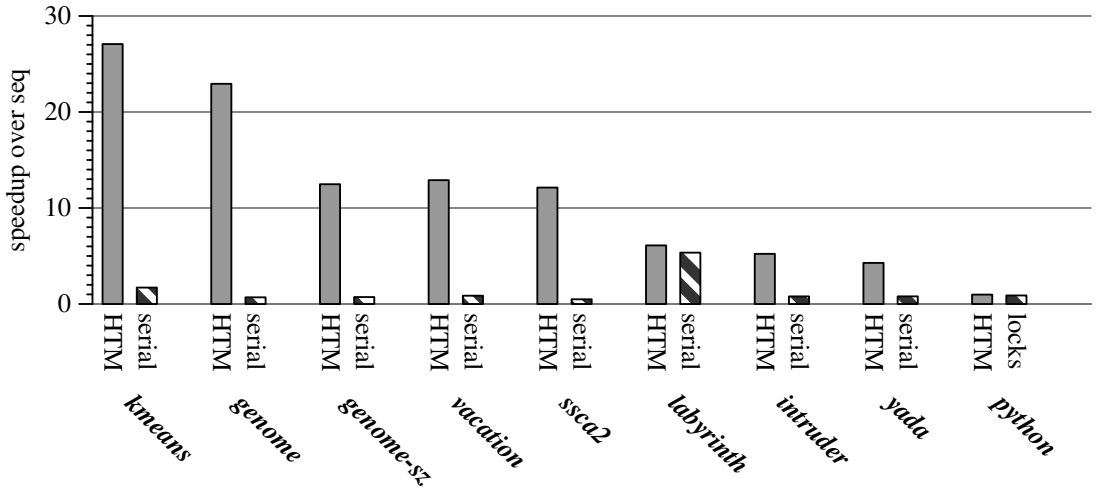


Figure 3.1: **Scalability of workloads under idealized HTM.** Execution is on 32 cores, meaning that a speedup of 30 is near-ideal. As a reference point for workloads that were created with lock-based synchronization, the right bar of each group represents performance using locks. The right bar of the STAMP workloads ("serial") represents performance if transactions are executed serially.

## 3.3 Are Conflicts a Performance Problem?

This section focuses on determining the extent to which conflicts limit performance. We do so via evaluation of the performance of the idealized unbounded HTM, in which transaction size does not form a performance bottleneck. In essence, this configuration allows us to determine the best result that could be hoped to achieve by extending the bounded HTM described in Section 2.6 to support transactions of unbounded size in the same fashion. After finding that the performance of this configuration is limited by conflicts on many workloads, we also evaluate idealized HTM's employing lazy and value-based conflict detection to determine whether these conflicts are an artifact of employing eager conflict detection.

**Overall performance.** Figure 3.1 on page 47 presents the scalability of this idealized unbounded HTM over sequential execution on 32 cores. As a reference point we also present the performance of these workloads if synchronization is implemented conservatively by executing critical sections or transactions serially rather than speculatively in parallel. The right bar of each group in Figure 3.1 on page 47 represents such a configuration. For `python`, this bar shows the performance of the lock-based version ("locks"). For the STAMP workloads, which were originally created using transactions, this bar shows the performance of a system in which transactions are executed serially ("serial").

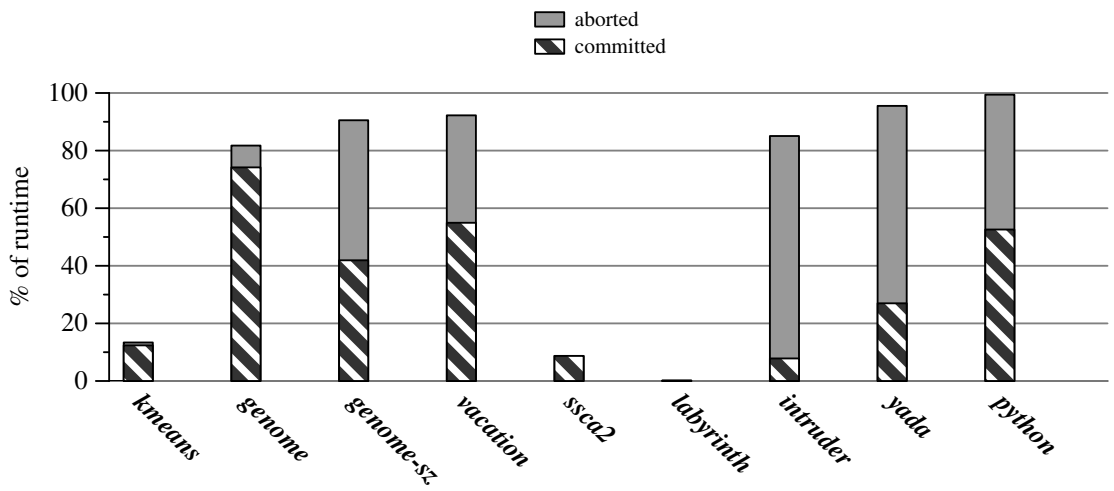


Figure 3.2: **Percentage of total execution time that is spent in committed and aborted transactions.**

---

We first note that speculative parallelization is necessary to achieve performance. On all workloads but one ("labyrinth", discussed in more detail below), the versions in which transactions are executed serially achieve no performance over sequential execution. This result provides basic validation of the use of these workloads for evaluating transactional memory design.

However, many of these workloads have only fair to poor scaling on the idealized HTM as well. While two workloads achieve near-linear speedups (`kmeans`, `genome`), all others achieve a speedup of only 13X or less on 32 cores. `python` has essentially no speedup over sequential execution. In the remainder of this section, we determine the role of conflicts in this limited performance. In the next section, we explore the nature of these conflicts in order to understand whether there is a potential role for hardware to play in eliminating their performance impact.

**Time lost to conflicts.** Figure 3.2 on page 48 presents the percentage of total execution time that processors spend in committed and aborted transactions. All but two workloads spend over 80% of execution time in transactions, providing insight into why speculative parallelization is necessary to obtain speedups. In fact, as `kmeans` and `ssca2` demonstrate, such parallelization can be necessary to achieve performance even for workloads that spend 15% or less of execution time in transactions. `labyrinth` spends essentially no time in transactions, explaining why it is insensitive to whether transactions are executed in parallel or serially.

This result also indicates that many workloads spend a large percentage of overall execution time in conflicts. All but three workloads lose over 30% of execution time to conflicts. The amount

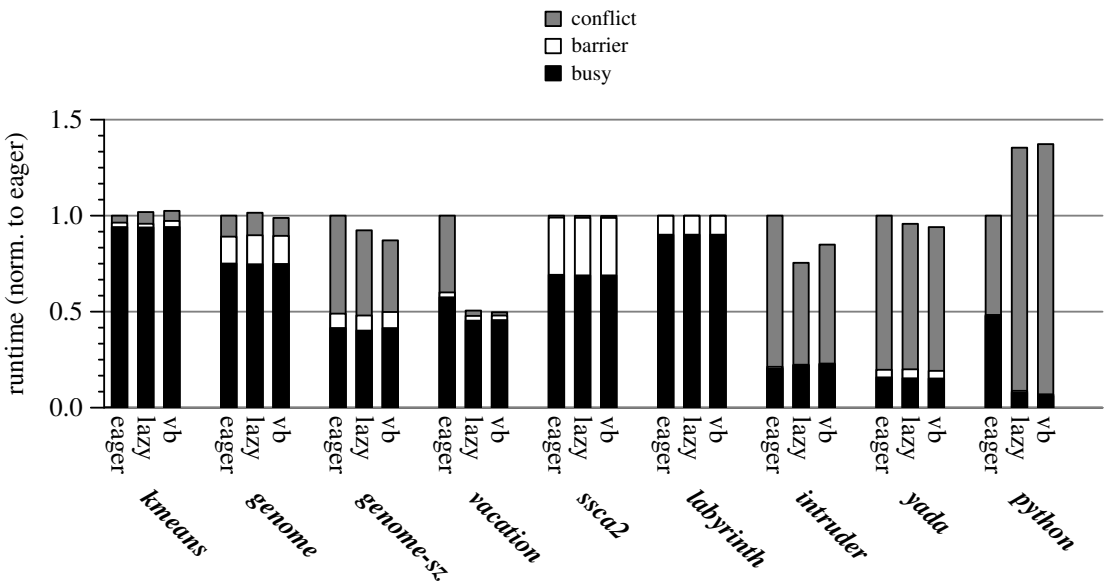


Figure 3.3: **Runtime breakdown of workloads under idealized HTM.** "eager" represents the idealized HTM shown in Figure 3.1 on page 47. "lazy" represents a variant in which conflict resolution is deferred until commit in the hopes of reducing stalling/rollbacks. "vb" (value-based) represents a further variant that incorporates lazy conflict resolution as just described and additionally performs conflict detection by checking equality of values read at a byte granularity (thus eliminating conflicts due to false sharing). "conflict" represents time spent stalled by other transactions or in transactions that ultimately abort. "barrier" represents time stalled at internal barriers as well as time spent waiting for other threads to finish at the end of execution. "busy" represents all other execution time.

---

of time lost to conflicts generally inversely correlates with the overall performance of these workloads presented in Figure 3.1 on page 47. `labyrinth` and `ssca2` are exceptions. Further examination revealed that the algorithm used in `labyrinth` induces load imbalance, whereas `ssca2` suffers from poor locality due to threads' concurrent graph updates and traversals.

As described in Section 3.2, this HTM configuration employs cache-block granularity eager conflict detection. In Chapter 2, we noted that an eager conflict detection TM could suffer from conflicts that a lazy TM could avoid, and additionally noted that HTM could suffer from false conflicts due to the fact that conflict detection is performed at cache block granularity. We next determine to the extent to which these conflicts are artifacts of this HTM's conflict detection mechanism.

**Sensitivity of performance to conflict detection mechanism.** To analyze the impact of lazy and value-based conflict detection on performance, we evaluated two variants of the idealized unbounded HTM: a variant that defers conflict resolution until transaction commit ("lazy") and a further variant that performs conflict detection by checking for equality of values read at byte gran-

ularity ("vb"). The "lazy" variant allows conflicts to occur during execution by maintaining the locally-modified versions of conflicting blocks in a private buffer. At commit, it requires all blocks that have been lost due to conflicts and checks that the values of these blocks have not changed. The "vb" configuration is similar, but it additionally maintains read bits at a byte granularity and checks only that the values of bytes read have not changed. We present a breakdown of the execution of these variants in Figure 3.3 on page 49. For reference we also present the default eager HTM that had been evaluated above (represented by "eager" on this graph).

In most cases these variants have essentially unchanged performance characteristics, meaning that the conflicts occurring in these workloads are *true conflicts*, *i.e.*, they are cases where transactions actually access the same piece of data in a conflicting way. One exception is `vacation`, in which lazy conflict detection increases concurrency by allowing transactions reading the databases to commit in the midst of transactions that write the databases. On `python` laziness causes decreased performance, illustrating the performance tradeoff described in Section 2.3.

In this section we thus determined that (1) conflicts are a performance bottleneck and (2) these conflicts are true conflicts. In the next section we examine the nature of these conflicts in more detail to determine whether simple software restructurings can eliminate them or whether there is a role for additional hardware support.

## 3.4 Analysis of Conflicts

This section analyzes the nature of conflicts in these workloads, with the goal of determining whether hardware support has a role to play in eliminating the performance impact of these conflicts. We (1) quantify the impact that straightforward software restructurings can have on reducing these conflicts and (2) characterize the conflicts that remain after such restructurings.

**Opportunities for software restructurings.** We find several opportunities for straightforward software restructurings. First, `python` contains global variables that are conceptually thread-private but were not made so due to the assumption that only one thread would be operating on them at a time. We trivially made these variables thread-private using the C "`__thread`" variable annotation supported by GCC and other compilers. Second, `intruder` dequeues work from one highly contended queue and enqueues work onto another highly contended queue; we split these queues to be thread-private. Third, both `intruder` and `vacation` have aborts due to rebalancing operations

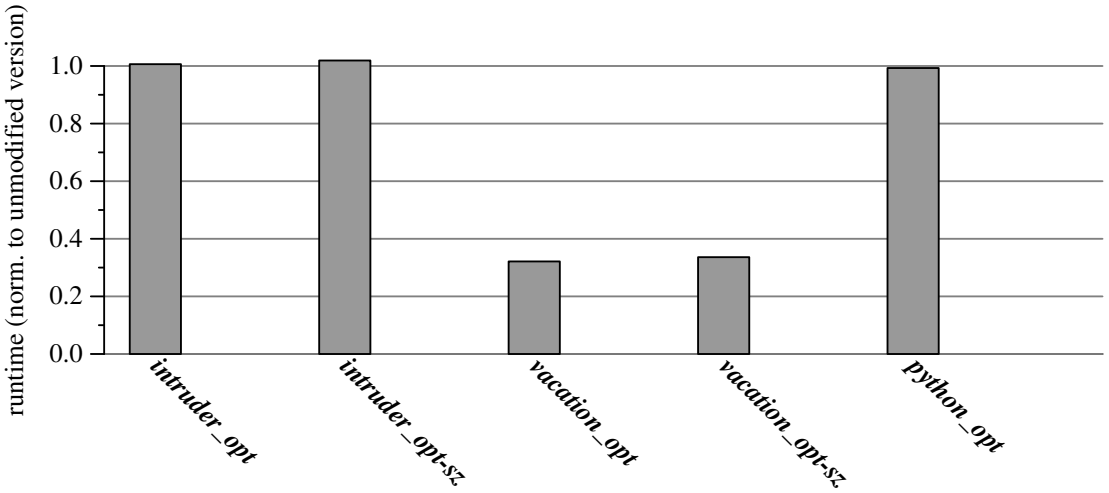


Figure 3.4: **Sequential runtimes of optimized workloads relative to the unoptimized versions.**

---

of a red-black tree used to implement an unordered map interface. We replaced the tree implementation with STAMP's hashtable implementation (as described in Section 3.1, this restructuring results in two variants: one with the non-resizable hashtable, and one with the resizable hashtable). The conflicts in `yada` are due to irregular traversals of a shared mesh; we have not found a way to reduce these conflicts short of restructuring the algorithm, which is beyond the scope of straightforward software restructuring. Table 3.1 on page 42 summarizes the optimized workloads (the workloads with a `_opt` suffix).

**Impact of restructurings on sequential runtime.** These restructurings are designed to increase the scalability of the workloads. However, it is important to determine whether they induce overheads on sequential execution. Figure 3.4 on page 51 shows that these restructurings in fact do not induce overhead in sequential performance. For `vacation` the restructurings actually increase sequential performance by almost 3X, as the hashtable has faster lookups than the red-black tree.

**Impact of restructurings on scalability.** Figure 3.5 on page 52 presents the scalability of the optimized workloads together with their unmodified variants. This graph shows first that these simple changes have a dramatic effect on the behavior of `vacation_opt` and `intruder_opt`, increasing scalability from 13x and 5x respectively to over 20x in both cases[1]. For the other variants,

[1]Note that this fact means that `vacation_opt` has a scalability of **80X** over the sequential run of the unoptimized `vacation`.

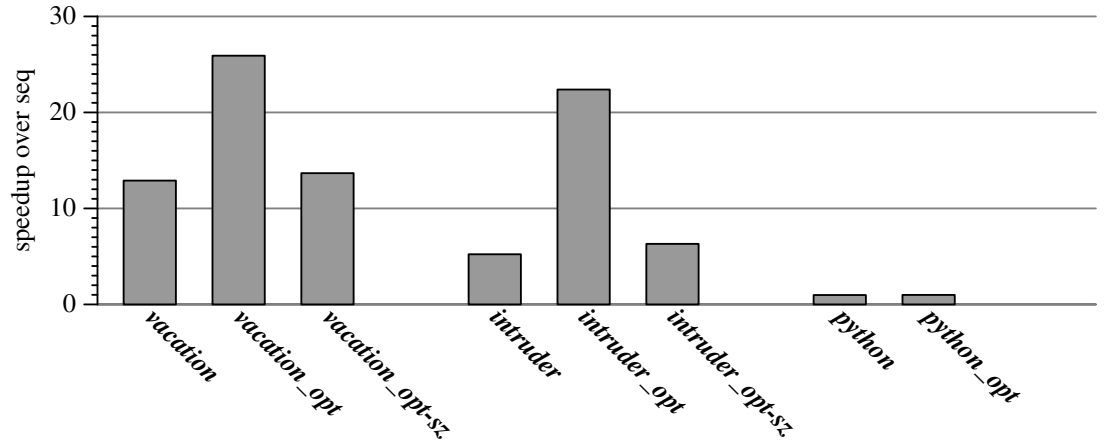


Figure 3.5: **Scalability of unoptimized and optimized versions of workloads under idealized HTM.**

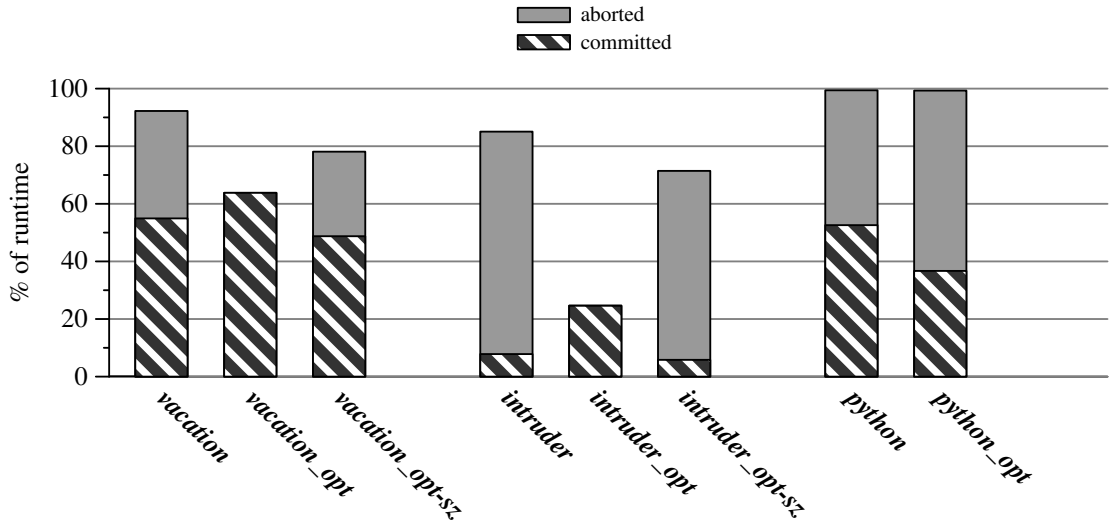


Figure 3.6: **Percentage of execution time that is spent in transactions for unoptimized/optimized workloads.**

however, the picture is less rosy: the software changes do not improve scalability, and Figure 3.6 on page 52 indicates that these workloads remain conflict-bound even after elimination of the most obvious sources of conflicts.

**Characterizing remaining conflicts.** The `python_opt` workload conflicts on reference counts of shared objects; `vacation_opt-sz`, `intruder_opt-sz`, and `genome-sz` conflict on the occupancy field that the resizable hashtable increments on every insert to determine when to resize; and as mentioned above, `yada` conflicts on mesh operations. We note that except for `yada`, all of the remaining conflicts occur on operations that are auxiliary to the main computation of the workload. Unfortunately, these conflicts are less amenable to straightforward software restructuring (for example, distributing reference counts per-thread would likely result in high storage and performance overheads). In Chapter 8, we propose a hardware mechanism to eliminate the performance impact of auxiliary data conflicts transparently to the programmer.

**Summary.** In the previous two sections we have demonstrated that conflicts form the primary bottleneck to performance *assuming* that hardware can support unbounded transactions with no performance overheads. However, as discussed in Section 2.6, basic hardware transactional memory can support only transactions that are bounded in size. The goal of the first part of this dissertation is to extend this basic HTM to support unbounded transactions with low design complexity (Chapter 5 and Chapter 6). We first perform an analysis of transaction sizes in the next section to understand the common-case behavior of these workloads.

## 3.5 How Large Do Transactions Become?

In this section, we examine the sizes of the transactions that occur in these workloads. We first examine transaction length to give an idea of the characteristics of the different workloads. We then examine the amount of data that transactions touch.

In analyzing this data, an intuitive way to organize the results would be to look at the percentage of transactions executing for a given number of cycles (or touching a certain amount of data). Figure 3.7 on page 54) shows such a breakdown for transaction length (organized as a cumulative histogram). This presentation, however, obscures the fact that longer transactions occupy more execution time than shorter transactions. Figure 3.8 on page 55 illustrates this fact by pre-

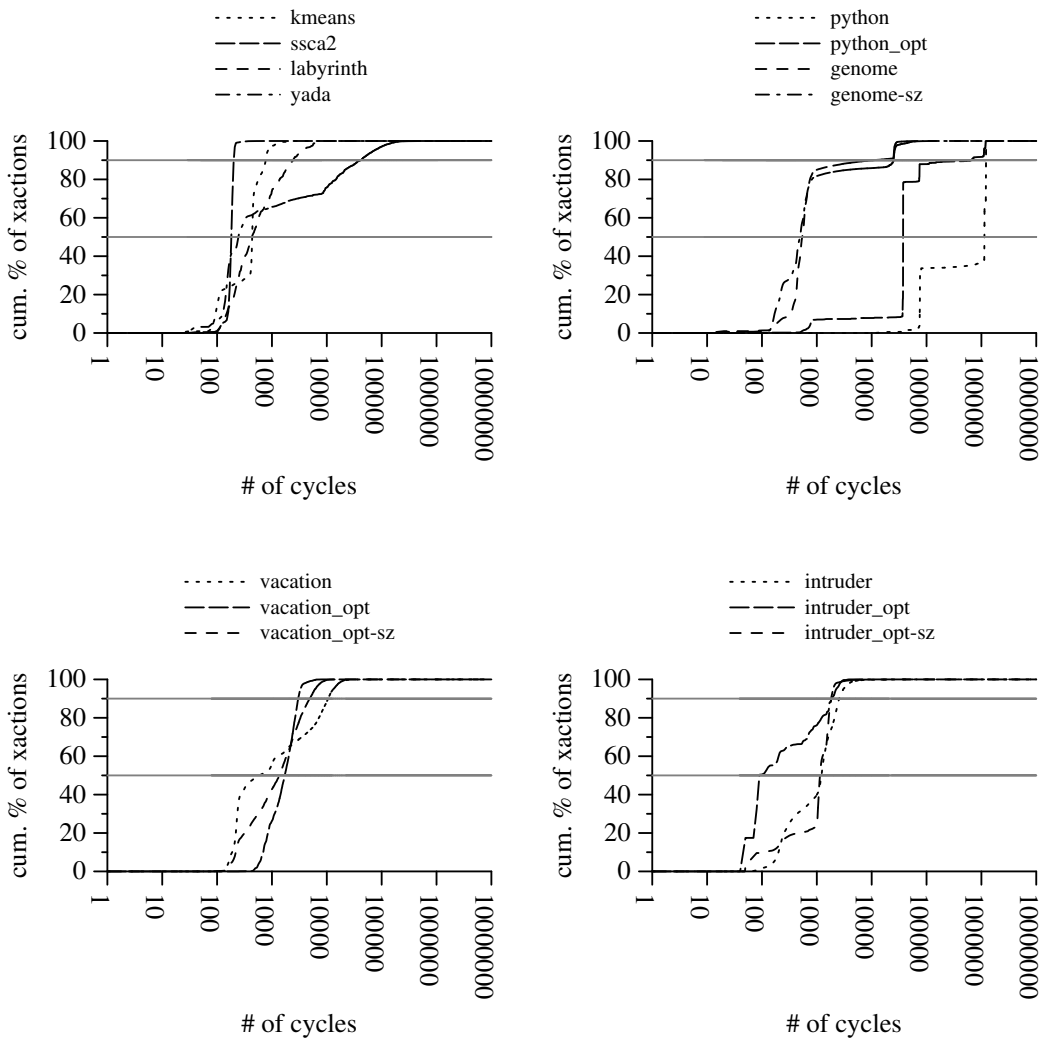


Figure 3.7: **Breakdown of transaction lengths by percent of transactions.** The x-axis is transaction length, and the y-axis is the cumulative percentage of transactions of that length or fewer. Horizontal lines are drawn to indicate the 50% and 90% marks.

---

senting the breakdown of transaction length in terms of *total transactional cycles* spent executing in transactions of a given length. As can be seen by looking at these two figures, looking only at the percentage of transactions of a given length (or size) underrepresents the contribution of longer transactions. For all of the results in this section, we instead look at percentage of cycles spent executing in transactions of a given length or size.

**Transaction length.** As Figure 3.8 on page 55 shows, the transactions in these workloads are generally short. Over half of the workloads spend 90% or more of their total transactional execution

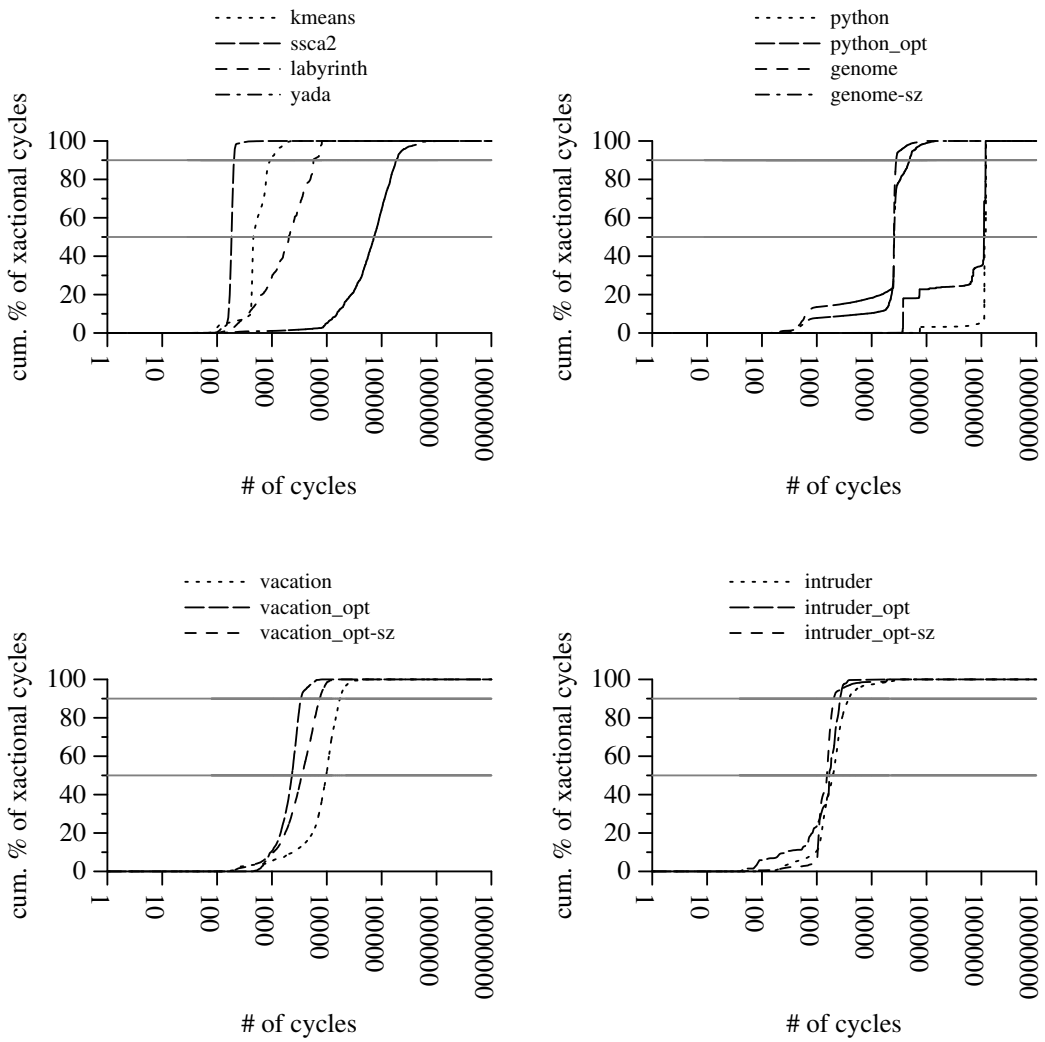


Figure 3.8: **Breakdown of transaction lengths by percent of transactional cycles.** The x-axis is transaction length, and the y-axis is the cumulative percentage of transactional cycles spent in transactions of that length or fewer. Horizontal lines are drawn to indicate the 50% and 90% marks.

---

in transactions of 10,000 cycles or fewer. `python`, however, has significantly longer transactions than the STAMP workloads, as over 50% of its transactions execute for a million cycles or longer.

**Transaction size.** We next examine transaction size, *i.e.* the total number of bytes that have been touched (read or written) by a transaction at the time that it commits or aborts (assuming the 64-byte block granularity of our memory system). Figure 3.9 on page 56 presents a cumulative histogram showing the percentage of transactional execution spent in transactions that access $n$ bytes or fewer. Eight of the workloads (`kmeans`, `ssca2`, `labyrinth`, `intruder` and its variants, and the optimized variants of `vacation`) spend all of their transactional execution in transactions of four

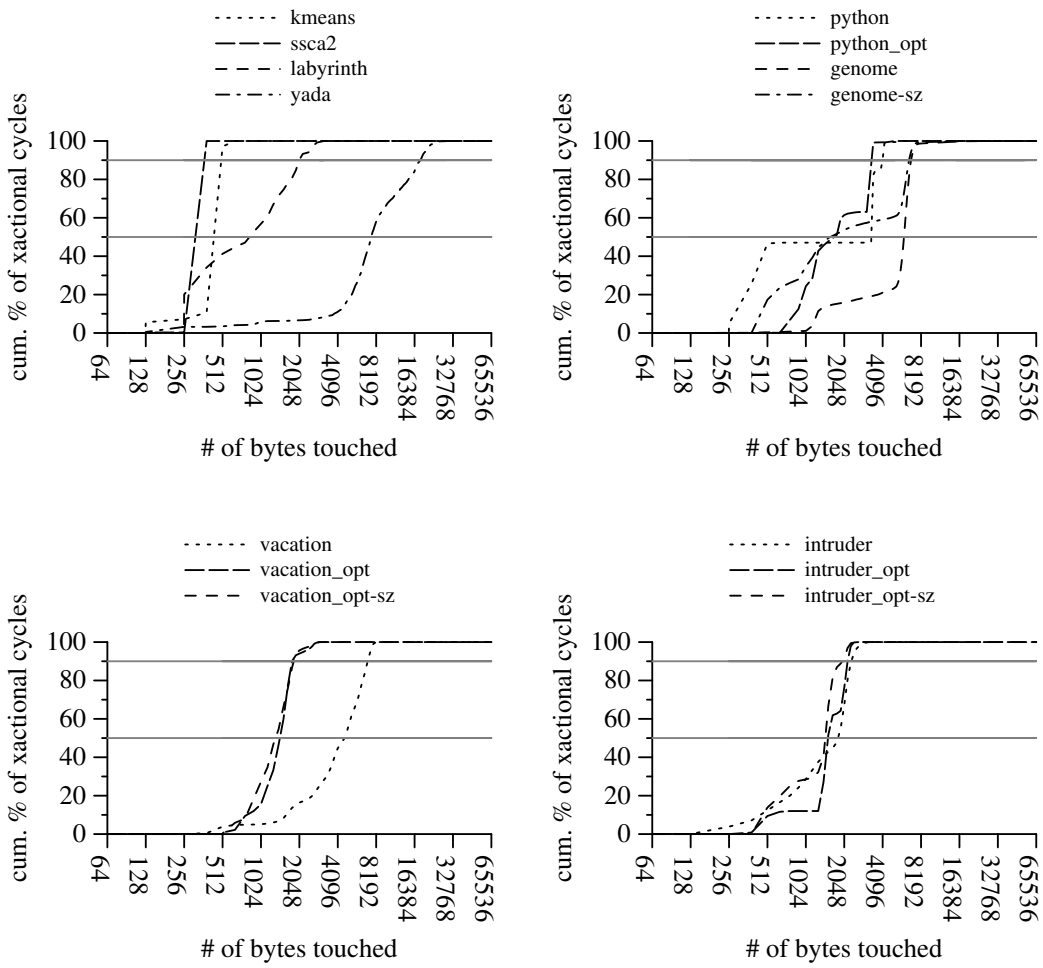


Figure 3.9: **Breakdown of transaction sizes by percent of transactional cycles.** The x-axis is the total number of bytes touched, and the y-axis is the cumulative percentage of transactional cycles spent in transactions touching that number of bytes or fewer. Horizontal lines are drawn to indicate the 50% and 90% marks.

---

kilobytes or fewer. All workloads but `yada` spend 90% of transactional execution in transactions of eight kilobytes or fewer.

Figure 3.10 on page 57 and Figure 3.11 on page 58 present similar breakdowns of transactions' read and write set sizes. The read set graphs look qualitatively similar to the graphs of total transaction sizes, indicating that little transactional data is write-only. The write set sizes, however, are considerably smaller: all workloads spend 90% of transactional execution time in transactions that write eight kilobytes or fewer, and only one workload (`yada`) writes more than four kilobytes in any transaction.

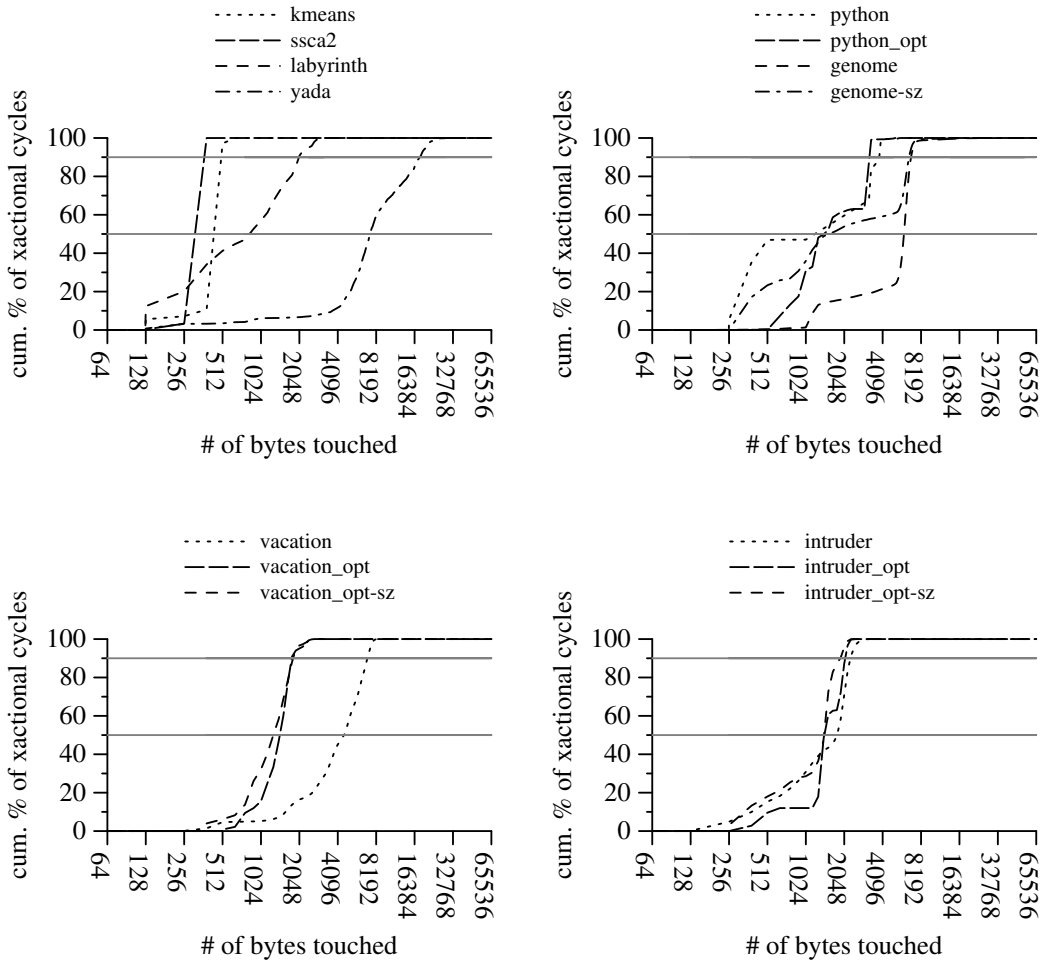


Figure 3.10: **Breakdown of transaction read set sizes by percent of transactional cycles.** The x-axis is the total number of bytes read, and the y-axis is the cumulative percentage of transactional cycles spent in transactions reading that number of bytes or fewer. Horizontal lines are drawn to indicate the 50% and 90% marks.

---

To place these results into context, Figure 3.12 on page 59 presents a breakdown of the percentage of *total* cycles (as opposed to transactional cycles) spent executing in transactions that access $n$ or fewer bytes. These graphs show that these workloads fall into three groups. The first group (`kmeans`, `labyrinth`, `ssca2`, `intruder_opt`) spend a small percentage of total execution time in transactions, which are themselves small (touch four kilobytes or fewer). The second and largest group (`intruder`, `intruder_opt`, `vacation_opt`, `vacation_opt-sz`, `python`, `python_opt`, `vacation`, `genome`, and `genome-sz`) has transactions that are still relatively small (touch eight kilobytes or fewer), but spend a large percentage of total execution time in trans-

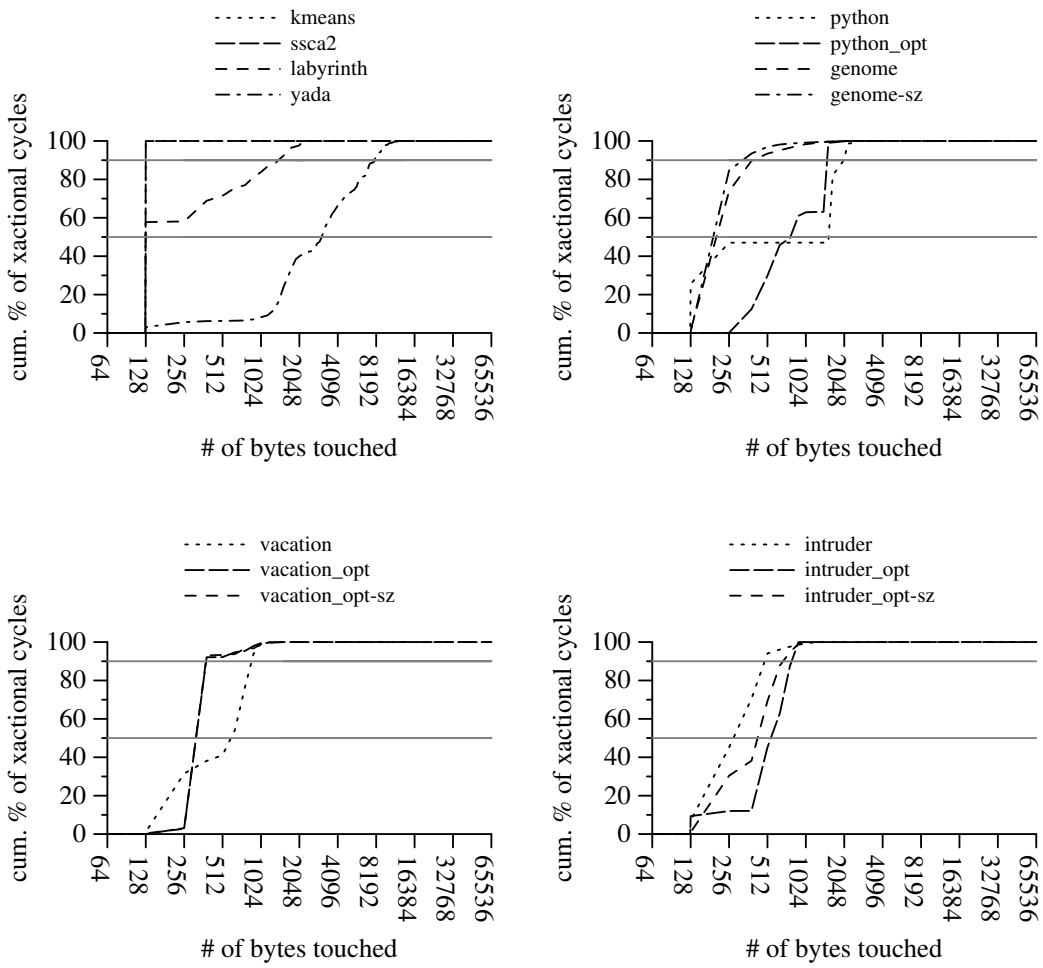


Figure 3.11: **Breakdown of transaction write set sizes by percent of transactional cycles.** The x-axis is the total number of bytes written, and the y-axis is the cumulative percentage of transactional cycles spent in transactions writing that number of bytes or fewer. Horizontal lines are drawn to indicate the 50% and 90% marks.

---

actions. Finally, `yada` spends most of its execution time in transactions, and over 50% of execution time in transactions that touch between 8 and 32 kilobytes of data.

## 3.6 Summary

In this chapter we introduced the workloads that we will use to drive and quantitatively evaluate our proposals. We found that even with a system that can handle unbounded transactions with no performance overheads, many workloads perform badly due to conflicts between transactions. We additionally showed that most of these conflicts are true conflicts. After performing straightforward

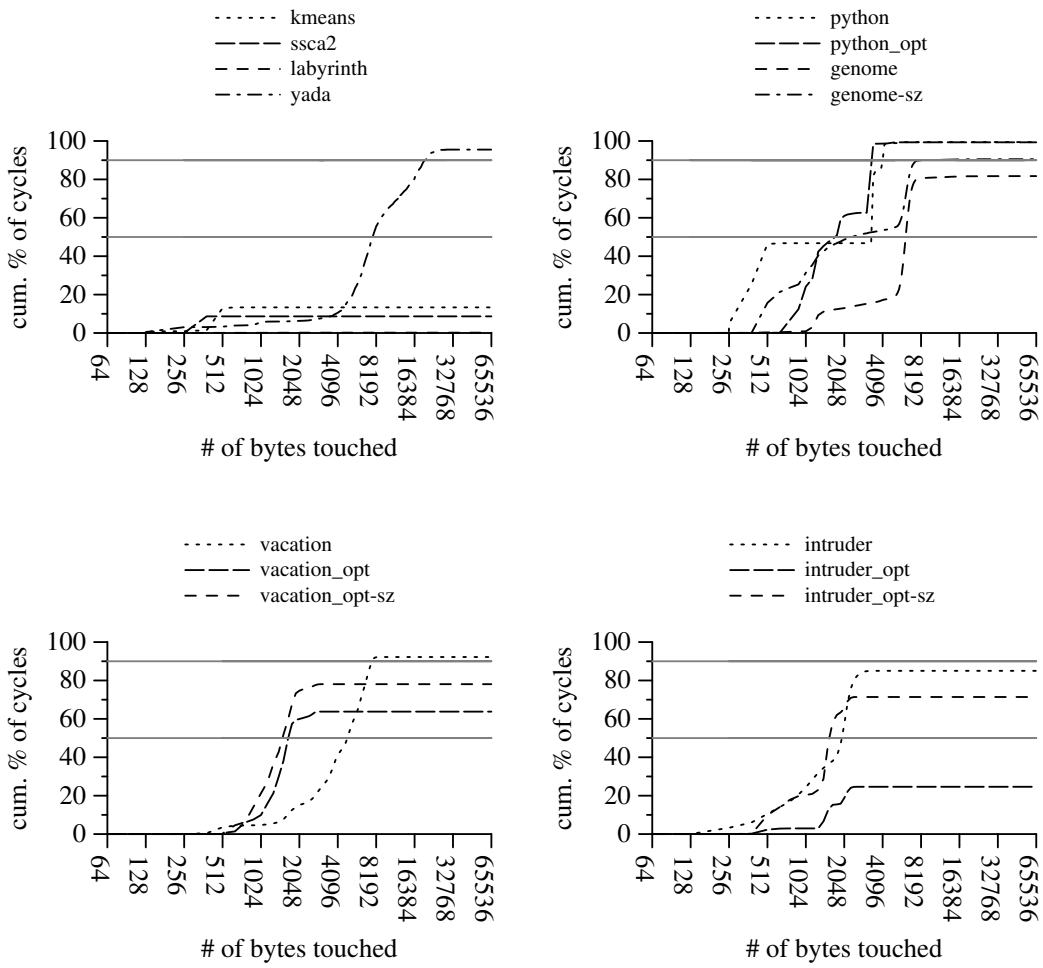


Figure 3.12: **Breakdown of transaction sizes by percent of total execution.** The x-axis is the total number of bytes touched, and the y-axis is the cumulative percentage of *total* execution time (as opposed to transactional execution time) spent in transactions touching that number of bytes or fewer. Horizontal lines are drawn to indicate the 50% and 90% marks.

---

software restructuring, we found that many of the remaining conflicts were on auxiliary data such as reference counts and hashtable occupancy fields. In the last part of this dissertation (Chapter 8 and 9), we will propose and evaluate RETCON, a hardware mechanism that increases the robustness of transactional memory to such auxiliary data conflicts.

Our first goal, however, is to devise an unbounded hardware transactional memory system that performs similarly to the idealized fully-concurrent unbounded system while seeking to retain the low complexity of the bounded hardware transactional memory. Our examination of the sizes of the transactions in these workloads in Section 3.5 shows that on these workloads, overflows will

likely be rare. However, it is also not clear whether these workloads are representative of future transactional workloads or not.

In response to the fact that the performance characteristics of future transactional workloads are not known, researchers have proposed designs that, like the idealized system, support unbounded transactions with full concurrency. Unfortunately, as Chapter 4 details, tracking the conflict detection information for an unbounded number of unbounded transactions and detecting conflicts between these transactions appears to entail significant complexity. We will instead take a decoupled approach to the problem of supporting unbounded transactions in hardware. In Chapter 5 we detail the permissions-only cache, a proposal for extending the range of bounded HTM from kilobytes to megabytes. In Chapter 6 we describe ONETM, our proposal for handling overflows of bounded HTM. ONETM exploits the fact that the permissions-only cache can likely be configured to make overflows rare in order to simplify their handling.

# Chapter 4

# Prior Approaches to Handling Overflows in Hardware

In this chapter we survey prior proposals for supporting unbounded transactions in hardware[1]. We describe six proposals: TCC [41], UTM [5], VTM [90], PTM [23], Bulk [19], and LogTM-SE [117].

TCC was the first proposal to support unbounded transactions in hardware. TCC directly implements the lazy conflict detection/lazy version management algorithm presented in Figure 2.9 on page 26: the original TCC design employed a global commit token and an update-based coherence protocol, with chunks broadcasting both addresses and data to all other chunks on acquiring commit permissions via global arbitration for the commit token [41]. The authors noted that in such a system, overflows could be handled by having transactions acquire the commit token early (similar to the handling of starvation). By doing so, the overflowed transaction prevents any other transaction from committing until it itself has committed and aborted any conflicting transactions. A subsequent design [20] refined the implementation to employ a distributed arbitration mechanism and an invalidation-based coherence protocol in which chunks send addresses but not data of write sets after committing; extending the pre-commit approach to work in this context was not discussed.

Subsequently, researchers sought to support unbounded transactions in hardware with full concurrency (*i.e.*, an unbounded number of unbounded transactions can be executing at a given time). The primary challenge in doing so is the need to detect conflicts between a potentially unbounded

[1] We defer discussion of proposals subsequent to ONETM ([14, 50]) until Section 6.4.

number of concurrently-executing transactions. Other significant challenges include the need to track an unbounded amount of conflict detection information for each transaction, the need to track an unbounded amount of version management information for each transaction, the need to support context switching of a transaction, and the need to support paging of transactionally-accessed data.

In the remainder of this chapter we describe the operation of UTM, VTM, PTM, Bulk and LogTM-SE, focusing on how each meets the challenges discussed above. In the next section we describe UTM, VTM and PTM, which utilize precise conflict detection up to cache-block granularity (similar to the bounded HTM). Bulk and LogTM-SE, by contrast, utilize *signatures* (finite-sized conservative representations of transaction read- and write-sets) for conflict detection. We describe the operation of these proposals in Section 4.2. After describing these proposals, we close this chapter by discussing their design complexity in Section 4.3.

## 4.1 UTM, VTM, and PTM

Below we describe UTM [5], VTM [90], and PTM [23]. These three proposals support supporting unbounded transactional memory in hardware with full concurrency and precise conflict detection up to cache-block granularity. Each uses the bounded HTM presented in Section 2.6 to implement transactions that do not overflow the local cache hierarchy. The proposals then add mechanisms to handle the case where a transaction overflows the local cache hierarchy (*i.e.*, the case where the bounded HTM would have to abort because it can no longer perform conflict detection for the transaction). We describe these overflow-handling mechanisms below.

**UTM.** UTM [5] implements the eager conflict detection/eager version management algorithm presented in Figure 2.8 on page 25. UTM maintains the state of overflowed transactions in a single, shared, memory-resident data structure called the xstate. The xstate structure contains (1) logs for each active transaction to record read and written addresses and the original data values at the written addresses, and (2) for each block in memory, a read/write bit and a linked list of pointers into the log entries for that block. As there can be an unbounded number of concurrently-executing transactions and each transaction can be unbounded in size, both the logs and the linked lists must also be unbounded; placing the xstate in memory supports this requirement.

On an overflow, a processor adds an entry to its log, optionally walking the list of entries for the overflowing block to avoid redundant logging. Conflicts are detected by first inspecting the

RW bit; if that bit signals a conflict, the transaction walks the linked list associated with the block to determine whether other transactions have accessed the block (*i.e.*, whether there is actually a conflict). An aborting transaction walks its list of accesses, destroying the log and reverting memory state. A committing transaction traverses the list of accesses to clean up the log. The xstate structure may be concurrently updated and read by multiple threads. To support paging of transactional data, the paper proposes that the system employ global virtual addresses, which are system-wide addresses that remain valid even if paging occurs [8, 59]. Supporting global virtual addresses typically entails an additional layer of address translation.

**VTM.** VTM [90] implements the eager conflict detection/lazy version management algorithm presented in Figure 2.7 on page 23. VTM tracks overflowed transactional state using a shared data structure mapped into the virtual address space (called the XADT). Entries in the XADT are allocated when blocks overflow the cache. Much like UTM's xstate, VTM's XADT uses linked lists and supports accessing all entries for a specific virtual memory block or all entries for a specific transaction. XADT operations include concurrently adding an entry on overflow, looking up an entry for a block, committing a transaction, aborting a transaction, and saving state on context switches. Each transactional load or store miss checks for conflicting transactional accesses before it completes. As VTM uses lazy version management, it buffers speculative updates in the XADT itself, propagating these updates only when a transaction commits. Because VTM operates on virtual rather than physical addresses, it supports paging of transactional data with no extra effort. However, this choice also complicates the task of context-switching a transaction, as discussed below.

To reduce expensive walks of the XADT, VTM introduces two caching mechanisms. First, VTM introduces a counting-Bloom-filter-based table (the XF) accessed on cache misses to quickly rule out conflicts with other transactions. Only when the XF indicates a potential conflict must the processor walk the XADT. The XF is mapped into the virtual memory space, shared among all threads, and accessed with cacheable loads and stores; as such, the XF can create overheads due to coherence sharing misses. Second, VTM employs another table, the XADC, to cache XADT entries for blocks that have been accessed by the current transaction.

On commit, VTM walks all the XADT entries for the transaction in hardware, copies the non-committed values into the memory, updates the shared XF, and deallocates and unlinks the XADT entries. Although non-transactional loads and stores do not normally need to check the XADT/XF,

they must do so when a transaction is committing. An abort similarly walks the list of entries for the aborting transaction; this walk can be done in the background.

On a context switch, VTM walks the cache and overflows any transactionally read or written blocks. As updating the XADT requires virtual addresses and most caches are physically tagged, VTM's cache is augmented with virtual address tags. When a transaction is swapped back in after a context switch, all of the values read by that transaction are validated by comparing the current value of the block with the value previously read by the transaction for the block, requiring the buffering of both reads and writes.

**PTM.** PTM [23], like UTM, implements the eager conflict detection/eager version management TM algorithm (Figure 2.7 on page 23). PTM maintains the state of overflowed transactions on a per-page basis. PTM's shadow pages behave similarly to UTM's log pointers, except that PTM's Transaction Access Vector (TAV) lists track data for an entire page. Like both UTM and VTM, the transactional state data structure supports iterating over all entries associated with both a particular memory location and a particular transaction. PTM simplifies data logging by making the observation that because only one transaction can be writing a block at a time (because of its use of eager conflict detection), one shadow copy for each block is the maximum ever needed. PTM supports paging by swapping in and out shadow pages together with their associated home pages.

In PTM, all of the transactional state is maintained and accessed at the memory controller during cache misses. The memory controller is responsible for all conflict detection, updating transactional state, and aborting/committing transactions. To avoid performing a list walk of TAVs on each cache miss, PTM employs a TAV summary cache at the memory controller (different from, but analogous to VTM's XF). When a cache block overflows the cache, it is the memory controller that is responsible for recording the original and overflowed value.

On commit, the memory controller walks and updates all of the TAVs for the transaction and updates the summary vectors. Abort is similar, but the controller copy-restores the original values. The proposal also describes an optimized version in which non-speculative blocks can reside in either the home or shadow page, with a bit vector specifying which page has the non-speculative copy of each block. In this version, the memory controller toggles the bits for transactionally-accessed blocks on commit but does not have to copy-restore blocks on abort (it still walks the TAV list for the transaction to free its entries, however). Between the time that a transaction logically

commits and completes clearing its transactional state, the transaction is marked as committed, signaling that conflicts due to the committing transaction can be ignored.

To avoid overflowing all transactional blocks on a context switch, PTM associates a transaction identifier with each block in the cache. The PTM proposal assumes that the in-cache transaction identifiers are cleared in the case when a transaction resumes execution and commits on another processor, but does not explain how that is accomplished.

## 4.2 Bulk and LogTM-SE

Below we describe Bulk [19] and LogTM-SE [117], which share the property of performing conflict detection for all transactions (whether overflowed or non-overflowed) via *signatures*, finite-sized conservative representations of transactions' read and write sets. A signature is simply a Bloom filter, *i.e.* a finite-sized table that can be queried for membership of a given address and can return false positives (but not false negatives). In these proposals, each transaction maintains a read set signature and a write set signature. On a load, the address of the load is added to the read set signature, and similarly, on a store, the address of the store is added to the write set signature. As we describe below, Bulk employs signatures to enforce lazy conflict detection, while LogTM-SE does so to enforce eager conflict detection.

**Bulk.** Bulk [19] first introduced the idea of performing conflict detection via signatures. Bulk implements the lazy conflict detection/lazy version management algorithm presented Figure 2.9 on page 26 (with a slight twist, described below). During transaction execution, transactions acquire all blocks via read requests (even for stores). Once a transaction has received permission to commit from the arbiter, it broadcasts its write signature to all other processors, which intersect this write signature with their own read and write signatures. A non-empty intersection causes the receiving processor to abort its transaction. When a processor receives such a write signature, it also invalidates all blocks in its cache whose addresses are contained in the signature, thus ensuring that it will receive the most recent copy of these blocks on next access. At this time, the directory also updates its state to make the committing transaction the exclusive owner of all blocks in its write signature. Once all processors have completed this process and sent an acknowledgement, the transaction's commit is finished and another transaction's commit can begin. To avoid arbitration for commit becoming a performance bottleneck, the arbiter can be distributed [18].

Bulk supports version management for non-overflowed transactions via cleaning (Section 2.6). As Bulk maintains the write set via a signature rather than by speculatively-written bits on L1 cache blocks, however, the process of invalidating speculatively-written blocks is somewhat more involved. In particular, Bulk must ensure that the conservative nature of signatures does not cause incorrect invalidations of blocks that were not speculatively written (but may, for example, hold non-speculative dirty data). Bulk ensures correct operation via two mechanisms. First, it builds the write signature in such a way that the set of cache indices of speculatively-written blocks can be generated exactly from the signature (the details are presented in Section 3.2 of Ceze et al. [19]). Second, Bulk restricts L1 cache sets to having dirty data that is either (a) non-speculative or (b) belongs to a single speculative transaction. On a transaction abort, Bulk generates the set of indices of cache sets containing speculatively-written blocks, and then walks through each cache set to invalidate all dirty blocks.

The use of signatures implies that unlike the work discussed above, a transaction in Bulk can let a block that has been read or written escape the cache without losing the ability to perform conflict detection on that block. However, supporting unbounded transactions in Bulk still requires solutions to the problems of unbounded version management and virtualization of transactions.

Bulk supports unbounded version management via per-thread overflow areas that reside in virtual memory. When a speculatively-written block overflows a processor's cache, it is moved to the thread's overflow area. To manage forwarding from overflowed blocks to subsequent loads of the transaction, Bulk sets an overflowed bit on an overflow. If the overflowed bit is set, transactional reads check for membership in the write signature and check the overflow area for forwarding if the signature indicates that the block may have previously been speculatively written. The process of moving writes from the overflow area back to their architectural locations on transaction commit is not explicitly described in Bulk; presumably, this occurs in between the time that a transaction acquires commit permissions and the time that it broadcasts its write signature to other processors for conflict detection. Bulk thus uses a combination of eager and lazy version management: it employs eager version management for blocks that do not overflow the cache, and lazy version management for blocks that do overflow the cache.

When a transaction is pre-empted, its read and write signatures remain active at the processor at which it had previously been executing; a new transaction on this processor is assigned a separate read and write signature (to facilitate this, each processor has several read/write signatures available

as contexts). Conflict detection at a processor is performed on all active signatures. If, however, there are no available signature contexts when a new transaction begins, then some pre-empted transaction's signature has to be moved to memory (and conflict detection still must be performed against this signature while it is in memory). In this case, the transaction's speculatively-written blocks are moved to its overflow area as well. The issue of paging of transactional data is not discussed.

**LogTM-SE.** Like Bulk, LogTM-SE [117] uses signatures to perform conflict detection. However, LogTM-SE differs from Bulk in that it implements the eager conflict detection/eager version management algorithm of Figure 2.7 on page 23. These differences arise due to the fact that LogTM-SE builds on LogTM [77], an earlier-proposed system supporting transactions that are unbounded in space but not time.

LogTM employs (and proposed) the namesake log described in Section 2.6 for version management. LogTM uses read and written bits on L1 cache lines to perform conflict detection for bounded transactions. To allow transactions to overflow the L1 cache while still supporting conflict detection, LogTM extends its baseline directory coherence with a mechanism called the "sticky state." This mechanism is simply a set of per-processor overflow bits that are associated with each block at the directory. When a given processor's overflow bit is set, the directory forwards all requests for the block to that given processor in addition to sending all other messages dictated by normal coherence protocol operation. Before a transaction allows a speculatively-accessed block to escape the L1 cache, it puts this block in the sticky state, thus ensuring that it will continue to see future requests for the block. When a processor sees a forwarded request for a given block due to the sticky state being set on that block, it conservatively performs conflict detection on the block as follows. If the processor is either not in a transaction or is in a transaction that has not overflowed the cache, then it can infer that the sticky state was set by a no-longer-active transaction. In that case, the processor simply clears the sticky state at the directory. However, if an overflowed transaction is currently executing on the processor, then a conflict is conservatively signaled.

LogTM's conflict detection and version management mechanisms together allow for transactions that are unbounded in space. However, LogTM does not support the suspension of transactions, as its conflict detection logic implicitly assumes that any transaction that previously set a

sticky state for a given processor but is not currently executing at that processor has either committed or aborted. Thus, transactions in LogTM are still bounded in duration.

LogTM-SE supports transactions that are unbounded in time by modifying LogTM to perform conflict detection through signatures rather than bits in the cache. LogTM-SE employs a summary signature to perform conflict detection on transactions that are switched out (*i.e.*, not presently executing on any processor). All processors perform conflict detection against this summary signature. When a conflict is detected against the summary signature, LogTM-SE traps to a software handler to resolve the conflict. When a transaction is context-switched, LogTM-SE adds its signature to the summary signature via an inclusive-or operation. To support removing a transaction's signature from the summary signature on commit, the summary signature maintains a count of how many transactions have set each bit.

LogTM-SE supports paging of transactional data as follows. When moving a page from one page context to another, the system first interrupts each processor and has it walk over the old page testing whether it contains each block. If so, the processor adds the new address of the block to its signature. The authors note that this process could potentially be optimized if it becomes a performance bottleneck.

## 4.3 Discussion

As discussed earlier, UTM, VTM, PTM, Bulk and LogTM-SE all support transactions of unbounded size and duration with full concurrency. Achieving this goal, however, comes at a complexity cost to both conflict detection and version management.

The primary challenge in conflict detection is *the need to detect and resolve conflicts between an unbounded number of transactions*. The set of proposals discussed in Section 4.1 require the hardware to dynamically allocate/deallocate, maintain, and concurrently manipulate complex link-based structures (UTM's xstate, VTM's XADT, and PTM's Transaction Access Vectors) and the corresponding cached versions of these structures. Manipulating and accessing these structures can add overhead to both overflowed transactions and concurrently-executing non-overflowed transactions, which need to access these structures to perform conflict detection. More importantly, the hardware for correctly manipulating these structures is not simple. While the signature-based

proposals discussed in Section 4.2 avoid these link-based structures, they must still traverse the signatures of all executing transactions for conflict detection and/or conflict resolution.

The support of multiple concurrently-executing unbounded transactions has implications on version management as well. In a conflict between two unbounded transactions, it may be necessary to abort one in order to maintain forward progress (*e.g.*, in the case of a cyclic pattern of data accesses that would otherwise result in each transaction stalling the other). This fact implies that any system supporting the concurrent execution of more than one unbounded transaction must support unbounded version management as well.

In the next two chapters we will propose a different approach to unbounded hardware transactional memory. First, we propose a hardware mechanism, the *permissions-only cache*, that reduces the rate at which less-efficient overflow handling mechanisms are invoked. With the knowledge that overflows will likely be rare we propose ONETM, a system for handling unbounded transactions that revisits the idea of supporting one overflowed transaction at a time.

# Chapter 5

# The Permissions-Only Cache: Reducing the Frequency of Overflows

In this chapter we introduce a mechanism whose goal is to reduce the frequency with which transactions overflow the bounded HTM presented in Section 2.6.4. This mechanism, the *permissions-only cache*, expands the range of transactions that the HTM can support from tens of kilobytes (*i.e.*, the size of the L1 cache) to megabytes. Overflows of this system can still occur, meaning that this system does not yet support transactions that are unbounded in size. However, by decreasing the frequency of overflows, the permissions-only cache will allow us to consider mechanisms for handling these overflows that trade off performance for implementation simplicity (Chapter 6).

In seeking to reduce the frequency of overflows, we first re-examine why overflows are a problem. In Section 2.6.6, we noted that for the bounded HTM to be able to perform conflict detection on a given block, it must (a) have coherence permissions to the block (in order to be guaranteed to see conflicting requests) and (b) have the read and written bits for the block (to be able to determine what constitutes a conflict). When a transactionally-accessed block overflows the L1 cache, the cache loses both coherence permissions and the transactionally-accessed bits for the block. Hence, it can no longer detect conflicts on that block and must abort the transaction running on that processor.

The above observations conversely suggest that if a transaction had a way of *retaining* coherence permissions and the read/written bits for a given block when it escapes the cache, the transaction could continue executing. This observation, coupled with the observation that it is *not* necessary to

have the data for a block in order to perform conflict detection on that block, serves as the foundation of the permissions-only cache. The permissions-only cache supports conflict detection on blocks that have been replaced from the processor's data cache by retaining coherence permissions and transactionally-accessed bits — but not data — for these blocks. Because the permissions-only cache needs to track only two bits per data block, it can track conflict detection information for a large number of memory blocks with a comparatively small amount of storage; for example, if data blocks are 64 bytes, the permissions-only cache can track the transactionally-accessed bits for 256 blocks in the same amount of space occupied by the data of a single block. Only when the permissions-only cache itself overflows does the system need to fall back on some other mechanism for detecting conflicts for overflowed blocks.

In the remainder of this chapter we present the permissions-only cache in detail. We first describe its basic operation in the next section. In Section 5.2 we discuss how the permissions-only cache can be organized to efficiently encode the conflict detection information of a large amount of state. In Section 5.3 we discuss how the range of the permissions-only cache can be expanded further by maintaining its information in the L2 cache rather than a dedicated structure. Finally, we discuss related work in Section 5.4 and conclude the chapter in Section 5.5.

## 5.1 Operation

The permissions-only cache tracks conflict information for blocks that have exceeded the capacity of the data cache. Like the data cache, it is organized as a tagged, set-associative structure. Each entry contains a read bit and optionally a write bit. Below, we first describe the usage of the permissions-only cache to allow read sets to grow beyond the size of the L1 cache without overflow. We then discuss how the coupling of the permissions-only cache and a mechanism that supports version management of transactionally-written blocks that escape the L1 cache (such as the log discussed in Section 2.6.3) can also allow the write set to grow beyond the size of the L1 cache. After discussing how aborts can still occur due to the permissions-only cache itself overflowing, we present the complete permissions-only cache algorithms. We close this section by discussing tradeoffs between this approach and extending the L2 cache with transactionally-accessed bits.

**Allowing transactionally-read blocks to escape the cache.** When a block that has been transactionally read (but not written) is replaced from the data cache, the cache controller sets the appro-

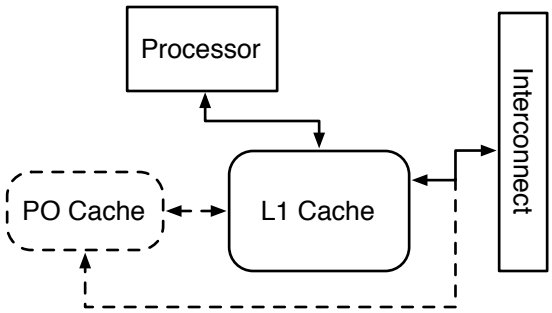


Figure 5.1: **Incorporation of the permissions-only cache into the system.** The solid lines show communication between existing system components: the processor and L1 cache communicate on local requests, while the L1 cache and the interconnect communicate for remote requests, fills of those requests, and incoming invalidations. The dashed lines show communication between the permissions-only cache and the rest of the system. As detailed in Figure 5.3 on page 75, the permissions-only cache communicates with the L1 cache on L1 cache evictions and fills, and it communicates with the interconnect on incoming invalidations. The permissions-only cache does not directly communicate with the processor.

priate read bit in the permissions-only cache, allocating an entry if necessary. The cache controller does *not* give up coherence permissions to the block in this case. Instead, the coherence state for the block is now *shared without data*. Externally, this state is indistinguishable from the traditional *shared* ($S$) coherence state. If the processor later brings such a block back into the L1 cache, the transactionally-accessed bits from the block's permissions-only cache entry are copied back over to the L1 cache entry.

By maintaining coherence permissions to the block, the cache also maintains the guarantee of seeing all potentially-conflicting requests to the block. To detect conflicts on blocks in the permissions-only cache, external requests check the read bit in the permissions-only cache in addition to the bits in the data cache (the two checks are performed in parallel). On a transaction commit or abort, the permissions-only cache is cleared by flash-invalidating all its blocks. Existing protocols commonly allow silent replacement of $S$ blocks, and thus already implicitly support this operation.

Figure 5.1 on page 72 details the incorporation of the permissions-only cache into a conventional multiprocessor. As discussed above, the permissions-only cache communicates with the L1 cache on evictions and fills, and with the interconnect on incoming invalidations. Local memory operations do not access the permissions-only cache.

We note that it may occur that a block is both transactionally-read and non-transactionally dirty. In this case, the cache controller both writes back the dirty data and maintains read permissions to the block. This operation is conceptually similar to the cleaning operation discussed in Section 2.6.3. We also note that in systems enforcing coherence inclusion, all blocks in the permissions-only cache would also have to be present in the L2 cache.

**Allowing transactionally-written blocks to escape the cache.** If the bounded HTM uses a log for version management (Figure 2.12 on page 36), the permissions-only cache can allow transactionally-written blocks as well as transactionally-read blocks to escape the L1's cache. When evicting a transactionally-written block, the block is written back to the second-level cache or memory and the appropriate written bit is set in the permissions-only cache. Dirty blocks may safely escape because any remote read to these addresses will conflict with the written bit in the permissions-only cache, preventing any access to the block until the subsequent abort has successfully restored the pre-transactional value (which is maintained in the log). When a block's written bit is set in the permissions-only cache, the local coherence state of the block is *clean-exclusive without data*, externally indistinguishable from the *clean-exclusive* ($E$) state (which may be replaced silently similarly to the $S$ state).

If using the cleaning version management scheme presented in Section 2.6.3, speculatively-written blocks cannot escape the L1 cache because the speculative data would overwrite the non-speculative data preserved in the lower levels of the memory hierarchy. In this case, the permissions-only cache would be used to hold transactionally-read data only. We call this variant a *read-only permissions-only cache*, as distinguished from the *read-write permissions-only cache* discussed above[1].

**Overflows of the permissions-only cache.** As the permissions-only cache is a finite-sized structure, it is possible that it could also overflow. In this case, the transaction would have to abort. Thus, as noted above, while the permissions-only cache extends the range of bounded HTM, it does not provide support for unbounded transactions. In Chapter 6 we propose complementary hardware support for unbounded transactions. Our proposal exploits the fact that the permissions-only cache will likely make overflows rare in order to simplify their handling.

[1] It would be possible to employ a read-write permissions-only cache in this situation by adding a separate structure that holds the data of transactionally-written blocks that have overflowed. We do not explore such a configuration in this dissertation.

**Adding a Read-Only Permissions-Only Cache to Bounded HTM**

**abort**

```
flash-invalidate writes
flash-clear written bits
flash-clear read bits
flash-invalidate POcache
in_transaction = false
```

**commit**

```
flash-clear written bits
flash-clear read bits
flash-invalidate POcache
in_transaction = false
```

**evict(A)**

```
if Cache[A].written_bit
  abort()
if Cache[A].read_bit
  move_to_POCache(A)
if Cache[A].dirty:
  write back A
Cache[A].valid = false
```

**handle_downgrade_request(A,ts)**

```
if Cache[A].written_bit:
  resolve_conflict(A,ts)
if Cache[A].state == M:
  Cache[A].state = S
send acknowledgement
```

**handle_invalidate_request(A,ts)**

```
if Cache[A].read_bit or
   Cache[A].written_bit or
   POCache[A].read_bit:
  resolve_conflict(A,ts)
Cache[A].state = I
send acknowledgement
```

**move_to_POCache(A)**

```
if no entry for A in POCache:
  abort()
POCache[A].read_bit =
  Cache[A].read_bit
```

Figure 5.2: **Adding a read-only permissions-only cache to bounded HTM.** This figure shows the incorporation of a read-only permissions-only cache into a bounded HTM using cleaning for version management (presented in Figure 2.11 on page 35). On eviction of a transactionally-read block, the block's transactionally-read bit is moved to the permissions-only cache. Because speculatively-written data cannot be stored to lower levels of memory, the the bounded HTM must still abort when evicting a transactionally-written block from the cache. In addition, the transaction aborts if there is no free permissions-only cache entry when evicting a transactionally-read block. When handling an external coherence request for a block, the permissions-only cache is checked for a conflict in parallel with the regular cache. At commit and abort, the permissions-only cache is flash-invalidated. Functions not shown are unchanged from Figure 2.11 on page 35.

**Algorithms.** Figure 5.2 on page 74 shows the incorporation of a read-only permissions-only cache into a bounded HTM that uses cleaning for version management. The key additions are (1) the use of the permissions-only cache to hold transactionally-accessed bits when a block is evicted from the L1 cache (`move_to_POCache`) and (2) the corresponding checking of the permissions-only cache in conflict detection (`handle_downgrade_request` and `handle_invalidate_request`). Figure 5.3 on page 75 presents the corresponding incorporation of a read-write permissions-only cache into a bounded HTM that uses a log for version management (Figure 2.12 on page 36).

**Adding a Read-Write Permissions-Only Cache to Bounded HTM**

**abort**

```
flash-invalidate writes
flash-clear written bits
flash-clear read bits
flash-invalidate POcache
in_transaction = false
```

**commit**

```
flash-clear written bits
flash-clear read bits
flash-invalidate POcache
in_transaction = false
```

**evict(A)**

```
if (Cache[A].read_bit) or
   (Cache[A].written_bit)
  move_to_POCache(A)
if Cache[A].dirty:
  write back A
Cache[A].valid = false
```

**handle_downgrade_request(A,ts)**

```
if Cache[A].written_bit or
   POCache[A].written_bit:
  resolve_conflict(A,ts)
if Cache[A].state == M:
  Cache[A].state = S
send acknowledgement
```

**handle_invalidate_request(A,ts)**

```
if Cache[A].read_bit or
   Cache[A].written_bit or
   POCache[A].read_bit or
   POCache[A].written_bit:
  resolve_conflict(A,ts)
Cache[A].state = I
send acknowledgement
```

**move_to_POCache(A)**

```
if no entry for A in POCache:
  abort()
POCache[A].read_bit =
  Cache[A].read_bit
POCache[A].written_bit =
  Cache[A].written_bit
```

Figure 5.3: **Adding a read-write permissions-only cache to bounded HTM.** This figure shows the incorporation of a read-write permissions-only cache into a bounded HTM using log-based version management (presented in Figure 2.12 on page 36). This algorithm is similar to Figure 5.2 on page 74. However, because the log is used for version management, both transactionally-read and transactionally-written blocks can be placed into the permissions-only cache. Functions not shown are unchanged from Figure 2.12 on page 36.

**Comparison to putting transactionally-accessed bits in the L2 cache.** An alternative way to extend the range of bounded hardware transactional memory would be to allow transactionally-accessed data to reside in the L2 cache. In this case, L2 cache lines would be extended with read and written bits. While certainly viable, this approach has two potential disadvantages. First, the L2 cache may be shared between cores. In this case, each core would have to have its own pair of read and written bits. Second, as L2 caches are much larger than L1 caches (as well as the permissions-only cache), supporting the flash-clear and conditional flash-invalidate operations may become expensive. Finally, invalidating speculatively-written blocks from the L2 on abort (as would

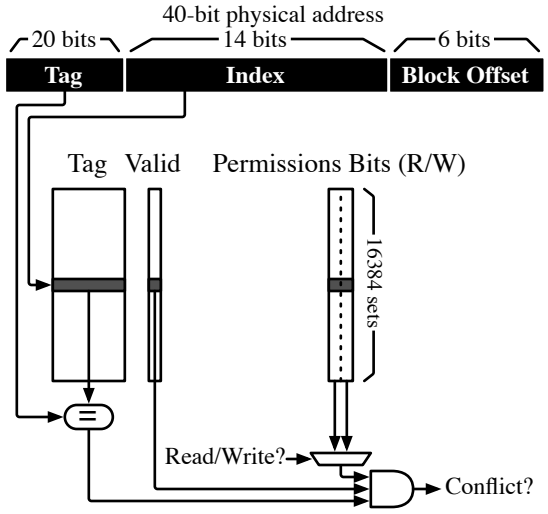


Figure 5.4: **Naive organization of a 4KB permissions-only cache.** Each entry requires two bits (read bit and written bit). Consequently, this 4KB cache has 16,384 entries (32,768 bits ÷ 2), *i.e.* it can store conflict detection information for 16,384 datablocks. Assuming 64-byte datablocks, the cache is thus able to store the information for one megabyte of data (16,384 datablocks × 64 bytes per datablock). However, in this naive organization the tags consume 40 kilobytes (16,384 entries × 20 bits per entry). We show how this overhead can be lowered via a sector cache organization in Figure 5.5 on page 77.

be done if cleaning is used as the version management mechanism) would result in costly misses to memory the next time that these blocks are accessed unless the L2 is backed by an L3.

## 5.2 Efficient Encoding

Because the permissions-only cache does not contain data, it can more efficiently encode the transactional read/write bits (just a few bits per block) than other on-chip caches that hold data as well as addresses. Figure 5.4 on page 76 illustrates the potential for space reduction: assuming 64-byte blocks, a 4-kilobyte permissions-only cache can hold conflict detection information for one megabyte of data.

A naive implementation of a permissions-only cache, however, would also incur the overhead of a full cache tag for each two-bit entry. For example, assuming 40-bit physical addresses as in Figure 5.4 on page 76, the 4-kilobyte cache would have a 40-kilobyte overhead for the tags. If the physical address space is larger, the tag overhead would correspondingly grow.

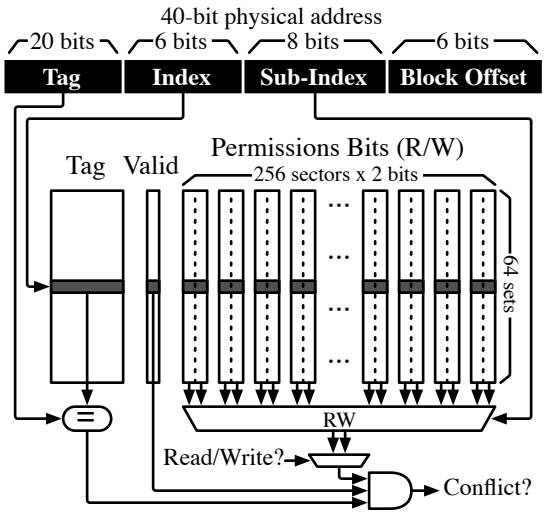


Figure 5.5: **4KB direct-mapped permissions-only cache in a 256-sector organization.** Each entry is a sector containing the read and written bits for 256 datablocks (512 bits or 64 bytes in total). The low-order bits of the index are now a sub-index that is used to offset into the sector. As each entry contains 64 bytes, the cache contains 64 entries. Compared to Figure 5.4 on page 76, the tag overhead has been reduced from 40 kilobytes to 160 bytes (20 bits $\times$ 64 entries). The tradeoff is the possible loss of coverage if there is poor spatial locality.

By using sector cache techniques [64] the tag overhead of the permissions-only cache can be reduced dramatically. A 512-bit (*i.e.*, 64-byte) entry per tag would provide for 256 two-bit sectors (containing a read bit and a write bit). The low-order bits of the cache index would be used to offset into the sector to find the relevant pair of read/write bits (Figure 5.5 on page 77). To ensure that stale data from previous transactions does not cause erroneous conflicts, a sector would be cleared before it is used for the first time in a transaction.

The advantage of this organization is that single 256-sector entry maps a contiguous 16KB region of memory (256 sectors $\times$ 64B cache lines), a 256-to-1 compression ratio in the best case. Figure 5.5 illustrates the reduction in tag overhead afforded by the sector cache organization: in this example, the tag overhead is reduced from 40 kilobytes (Figure 5.4 on page 76) to 160 bytes. The appeal of organizing the permissions-only cache as a sector cache is increased by the fact that it does not have to support eviction (as any cache conflict in the permissions-only cache forces a transaction abort), which is more complex in a sector cache than a conventional cache.

The tradeoff of the sector cache organization is the possibility of reduced coverage if spatial locality is poor. In the worst case, a 4KB permissions-only cache would track the read/write bits for

only 4KBs of blocks. However, with good page-level spatial locality, this cache would maintain the property of being able to track a megabyte of data.

## 5.3 Employing the L2 Cache to Store Permissions-Only Information

To support even larger transactions without overflow, instead of using a dedicated structure, the processor could dynamically share the second-level cache's storage capacity by allowing second-level cache frames to contain either data *or* an array of read/write bits. A second valid bit—a *permissions-only valid bit*—would be added to each entry's cache tag to indicate when the frame holds transactional read/write bits. When a transactional block is replaced from the data cache, its transactional read/write bits would be updated in the corresponding bits in the second-level cache's data array (replacing and allocating another entry as needed). On a commit or abort of a transaction, all the read/write bits would be discarded by flash-clearing the permissions-only valid bits. For shared second-level caches, an additional core identifier field could be associated with each cache frame containing permissions data.

When external coherence invalidations query the second-level cache, the cache tags would be accessed twice. The first tag lookup is the normal lookup, but it would match only for frames that hold data blocks (by checking the permissions-only valid bit). The second tag lookup—which would use the sector cache indexing similar to a stand-alone permissions-only cache—would check for matching frames of read/write bits. If a tag hit occurred on a tag for a frame of read/write bits, the data array would be accessed to query the corresponding bit (sector) to detect conflicts. If no permissions-only blocks had been allocated in the second-level cache, the second lookup could be skipped.

With such an organization, a 4MB second-level cache with 64-byte blocks could hold enough permissions-only information to allow a transaction to access up to 1GB of data (64K entries of 256 read/write bit pairs and each entry maps 16KBs) without overflow.

## 5.4 Related Work

As further discussed in Chapter 4, signatures [19] provide conflict detection via finite-sized Bloom filters that are conservative representations of a transaction's read and write set. Signatures are

similar to the permissions-only cache in that they both allow a transaction's read set to be larger than its L1 cache. However, the two approaches have significant tradeoffs due to the fact that entries in the permissions-only cache are tagged and thus can be identified precisely. The permissions-only cache thus maintains the property of precise conflict detection up until it itself overflows, at which point a different conflict detection scheme needs to be invoked. By contrast, signatures provide a uniform scheme for all transactions; however, research indicates that the imprecision that they introduce can cause significant numbers of false conflicts as transactions grow in size [14, 119].

The efficient data-less encoding of coherence permissions employed by the permissions-only cache is similar to the Store Miss Accelerator [22], which retains exclusive coherence permission to evicted blocks. Whereas the purpose of the permissions-only cache is to avoid transaction overflows, the Store Miss Accelerator aims to avoid incurring the latency of invalidation requests on a store that misses in the local cache hierarchy and is not present in any other cache in the system.

## 5.5 Discussion

By efficiently tracking transactions' read and write sets, the permissions-only cache increases the size of transactions that can successfully complete without invoking an overflowed execution mode, largely independent of the particular scheme used to handle overflows. This reduction in the frequency of overflows may reduce the runtime overheads of previously-proposed hardware-based unbounded transactional memory schemes that employ a higher-overhead conflict detection mechanism for transactions that overflow the local cache hierarchy such as UTM [5], VTM [90], and PTM [23] (Chapter 4). With a sufficiently large permissions-only cache, however, the occurrence of overflowed transactions will likely be rare. In the next chapter we propose ONETM, a novel approach for handling overflows that exploits this assumption with the goal of reducing hardware complexity.

# Chapter 6

# ONETM: Handling Overflows via Selective Serialization

In this chapter we propose ONETM, a transactional memory system in which only a single overflowed transaction per process can be active at a time. The principal advantage of this approach is that it has the potential to significantly ease implementation requirements by eliminating prior proposals' need to perform conflict detection between an unbounded number of unbounded transactions. We present two instantiations of ONETM, ONETM-Serialized and ONETM-Concurrent. In Figure 6.1, we illustrate the differences in concurrency between these systems as well as a system that places no concurrency restrictions on overflowed transactions (such as the systems described in Chapter 4).

The first instantiation of ONETM that we present is ONETM-Serialized, in which overflowed transactions serialize the system. To support overflowed transactions, ONETM-Serialized adds a bit called the *overflowed bit* to the bounded hardware transactional memory presented in Section 2.6 as well as machinery to manipulate this bit. Overflowed transactions set this bit, while non-overflowed transactions check the bit and stall if it is set. The overflowed transaction is assigned highest priority in conflict resolution, meaning that it will not abort due to a conflict and thus eliminating prior proposals' need to support unbounded version management.

While simple, the overflow handling mechanism of ONETM-Serialized can result in performance degradation if the number of overflows is not negligible. We thus also propose ONETM-Concurrent, a system that supports more concurrency on overflows than ONETM-Serialized while

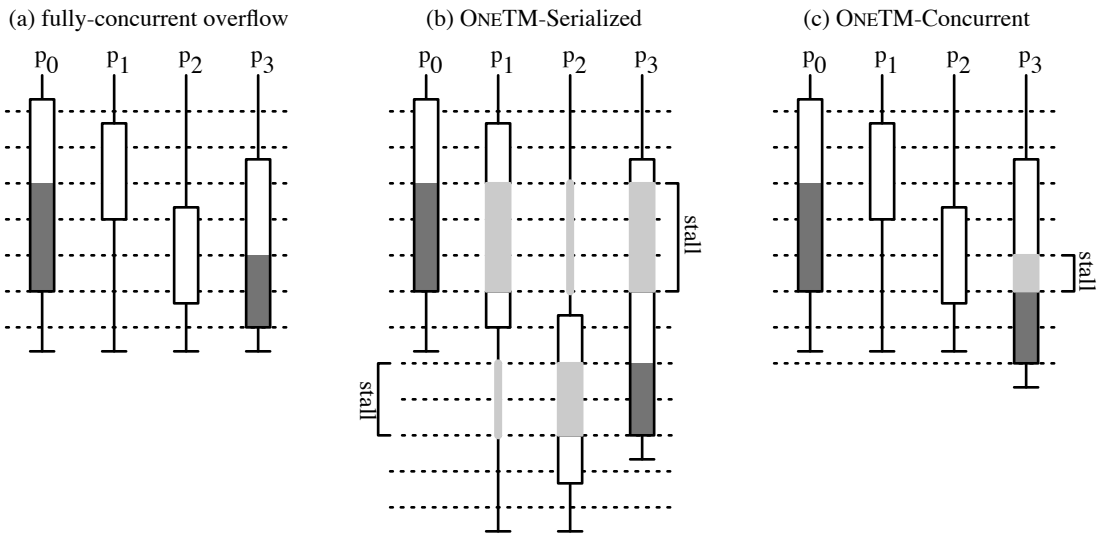


Figure 6.1: **An example execution on three systems for handling overflowed transactions.** The white bars represent non-overflowed transactions, the dark gray bars overflowed transactions, and the straight lines non-transactional execution. A light gray color means that the processor is stalled. In this execution, there are no conflicts, and the same amount of useful work is performed on each system. The example assumes strong atomicity.

still having fewer structures and mechanisms than the systems described in Chapter 4. In ONETM-Concurrent any number of non-overflowed transactions (as well as non-transactional code) are permitted to execute concurrently with the single overflowed transaction. To support this property, it associates a single pair of read and written bits with each memory block, analogous to the bounded HTM's pair of transactionally-accessed bits per cache block. These bits travel coherently with data. The overflowed transaction sets them on an access and other transactions check them to determine conflicts.

Section 6.1 and Section 6.2 respectively describe ONETM-Serialized and ONETM-Concurrent, including qualitative comparisons of these proposals to the prior work discussed in Chapter 4. In Section 6.3, we discuss how ONETM handles the advanced semantic issues discussed in Section 2.2.2. Finally, Section 6.4 discusses work published subsequently to the ISCA 2007 paper that introduced ONETM [9] and Section 6.5 summarizes the chapter.

**(a) Shared Transaction Status Word (STSW)**
Fixed address in virtual memory

| STSW Field | Description |
|---|---|
| overflowed | is an overflowed transaction active? |
| OTID | ID of active overflowed transaction |

**(b) Private Transaction Status Word (PTSW)**
Per-thread architected register

| PTSW Field | Description |
|---|---|
| overflowed | is this thread in an overflowed transaction? |
| TND | nesting depth of current transaction |

Figure 6.2: **Description of transaction status words.**

## 6.1 ONETM-Serialized

Our first implementation, ONETM-Serialized, revisits the idea of serialization of overflowed transactions first proposed in TCC [39] (described in Chapter 4), adapting this idea to work within the context of the conventional memory system used by our baseline multiprocessor. ONETM-Serialized stalls all other threads in an application when one of the threads needs to execute an overflowed transaction, as illustrated in Figure 6.1b (threads executing non-transactionally stall to maintain strong atomicity [10]). In this section we first describe the structures that ONETM-Serialized requires and its operation using these structures. We then discuss the required operation system involvement. Finally, we close the section with a summary.

### 6.1.1 Structures

ONETM-Serialized employs the bounded hardware transactional memory presented in Section 2.6 as a foundation. To support overflowed transactions, ONETM-Serialized adds two *transaction status words*, the *shared (per-process) transaction status word* (STSW) and the *private (per-thread) transaction status word* (PTSW). The STSW resides in a fixed location in the virtual address space of each process and contains an *overflowed* bit, which is set while any thread in the application is executing an overflowed transaction. The PTSW, by contrast, is an architected machine register (*i.e.*, it persists across context switches because the operating system saves and restores this register along with all the other architected registers). The PTSW also contains an *overflowed* bit (set only when the *current* thread is executing an overflowed transaction), as well as a *transaction*

**ONETM-Serialized Algorithm**

**load(A)**

```
while STSW.overflowed and
      not PTSW.overflowed:
  stall
if Cache[A].state == I:
  obtain_permissions(A,read)
if in_transaction:
  Cache[A].read_bit = true
return Cache[A].data
```

**store(A,v)**

```
while STSW.overflowed and
      not PTSW.overflowed:
  stall
if Cache[A].state != M:
  obtain_permissions(A,write)
if (in_transaction) and
   (not Cache[A].written_bit):
  write back A
  Cache[A].written_bit = true
Cache[A].data = v
```

**commit**

```
while STSW.overflowed and
      not PTSW.overflowed:
  stall
flash-clear written bits
flash-clear read bits
if PTSW.overflowed:
  PTSW.overflowed = false
  STSW.overflowed = false
in_transaction = false
```

**evict(A)**

```
if (Cache[A].read_bit) or
   (Cache[A].written_bit)
  if not PTSW.overflowed:
    while STSW.overflowed:
      stall
    atomically set STSW.overflowed
    set PTSW.overflowed
    timestamp = -1
if Cache[A].dirty:
  write back A
Cache[A].valid = false
```

Figure 6.3: **ONETM-Serialized algorithm** The algorithm builds on the bounded hardware transactional memory presented in Figure 2.11 on page 35 by using the structures detailed in Figure 6.2 on page 82 to implement overflow handling. On an overflow, a transaction stalls until the overflowed bit of the STSW is clear and then atomically sets this bit. Other processors check the overflowed bit, stalling if it is set and not resuming execution until it is cleared as part of the commit of the overflowed transaction. Functions not shown are unchanged from Figure 2.11 on page 35.

*nesting depth* (TND) field (the overflowed analogue of the TND field described in Section 2.6). These status words are summarized in Figure 6.2 (the OTID field of the STSW will be introduced in Section 6.2.2).

### 6.1.2 Operation

Here we describe the basic operation of ONETM-Serialized, including when and how a transaction transitions to overflowed mode, how the system maintains isolation while an overflowed transaction is executing, and commit of overflowed transactions. Figure 6.3 on page 83 presents the extensions that ONETM-Serialized makes to the bounded HTM algorithm from Figure 2.11 on page 35 in order to support overflowed transactions.

**When to transition to overflowed mode.** A transaction transitions to overflowed mode when it has to evict a transactionally-accessed block. In addition, a transaction that experiences an interrupt during execution aborts and restarts execution in overflowed mode.

**Transitioning to overflowed mode.** To transition to overflowed execution, the processor must ensure that no other thread in the application is executing in overflowed mode. The key to enforcing this property is the overflowed bit of the STSW, which acts much like a mutex lock on overflowed execution. A transaction may only transition to overflowed execution after it has atomically changed the bit from unset to set. When transitioning to overflowed execution, the transaction also sets the overflowed bit of the PTSW. The TND field of the PTSW is used to implement the subsumption of nested transactions (*i.e.*, nested transaction initiation and commit are treated as no-ops, except for the manipulation of the TND field). When a transaction transitions to overflowed mode, it sets its timestamp to $-1$, *i.e.* the lowest timestamp in the system. This action ensures that it will be prioritized in conflict resolution. In Figure 6.3 on page 83, `evict` details the transition to overflowed mode when evicting a transactionally-accessed block.

**Maintaining isolation.** To serialize execution during overflow, all threads in the application that are not executing an overflowed transaction monitor the overflowed bit in the STSW and stall if it is set (`load`, `store`, and `commit` in Figure 6.3 on page 83). To prevent overflowed transactions' STSW accesses causing all concurrently-executing bounded transactions to abort, STSW accesses are not made part of a transaction's read or write sets. Nonetheless, while an overflowed transaction is executing, it may conflict with a (stalled) bounded transaction. Any such conflict will be detected locally by the stalled bounded transaction. Because the overflowed transaction is guaranteed to have a lower timestamp, the conflict will be resolved by aborting the bounded transaction.

To make processors' reads of the STSW's overflowed bit inexpensive, the STSW can be coherently cached in a special register. Rather than having the processor read this register on all memory

accesses, external write requests would snoop the register. A write request to the STSW would trigger a pipeline flush, at which point the local processor would have to re-acquire read permissions to the STSW before resuming execution (stalling if it sees the overflowed bit as set after the re-acquire). This implementation is similar to that used to support speculative out-of-order execution of loads in current multiprocessors [35].

**Committing an overflowed transaction.** An overflowed transaction clears the overflowed bits in the STSW and the PTSW as part of a non-nested commit. This operation unstalls all other processors.

**Discussion.** The STSW and PTSW together provide support for virtualization of overflowed transactions. If an overflowed transaction is context-switched out, the other threads continue to stall on the overflowed bit in the STSW. Because the PTSW persists across context switches and migrations, a thread will not forget that it is executing an overflowed transaction nor the nesting depth of the current transaction.

ONETM-Serialized also supports paging of transactionally-accessed data. The potential danger here is that a bounded transaction could access a given virtual memory address, the address be remapped to a different physical address, and then the overflowed transaction access the *new* physical address. In this case, the conflict between the bounded and overflowed transaction would not be detected. To eliminate this case, when a page of virtual memory is remapped, all currently-executing bounded transactions are aborted as part of the process of invalidating the current mapping from processors' translation lookaside buffers (the so-called "TLB shootdown"). If a transaction is continually aborted in such a fashion, it can restart execution in overflowed mode (similar to the handling of interrupts within a transaction).

### 6.1.3 Runtime Involvement

The runtime must inform each processor of the location of the STSW. In addition, the OS must save and restore the PTSW on context switches as with all other architected registers.

### 6.1.4 ONETM-Serialized Summary

ONETM-Serialized extends the bounded HTM described in Section 2.6 to support unbounded transactions by adding the STSW and the PTSW (Figure 6.2 on page 82). The price of this relatively

small change over the baseline, however, is the loss of all concurrency when a transaction overflows, which will have a significant negative impact on performance if overflows are frequent. We next propose ONETM-Concurrent, an implementation that allows concurrent execution of non-overflowed transactions, non-transactional code, and a single overflowed transaction.

## 6.2 ONETM-Concurrent

If overflows are truly rare, then ONETM-Serialized may be sufficient to handle them without performance loss. However, if overflows are *not* extremely rare, it is likely that the serialization induced by ONETM-Serialized will result in performance degradation. To provide greater performance robustness to overflows, ONETM-Concurrent extends ONETM-Serialized to allow other code to execute concurrently with the single overflowed transaction (Figure 6.1c). To achieve this property, it introduces per-block persistent transaction metadata as part of the architected state. The system uses this metadata to track the read and write set of the single overflowed transaction; other threads then check the metadata to detect conflicts. To efficiently provide this metadata, each cache-block-sized block of physical memory is augmented with additional bits; these bits are the overflowed equivalents of the read/written bits described in Chapter 2. When the overflowed transaction writes (reads) a block, it sets the overflowed metadata write (read) bit. A single set of bits per memory block is sufficient because there can be only one overflowed transaction at a time.

By inspecting the overflowed metadata for a given block, non-overflowed transactional and non-transactional accesses can detect conflicts with the overflowed transaction. The metadata thus ensures that all other threads will detect conflicts with the overflowed transaction, even if has been pre-empted or has migrated to a different processor than that on which it started execution. Below, we first describe the storage and manipulation of the metadata. We then discuss how *lazy clearing* can be employed to eliminate the requirement that an overflowed transaction clear all its metadata as part of commit, followed by a discussion of how ONETM-Concurrent supports having overflowed transactions set the overflowed read bit for a given block without needing to have write permissions for the block. We then detail required operating system support. We close the section by comparing ONETM-Concurrent to the prior work described in Chapter 4.

**ONETM-Concurrent Algorithm with Active Clearing**

**begin**

```
in_transaction = true
if start_as_overflowed:
  while STSW.overflowed and
      not PTSW.overflowed:
    stall
  set STSW.overflowed
  set PTSW.overflowed
  timestamp = -1
  start_as_overflowed = false
else:
  timestamp = clock
```

**commit**

```
foreach address A in Cache:
  Cache[A].read_bit = false
  Cache[A].written_bit = false
if PTSW.overflowed:
  for A in read_set:
    obtain perms to A
    A.read_bit = false
  for A in write_set:
    obtain perms to A
    A.written_bit = false
  PTSW.overflowed = false
  STSW.overflowed = false
in_transaction = false
```

**evict(A)**

```
if (Cache[A].read_bit) or
   (Cache[A].written_bit)
  start_as_overflowed = true
  abort()
if Cache[A].dirty:
  write back A
Cache[A].valid = false
```

**load(A)**

```
if Cache[A].state == I:
  obtain_permissions(A,read)
if PTSW.overflowed:
  A.read_bit = true
else:
  while A.written_bit:
    stall
if in_transaction:
  Cache[A].read_bit = true
return Cache[A].data
```

**store(A,v)**

```
if Cache[A].state != M:
  obtain_permissions(A,write)
if PTSW.overflowed:
  A.written_bit = true
else:
  while (A.read_bit or
         A.written_bit):
    stall
if (in_transaction) and
   (not Cache[A].written_bit):
  write back A
  Cache[A].written_bit = true
Cache[A].data = v
```

Figure 6.4: **ONETM-Concurrent algorithm with active clearing.** The algorithm builds on the bounded hardware transactional memory presented in Figure 2.11 on page 35. Each memory block is extended with a pair of read and written bits that travel coherently with the data for the block (denoted as `A.read_bit` and `A.written_bit`). The overflowed transaction sets these bits and other threads check them to determine conflicts. At commit, the overflowed transaction clears these bits for all blocks it has accessed. Functions not shown are unchanged from Figure 2.11 on page 35.

### 6.2.1 Metadata Operation

The metadata used by ONETM-Concurrent comprises two bytes per memory block (two bits indicating transaction read and write and a 14-bit identifier to be described later). This metadata is part of the system's architected state, existing both in caches (in addition to transactional read/written bits used by non-overflowed transactions) and memory. As the metadata is logically associated with every block of data, the metadata travels with the data anytime the data block is transferred (*e.g.*, cache misses, responses from memory, cache-to-cache data transfers, and cache evictions). When responding to a cache miss, the memory controller provides both the data and metadata bits from the memory in parallel. Although this metadata increases the size of the data payload, the coherence protocol control logic itself need not change, and thus no special logic is required to communicate and manage the metadata. Non-overflowed transactions check for conflicts by simply examining the overflowed metadata of a cache block after they have brought the block into their cache.

Figure 6.4 on page 87 details the basic operation of ONETM-Concurrent. This algorithm assumes that an overflowed transaction clears the overflowed read and written bits as part of commit. We discuss problems with this assumption and a mechanism for relaxing it in Section 6.2.2.

**Metadata storage.** The problem of where to store the metadata of ONETM-Concurrent in memory is similar (but not identical) to the classic problem of where to store directory state in a directory-based implementation of cache coherence (or token count state in a token coherence-based implementation [70]). One implementation option for storage of this metadata is to add dedicated storage at each memory controller. Assuming a memory system with 64-byte blocks, this dedicated storage represents a 3% memory overhead. This approach is similar to that taken by several pioneering implementations of directory coherence [58, 60, 61, 65]. A conceptually similar approach that avoids adding dedicated storage is for each memory controller to allocate a fixed-sized region in its physical memory to store the metadata associated with its remaining addressable memory.

Another common approach to storing directory state is to use part of the per-block space allocated for error-correction codes [6, 34, 38, 51, 81]. As discussed by Gharachorloo et al. [34], it is possible to free sufficient space for metadata storage at the cost of a slight loss in error coverage by coarsening the granularity at which error correction and detection is performed. ONETM-Concurrent can use this approach as well, and in fact, it is simpler to implement in our context. One known disadvantage of using ECC codes to store coherence protocol state is that operations

that were previously simply reads or writes must now be read-modify-writes in order to update the coherence state [34, 69]. In contrast, the metadata used by ONETM-Concurrent is semantically simply extra data from the point of view of the cache coherence protocol (as discussed above), and the maintenance of this metadata imposes little extra control logic on memory controllers.

We finally note that one mechanism from directory protocol implementations that is not relevant to ONETM-Concurrent is the directory cache (see Section 3.2.4 of Martin's dissertation [69]). In the context of coherence, a directory cache has two potential purposes: first, it can reduce latency for memory accesses (as the directory lookup is on the critical path of a miss), and second, it can serve to eliminate the need to store directory state in memory entirely (by invalidating a block from all processors' caches if its directory entry must be evicted from the directory cache). In ONETM-Concurrent, however, accessing metadata incurs no extra latency over accessing data (eliminating the first potential benefit of a metadata cache). Additionally, as there is no bound on the number of blocks that a transaction may access, metadata must be maintained for every block in the system (eliminating the second potential benefit of a metadata cache).

**Transitioning to overflowed mode.** When a transaction overflows, it transitions to overflowed execution mode. A simple way to accomplish this transition is to abort the transaction and restart it in overflowed mode after ensuring that no other thread in the application is already executing in overflowed mode (by checking the overflowed bit of the STSW). This is the specific implementation that we evaluate in Chapter 7. Alternatively, ONETM-Concurrent could avoid an abort by more gracefully transitioning to overflowed mode. As before, the processor must first ensure that no other thread in the application is executing in overflowed mode and transition itself to overflowed mode by setting the overflowed bit of the STSW. Next, the processor walks the data cache to set the overflowed metadata for blocks read or written by the transaction; this action ensures that the conflict detection information for these blocks is not lost if the overflowed transaction is context switched. As a further optimization, the processor could update the metadata gradually as blocks overflow the caches and defer the metadata updates for non-overflowed blocks until a context switch actually occurs.

**ONETM-Concurrent Algorithm with Lazy Clearing**

**load(A)**

```
if Cache[A].state == I:
  obtain_permissions(A,read)
if PTSW.overflowed:
  A.read_bit = true
  A.OTID = STSW.OTID
else:
  if A.written_bit:
    if metadata_is_stale(A):
      A.written_bit = false
    else:
      while STSW.overflowed:
        stall
if in_transaction:
  Cache[A].read_bit = true
return Cache[A].data
```

**metadata_is_stale(A)**

```
 if not STSW.overflowed:
   return true
 if A.OTID != STSW.OTID:
   return true
 return false
```

**commit**

```
foreach address A in Cache:
  Cache[A].read_bit = false
  Cache[A].written_bit = false
if PTSW.overflowed:
  STSW.OTID++
  PTSW.overflowed = false
  STSW.overflowed = false
in_transaction = false
```

**store(A,v)**

```
if Cache[A].state != M:
  obtain_permissions(A,write)
if PTSW.overflowed:
  A.written_bit = true
  A.OTID = STSW.OTID
else:
  if A.read_bit or
     A.written_bit:
    if metadata_is_stale(A):
      A.read_bit = false
      A.written_bit = false
    else:
      while STSW.overflowed:
        stall
if (in_transaction) and
   (not Cache[A].written_bit):
  clean Cache[A].data
  Cache[A].written_bit = true
Cache[A].data = v
```

Figure 6.5: **Addition of lazy clearing to ONETM-Concurrent algorithm.** This algorithm adds lazy clearing to the algorithm presented in Figure 6.4 on page 87. Rather than the overflowed transaction clearing the overflowed read and written bits at commit, processors check for staleness of these bits on conflicts. Functions not shown are unchanged from Figure 6.4 on page 87.

### 6.2.2 Lazy Metadata Clearing

The above discussion assumes that an overflowed transaction clears all the overflowed read and written bits at commit. However, this assumption is not practical: the number of blocks with non-zero overflowed transactional metadata is unbounded, and such blocks could be in any cache, memory module, or even swapped to disk. As such, it is not possible to easily clear all the overflowed transactional metadata. In this section we detail *lazy clearing*, a mechanism that eliminates the need for overflowed transactions to clear metadata at commit.

Instead of actively clearing the metadata, the system clears the metadata lazily by using an *overflowed transaction identifier* (OTID) to differentiate between stale and current metadata. The per-block metadata is extended to hold an OTID (the 14-bit identifier mentioned earlier) that is updated anytime the metadata read/written bits are set. The OTID of the active overflowed transaction is also stored in the STSW (see Figure 6.2a), allowing all processors to fetch the current OTID by executing a coherent read request to its location. When an overflowed transaction commits, it increments the OTID in the STSW. Instead of explicitly clearing the metadata bits when it completes, the overflowed transaction simply clears the overflowed bit in the STSW as before.

A processor checks for conflicts by checking the metadata as described above; the processor elides this check if the overflowed bit in the PTSW is set. If the processor detects a possible conflict, it then proceeds to check whether the OTID associated with the conflicting memory block is equal to the currently active OTID (by reading the STSW). If the IDs do not match, the processor proceeds without stalling or aborting (*i.e.*, the metadata is stale). If the IDs match, a conflict exists and the requesting processor stalls until the overflowed transaction clears the STSW's overflowed bit during commit. While a processor is stalling, another conflict can cause its transaction to abort. Figure 6.5 on page 90 details the addition of lazy clearing to the basic ONETM-Concurrent algorithm presented in Figure 6.4 on page 87.

If OTIDs were never reused, this approach would avoid the need to ever clear the metadata. However, the OTID width is finite and small. As a result, OTIDs will eventually wrap around, creating the potential for false conflicts and unnecessary delay. Such false conflicts can occur only when (1) an overflowed transaction is active (otherwise the metadata is ignored) and (2) the thread attempting the access is not executing in overflowed mode. The stall due to the false conflict will

be temporary, because once the active overflowed transaction completes it clears the overflowed bit in the STSW, thus un-stalling the victim of the false conflict.

To reduce false stalls, the processor opportunistically clears stale overflowed transaction metadata whenever possible. Whenever a processor not executing an overflowed transaction writes a cache block, it clears the associated metadata. Thus, as long as a block has been written since the last time the current OTID was used, no false conflicts will occur on that block. The metadata can also be cleared whenever a processor manipulates a cache block in which the current OTID does not match the block's OTID. Lazily clearing metadata does not impact correctness or forward progress; it is only a performance optimization.

### 6.2.3 Lazily Coherent Metadata

Although the metadata can be kept exactly coherent by requiring a processor to have write permission to a block to modify its metadata, such a requirement causes unnecessary invalidations of blocks in shared state (and thus transaction conflicts) when only the metadata needs to be modified, and also inhibits the efficient lazy clearing of metadata.

Instead, we would like a processor executing an overflowed transaction to be able to set the metadata without needing exclusive permissions to the block. As there is only one active overflowed transaction at a time, there will be at most a single writer (even if there are multiple readable copies in the system). However, to prevent out-of-order writebacks from overwriting more recent metadata with stale metadata, the system allows only the owner of the block (non-exclusive or exclusive) to set the metadata. Many cache coherence protocols already include the notion of a single non-exclusive read-only dirty owner (the "O" state [107]) that is responsible for writing back the block to memory upon eviction. Once the metadata has been written, it is the owner's responsibility to ensure the data is eventually written back to memory (or transfer the ownership, and thus the responsibility, on to another processor). Some protocols already support a non-dirty owner as part of determining which processor responds with data during a shared-intervention [70, 110]. In protocols that grant non-exclusive owner status to the most recent requester, whenever a processor in an overflowed transaction requests a block, it will be able to set the read bit immediately after the miss completes. In the case where a block is in the cache in shared but not owned state, the overflowed transaction issues a remote request to obtain ownership of the block.

The key to the correctness of this lazy updating of metadata is that the system guarantees that any *new* requests for the block receive the most recent version of the metadata. Once an overflowed transaction has set the read bit (and thus has the block in owned state), any other processor that tries to write the block will issue a cache request and receive the most recent version of the metadata, indicating the conflict. Processors will only set the written bit when they are writing the block, in which case they have exclusive permissions to it; thus, any subsequent read or write will again receive the most recent copy of the metadata and detect the conflict. Any processor can clear the metadata opportunistically as described above; if the processor owns the block, then its clearing of the metadata will propagate to other processors.

### 6.2.4 Example Execution

Figure 6.6 illustrates the lazy coherence and clearing of metadata. At time $t_1$, processor $P_1$ loads the block A into its cache; at that time, there is no overflowed transaction executing and the metadata for A is $\emptyset$. At $t_2$, $P_0$ overflows, setting the overflowed bit of the STSW and incrementing the OTID. At $t_3$, $P_0$ loads A into its cache in owned state and sets the read bit for A, as well as writing its OTID into the OTID metadata field for A. $P_1$ now has stale metadata in its cache, but there is no conflict. At $t_4$, $P_2$ loads A into its cache; because $P_0$ owns A, it supplies the data (and metadata) to $P_2$. Again, there is no conflict. At $t_5$, $P_3$ requests A in modified state; as the owner, $P_2$ supplies $P_3$ with the data. $P_3$ now stalls, because the read bit of A is set and the STSW indicates that the OTID of the active overflowed transaction matches the OTID in the metadata of A. At $t_6$ $P_0$ commits its overflowed transaction, clearing the overflowed bit of the STSW. A short time later, $P_3$ sees that the overflowed bit of the STSW is now clear and unstalls itself to perform its write of A. It also opportunistically clears the metadata of A at this point. Because $P_3$ has A in modified state, it will ensure that its version of the metadata for A is given to anyone requesting A in the future.

### 6.2.5 Operating System Involvement

As in ONETM-Serialized, the operating system must save and restore the PTSW register as part of thread state. Additionally, when swapping pages to and from disk, the operating system must save and restore the associated metadata and OTIDs (as implemented in other systems [28, 102]). The operating system may optionally clear metadata and OTIDs when zeroing pages before reallocation.

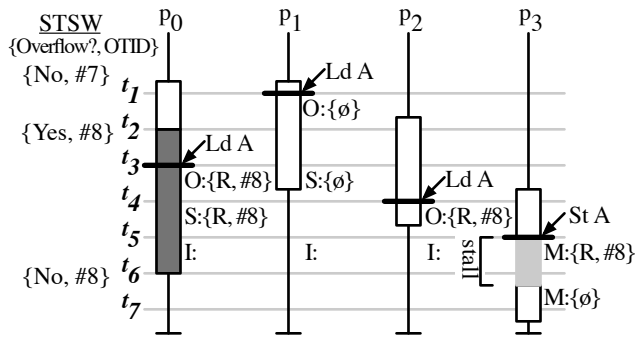


Figure 6.6: **Example illustrating lazy coherence and clearing of metadata in ONETM-Concurrent.** The white bars are non-overflowed transactions, the dark gray bars are overflowed transactions, the straight lines are non-transactional execution, and the light gray color indicates stalled execution. The example centers around a memory block with address A; the text to the right of each processor is that processor's MOESI coherence state and local metadata for A.

### 6.2.6 Comparison to Prior Work

Supporting only one unbounded transaction at a time enables ONETM-Concurrent to avoid facing challenging problems of the prior proposals discussed in Chapter 4, which seek to support unbounded transactions with full concurrency. First, any system that supports more than one unbounded transaction at a time must also be able to abort these transactions in order to be able to guarantee forward progress in the case of cyclic conflicts. Thus, any such system must support unbounded version management. ONETM-Concurrent, by contrast, avoides the need for unbounded version management due to its policy of making the unbounded transaction highest-priority in conflict detection. Second, systems that support an unbounded number of unbounded transactions face the challenge of detecting conflicts between these transactions. While prior proposals provide solutions to this problem – *e.g.*, the linked-list traversals of UTM, PTM, and VTM, Bulk's traversal of the signatures of swapped out transactions, and LogTM-SE's maintenance of the summary signature – ONETM-Concurrent avoids the problem altogether.

The price that ONETM pays for the above simplifications is, of course, a limit on concurrency for overflowed transactions. In an $n$-processor system, the impact of this restriction is likely to be small as long as the fraction of execution time that each processor spends in overflows is less than $1 \div n$. In Chapter 7, we show that the combination of the permissions-only cache and ONETM

can provide similar performance to an idealized unbounded hardware transactional memory on the workloads that we use.

## 6.3 Semantic Considerations in ONETM

This section describes the impact of various semantic choices on ONETM, including weak/strong atomicity, starvation avoidance, support for IO within transactions, and support for an explicit abort operation.

**Weak and strong atomicity.** As described in Section 2.2.2, strong atomicity is a model in which transactions are guaranteed to be isolated from non-transactional memory accesses, whereas weak atomicity is a model in which only conflicts between transactions are guaranteed to be detected. As presented above, both ONETM-Serialized and ONETM-Concurrent enforce strong atomicity. In ONETM-Serialized, all other processors stall while an overflowed transaction is executing, whether these other processors are executing transactions or not. In ONETM-Concurrent, both transactonal and non-transactional memory accesses check for conflicts with the overflowed metadata. Here, we consider the possible impact on performance and complexity of instead supporting weak atomicity in these systems.

In ONETM-Serialized, enforcing weak rather than strong atomicity can potentially increase performance, as threads not executing transactions would not have to stall while an overflowed transaction is executing. In addition, subsequent work by Hofmann et al. [50] made the observation that the decision to enforce weak isolation could simplify implementation requirements. We discuss this proposal in Section 6.4.

In ONETM-Concurrent, by contrast, enforcing weak rather than strong atomicity does not significantly impact the design of the system and is unlikely to impact the system's performance. The only change to the system would be that only processors in transactions would check the overflowed metadata on a load or store. Similarly, this policy choice would have a performance impact only in the case where there actually are conflicts between transactions and non-transactional memory accesses.

**Starvation avoidance.** As detailed in Section 2.6, the bounded hardware transactional memory system on which ONETM builds guarantees that all transactions will eventually become highest-

priority in conflict resolution through its use of timestamps. In ONETM, however, the fact that the overflowed transaction always acquires the lowest timestamp means that bounded transactions are always vulnerable to being aborted by the overflowed transaction. To maintain the property of starvation avoidance, if a transaction fails to make progress due to conflicts, it arbitrates to become the overflowed transaction. As long as this arbitration process is fair (*e.g.*, it could employ the timestamps of the competing *bounded* transactions), this policy allows a transaction to avoid starvation.

**IO.** The fact that unrestricted transactions never abort allows support for general IO within transactions in ONETM. Although many common system calls and I/O may be handled via input/output buffering, compensation actions [16], and/or transactional OS interfaces [86], some operations are not easily handled within a transaction that may later be rolled back (*e.g.*, sending a network request and receiving its response). To handle cases in which a transaction wants to perform a non-transactional system call, the runtime system can transition the transaction into overflowed mode. As the transaction will then never abort, programs can rely on this property to, for example, perform arbitrary system calls or input/output within the transaction [11]. A similar approach to the problem of supporting IO within transactions was proposed by the TCC project [41].

The performance impact of this mechanism is dependent on the frequency of IO within transactions. The workloads described in Chapter 3 and used throughout this dissertation do not perform IO within transactions. However, in a study of IO within critical sections in Firefox and MySql, Baugh and Zilles [7] found that concurrent execution of multiple critical sections containing IO was relatively infrequent. As such, ONETM-Concurrent in particular could potentially provide sufficient performance.

**Explicit abort.** ONETM's policy of allowing only one overflowed transaction at a time and prioritizing the overflowed transaction in conflict resolution means that the system never has to abort the overflowed transaction. Consequently, unlike transactional memory systems that support the concurrent execution of multiple overflowed transactions (and must resolve conflicts between them), ONETM does not require unbounded version management in order to resolve conflicts. It is true, however, that unbounded version management is required if an explicit abort operation is desired as part of the transactional interface to the programmer. The most straightforward way to support such

**commit**

```
if not overflowed:
  while global lock held:
    stall
flash-clear written bits
flash-clear read bits
if overflowed:
  release global lock
  overflowed = false
```

**evict(A)**

```
if (Cache[A].read_bit) or
   (Cache[A].written_bit)
  while global lock held:
    stall
  acquire global lock
  overflowed = true
if Cache[A].dirty:
  write back A
Cache[A].valid = false
```

Figure 6.7: **Overflow handling algorithm of Hofmann et al. [50].** The algorithm employs a global lock. To execute in overflowed mode, a transaction must first acquire the lock. Non-overflowed transactions can commit only when the lock is free. The algorithm is conceptually similar to Figure 6.3 on page 83, but differs in (1) not requiring custom hardware for overflow handling, (2) stalling non-overflowed transactions at commit rather than "in place", and (3) providing weak rather than strong atomicity.

unbounded version management in ONETM is to use the log-based version management presented in Figure 2.12 on page 36.

If support for explicit abort is combined with the above-mentioned support for IO within transactions, the system must ensure that a transaction that has performed irrevocable IO is not later rolled back by the user. To maintain such a guarantee, the system could add a "performed-IO" bit to the PTSW. This bit would be set on performance of an IO operation and checked to be clear on a rollback, with a failed check causing an exception to be raised.

## 6.4 Subsequent Work

Hofmann et al. [50] propose a system that employs a global lock to serialize overflowed transactions. Before beginning execution in overflowed mode, a transaction must acquire this global lock. Non-overflowed transactions are allowed to continue execution in the presence of an overflowed transaction, but can commit only when no overflowed transaction is concurrently executing (*i.e.*, the lock is free). As in ONETM-Serialized, all conflicts between an overflowed transaction and a non-overflowed transaction will be detected by the non-overflowed transaction and are resolved in favor of the overflowed transaction. Figure 6.7 on page 97 shows the pseudocode for this system.

This system is conceptually similar to ONETM-Serialized but differs by (1) enforcing weak atomicity rather than strong atomicity and (2) stalling non-overflowed transactions at commit rather

than in place when an overflowed transaction begins execution. The benefit of enforcing only weak atomicity is that the processor does not need to check for the presence of an overflowed transaction on each load and store. It is not clear whether the difference in stalling policy would have a significant performance impact, as non-overflowed transactions are likely to be short relative to overflowed transactions.

TokenTM [14] details an approach to unbounded hardware transactional memory that supports multiple overflowed transactions executing concurrently via *transactional tokens*. Adapting the use of tokens from token coherence [70], a transaction is allowed to write a block if it holds all tokens and is allowed to read the block if it holds at least one token. Once a transaction acquires a token for a given block, it retains that token until commit or abort, at which time it releases all its tokens[1].

TokenTM employs logging in order to support abort of unbounded transactions (necessary for resolving cyclic conflicts) as well as to record all tokens acquired by the transaction (necessary for releasing these tokens on commit or abort). If a transaction cannot acquire sufficient tokens to complete its access, it signals a conflict. TokenTM generalizes the lazy coherence employed by ONETM-Concurrent to support simultaneous modifications of a block's metadata by different transactions.

The approach taken by TokenTM is conceptually similar to ONETM-Concurrent in that both employ metadata that travel coherently with data. TokenTM, however, avoids ONETM's restriction of allowing only one overflowed transaction at a time. This significant increase in concurrency has several costs: (1) requiring a mechanism for unbounded version management, (2) requiring a mechanism for recording all tokens held by a transaction, (3) performing clearing of metadata (*i.e.*, token release) actively rather than lazily at the end of a transaction, and (4) needing to walk the logs of all active transactions to detect conflicts in the worst case.

## 6.5 Summary

This chapter described ONETM, a proposal for supporting unbounded hardware transactional memory that operates via serialization of overflowed transactions. The key concept of ONETM is the use of an *overflowed bit* that resides in an application's virtual address space. Transactions read

[1]The authors propose an optimization that enables token release in constant time for transactions that have not overflowed the cache.

and modify this overflowed bit in order to determine when they must stall and/or can begin an overflowed transaction.

We presented two instantiations of ONETM, ONETM-Serialized and ONETM-Concurrent. In ONETM-Serialized, one thread beginning an overflowed transaction results in all other threads in the application stalling. In ONETM-Concurrent, by contrast, non-conflicting bounded transactions (and non-transactional operations) can execute and commit in the presence of an overflowed transaction. To support conflict detection, ONETM-Concurrent adds overflowed metadata that (1) travels coherently with data blocks, (2) is set on an access by an overflowed transaction, and (3) is read by other threads to determine conflicts.

In the next chapter we quantitatively evaluate the performance of ONETM. We find that on the workloads that we use, the combination of ONETM and the permissions-only cache provides similar performance to that of an idealized unbounded hardware transactional memory system.

# Chapter 7

# Experimental Evaluation of ONETM and the Permissions-Only Cache

In this chapter we experimentally evaluate ONETM and the permissions-only cache using the workloads and infrastructure described in Chapter 3. We first analyze the performance of ONETM-Serialized and ONETM-Concurrent relative to the idealized unbounded HTM system described in Chapter 3, which handles overflows with full concurrency and no overheads. We focus on the questions of (a) how much the policy employed by ONETM-Serialized of serializing the system on overflows degrades overall performance and (b) how much the increased concurrency of ONETM-Concurrent reduces this negative performance impact. We then analyze whether enforcing weak rather than strong atomicity could increase the performance of ONETM-Serialized and ONETM-Concurrent; the answer to this question can help system designers reason about which policy to implement. Next, we analyze the impact of lazy clearing on the performance of ONETM-Concurrent, focusing on the questions of (a) whether finite-length OTID's result in spurious conflicts and (b) whether OTID's are necessary to avoid performance degradation. Finally, we examine the impact on performance of adding a permissions-only cache to ONETM-Serialized and ONETM-Concurrent. Our main objective is to determine whether the combination of ONETM and the permissions-only cache can achieve the performance of the idealized HTM. We also examine whether the sector cache organization employed by the permissions-only cache to reduce tag overhead results in performance degradation due to increased cache conflicts.

In Section 7.1 we detail the configurations that we evaluate in this chapter. We evaluate the performance of ONETM-Serialized and ONETM-Concurrent in Section 7.2. We next examine the performance impact of weak atomicity in Section 7.3. Section 7.4 examines the impact of lazy clearing on the performance of ONETM-Concurrent. We examine the impact of the permissions-only cache on the performance of ONETM in Section 7.5. We discuss the power implications of our proposals in Section 7.6. We summarize the main results of this chapter in Section 7.7.

## 7.1 Experimental Methodology

We use the simulator infrastructure and workloads described in Chapter 3. We also reuse the idealized unbounded HTM that handles overflows with full concurrency and no overheads; we will measure the performance of our systems against the performance of this idealized system.

In this chapter, ONETM-Serialized and ONETM-Concurrent implement strong atomicity unless indicated otherwise. In addition, unless indicated otherwise, ONETM-Concurrent implements lazy clearing with 14-bit OTID's as described in Section 6.2.2.

In all configurations of the permissions-only cache that we present, the metadata for a given block occupies two bits. Our default configuration of the permissions-only cache holds 256 bytes of metadata. This default permissions-only cache can thus hold the metadata for 1024 blocks (64 kilobytes of data). Unless indicated otherwise, the permissions-only is organized as a sector cache, with each sector holding 64 bytes of metadata (*i.e.*, metadata for 256 blocks). In all configurations of the permissions-only cache it is 4-way set-associative.

Our default configurations of ONETM use a foundation of bounded HTM employing cleaning for version management (Figure 2.11 on page 35). Thus, by default the permissions-only cache can hold metadata for blocks that have been transactionally read but not written only. When choosing a victim to evict, the L1 cache prioritizes blocks whose eviction will not cause a transaction abort.

## 7.2 Evaluation of ONETM

We present and analyze the performance of ONETM in this section. We first study the question of how the policy of serializing the system on overflow employed by ONETM-Serialized degrades overall performance on our workloads relative to the idealized fully-concurrent unbounded HTM.

We then examine the extent to which the increased concurrency afforded by ONETM-Concurrent, which serializes only overflowed transactions, increases performance over full system serialization.

### 7.2.1 What is the Impact of Serializing the System on Overflow?

Figure 7.1 on page 103 presents the scalability of ONETM-Serialized over sequential execution. We also present the performance of the idealized unbounded HTM described in Chapter 3. On many workloads, ONETM-Serialized matches or nearly-matches the idealized system's performance. However, on several workloads (most notably `genome`) ONETM-Serialized suffers a substantial slowdown. To gain insight into these results, Figure 7.2 on page 103 presents a breakdown of execution time for the two systems. This figure indicates that the performance losses of ONETM-Serialized relative to the idealized system correspond directly to time spent stalled due to an overflowed transaction.

Figure 7.3 on page 104 presents the amount of time that each processor spends in overflowed transactions on average under ONETM-Serialized (we discuss the second bar below). We first note that this number is uniformly small: for all workloads, this figure is less than 4%, and for all but three workloads, it is less than 1%. Several workloads spend no time in overflowed transactions, meaning that ONETM-Serialized can match the performance of the idealized system on these workloads (Figure 7.1 on page 103).

However, even a small amount of time spent in overflowed transactions can result in performance degradation for ONETM-Serialized. Each cycle that a processor spends in an overflowed transaction results in all other processors being stalled. Thus, for example, the 1% of time that each processor spends in overflowed execution in `genome` results in each processor spending 30% of its execution time stalling due to overflows (Figure 7.2 on page 103).

### 7.2.2 Does Serialization of Only Overflowed Transactions Increase Performance?

In this subsection we analyze the performance of ONETM-Concurrent, focusing on the question of whether serializing only overflowed transactions increases overall performance over serializing the entire system on an overflow.

As shown in Figure 7.3 on page 104, processors spend a similar amount of time in overflowed transactions under ONETM-Serialized and ONETM-Concurrent. However, this overflow

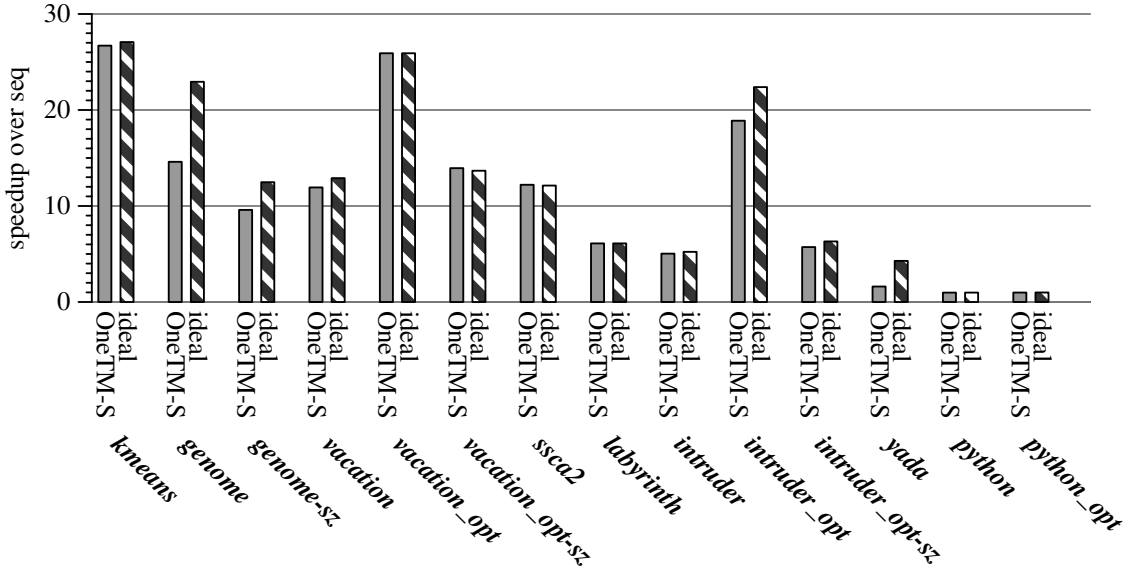


Figure 7.1: **Scalability of workloads under ONETM-Serialized.** Execution is on 32 cores, meaning that a speedup of 30 is near-ideal. The first bar ("OneTM-S") of each group shows the performance of ONETM-Serialized, while the second bar ("ideal") shows the performance of the idealized HTM described in Chapter 3. All configurations implement strong atomicity.

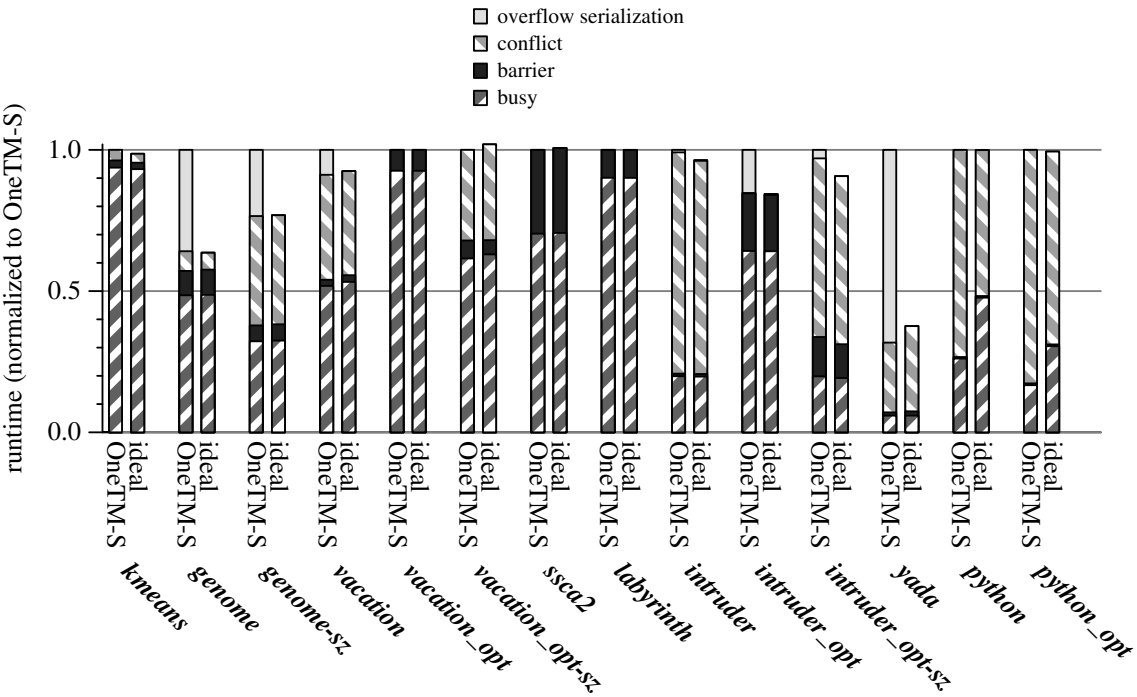


Figure 7.2: **Time breakdown of ONETM-Serialized.** "busy" represents cycles in which the processor is not stalled for a synchronization-related reason. "barrier" represents cycles stalled at a barrier. "conflict" represents cycles lost due to conflicts. "overflow serialization" represents cycles that the processor is stalled due to another thread executing an overflowed transaction.

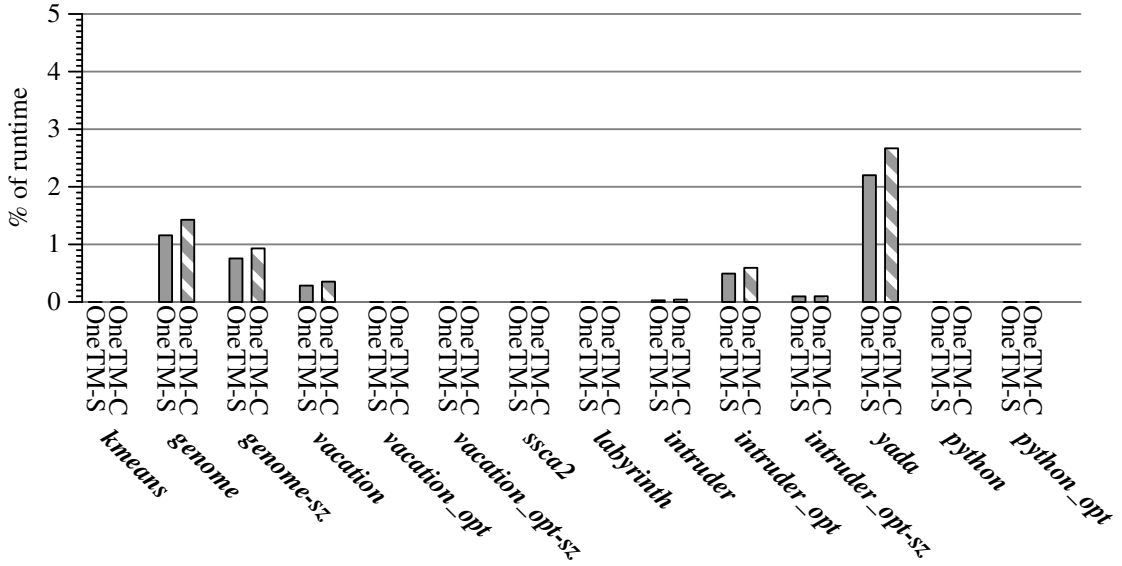


Figure 7.3: **Percentage of execution time that is spent in overflowed transactions in ONETM-Serialized and ONETM-Concurrent.**

time results in significantly less performance degradation for ONETM-Concurrent. Figure 7.4 on page 105 presents the scalability of ONETM-Concurrent. For reference we include the scalability of ONETM-Serialized and the idealized unbounded HTM. ONETM-Concurrent significantly improves on the performance of ONETM-Serialized: with the exception of `yada`, it is able to approach the performance of the idealized HTM. As Figure 7.5 on page 105 illustrates, it accomplishes this improvement by reducing the time spent stalling on overflowed transactions.

Figure 7.6 on page 106 analyzes this reduction of time spent stalling on overflows in more detail, showing the number of cycles spent stalling for each cycle of overflowed transaction execution. For ONETM-Serialized this number is uniformly 31: while an overflowed transaction is executing, all other processors must stall. For ONETM-Concurrent, however, this number corresponds to the number of processors that want to begin an overflowed transaction while an overflowed transaction is already executing. On all but two workloads this number is 5 or less.

### 7.2.3 Summary

In this section, we first showed that ONETM-Serialized matches the performance of the idealized HTM on several of our workloads (those with no time spent in overflows). However, even 1% of time spent in overflows causes the performance of ONETM-Serialized to degrade significantly rel-

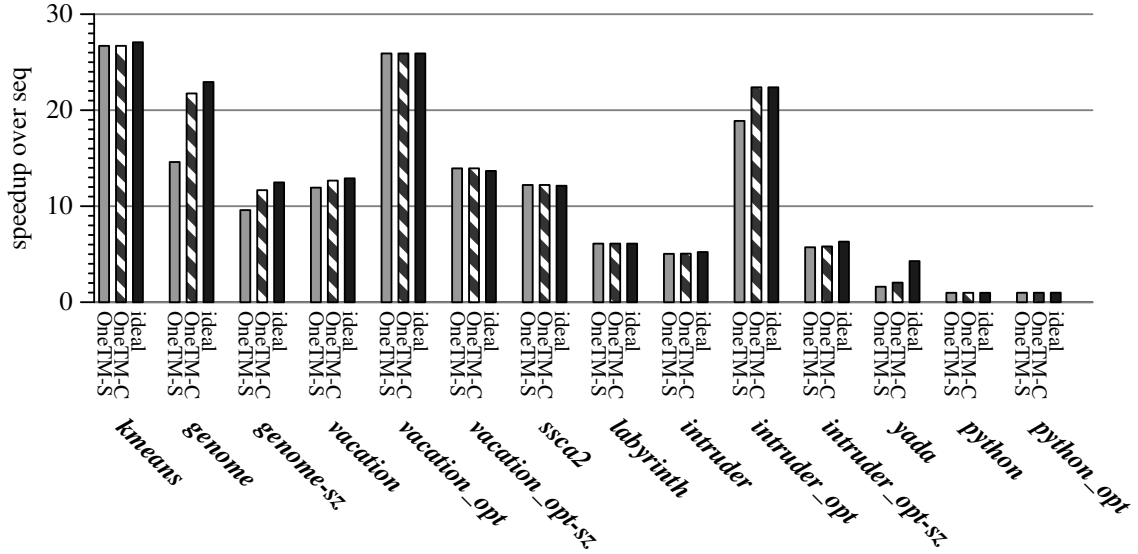


Figure 7.4: **Scalability of workloads under ONETM-Concurrent.**

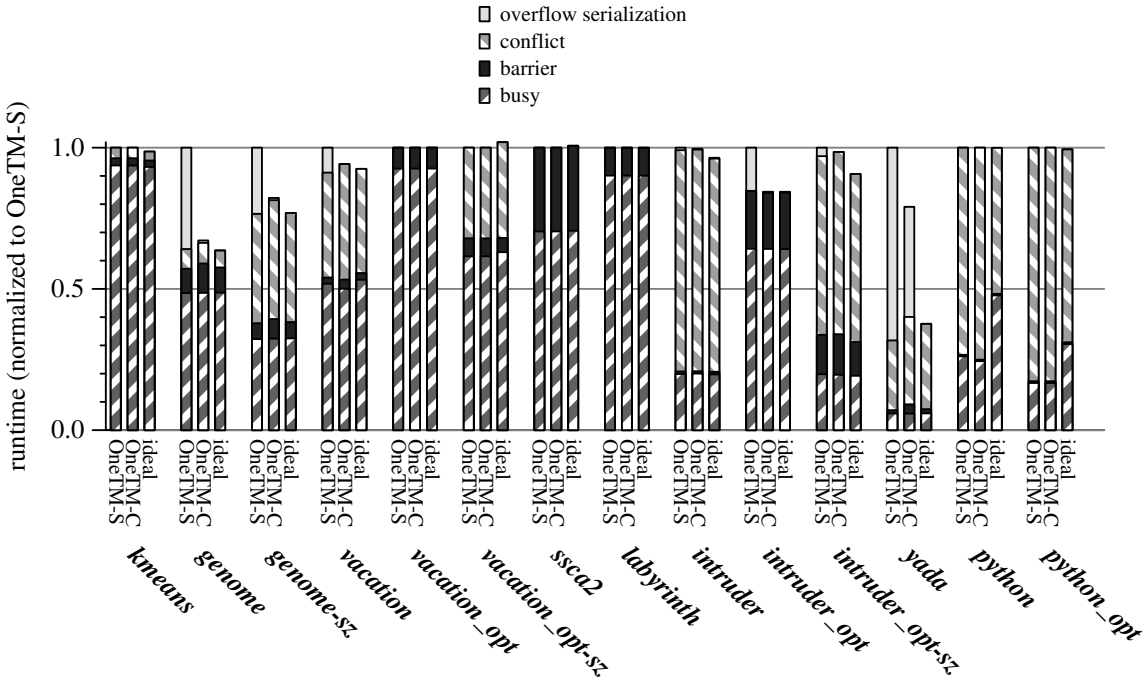


Figure 7.5: **Time breakdown of ONETM-Concurrent.**

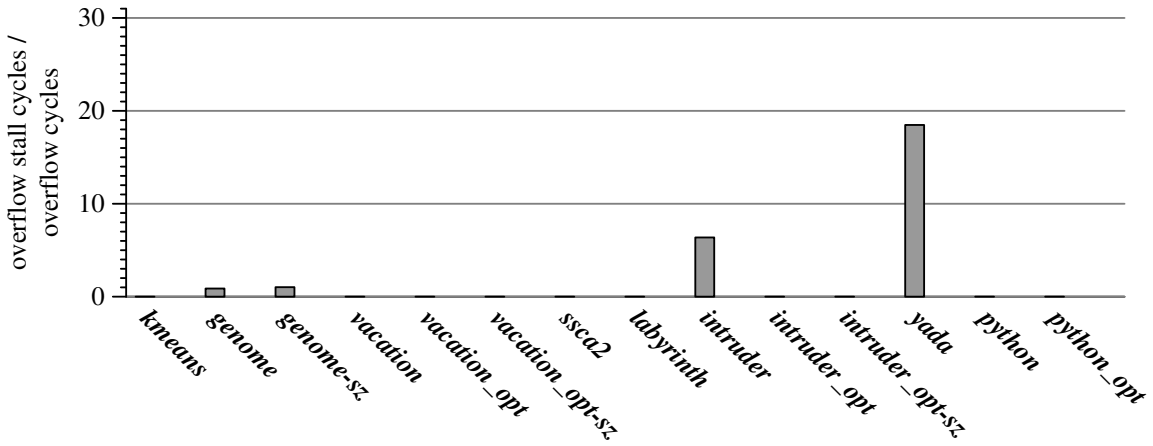


Figure 7.6: **Stall time due to serialization of oveflowed transactions in ONETM-Concurrent.**

ative to the fully-concurrent unbounded HTM. We then showed that by serializing only overflowed transactions, ONETM-Concurrent increases robustness to overflows over ONETM-Serialized. For ONETM-Concurrent, a processor stalls only when that processor overflows *and* there is already another processor executing in overflowed mode. Figure 7.6 on page 106 shows that the number of such simultaneous overflows is generally small, enabling ONETM-Concurrent to match or nearly-match the performance of the idealized system on most workloads. However, ONETM-Concurrent still suffers performance degradation from the fully concurrent system in a minority of cases.

## 7.3 Impact of Weak Atomicity on ONETM

We noted in Section 6.3 that the choice to support weak atomicity rather than strong atomicity could have a positive performance impact on ONETM. In particular, enforcing weak atomicity could potentially increase ONETM-Serialized performance by allowing non-transactional threads to continue executing in the presence of an overflowed transaction. In ONETM-Concurrent, the impact of enforcing weak rather than strong atomicity is that non-transactional memory accesses do not have to check for conflicts with the overflowed bits. We quantitatively examine the impact of weak atomicity on ONETM-Serialized and ONETM-Concurrent here. If this performance impact was large, it could help influence system designers to enforce a semantics of weak rather than strong atomicity.

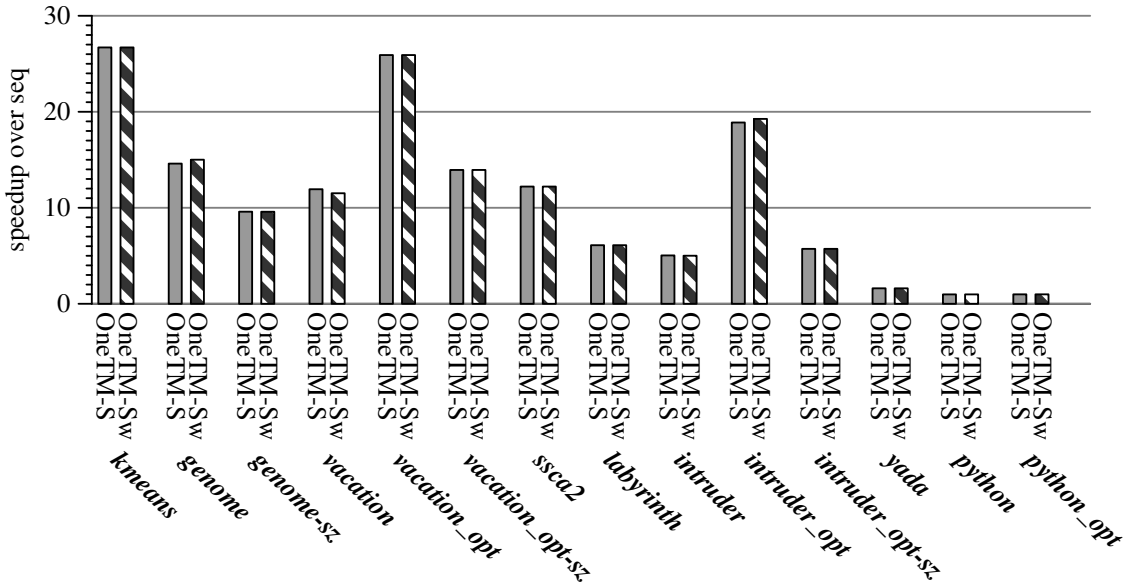


Figure 7.7: **Impact of weak atomicity on ONETM-Serialized.** Execution is on 32 cores, meaning that a speedup of 30 is near-ideal. The first bar of each group ("OneTM-S") shows ONETM-Serialized with its default configuration of strong atomicity, and the second bar of each group ("OneTM-Sw") shows ONETM-Serialized configured to enforce weak atomicity.

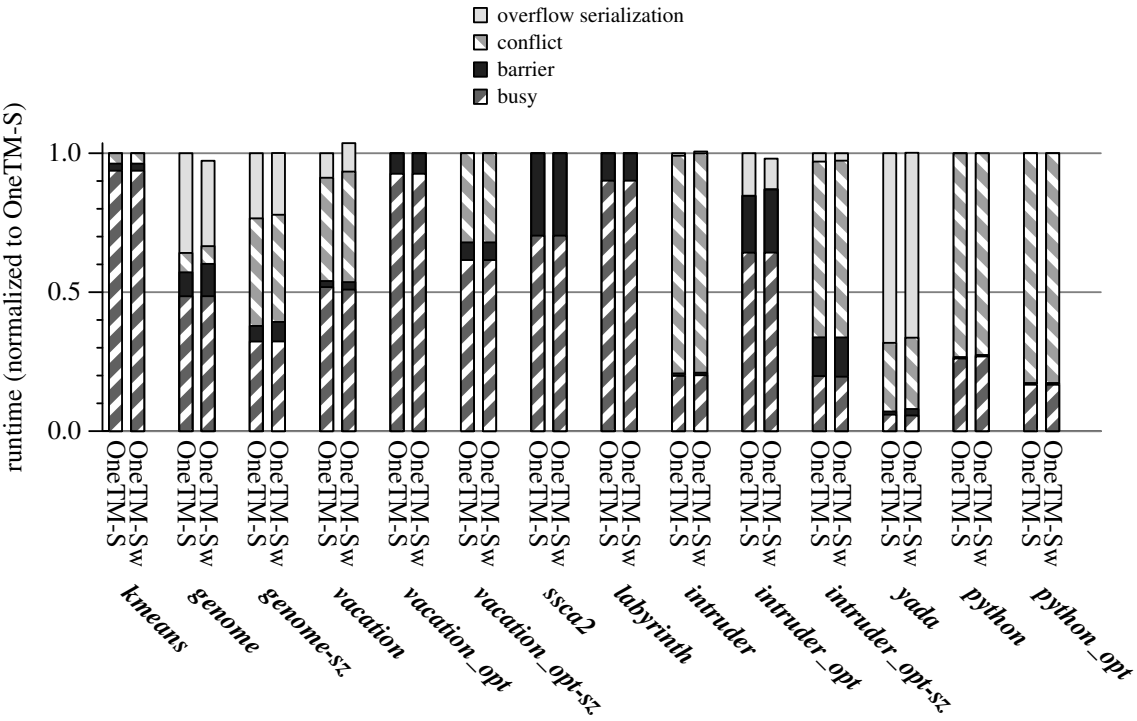


Figure 7.8: **Time breakdown of ONETM-Serialized with strong and weak atomicity.**

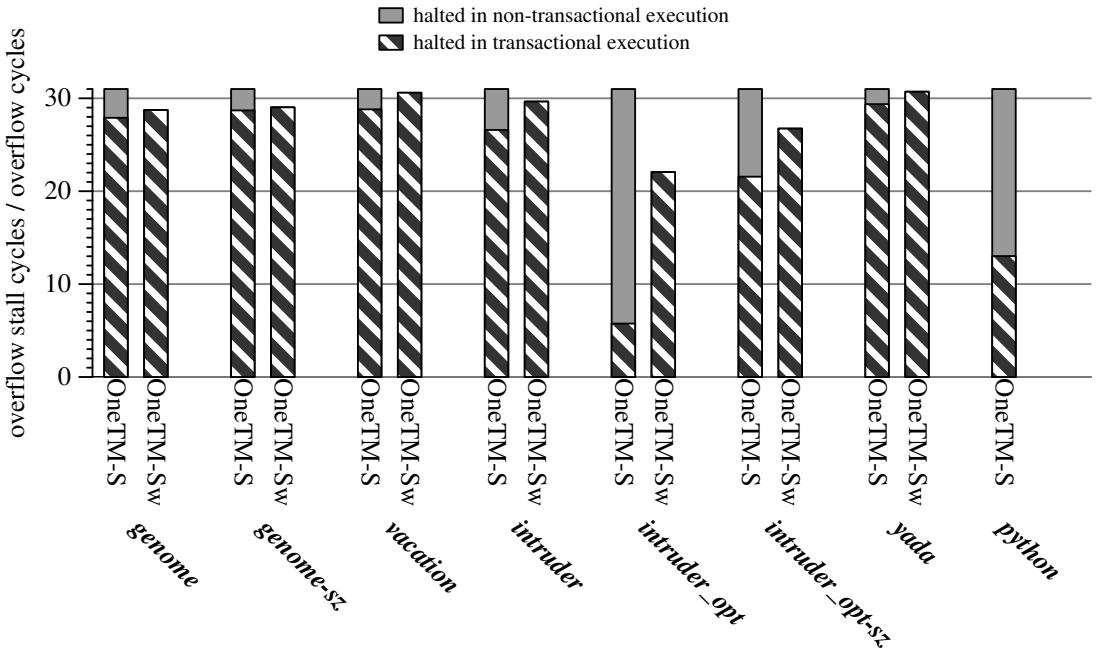


Figure 7.9: **Stall time due to serialization of overflowed transactions in ONETM-Serialized.** The two components represent cycles wherein processor execution is halted due to an active overflowed transaction on another processor, broken down by what state the processor is in at the time that it is stalled. Workloads not included do not execute any overflowed transactions. `python` has overflowed transaction time when executed under strong atomicity but does not when executed under weak atomicity.

### 7.3.1 Does Weak Atomicity Help ONETM-Serialized Performance?

Figure 7.7 on page 107 and Figure 7.8 on page 107 present the performance of ONETM-Serialized configured to implement weak atomicity. For reference, we include the performance of ONETM-Serialized configured to implement strong atomicity (*i.e.*, the results from above). These graphs show the somewhat unintuitive result that in fact the two configurations perform similarly: there is essentially no performance benefit from implementing weak atomicity in ONETM-Serialized.

We examine this result in more detail in Figure 7.9 on page 108. This graph presents the number of cycles that are spent stalled for each cycle that a processor is executing an overflowed transaction. This number is subdivided into time spent stalling in transactional execution and time spent stalling in non-transactional execution. As discussed above, for ONETM-Serialized configured with strong atomicity, this number is always 31: when one processor is executing an overflowed transaction, all other processors stall for the duration of the transaction. For ONETM-Serialized configured with

weak atomicity, this number corresponds to the number of other processors that are concurrently executing bounded transactions.

This graph illustrates the reasons for the general lack of performance gain from weak atomicity in ONETM-Serialized. The majority of time spent stalling under strong atomicity on most workloads is within transactions, meaning that these stalls must occur even under weak atomicity. Furthermore, time spent stalling in non-transactional execution is often simply replaced by increased time spent stalling in transactional execution.

### 7.3.2 Does Weak Atomicity Help ONETM-Concurrent Performance?

Figure 7.10 on page 110 shows that the difference between strong and weak atomicity has little performance impact on the performance of ONETM-Concurrent. Figure 7.11 on page 110 details the reasons that weak atomicity does not have a performance impact on our workloads: first, processors spend a small amount of time stalling on conflicts with the overflowed bits (less than 1% of cycles in all cases), and second, most of that time is spent within (bounded) transactions.

### 7.3.3 Summary

In this section, we evaluated the potential of weak atomicity to increase the performance of ONETM-Serialized and ONETM-Concurrent. We found that in almost all cases, enforcing weak atomicity provides little to no performance benefit over enforcing strong atomicity. The predominant reason is that even when ONETM-Serialized and ONETM-Concurrent are configured to enforce strong atomicity, processors generally spend little time stalling on overflowed transactions in non-transactional execution.

## 7.4 Impact of Lazy Clearing on ONETM-Concurrent Performance

As described in Section 6.2.2, lazy clearing of overflowed transaction metadata eliminates the requirement that overflowed transactions actively clear this metadata as part of transaction commit in ONETM-Concurrent. However, it can also result in performance degradation due to spurious conflicts with stale metadata. The default configuration of ONETM-Concurrent employs 14-bit overflowed transaction identifiers (OTID's) to reduce the probability of such spurious conflicts: a

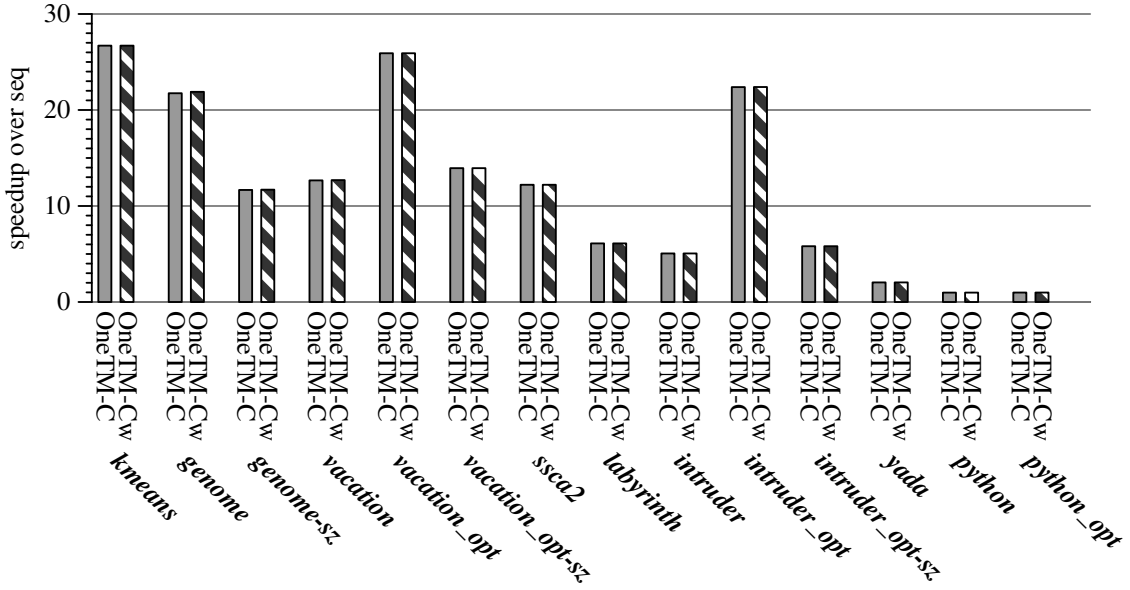


Figure 7.10: **Impact of weak atomicity on ONETM-Concurrent.** The first bar of each group ("OneTM-C") shows ONETM-Concurrent with its default configuration of strong atomicity, and the second bar of each group ("OneTM-Cw") shows ONETM-Concurrent configured to enforce weak atomicity.

---

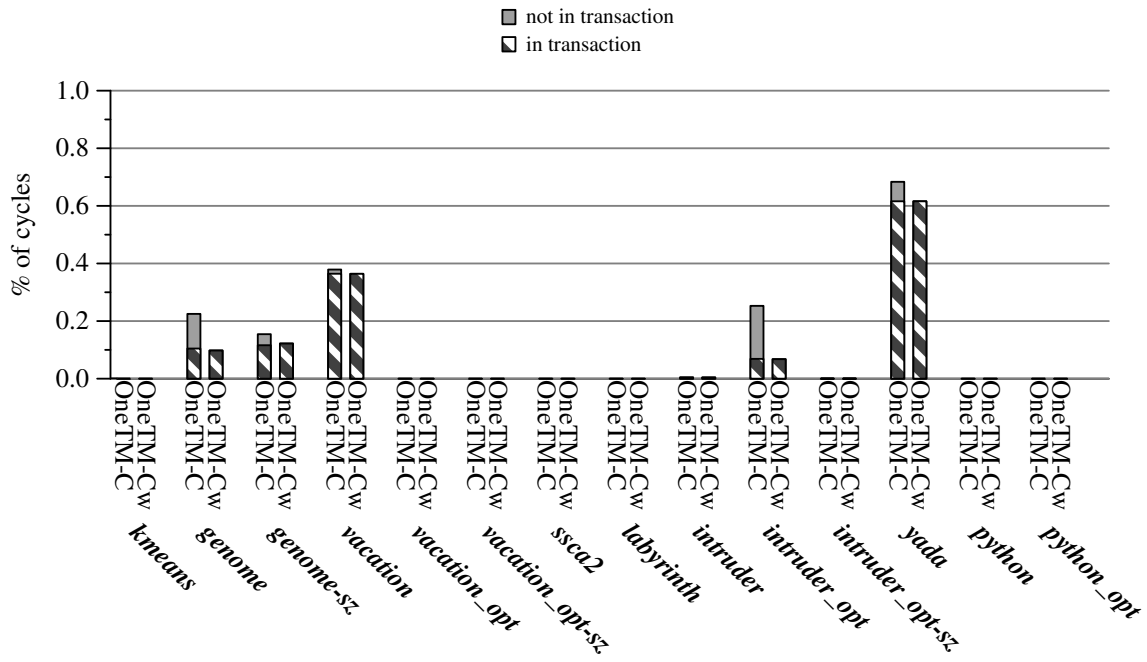


Figure 7.11: **Breakdown of stall time due to conflicts with overflowed transactions in ONETM-Concurrent.** The graph shows the percentage of total cycles that processors spend stalling on conflicts with overflowed transactions, divided into stall cycles spent in bounded transactions and stall cycles spent in non-transactional execution.

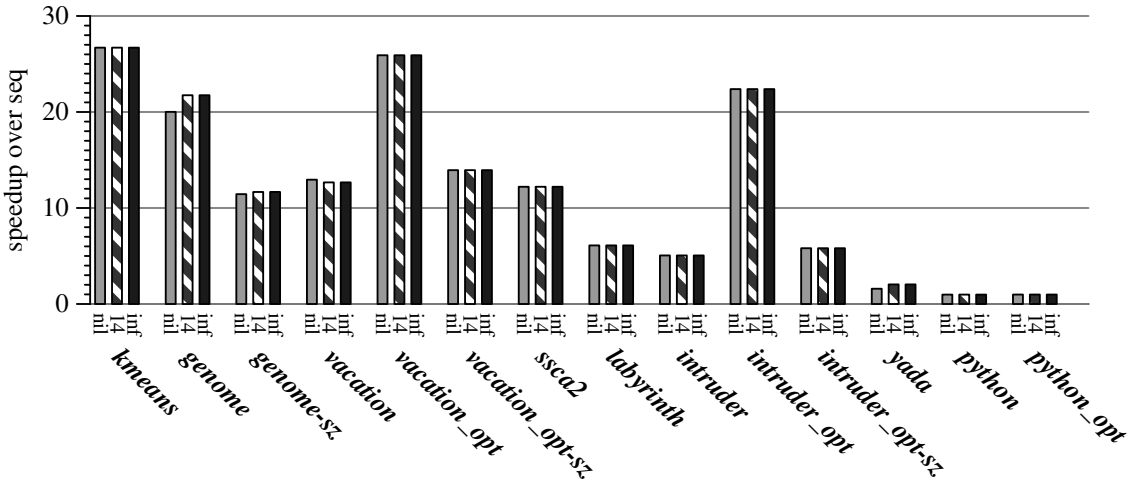


Figure 7.12: **Impact of OTID length on ONETM-Concurrent.** The middle bar of each group ("14") shows ONETM-Concurrent with the default OTID length of 14 bits. The right and left bars show variants with nil OTID's (as if all overflowed transactions had the same OTID) and infinite-length OTID's respectively.

---

spurious conflict occurs only when an access conflicts with stale metadata *and* the OTID of the metadata corresponds to the OTID of the currently-executing transaction.

In this section we examine the sensitivity of ONETM-Concurrent to OTID length and usage. We first study the question of whether 14-bit OTID's are sufficient to avoid performance degradation from infinite-length OTID's (*i.e.*, OTID's that have no possibility of aliasing). To answer this question, we examine the performance of ONETM-Concurrent in its default configuration of 14-bit OTID's relative to that of a variant configuration employing infinite-length OTID's. Figure 7.12 on page 111 presents these results (we discuss the first bar of this graph below). This figure indicates that the default OTID length of 14 bits is sufficient to achieve the performance of infinite-length OTID's.

This result indicates that 14-bit OTID's are sufficient to avoid spurious conflicts on our workloads. We next examine the question of whether they are *necessary* to do so. To answer this question, we evaluate a variant configuration of ONETM-Concurrent that does not employ OTID's at all. Instead, if a transaction detects a conflict with the overflowed metadata, it simply checks whether there is an overflowed transaction currently executing; if so, it assumes that this conflict is with the active overflowed transaction and stalls. Transactions can still detect false conflicts (and clear the overflowed metadata) in the case where there is a conflict with the overflowed metadata and there is no currently-executing overflowed transaction.

In Figure 7.12 on page 111, the first bar of each group shows the performance of the no-OTID variant of ONETM-Concurrent. Perhaps surprisingly, this configuration achieves the performance of the default ONETM-Concurrent configuration on all workloads but `genome`. This result occurs because (1) processors spend little time in overflowed execution (Figure 7.3 on page 104) and (2) as described above, processors can still clear the overflowed transaction metadata on detecting that it is stale.

### 7.4.1 Summary

In this section we evaluated whether the lazy clearing of metadata employed by ONETM-Concurrent resulted in performance degradation due to spurious conflicts with stale metadata. We found that with the default OTID length of 14 bits, ONETM-Concurrent experiences essentially no performance loss from an idealized configuration with infinite-length OTID's. Moreover, we found that even if ONETM-Concurrent does not employ OTID's at all, it still does not experience spurious conflicts in most cases due to the fact that processors spend so little time in overflowed transactions.

## 7.5 Impact of the Permissions-Only Cache on ONETM Performance

In this section we evaluate the performance impact of the addition of a permissions-only cache to ONETM. We first examine the question of how much a 256-byte read-only permissions-only cache can increase the performance of ONETM-Serialized and ONETM-Concurrent. We then examine the question of whether the fact that this permissions-only cache is organized as a sector cache results in performance degradation from a non-sector cache organization. Finally, we examine the performance impact of smaller and larger permissions-only caches and close the section by analyzing the causes of (generally small) remaining performance differences between ONETM and the idealized unbounded HTM.

### 7.5.1 Impact of the Read-Only Permissions-Only Cache on ONETM-Serialized

Figure 7.13 on page 113 and Figure 7.14 on page 113 present the impact of the addition of the read-only permissions-only cache on ONETM-Serialized. The permissions-only cache significantly reduces the time spent in overflows on several workloads, most notably `genome`. However, the performance of ONETM-Serialized on `yada` suffers relative to the idealized system even after the

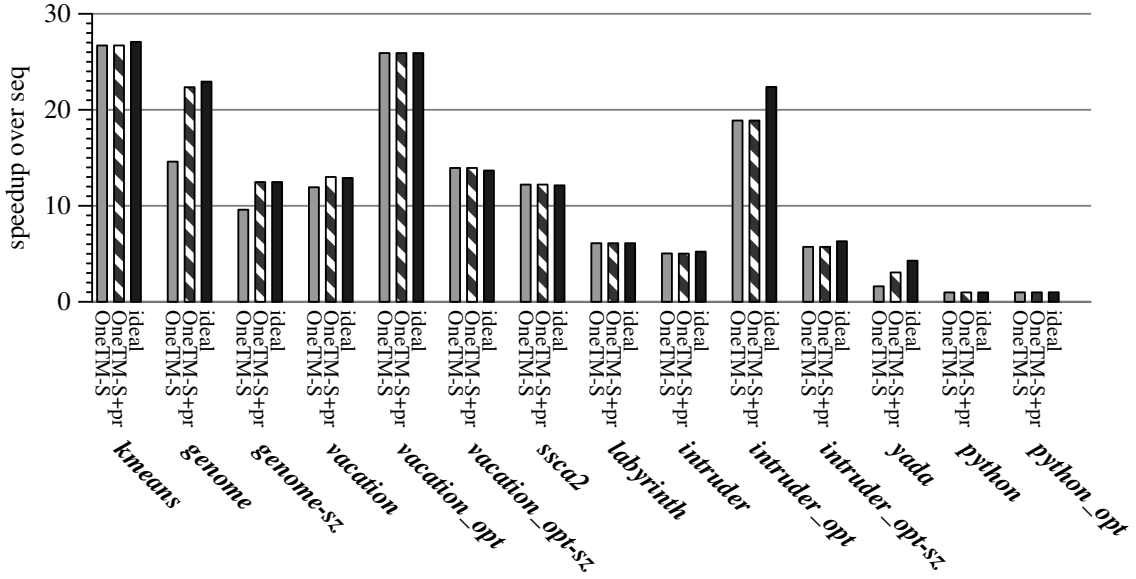


Figure 7.13: **Impact of read-only permissions-only cache on ONETM-Serialized performance.** "+pr" represents the addition of a 256-byte read-only permissions-only cache.

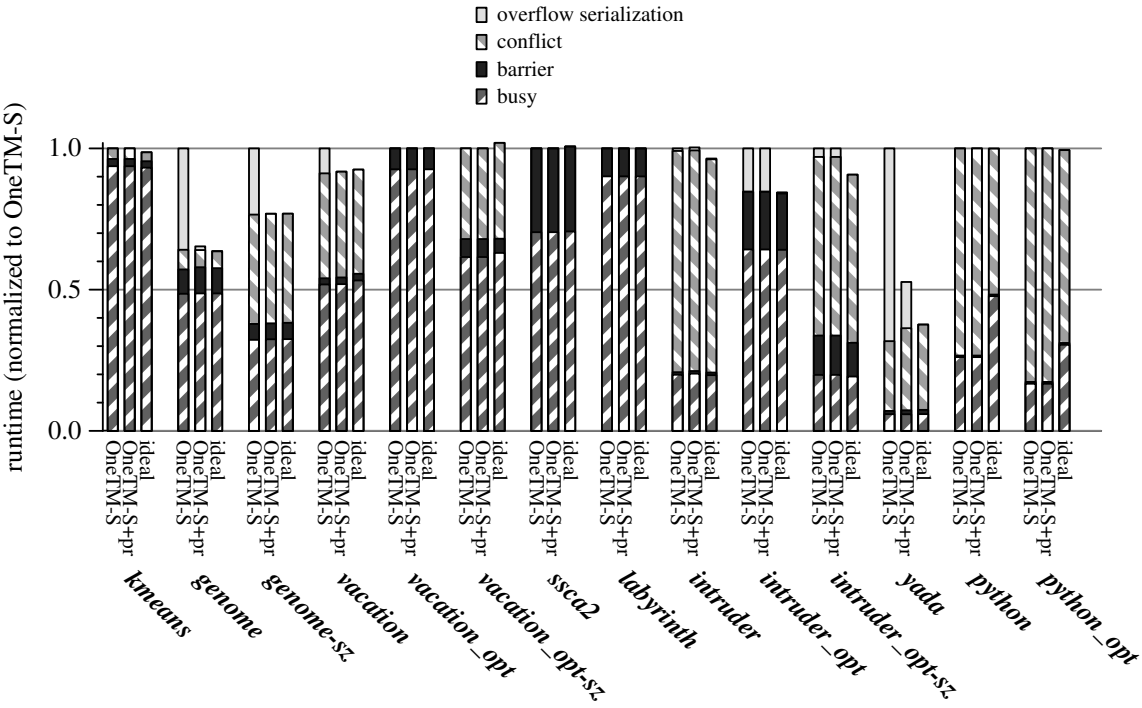


Figure 7.14: **Impact of read-only permissions-only cache on ONETM-Serialized execution time.**

addition of the permissions-only cache. We note that `intruder_opt` is unaffected by the addition of the permissions-only cache; further investigation revealed that the unbounded transactions in that workload are in fact due to pagefaults (Section 6.3).

### 7.5.2 Impact of the Read-Only Permissions-Only Cache on ONETM-Concurrent

We next examine the impact of the permissions-only cache on ONETM-Concurrent performance. As Figure 7.15 on page 115 illustrates, the addition of the permissions-only cache serves to essentially equalize the performance of ONETM-Concurrent with that of the idealized HTM. Figure 7.16 on page 115 indicates that ONETM-Concurrent augmented with the permissions-only cache suffers virtually no serialization from overflows.

### 7.5.3 Sensitivity to Sector Cache Organization

As described in Section 5.2, we organize the permissions-only cache as a sector cache in order to reduce tag overhead. This organization, however, also has the potential to reduce performance from a non-sector cache organization by reducing coverage if spatial locality is poor. We quantitatively evaluate the impact of the sector cache organization here by comparing the performance of the permissions-only cache to a variant configuration that is organized as a non-sector cache (*i.e.*, the metadata for each block resides in its own cache line). Figure 7.17 on page 116 and Figure 7.18 on page 116 present these results for ONETM-Serialized and ONETM-Concurrent respectively. These figures indicate that in both cases the sector cache organization results in almost no performance loss from a non-sector cache organization.

### 7.5.4 Sensitivity to Permissions-Only Cache Size

In the previous analysis, we have evaluated the performance impact of a 256-byte permissions-only cache that can hold the conflict information of 64 kilobytes of data (Section 7.1). Here we study the impact of varying the size of the permissions-only cache by evaluating two variant configurations. The first variant configuration that we study is an 2-byte permissions-only cache. This variant can hold the conflict information of 8 blocks. It can thus eliminate transaction aborts due to cache conflict evictions, but likely cannot eliminate aborts due to cache capacity issues (*i.e.*, its impact is similar to that which a victim buffer would have). This variant is not organized as a sector cache as

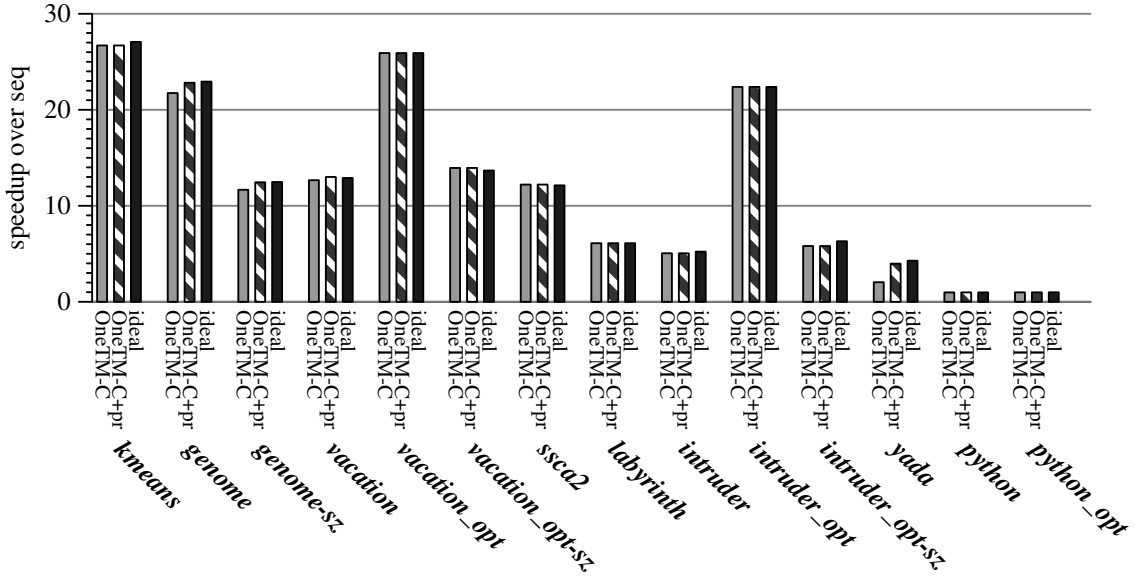


Figure 7.15: **Impact of read-only permissions-only cache on ONETM-Concurrent performance.** "+pr" represents the addition of a 256-byte read-only permissions-only cache.

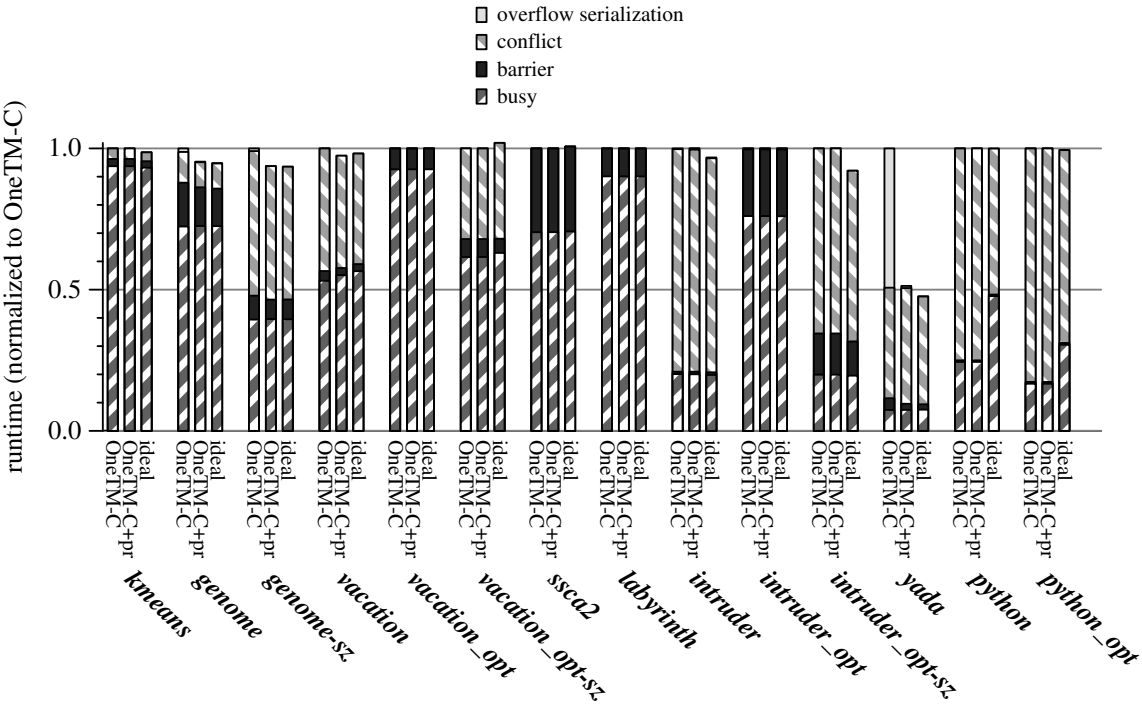


Figure 7.16: **Impact of read-only permissions-only cache on ONETM-Concurrent execution time.**

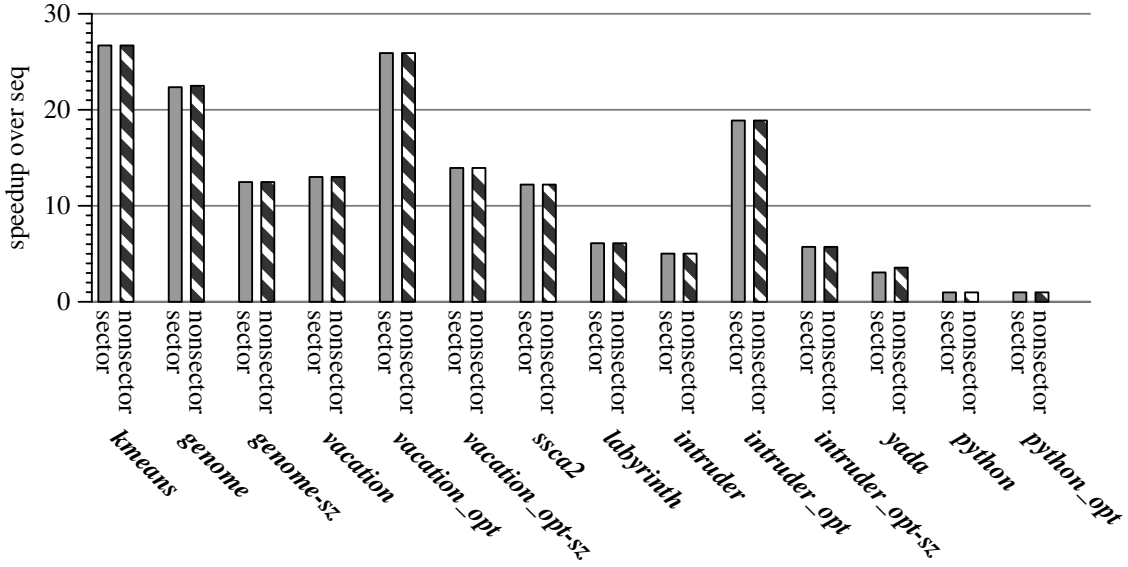


Figure 7.17: **Impact of read-only permissions-only cache sector cache organization on ONETM-Serialized.**

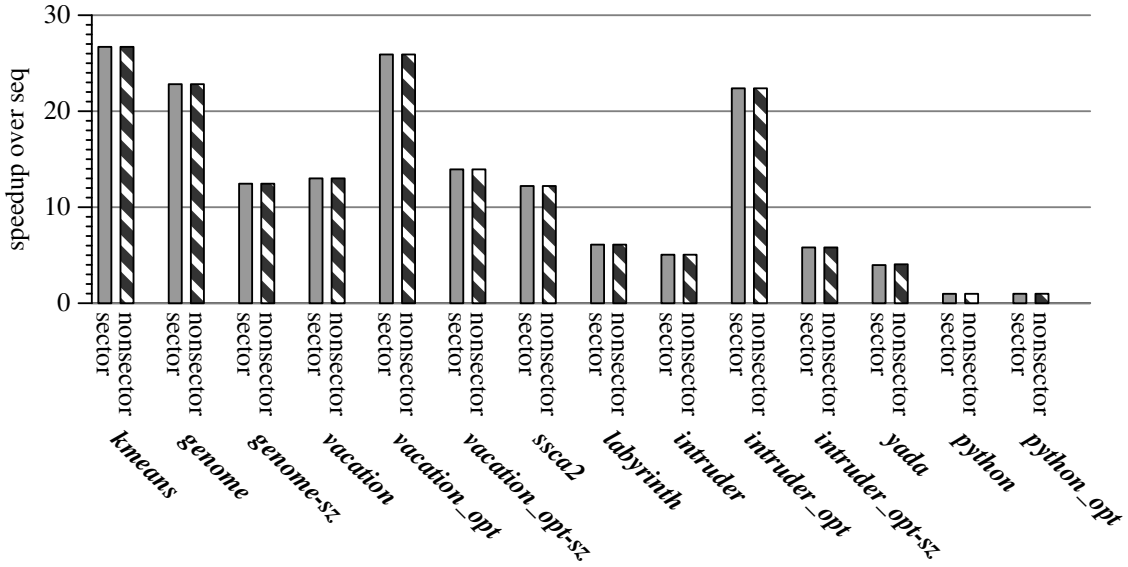


Figure 7.18: **Impact of read-only permissions-only cache sector cache organization on ONETM-Concurrent.**

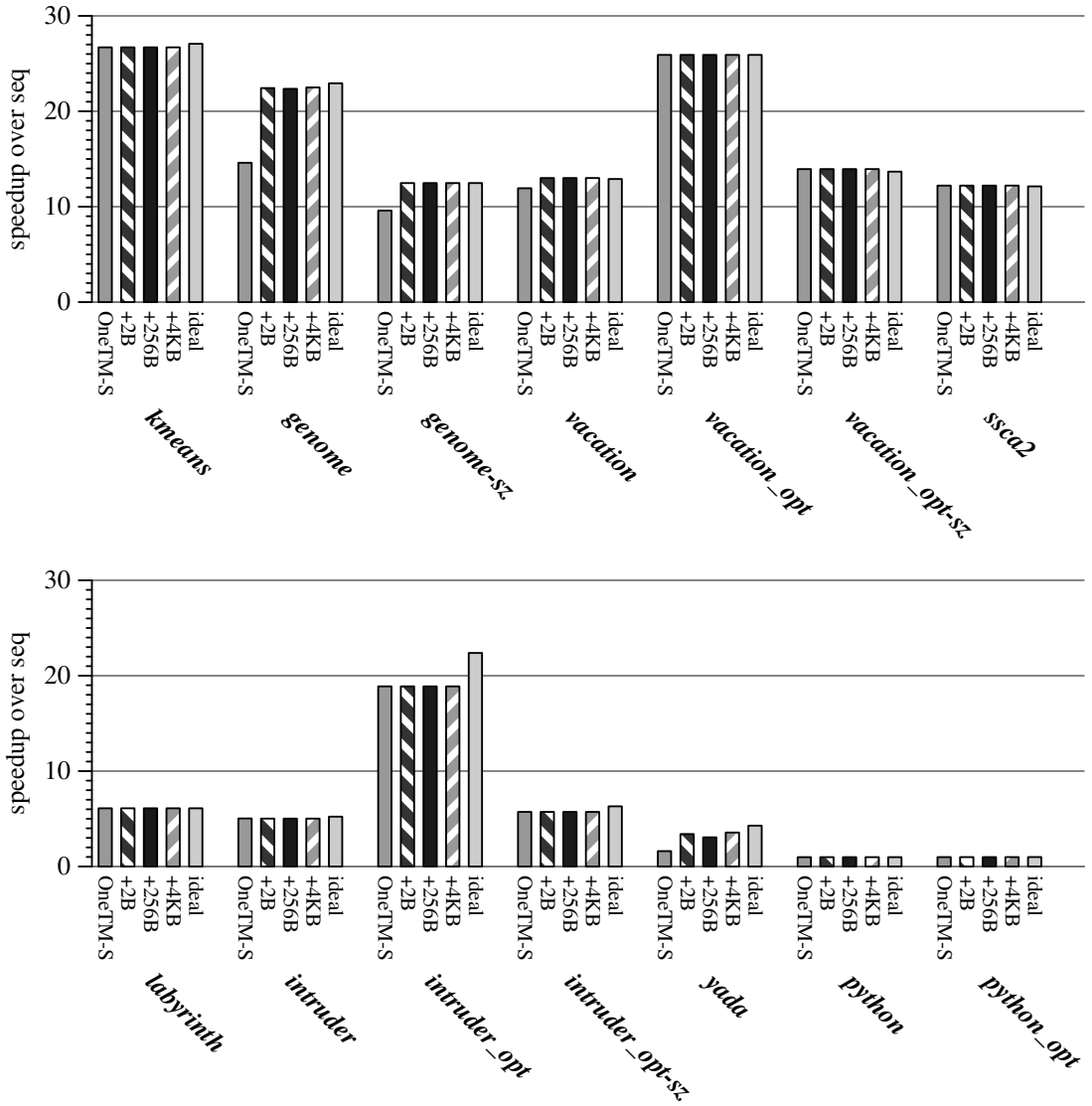


Figure 7.19: **Impact of read-only permissions-only caches of various sizes on ONETM-Serialized performance.**

---

it only has 8 cache lines. The second variant that we study is a 4-kilobyte permissions-only cache. This variant (which is organized as a sector cache) can hold the conflict information of a megabyte of data. As the data presented in Chapter 3 indicates that the transactions in these workloads touch roughly 64 kilobytes of data at a maximum, this variant effectively serves as an upper bound for the impact of adding a permissions-only cache.

Figure 7.19 on page 117 and Figure 7.20 on page 118 present the performance impact of these variant permissions-only cache configurations on ONETM-Serialized. The 4-kilobyte permissions-

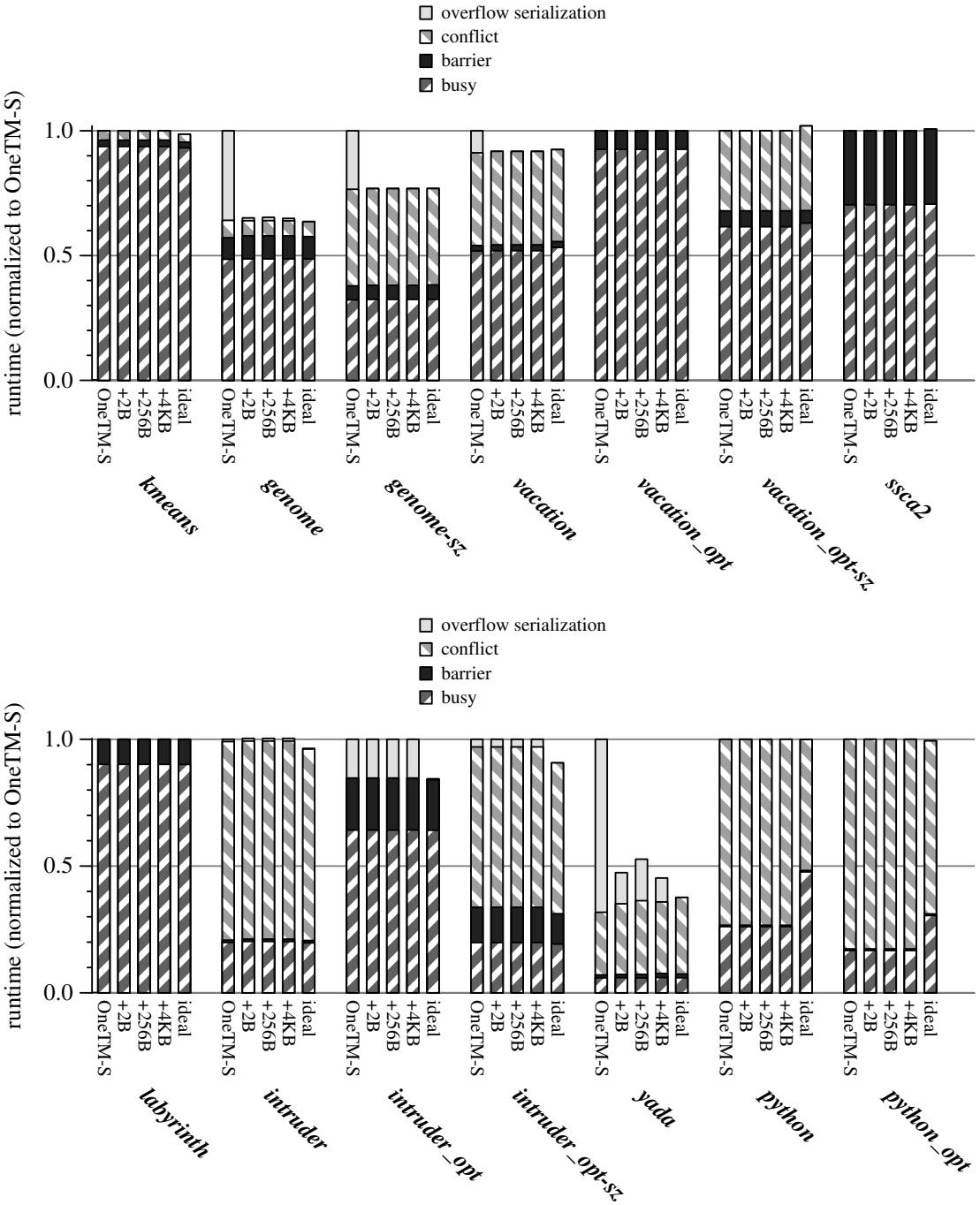


Figure 7.20: **Impact of read-only permissions-only caches of varying sizes on ONETM-Serialized execution time.**

only cache eliminates cache overflows on all workloads but `yada` (the remaining time in unbounded transactions on the other workloads is due to pagefaults occurring within transactions). On `yada`, evictions of transactionally-written blocks occur.

We also note that in several cases the 2-byte permissions-only cache is able to match the performance of the 256-byte permissions-only cache. This fact suggests that many of the overflows in these workloads are indeed due to cache conflict evictions rather than capacity evictions, meaning that the permissions-only cache plays a victim buffer-like role on these workloads[1].

Figure 7.21 on page 120 and Figure 7.22 on page 121 present the analogous results for ONETM-Concurrent. Once again, the 4-kilobyte permissions-only cache does not increase performance over the 256-byte permissions-only cache. Moreover, the 2-byte permissions-only cache is generally able to match the performance of the 256-byte version.

[1]The permissions-only cache, however, has the ability to increase the capacity of the bounded HTM far beyond the addition of a victim buffer.

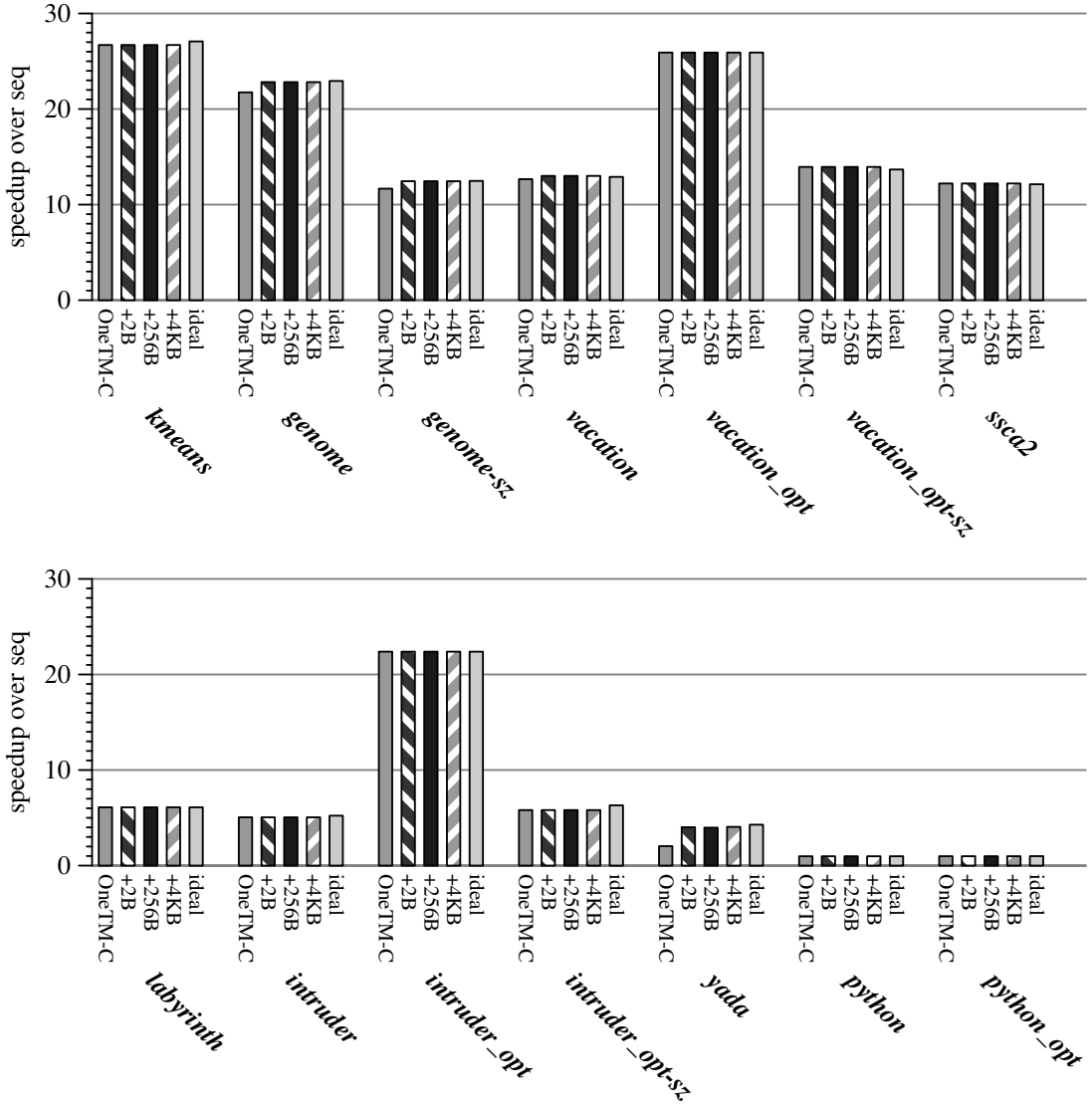


Figure 7.21: **Impact of read-only permissions-only caches of varying sizes on ONETM-Concurrent performance.**

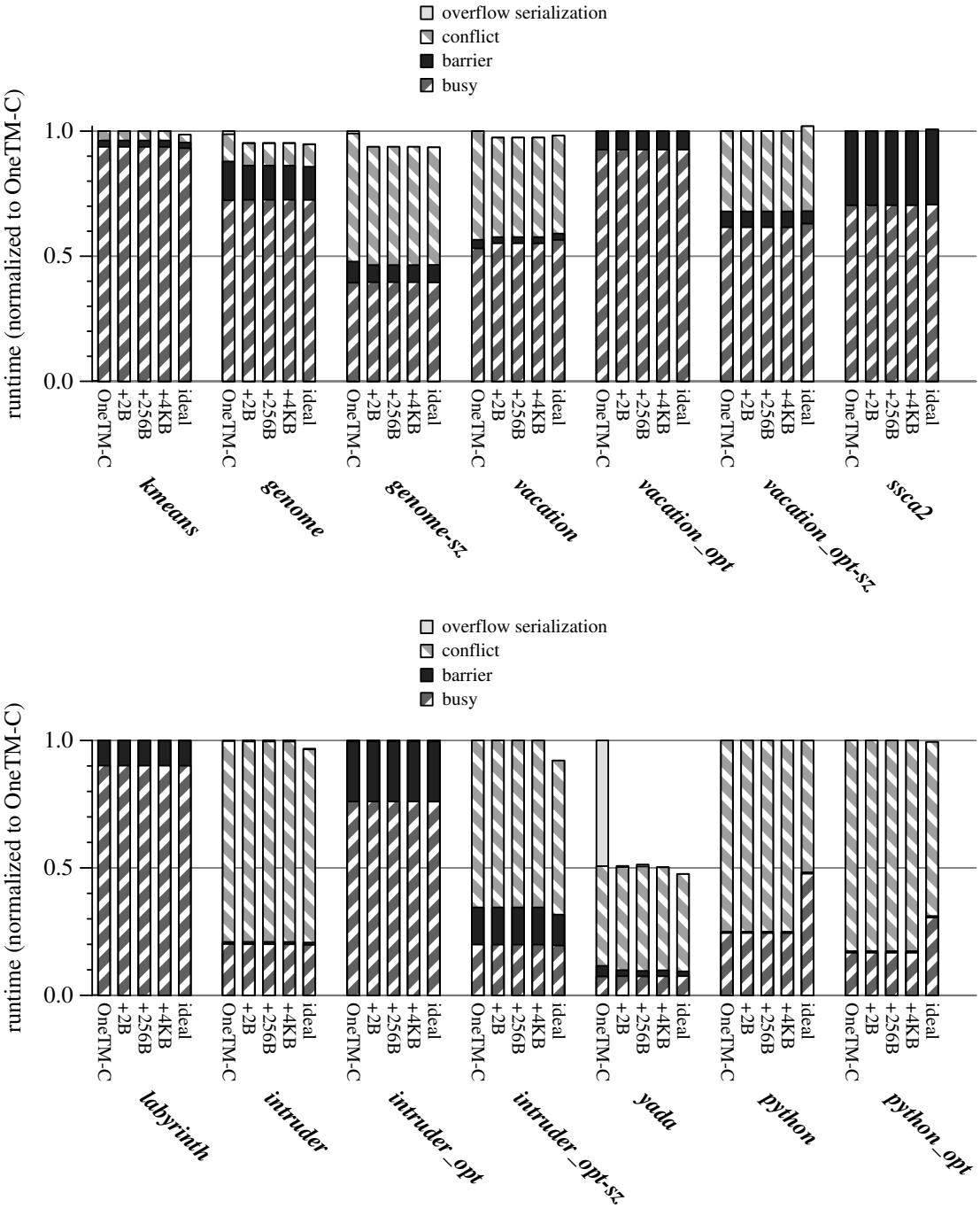


Figure 7.22: **Impact of read-only permissions-only caches of varying sizes on ONETM-Concurrent execution time.**

### 7.5.5 The Remaining Performance Gap between ONETM and the Idealized HTM

As illustrated in Figure 7.20 on page 118 and Figure 7.22 on page 121, there are small remaining differences between the performance of ONETM and that of the idealized system even after the addition of a large (4 kilobyte) permissions-only cache. There are several potential causes of these remaining differences. First, ONETM differs from the idealized system in that ONETM employs cleaning for version management while the latter employs an idealized no-cost unbounded log. Second, ONETM with the addition of the read-only permissions-only cache still must abort on eviction of a transactionally-written block, whereas the idealized system supports unbounded write sets as well as read sets. Finally, while all three systems handle non-abortable events in transactions (such as pagefaults) by restarting a transaction in non-abortable mode, ONETM induces system serialization in this case whereas the other two systems do not. In this subsection, we determine how these implementation differences contribute to the remaining performance differences between ONETM and the idealized system here.

We first analyze the reasons for the remaining performance differences between ONETM-Serialized and the idealized system. To do so, we study four configurations. The first configuration ("OneTM-S+4kb") is ONETM-Serialized configured with a 4-kilobyte permissions-only cache. This configuration eliminates the performance impact of transactions' read sets overflowing on these workloads. The second configuration ("+log") changes the version management mechanism to be the same no-cost unbounded log employed by the idealized system, thus eliminating this difference between ONETM-Serialized and the idealized HTM. The third configuration ("+r/w") builds on the second configuration by changing the permissions-only cache to be read/write. This configuration eliminates the performance impact of transactions' write sets as well as read sets overflowing. Finally, the fourth configuration ("ideal") is the idealized system. Any remaining performance difference between the third and fourth configurations is due to the fact that ONETM-Serialized serializes the system when a transaction must handle a non-abortable event, whereas the idealized system does not.

Figure 7.23 on page 123 presents a runtime breakdown of these configurations. The most common cause of performance difference between ONETM-Serialized is the use of cleaning rather than an idealized zero-cost log for version management. `yada` receives a performance benefit from

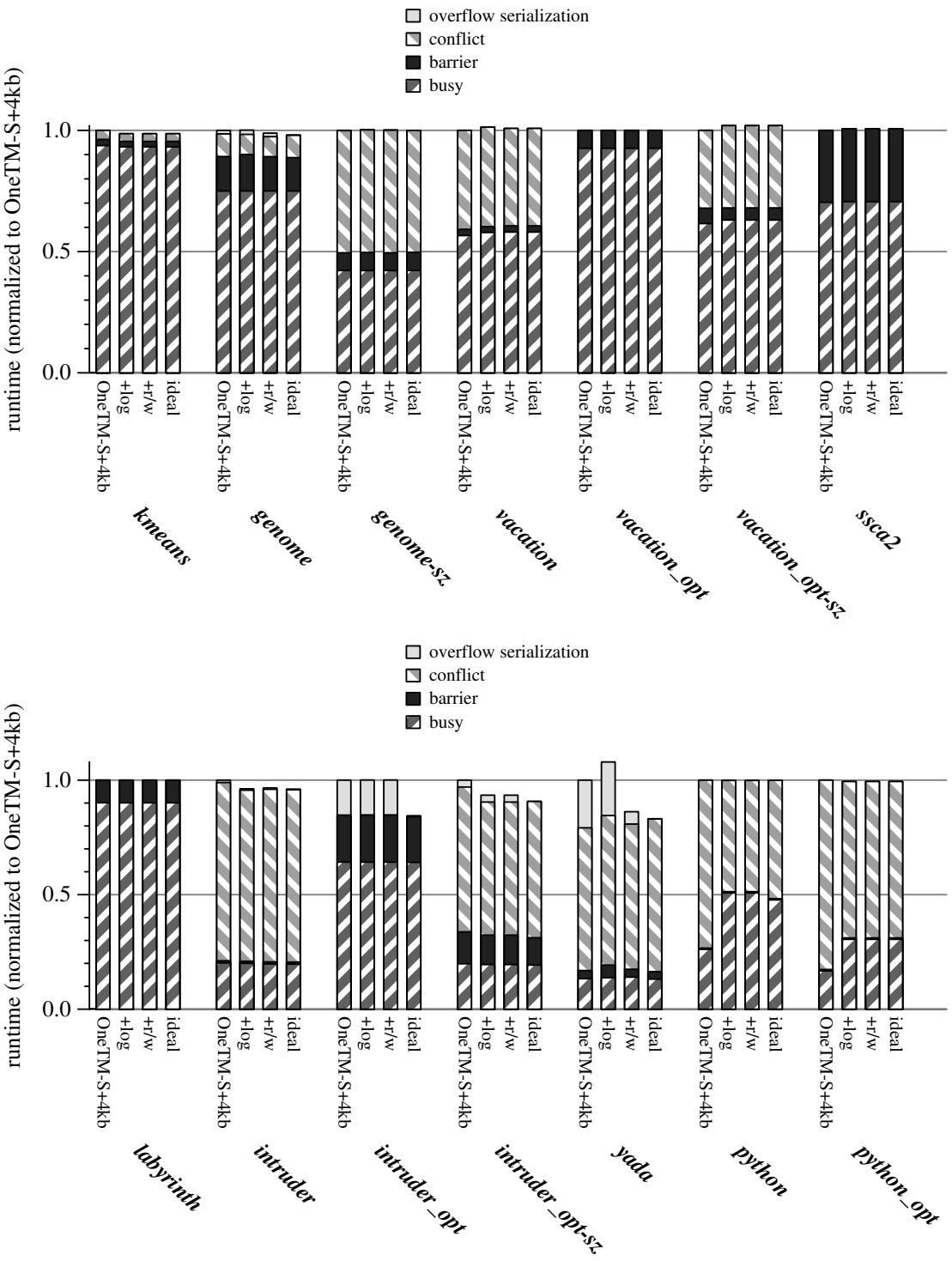


Figure 7.23: **Analysis of remaining performance differences between ONETM-Serialized and the idealized HTM.** "OneTM-S+4kb" shows the performance of ONETM-Serialized with a 4-kilobyte read-only permissions-only cache. "+log" changes the version management mechanism to idealized no-cost unbounded logging. "+r/w" additionally changes the permissions-only cache to be read/write. "ideal" represents the idealized unbounded HTM.

being able to overflow transactionally-written as well as transactionally-read blocks from the L1 cache. Finally, `intruder_opt` and `yada` experience pagefaults within transactions.

Figure 7.24 on page 125 presents the corresponding runtime breakdown for ONETM-Concurrent. In this case, the performance differences on all workloads except `yada` are solely due to the difference in version management. As noted above, `yada` benefits from the read/write permissions-only cache.

### 7.5.6 Permissions-Only Cache Summary

In this section we evaluated the impact of the permissions-only cache on the performance of ONETM. We found that a 256-byte permissions-only cache could significantly increase the performance of ONETM-Serialized, but could not equalize performance with the idealized HTM in all cases. The remaining difference is largely due to the fact that the idealized system handles non-abortable operations within transactions more gracefully than ONETM-Serialized. By contrast, the addition of the permissions-only cache enabled ONETM-Concurrent to perform similarly to the performance of the idealized unbounded HTM. We also found that the sector cache organization of the permissions-only cache does not degrade performance from a non-sector cache organization and that a permissions-only cache as small as 2 bytes could have a significant positive impact on the performance of ONETM.

## 7.6 Discussion of Power Implications of Our Proposals

In this section, we discuss the potential implications of our proposals on power. We first discuss ONETM and then discuss the permissions-only cache.

**ONETM.** In ONETM-Serialized, the overflowed bit of the STSW is (1) snooped on external write requests, (2) coherently written when an overflowed transaction begins and commits, and (3) re-read by all other processors after being invalidated by the overflowed transaction's write. As discussed above, in many modern processors external write requests already snoop the load queue [35]. The power impact of the two latter operations is directly correlated with how often transactions must transition to overflowed mode. The data in this chapter indicates that such transitions occur infrequently, and thus the overflowed bit is unlikely to have a significant impact on power. The most significant additional power implication of ONETM-Concurrent is that it potentially widens the

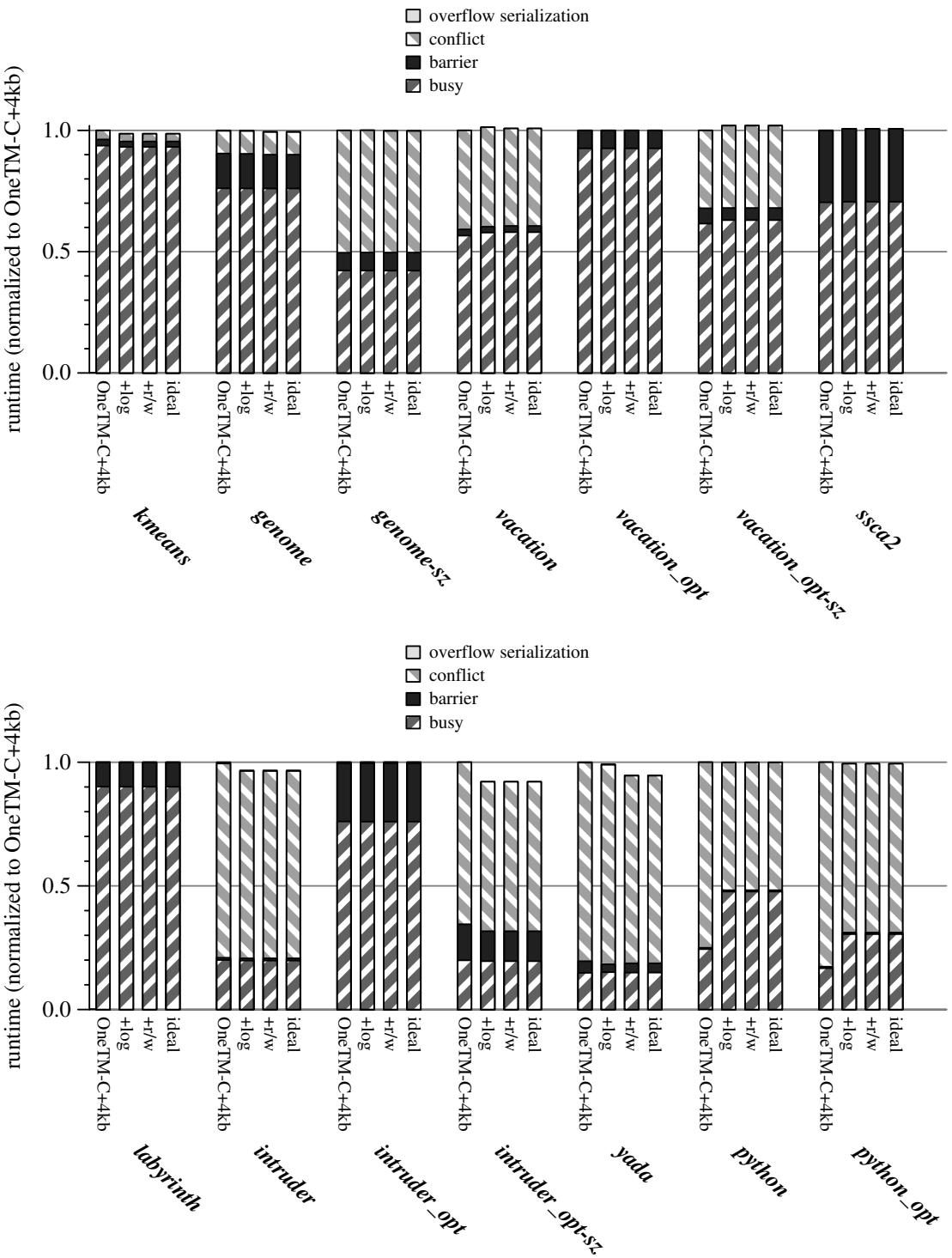


Figure 7.24: **Analysis of remaining performance differences between ONETM-Concurrent and the idealized HTM.** "OneTM-C+4kb" shows the performance of ONETM-Concurrent with a 4-kilobyte read-only permissions-only cache. "+log" changes the version management mechanism to idealized no-cost unbounded logging. "+r/w" additionally changes the permissions-only cache to be read/write. "ideal" represents the idealized unbounded HTM.

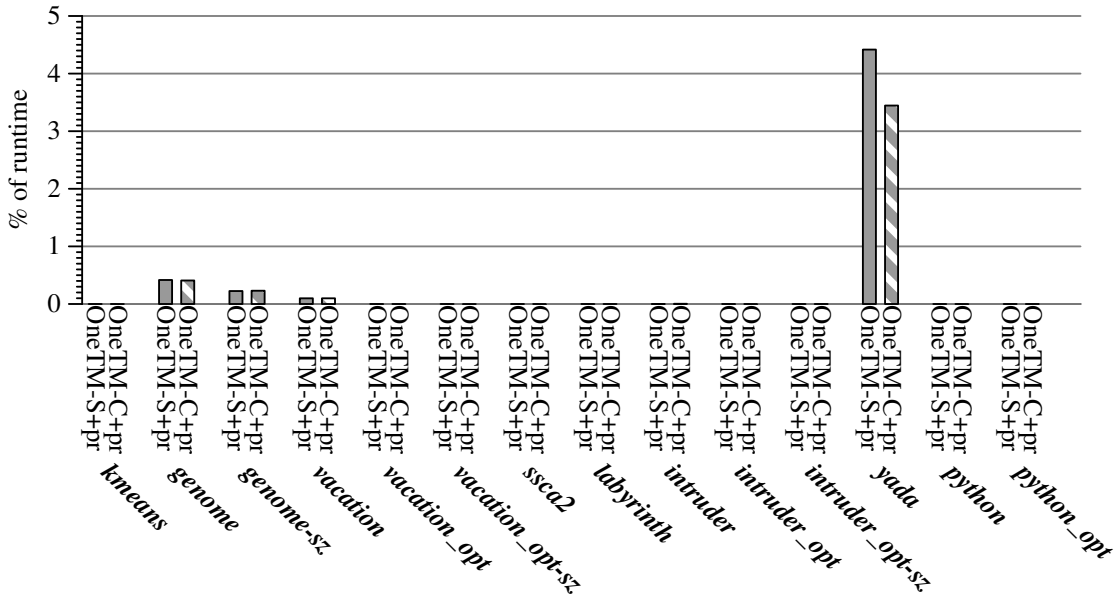


Figure 7.25: **Percent of time the permissions-only cache is non-empty.** In these configurations, the permissions-only cache is in its default 256-byte, read-only configuration.

---

data payload by two bytes. If datablocks are 72 bytes (64 bytes of data in addition to the 8-byte address header), this addition represents a 2.8% overhead.

**Permissions-only cache.** The permissions-only cache is (1) read by external coherence requests as part of conflict detection, (2) updated when a transactional block is replaced from the data cache, and (3) invalidated on a commit or abort. It is not accessed for processor-local memory operations. As such, it is accessed significantly less often than the L1 cache. Finally, the permissions-only cache is often empty. In these circumstances, it can be completely powered down to save dynamic and static power.

Figure 7.25 on page 126 shows the percentage of execution time that the permissions-only cache is non-empty (for the 256-byte read-only permissions-only cache). For all workloads except `yada`, the permissions-only cache is active for less than 1% of execution. While the usage of the permissions-only cache in `yada` is higher, it is still less than 5% of execution.

## 7.7 Summary

In this chapter we evaluated the performance of ONETM on our workloads. We found that on several workloads ONETM-Serialized can match the performance of an idealized HTM that handles

overflows with full concurrency and no overheads, as these workloads simply do not overflow. This result illustrates the fact that if overflows are extremely rare, they can be handled simply without a great impact on overall performance. However, we also found that even a small amount of time spent in overflowed execution could cause performance degradation in ONETM-Serialized. We found that the increased concurrency of ONETM-Concurrent increases robustness to overflows. Even ONETM-Concurrent, however, cannot match the performance of the idealized HTM on all workloads.

We then evaluated the impact of the addition of the permissions-only cache. We found that for ONETM-Serialized the permissions-only cache significantly increases performance but that the combination still falls short of the idealized HTM in some cases (largely due to the fact that ONETM-Serialized serializes the system to handle non-abortable events within transactions). Combining ONETM-Concurrent with the permissions-only cache, however, results in a system that matches the performance of the idealized fully-concurrent unbounded HTM across the workloads that we use.

As we discussed in Chapter 3, after overflows are eliminated as a performance bottleneck, the overall performance of many workloads is still limited by conflicts. In that chapter we also found a pattern of conflicts on auxiliary data that were challenging to eliminate via software restructuring. The next chapter details RETCON, our proposal for eliminating such auxiliary data conflicts in hardware.

# Chapter 8

# RETCON: Eliminating Auxiliary Data Conflicts

As the last chapter illustrated, the combination of ONETM and the permissions-only cache achieves the performance of an idealized unbounded HTM that handles overflows with full concurrency and no overheads on the workloads that we use. Nonetheless, performance in many cases is still limited by conflicts. In particular, as we discussed in Chapter 3, several workloads experience a common pattern of conflicts on auxiliary data, *i.e.*, data such as hashtable occupancy fields and reference counts that is peripheral to a transaction's main copmutation. As detailed in that chapter, these conflicts can be challenging to eliminate via software restructuring. In this chapter we present RETCON, our hardware-based proposal for eliminating auxiliary data conflicts.

We observe that auxiliary data is usually accessed by short, simple computations that do not affect the larger transaction. This property suggests an approach of allowing conflicts to occur without rollback, reacquiring lost data at commit, and using the current values of this data to repair local state as necessary. Such a repair-based approach was proposed by ReSlice [96] to lessen the impact of conflicts in thread-level speculation, and similar slicing has been employed in other contexts as well [21, 48, 103]. These proposals use instruction-based repair by saving the dependent instructions of a conflicting load to later re-execute these instructions with the updated value of the load (either in a special-purpose core or by re-using the resources of the processor itself).

Guided by the nature of this auxiliary data, we propose a different approach. As the processor executes a transaction, it also tracks the relationship between input values and output values symbol-

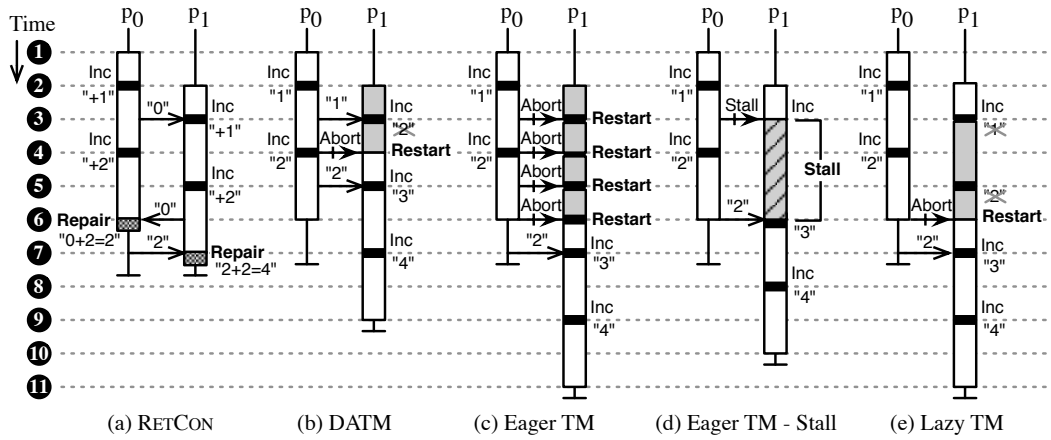


Figure 8.1: **Comparison of RETCON to other approaches.** $P_0$ and $P_1$ begin transactions at times ❶ and ❷ respectively. Each transaction performs two increments to a shared counter variable that is initialized to zero. *(a)* RETCON symbolically tracks the counter address and repairs its value at commit by adding the computed increment. *(b)* DATM [92] forwards the speculatively incremented counter variable at time ❸, but must abort a transaction when the second increment introduces a cyclic dependence at time ❹. *(c)* In EagerTM, $P_1$ suffers repeated aborts until $P_0$ commits at time ❻. *(d)* EagerTM-Stall stalls $P_1$'s first increment until $P_0$ commits at time ❻. *(e)* In LazyTM, $P_1$ performs both its speculative increments but then aborts when it detects a conflict on the commit of $P_0$ at time ❻.

---

ically. Conditionals form constraints on the acceptable range of values that an input can take when reacquired at commit. At commit time, all inputs that have been lost are reacquired, constraints are checked, and symbolic computation is reapplied to these values (see Figure 8.1 on page 129).

Our instantiation of this approach, RETCON,[1] is tailored to fit the needs of the auxiliary data present in our workloads. RETCON tracks an input symbolically through a sequence of loads, simple arithmetic operations, branches, and stores, with more complex computation creating a constraint that the input value be the same at commit. To track symbolic information, RETCON adds a buffer to hold the initial values of symbolically-tracked blocks, a buffer to hold constraints, and a buffer to hold symbolically-tracked stores.

Although RETCON's focus is on repairing remotely changed values, its mechanisms have the secondary benefit of reducing conflicts in other ways. RETCON's resolution of conflicts at commit implicitly provides selective lazy conflict detection [15, 98, 112]. In addition, by performing conflict detection based on values [82, 109], RETCON also avoids conflicts due to false sharing [52], silent sharing, and temporally silent sharing [62].

[1]Retcon, short for *retroactive continuity*, refers to soap operas' and comic books' practice of revising past events as necessary to match current reality.

In the remainder of this chapter, we first describe the architecture and high-level RETCON algorithm in Section 8.1, followed by a discussion of operational details in Section 8.2 and implementation optimizations in Section 8.3. We detail benefits that RETCON provides beyond the ability to repair conflicts on auxiliary data in Section 8.4. Finally, we discuss related work in Section 8.5 and summarize the chapter in Section 8.6.

## 8.1 RETCON Architecture and High-Level Operation

As described above, RETCON tracks the symbolic relationships between inputs and outputs. As transactions execute, values are tagged with a symbolic representation of the computation that produced that value. For example, if the processor loads a value and then increments it twice, the processor tracks information indicating that it can calculate the final value by adding two to whatever value the load eventually takes. Such *symbolic values* propagate through registers and memory, while conditional operations result in constraints that the symbolic value must satisfy at the time of commit. If computation that is too complex to track symbolically occurs on a given input, the system constrains that input to remain equal to its original value when reacquired. In this section, we describe the architecture and high-level operation that allow RETCON to support these tasks.

Key to RETCON are the concepts of *symbolic locations*, *symbolic values*, and *symbolic constraints*. A symbolic location is a memory address that RETCON decides to track symbolically (determined via a predictor trained by past history of conflicts). The symbolic value of a register or memory location is a representation of the computation that was performed to calculate the concrete value of that location. As an example, the symbolic value of the register output of the first load to symbolic location $A$ would be "$[A]$". Finally, symbolic constraints are a combination of a symbolic value, a boolean operator, and a constant. An *equality constraint* is a special type of constraint that simply specifies that a given symbolic location must be equal to the value first read for that location by the transaction.

RETCON optimizes for design simplicity by restricting the type of computation that it can track symbolically. It restricts operations to have at most one symbolic input and tracks only data (not memory addresses) symbolically. These restrictions allow RETCON to maintain symbolic information efficiently and admit a streamlined commit process, while still supporting our goal of being able

to repair the effects of peripheral data conflicts. We describe RETCON operation in detail below, followed by examples of code idioms causing conflicts that RETCON can and cannot repair.

### 8.1.1 RETCON Operation

**Initiating symbolic tracking.** During transaction execution, loads and stores not involved with symbolic repair use the conflict detection mechanism of the underlying TM system. A *symbolic load*, a load that reads from a symbolic location, initiates symbolic tracking of dependent operations by associating a symbolic value with the load's output register (recorded in the symbolic register file, described in Figure 8.2 on page 132). The value written to the register file by a symbolic load is the processor's best-guess value for the location (*i.e.*, the architectural value at the time or a value prediction [83, 84, 109]) and execution continues based on that concrete value. The first load to a symbolic location also records the *initial concrete value* of the location (recorded in the *initial value buffer*, described in Figure 8.2 on page 132). A load to a symbolic location does *not* set the read bit in the cache, thus allowing the block to be stolen away without triggering a conflict.

**Execution with symbolic tracking.** Transaction execution is determined entirely by the concrete values, but symbolic values are tracked and propagated from instruction to instruction. If an instruction's specific operation is one that can be tracked symbolically, the symbolic input is propagated to the symbolic value output. The processor performs as much computation as it can during this propagation. As an example, if a register with concrete value 5 and symbolic value "$[A]+7$" is used as input to an increment instruction, the output register's concrete value would be 6 and its symbolic value would be "$[A]+8$".

**Symbolic tracking through memory.** When writing a register with a symbolic value to memory, both the concrete and symbolic values are recorded and propagated to subsequent loads (using the *symbolic store buffer,* described in Figure 8.2 on page 132). Correspondingly, all loads check the symbolic store buffer (as well as the initial value buffer, as described above) in parallel with the data cache. When a load forwards from a store that has a symbolic value, it copies that symbolic value (rather than initializing its symbolic value to the address of the store). In essence, RETCON collapses all store-load forwarding during execution. Figure 8.3 on page 132 contains a flowchart of the operation of loads and stores in RETCON[2].

[2]If the symbolic store buffer overflows, the transaction is aborted and re-executed with RETCON disabled.

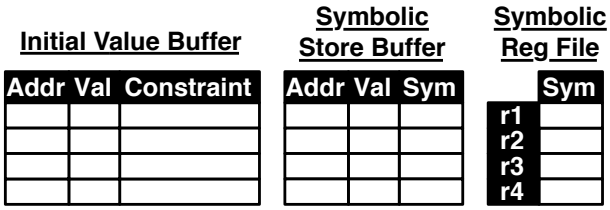


Figure 8.2: **RETCON structures.** The **initial value buffer** is a cache-like structure indexed by data address. Each entry contains the address tag bits, the initial concrete value of the symbolic memory location, and the symbolic constraints associated with that memory location (if any). The **symbolic store buffer** records symbolically-tracked stores. It is indexed by data address and accessed like a conventional cache-like unordered store buffer. Each entry contains the address tag bits, the store's concrete value, and the store's symbolic value (if any). The **symbolic register file** records the current symbolic value (if any) for each register. The value recorded in the traditional register file is the concrete value of each register, which is used to guide execution.

---

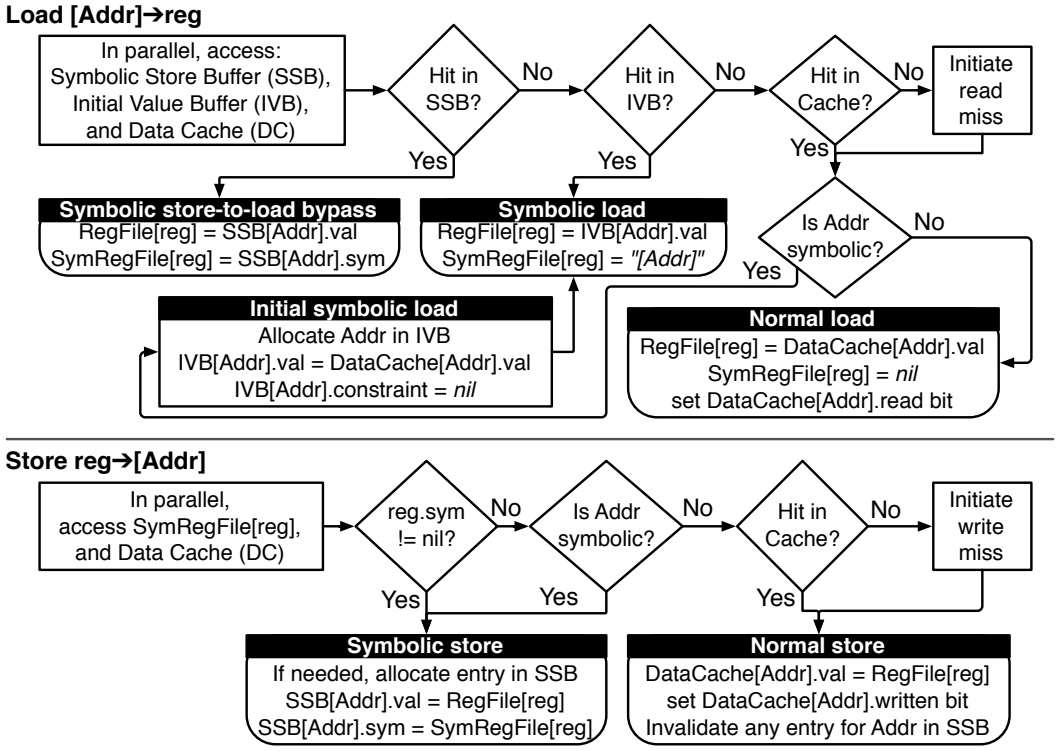


Figure 8.3: **RETCON memory operation flowchart.**

**Symbolic control-flow constraints.** If the source register of a branch has a symbolic value, RETCON adds a symbolic constraint to capture the necessary condition to ensure consistent control flow. For example, a taken branch based on a register with symbolic value "[$A$]+1" is greater than 5 would generate the constraint: "[$A$]+1>5" or, simplified, "[$A$]>4". Non-taken branches record the negation of the branch condition ("[$A$]<=4" in this case). The constraint is recorded in the initial value buffer entry corresponding to the root memory location of the symbolic value.

**Equality constraints.** An equality constraint is set whenever a symbolic input is used in a way that cannot be tracked symbolically, thus ensuring that any difference in the initial value and final value will result in an abort. Equality constraints are introduced when symbolic values are supplied as inputs to (1) complicated arithmetic operations the implementation has chosen not to track symbolically (*e.g.*, integer divide) or (2) the address calculation of loads or stores (but, critically, not the data input of store instructions). In addition, if an operation has multiple symbolic values as inputs, equality constraints are set on all but one to maintain the invariant that all operations have at most one symbolic input.

**Pre-commit repair.** As part of the commit process, the system enforces symbolic constraints by re-loading the *final concrete value* for each symbolic location. Symbolic constraints are evaluated using this final concrete value to ensure that the control flow remains valid. Next, RETCON generates the final concrete values for each symbolic register value and symbolic memory value (*i.e.*, stores with symbolic data input register values). This involves updating the concrete value in the register file and writing concrete values into the data cache. To ensure atomicity during the repair process, all loads and stores set the read/written bits in the cache and conflict detection reverts to that of the baseline system (described in Chapter 2). Once repair has completed, the normal baseline transaction commit commences. Figure 8.4 on page 134 describes the repair algorithm in detail.

**Example.** Figure 8.5 on page 135 presents a step-by-step example of RETCON's operation. This example shows the symbolic execution and commit operations of a processor due to block A that gets stolen away mid-transaction. The load to register r1 at time ❶ initiates symbolic execution, populates the initial value buffer and the symbolic and concrete register files for r1; note that the read bit in the cache is not set. The symbolic data flows to r2 via the register file at time ❷, which in turn introduces a constraint for A at time ❸. At time ❹, r2 is stored in address B, causing it to be tracked in the symbolic store buffer. The load from B at ❺ forwards from the symbolic store buffer. Also at this time, A is removed from the cache due to a remote request; no conflict is triggered

**RETCON Pre-Commit Process**

**Step #1.** Reacquire all lost blocks to obtain final concrete values, record them in the initial value buffer, and check that all control-flow constraints are satisfied by the current values of the blocks:

```
foreach Address A in Initial Value Buffer (IVB):
  if not already in cache, obtain read permission to A
  set DataCache[A].read bit
  IVB[A].value <- DataCache[A].value
  if new value does not satisfy IVB[A].constraint:
    abort
```

**Step #2.** Update memory and register state based on the values in the initial value buffer (which as of step #1 above, now contains the final concrete values):

```
foreach Address A in Symbolic Store Buffer (SSB):
  if not already in cache, obtain write permission to A
  set DataCache[A].written bit
  if SSB[A].sym == nil:
    DataCache[A].value <- SSB[A].value
  else:
    DataCache[A].value <- SSB[A].sym evaluated on value from IVB

foreach Register R in Symbolic Register File (SRF):
  if SRF[R].sym != nil:
     RF[R].value <- SRF[R].sym evaluated on value from IVB
```

Figure 8.4: **RETCON pre-commit repair algorithm.** To ensure atomicity of the commit process, the speculatively read/written bits are set when reacquiring lost blocks and before writing values into the data cache. If a conflict occurs during this pre-commit process, the baseline conflict management logic is invoked. Once the above repair has completed, the normal baseline transactional commit commences.

because A's read bit was not set. At time ❻, register r1 is overwritten with a new offset. The branch at time ❼ updates the initial value buffer with an additional constraint on A. At time ❽, the symbolic store to the symbolically tracked address A introduces another store buffer entry. The store to block B at time ❾ is non-symbolic, hence B's entry is invalidated in the symbolic store buffer and the store value is written speculatively into the cache. The commit process begins at time ❿ by fetching A into the cache speculatively, verifying that the new value of A (6) still satisfies the constraint and computing the new value to be stored to A in the symbolic store buffer. In the final step of the commit process, r1's concrete value is written back into the register file and store to A is drained from the store buffer to the cache, and any speculative bits in the cache are flash-cleared.

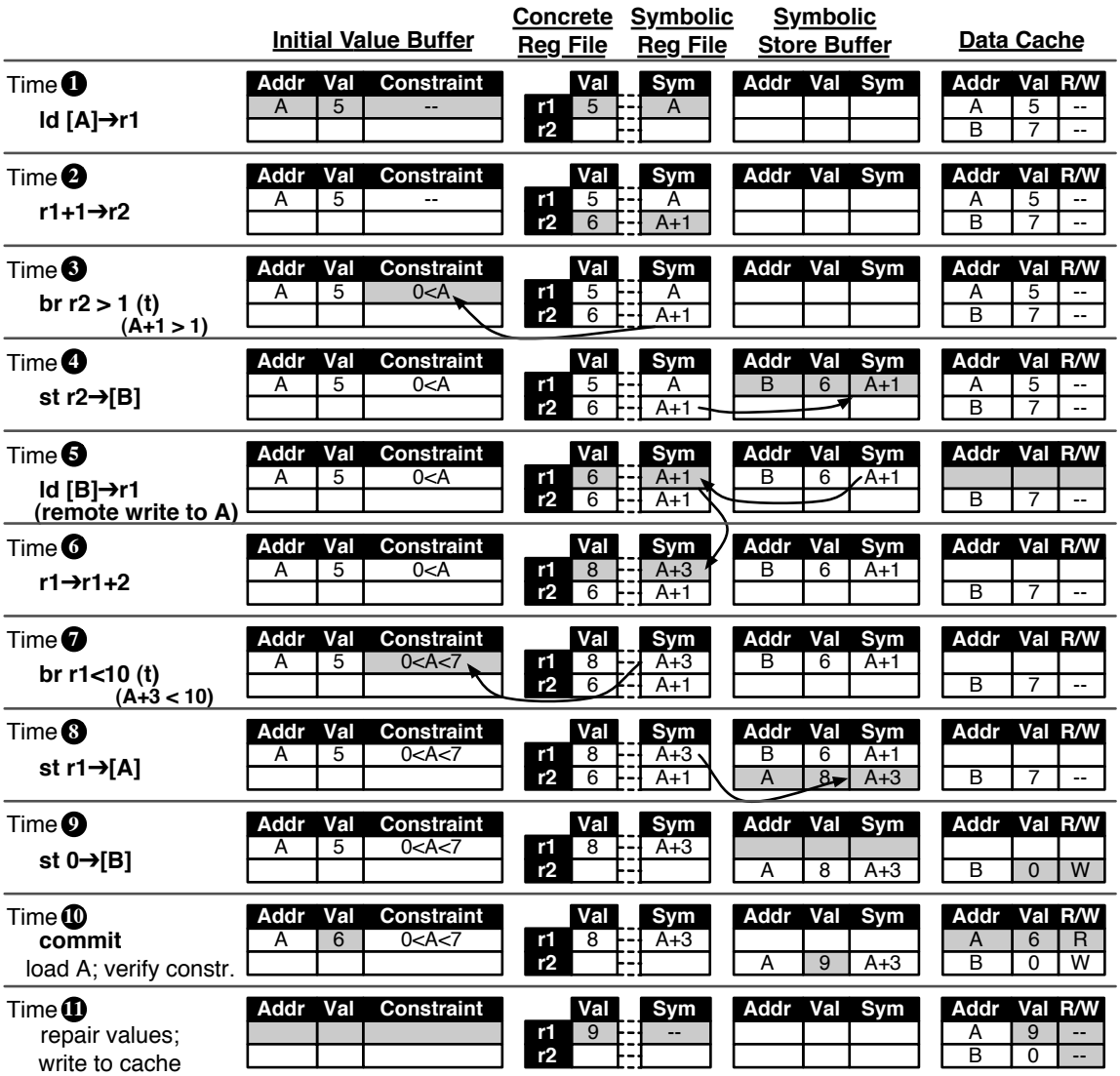


Figure 8.5: **Example of RETCON operation.** The operation of RETCON in this example is described at the end of Section 8.1.

### 8.1.2 Conflict Idioms that RETCON Can Repair

In this section, we detail the basic conflict idioms that RETCON can repair without requiring rollback. Figure 8.6 on page 137 details a case where two transactions concurrently perform additions to a shared variable. As described above, RETCON can track these additions symbolically and use the symbolic information to repair the effects of the conflict at transaction commit. Figure 8.7 on page 137 illustrates a similar but more complex example where RETCON tracks symbolic values through memory. These idioms are similar to those of performance counter and transaction ID fields.

Figure 8.8 on page 138 details a more complex case. In addition to performing addition on a shared variable, one transaction uses the variable as input to a branch. The conflict that occurs on the variable does not change the direction in which the branch is resolved. RETCON tracks the addition symbolically, while the control flow on the symbolically-tracked variable generates a constraint that its value must satisfy when reacquired to avoid an abort (in this case, that the reacquired value be less than 64). At commit RETCON reacquires the variable, verifies that the constraint is satisfied, and uses the symbolic information to repair output state. Figure 8.9 on page 138 illustrates an example where multiple branches occur on a symbolically-tracked input. These idioms are similar to those of conflicts on hashtable occupancy fields that do not affect resizing of the hashtable and conflicts on reference counts that do not affect whether the object is garbage collected.

### 8.1.3 Conflict Idioms that RETCON Cannot Repair

In this section, we detail conflict idioms that RETCON cannot repair. Figure 8.10 on page 139 details a case where a conflict results in the control flow of a transaction being changed. In this case, the constraint that had been generated by that control flow will be violated, and RETCON will abort the transaction. Examples of this idiom are a conflict on a hashtable occupancy field that causes a hashtable resize or a conflict on a reference count that affects whether the object is garbage collected.

Figure 8.11 on page 139 details a case where a conflict occurs on a variable that was used to index into memory. As the use of the variable to index into memory generates a constraint that this variable remain equal to its original value, RETCON will abort the transaction. An example of this idiom is transactions simultaneously enqueuing and dequeuing from a shared queue.

```
                int x = 42;

proc1(){                        proc2(){
  transaction{                    transaction{
    ...                             ...
    x += 7                          x += 5;
    ...                             ...
  }                               }
}                               }

                P = proc1() || proc2()
```

Figure 8.6: **An idiom that RETCON can repair: conflicts involving only addition.** `proc1` and `proc2` conflict on their additions to `x`. RETCON symbolically tracks these additions and reapplies them to the current value of `x` at transaction commit to repair this conflict without rolling back.

```
                int x = 42;
                int y = 0;

proc1(){                        proc2(){
  transaction{                    transaction{
    ...                             ...
    r1 = x;                         r2 = x;
    r1++;                           r2++;
    y = r1;                         x = r2;
    r3 = y;                         ...
    r3++;                           ...
    x = r3;                         ...
  }                               }
}                               }

                P = proc1() || proc2()
```

Figure 8.7: **Example of RETCON tracking through memory.** `x` and `y` are shared variables residing in memory, and `r1`, `r2`, and `r3` are registers. RETCON tracks symbolic values throughout the computation (including the forwarding through memory performed by `proc1`). When `proc1` commits, RETCON reacquires `x`, stores its current value plus one to `y`, and stores that value plus two back to `x`.

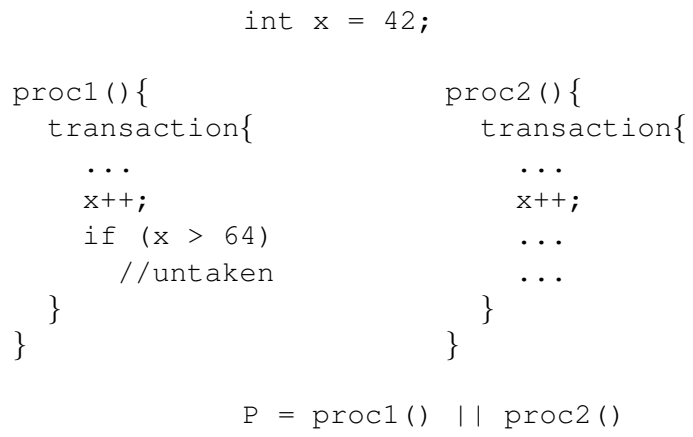


Figure 8.8: **An idiom that RETCON can repair: conflicts not changing control flow.** `proc1` and `proc2` conflict on their additions to `x`. RETCON symbolically tracks these additions. Additionally, the control flow on `x` in `proc1` generates a constraint on the value that `x` can take when reacquired by `proc1`: it must be less than 64. At commit RETCON reacquires `x`, checks that the constraint is satisfied, and reapplies the addition.

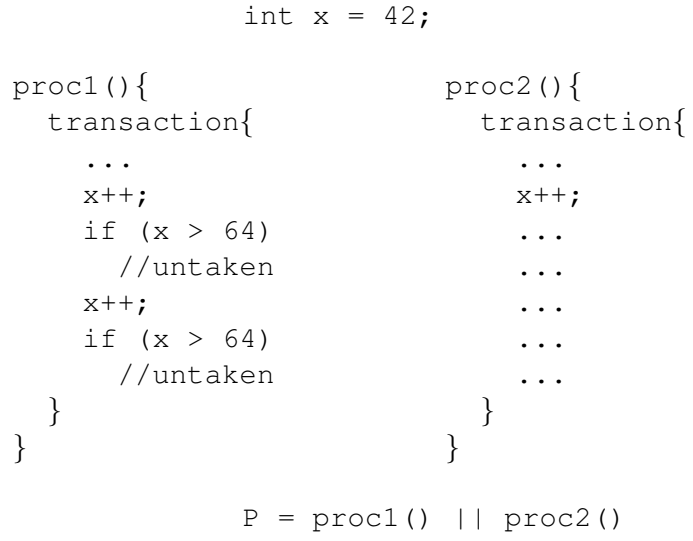


Figure 8.9: **Generating a constraint from multiple branches.** `proc1` and `proc2` conflict on their additions to `x`. RETCON symbolically tracks these additions. The first branch on `x` in `proc1` additionally generates a constraint that `x` must be less than 64 when reacquired. The second branch updates this constraint to the more restrictive constraint that `x` must be less than 63 when reacquired. At commit RETCON reacquires `x`, checks that the constraint is satisfied, and reapplies the addition.

```
int x = 63;

proc1(){                    proc2(){
  transaction{                transaction{
    ...                         ...
    x++;                        x++;
    if (x > 64)                 ...
      //untaken                 ...
  }                           }
}                           }

P = proc1() || proc2()
```

Figure 8.10: **An idiom that RETCON cannot repair: conflicts changing control flow.** `proc1` and `proc2` conflict on their additions to `x`. If `proc2` commits first, then the value of `x` will be 64 when it is reacquired at the commit of `proc1`. In this case, the `x < 64` constraint that was generated by the control flow on `x` would be violated, and RETCON would abort the transaction. Repairing the transaction in this case would require executing the other direction of the branch, which is outside the scope of RETCON.

```
node* head = 0xffbb;

proc1(){                  proc2(){
  transaction{              transaction{
    t = head->task;           ...
    do_work(task);            head = 0xcb0a;
  }                         }
}                         }

P = proc1() || proc2()
```

Figure 8.11: **An idiom that RETCON cannot repair: conflicts changing dataflow.** `proc1` and `proc2` conflict on `head`. Because `head` is used to index into memory in `proc1`, a constraint will be generated that it remain equal to its original value. If `proc2` commits first, the value of `head` will be changed when reacquired at the commit of `proc1`. In this case, RETCON would abort `proc1`.

```
                float x = .687;

proc1(){              proc2(){
  transaction{          transaction{
    ...                   ...
    r1 = sin(x);          r2 = arctan(x);
    x = r1;               x = r2;
    ...                   ...
  }                     }
}                     }

                P = proc1() || proc2()
```

Figure 8.12: **An idiom that RETCON cannot repair: conflicts on complex computation.** `proc1` and `proc2` conflict on `x`. The computation perform on `x` is outside the bounds of the symbolic tracking performed by RETCON, and hence it generates a constraint that `x` remain equal to its original value. The conflict will thus cause RETCON to abort.

Finally, Figure 8.12 on page 140 details a case where a conflict occurs on a variable that was used as input to a computation too complex for RETCON to track symbolically. Similar to above, this use generates a constraint that the variable remain equal to its original value, and hence RETCON will abort the transaction.

## 8.2 Operational Details

The above description does not explicitly discuss how RETCON would be implemented within the processor core. In a naive implementation, each register would be shadowed by a symbolic register. This symbolic register would have fields for the symbolic location of the register as well as for the expression denoting the computation that had been performed to arrive at the current value of the register. To minimize the amount of space required for the symbolic location, it could be represented as a pointer into the initial value buffer rather than as an explicit address. The representation of the computation performed would depend on the computation that RETCON tracked symbolically. As discussed below, RETCON currently tracks only addition and subtraction. In this case, the computation would be represented by a simple counter. It is likely that more optimized implementations could be devised; we leave such efforts as future work.

The above description also assumes word-granularity aligned memory operations and conditionals that operate directly on register values. However, RETCON must handle features from real-world architectures such as condition codes, sub-word memory operations and unaligned memory operations. To handle condition codes, each condition code register is extended with a symbolic constraint field. When an arithmetic operation with a symbolically-tracked input updates a condition code register, the symbolic constraint of the condition code is updated to reflect the constraint required for that condition code to retain the same value. The form of the constraint depends on the semantics of the condition code. For example, for the "equal-to-zero" condition code, the constraint operator is one of equality if the condition code is set to true and one of inequality if the condition code is set false. When a conditional operation is performed on a condition code being tracked symbolically, the condition code's constraint is added as a constraint on its root address.

To handle sub-word operations, RETCON adds a size field to symbolic values. If store-load communication becomes too complex (*e.g.*, an 8-byte load forwards from two 4-byte stores or a 4-byte store partially overwrites an 8-byte symbolic load), RETCON sets equality constraints on the relevant inputs. Similarly, RETCON treats unaligned memory operations as computation that cannot be tracked symbolically, adding equality constraints to the input word(s) of the operation.

The predictor that RETCON employs operates at a cache-block granularity (as this is the granularity at which blocks may be lost from the cache). It is possible (and indeed likely) that a single cache block could hold both peripheral data and other types of data. Once RETCON starts tracking a block, it symbolically tracks all accesses to the block, including generating constraints as necessary. The process of constraint generation and checking ensures correctness regardless of what data resides in a block being tracked by RETCON. For instance, it may occur that peripheral data shares a block with other data on which conflicts occur that RETCON *cannot* repair. RETCON will detect these conflicts via violated constraints and will abort the transaction.

Finally, the fact that RETCON allows transactions to execute past conflicts can result in cases where transactions raise spurious exceptions or enter infinite loops (Figure 8.13 on page 142). RETCON's approach to this so-called *inconsistent data problem* is similar to that taken by prior proposals [92]. When a transaction that has lost a block raises an exception, the hardware restarts the transaction with RETCON disabled. To prevent transactions from entering infinite loops due to inconsistent data, RETCON periodically reacquires stolen blocks and validates constraints.

```
                int x = 42;
                int y = 17;

proc1(){                        proc2(){
  transaction{                    transaction{
    if (x > 0)                      x = 0;
      r = x / y;                    y = 0;
  }                               }
}                               }

                P = proc1() || proc2()
```

Figure 8.13: **The inconsistent data problem.** `proc1` assumes that either both `x` and `y` are non-zero or both are zero. Because RETCON allows conflicts to occur without rollback, `proc1` could read the value of `x` from before `proc2` commits and the value of `y` from after `proc2` commits. This behavior would result in a spurious divide-by-zero exception.

## 8.3 RETCON Implementation Optimizations

The basic structures and operations described above admit several optimizations, which we describe below.

**Maintenance of initial value buffer entries at cache-block granularity.** To avoid reacquiring a stolen block each time a byte in the block is accessed for the first time, the initial value buffer maintains entries at cache-block granularity. A symbolic load starts symbolic tracking of the entire block. Constraints are maintained by a separate address-indexed and word-granularity buffer.

**Compressed representation of equality constraints.** RETCON represents equality constraints using per-byte "equality bits" added to entries in the initial value buffer. This optimization works synergistically with the previous one as it reduces pressure on the constraint buffer.

**Avoidance of upgrade misses during pre-commit.** As described thus far, during the commit process RETCON issues two misses for blocks that it has symbolically read and written (first to acquire the block via a read to check constraints followed by an upgrade miss when writing the block into the cache). To avoid this second miss, RETCON includes per-block written bits on initial value buffer entries. If the written bit is set, the block is acquired with write permission during the initial precommit phase.

```
                node* head = 0xffbb;

proc1(){                proc2(){
  transaction{            transaction{
    t = head->task;         ...
    do_work(task);          head = 0xcb0a;
  }                         ...
  ...                       ...
  ...                     }
}                       }

                P = proc1() || proc2()
```

Figure 8.14: **How RETCON can capture laziness.** `proc1` and `proc2` conflict on `head`. An eager conflict detection algorthim would detect this conflict and have to resolve it. In a lazy conflict detection algorithm, if `proc1` commits first, both transactions can commit. In this case RETCON would be able to commit both transactions as well, as the write to `head` by `proc2` would be kept in the symbolic store buffer until transaction commit. Note, however, that neither RETCON nor regular lazy conflict detection can avoid an abort if `proc2` commits first (Figure 8.11 on page 139).

**Efficient representation of symbolic computation.** Limiting the type of computation that can be symbolically tracked to be only additions and subtractions allows optimization of symbolic representations [85]. RETCON (1) tracks symbolic values succinctly as `(input_address, increment)` pairs, (2) collapses all arithmetic computation on symbolic values to cumulative increments, and (3) represents constraints by succinct intervals. RETCON precisely represents any number of constraints $(\leq, <, =, >, \geq)$ by the *most restrictive interval* bounding the symbolic value. Similarly, RETCON compactly represents any number of not-equal-to constraints—at the cost of some loss of precision—by tracking the interval in which the value cannot reside.

## 8.4 Other Benefits of RETCON

Although the purpose of RETCON is to enable recovery from remotely-changed inputs, its potential benefits extend further. First, RETCON's property of selectively delaying conflict resolution until transaction commit provides a form of lazy conflict detection; as discussed in Section 2.3, lazy conflict detection can enhance concurrency over eager conflict detection by allowing readers to

```
                struct mystruct{
                  int x;
                  int y;
                };
                struct mystruct s;

proc1(){                        proc2(){
  transaction{                    transaction{
    ...                             ...
    s.x = 5;                        s.y = 7;
    ...                             ...
  }                               }
}                               }

                P = proc1() || proc2()
```

Figure 8.15: **How RETCON can eliminate false sharing conflicts.** `proc1` and `proc2` access different fields of a shared struct `s`. These fields likely reside on the same cache block, causing a conflict in the baseline address-based HTM system (which detects conflicts at cache block granularity). In RETCON, constraints are generated at a word (or sub-word) granularity. Hence, if two transactions access separate words within a cache block, then each transaction's constraints are guaranteed to be satisfied (since the value that it accessed has not been remotely changed) and each transaction can commit.

commit before a conflicting writer. Figure 8.14 on page 143 illustrates how RETCON can capture the performance benefits of laziness by delaying writes until transaction commit.

Second, RETCON's value-based detection of conflicts at a sub-block granularity eliminates conflicts due to false sharing [52] and silent sharing [62] in a similar manner to prior proposals on value-based conflict detection [82, 109]. Figure 8.15 on page 144 provides an example of how RETCON can eliminate conflicts due to false sharing. We analyze the impact of these benefits in Section 9.3.

## 8.5 Related Work

Several proposals have focused on mitigating the performance limitations caused by conflicts in transactional memory and related contexts such as speculative locking [73, 88] and thread-level speculation [26, 40, 101, 105]. Bobba et al. [15] examine the performance pathologies present in conflict resolution schemes, including eager or lazy conflict detection [39, 77, 98], and recent work

proposes mixed eager/lazy strategies [98, 111, 112]. Value-based conflict detection has been used in the context of transactional memory for avoiding conflicts due to false sharing [109] as well as for compatibility with library code [82].

Researchers have proposed the use of speculative values to avoid conflicts due to true sharing. Dependence-aware transactional memory (DATM) [92] forwards speculatively written data between transactions. A globally enforced commit order guarantees atomicity and forward progress. DATM prevents aborts due to conflicts when transactions access shared data acyclically (such as incrementing a shared counter once), but aborts when there are repeated accesses to shared data between transactions (see Figure 8.1 on page 129). Other proposals have explored value prediction in the context of thread-level speculation and transactional memory [26, 83, 84, 105].

Proposals such as open nested transactions [16, 39, 78], abstract nested transactions [44], and transactional boosting [45] seek to increase concurrency by providing programming abstractions. RETCON differs from these proposals by trading off generality for programmer transparency.

RETCON is inspired by proposals for repair via selective instruction replay [21, 31, 48, 49, 96, 103]. The proposal most relevant in our context is ReSlice [96]. Within the context of thread-level speculation, ReSlice tracks the slice of instructions dependent on a load that is likely to result in a conflict, recording this slice in an instruction buffer. At commit, ReSlice sequentially re-executes the dependent slice of all loads whose input value has changed. As long as all branch outcomes are the same (*i.e.*, control flow is unchanged) and no memory dependences have changed, such replay can successfully repair speculation, allowing it to commit.

The tradeoffs in symbolic tracking versus instruction slicing are efficiency of representation vs. generality of recomputation. On the side of efficiency, a long transaction might, for example, perform many hashtable inserts or reference count updates; RetCon would perform a constant amount of recomputation vs. the linear amount of state tracked and computation reperformed by an instruction replay-based scheme. In contrast, an instruction replay-based scheme can support replay of more complex types of computation (*e.g.*, ReSlice allows memory addresses to change in re-execution as long as the new address does not change the dataflow of the slice).

Finally, other researchers have studied how to make Python interpreters more amenable to transactional memory via software restructuring. Tabba [108] performed an analysis of a transactionalized variant of the reference Python interpreter (the same interpreter used in this study). He describes a similar process of privatizing global variables to the one that we discussed in Chapter 3.

To handle the problem of reference counts, he instruments the `incref` and `decref` routines to avoid modifying the reference counts of objects that are known to never need deallocation (*e.g.*, the object representing `True`). This solution introduces performance overhead and does not easily generalize. Riley and Zilles [93] explore software restructuring and observe aborts due to false conflicts when studying the behavior of the `PyPy` python interpreter with transactional memory. One key difference in the tasks of parallelizing `PyPy` and parallelizing standard `python` is that `PyPy` uses a conservative garbage collector and thus does not raise the problem of conflicts on reference counts.

## 8.6 Summary

This chapter proposed RETCON, a mechanism for hardware transactional memory that allows transactions to lose data during execution and transparently repairs the effects of remote modifications at commit. RETCON uses symbolic tracking of the relationships between inputs and outputs to achieve repair without replay. Section 8.1 detailed the architecture and high-level operation of RETCON. We discussed operational details and optimizations in Section 8.2 and Section 8.3 respectively, followed by a discussion of other benefits of RETCON in Section 8.4 and related work in Section 8.5. In the next chapter we quantitatively evaluate the impact of RETCON on the workloads that we use.

# Chapter 9

# Experimental Evaluation of RETCON

This chapter experimentally evaluates the performance impact of RETCON on the workloads described in Chapter 3. Our most important objective is to determine whether RETCON can eliminate the performance impact of auxiliary data conflicts on these workloads. We also seek to answer the question of whether RETCON achieves performance benefits from its incorporation of laziness and value-based conflict detection. Finally, we seek to show that the amount of state that RETCON requires is small.

In the next section we describe the metholodogy that we use to evaluate RETCON. Section 9.2 evaluates the performance impact of RETCON on our workloads. Section 9.3 examines the question of how much of the performance impact of RETCON is due to the ability of RETCON to repair auxiliary data conflicts and how much is due to its incorporation of laziness and value-based conflict detection. Section 9.4 studies whether the inexact representation of constraints employed by RETCON results in performance degradation compared to a (less space-effective) exact representation. Section 9.5 studies the sensitivity of RETCON to parallelism of reacquires at transaction commit, and Section 9.6 studies the sensitivity of RETCON to structure size. Section 9.7 examines the sensitivity of RETCON to the configuration of the predictor that is used to determine whether to track blocks symbolically, and Section 9.8 discusses potential implications of RETCON on power. Finally, Section 9.9 summarizes the main results of the chapter and discusses remaining performance challenges.

| Parameter | Value |
|---|---|
| Processor | 32 in-order x86 cores, 1 IPC |
| L1 cache | 64 KB, 4-way set associative, 64B blocks, 1-cycle hit latency |
| L2 cache | Private, 1MB, 4-way SA, 64B blocks, 10-cycle hit latency |
| Memory | 100-cycle DRAM lookup latency |
| Permissions-only cache | 256B, 4-way set associative, read-only |
| Mechanism for overflows | ONETM-Concurrent |
| Coherence | Directory-based protocol, 20-cycle hop latency |
| RETCON structures | 8-entry initial value buffer, 8-entry constraint buffer |
| RETCON predictor | 4 sets, 4-way SA, 8-bit saturating counters, 1:100 up/down training ratio |

Table 9.1: **Simulated RETCON configuration.**

## 9.1 Methodology

We use the workloads and infrastructure described in Chapter 3. All of the configurations evaluated in this chapter employ the bounded HTM with cleaning-based version management, a 256-byte 4-way set-associative read-only permissions-only cache, and ONETM-Concurrent implementing strong atomicity to handle overflows. This includes a baseline configuration that employs purely address-based conflict detection (*i.e.*, the configuration evaluated in Section 7.5).

The version of RETCON that we evaluate employs the optimizations discussed in Section 8.3. Our default configuration of RETCON supports tracking at most eight cache blocks symbolically and maintaining constraints on up to eight word-granularity addresses. Vagaries of our simulator prevented us from easily bounding the size of the word-granularity symbolic store buffer; we analyze this number below. By default, we configure RETCON's commit-time repair process to reacquire all lost blocks in parallel and reperform stores serially after all blocks have been reacquired.

RETCON uses a predictor to determine which data blocks invoke value-based and symbolic tracking. The default configuration of this predictor is as follows. The predictor is a tagged table of 16 eight-bit saturating counters indexed by the data block address. It is 4-way set-associative (*i.e.*, the table is organized into 4 sets of 4 entries each). The predictor learns based on observed conflicts. To avoid elongating the amount of time that is spent in transactions that will eventually abort, a violated constraint causes the predictor to train down aggressively, requiring the observation of 100 conflicts on that block before attempting symbolic tracking on that block again.

We summarize the defaults of the default machine configuration used in this chapter in Table 9.1 on page 148.

## 9.2 Performance Impact of RETCON

Figure 9.1 on page 150 presents the impact of RETCON on workload scalability. In several cases, the ability of RETCON to repair conflicts changes the qualitative behavior of the workload. Most significantly, RETCON tranforms `python_opt` from a workload that has no scaling for the baseline configuration to one that has near-linear scaling (25x on 32 cores) by eliminating the performance impact of reference counter updates. Similarly, RETCON's ability to resolve conflicts on hashtable occupancy field updates without rollbacks changes the characteristics of `genome-sz`, `intruder_opt-sz`, and `vacation_opt-sz`. Whereas without RETCON the performance of these workloads is significantly worse than the corresponding variants with a fixed-size hashtable, the addition of RETCON makes them insensitive to whether the hashtable is fixed-size or resizable. As Figure 9.2 on page 150 illustrates, the cause of this performance increase is a reduction in time lost to conflicts.

## 9.3 What Contributes to RETCON Performance?

As discussed in Section 8.4, RETCON provides multiple benefits over a baseline HTM system employing eager conflict detection: in addition to admitting transaction commits wherein a value read has been changed remotely, RETCON can reduce conflicts versus the baseline system due to its selective use of laziness and value-based conflict detection. To provide insight into the sources of RETCON's performance gains, we evaluate two limited variants of RETCON in which values read are not allowed to change. In the first variant, which we refer to as *lazy*, values read are checked to have the same value at a block granularity at commit. This variant captures the performance benefits of lazy conflict detection. In the second variant, which we refer to as *vb*, values read are checked to have the same value at commit at a *byte* granularity. This variant captures the effects of eliminating false sharing conflicts as well as laziness. Neither variant, however, allows commits where a value read has been changed remotely.

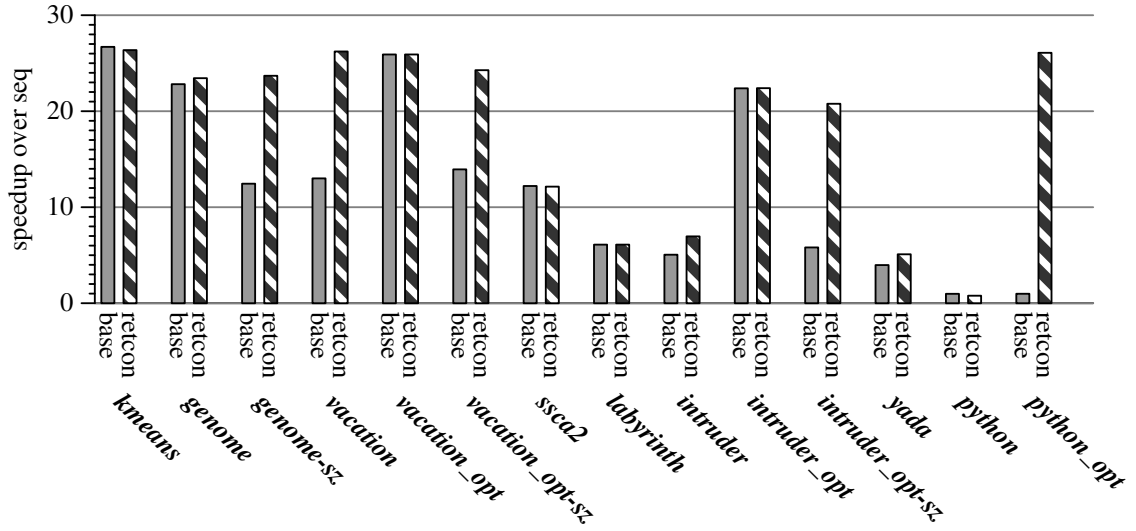


Figure 9.1: **Scalability of RETCON over sequential execution.** "base" represents the performance of ONETM-Concurrent (as evaluated in Section 7.5). "retcon" represents the performance that results when RETCON is added to this system.

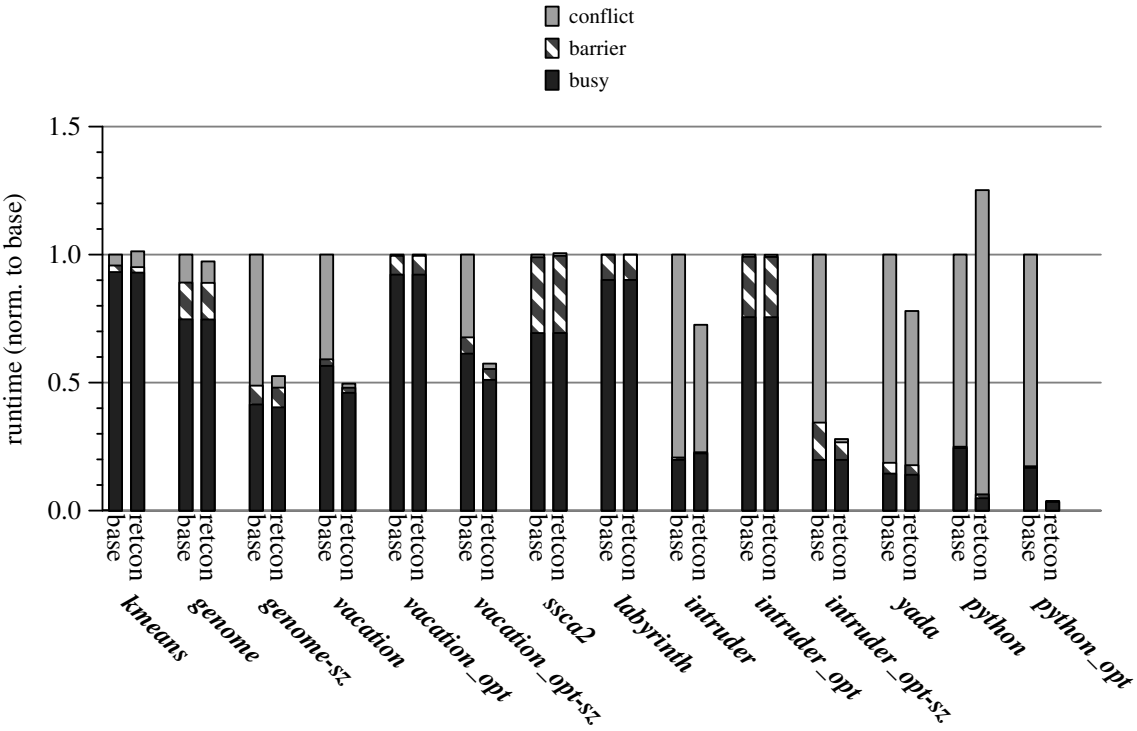


Figure 9.2: **Breakdown of RETCON execution time.**

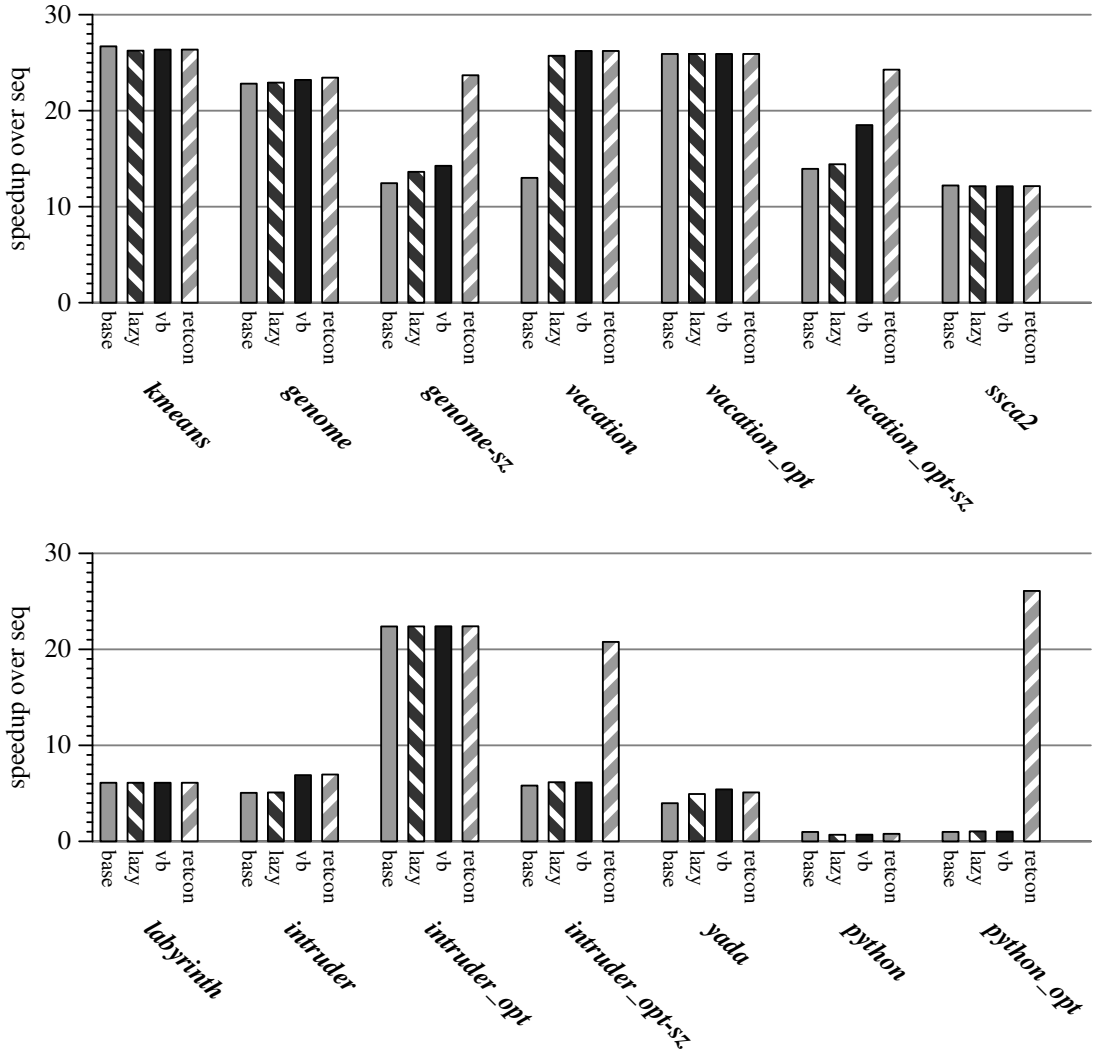


Figure 9.3: **Performance of variants of RETCON.** "lazy" captures the performance benefits of laziness. "vb" additionally captures the performance benefits of eliminating conflicts on false sharing.

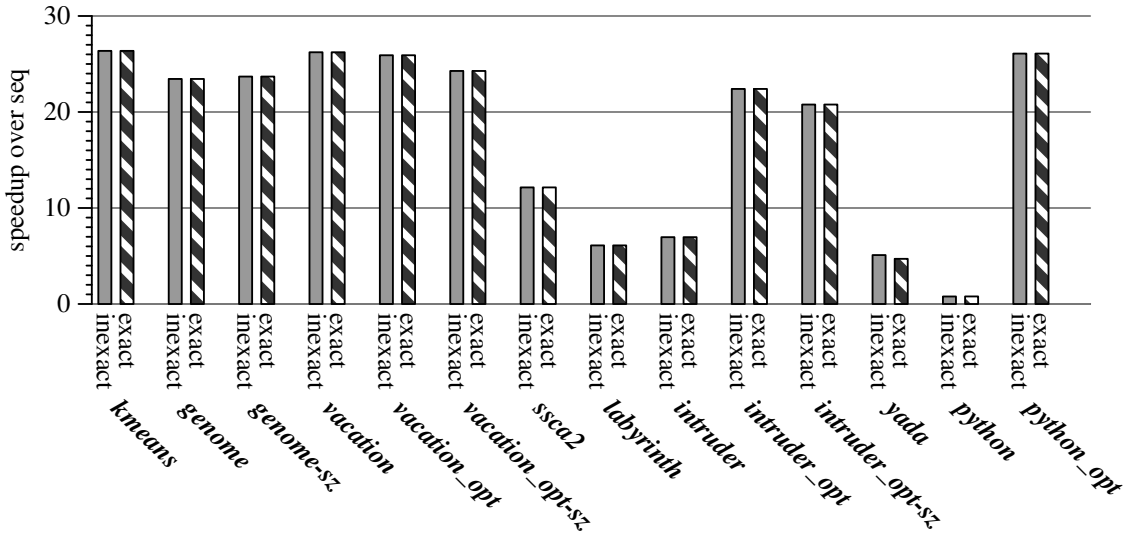


Figure 9.4: **Impact of inexact constraint representation on RETCON performance.**

Figure 9.3 on page 151 presents the scalability of these variants of RETCON. In most cases, the ability of RETCON to repair effects of remote modifications is needed to achieve the performance gains observed in the last section. One exception is `vacation`, in which the performance gains of RETCON are due to its incorporation of laziness.

## 9.4 Impact of Inexact Constraint Representation on RETCON

As described in Section 8.3, RETCON presents constraints via intervals: RETCON represents any number of constraints $(\leq, <, =, >>, \geq)$ by the *most restrictive interval* bounding the symbolic value, and it represents any number of not-equal-to constraints by tracking the interval in which the value cannot reside. The former representation is precise. The latter, however, is not. For example, RETCON represents the constraints that a given value must not be equal to 4 and also must not be equal to 8 by excluding the value from residing in the interval $[4, 8]$. If the value is (for example) 6 at the time of reacquire, RETCON will cause an abort that is in fact spurious. If such spurious aborts occur frequently, performance degradation could result.

In this section we evaluate the performance impact of this inexact representation of constraints. Figure 9.4 on page 152 presents the performance of the default configuration of RETCON as well

as a variant configuration that represents constraints exactly. This figure indicates that the inexact representation of constraints causes no performance degradation on the workloads that we use.

## 9.5 Sensitivity of RETCON to Parallelism of Commit-Time Reacquires

In its default configuration, RETCON reacquires lost blocks in parallel at transaction commit. Figure 9.5 on page 154 illustrates the performance impact of instead reacquiring blocks in serial at commit. As this figure shows, in most cases RETCON performance is not sensitive to this change. The only exception is `python_opt`, which loses significant performance if blocks are reacquired serially.

Figure 9.6 on page 154 gives insight into this result by showing the amount of time spent in transaction commit under the two variants. On almost all workloads, this time is nearly identical. For `python_opt`, however, reacquiring blocks serially causes a large increase in the time spent in transaction commit.

## 9.6 Sensitivity of RETCON to Structure Size

In this section, we examine the impact of varying RETCON structure sizes on performance. To provide context for this study, Table 9.2 on page 155 presents data for a limit study on the amount of state used by RETCON (for reference, we present the same data for the default RETCON configuration in Table 9.3 on page 156). In this limit study, RETCON structures were sized at 1024 entries. This limit study indicates that the number of blocks lost per transaction is generally quite small, on average below 4 on all workloads except `python` and `python_opt`. It does, however, reach a maximum greater than 8 on several workloads, indicating that there is a potential for performance increase by increasing structure sizes beyond those of the default RETCON configuration.

Inspired by these facts, we run two variant configurations of RETCON. In the first variant, the initial value buffer and constraint buffer can hold only 4 entries each. In the second variant, which serves as the above-mentioned limit study, the initial value buffer and constraint buffer can hold 1024 entries each. In this variant, we also increase the number of entries in the predictor to 1024 and the parallelism of commit-time reacquires to 32 to keep the system balanced.

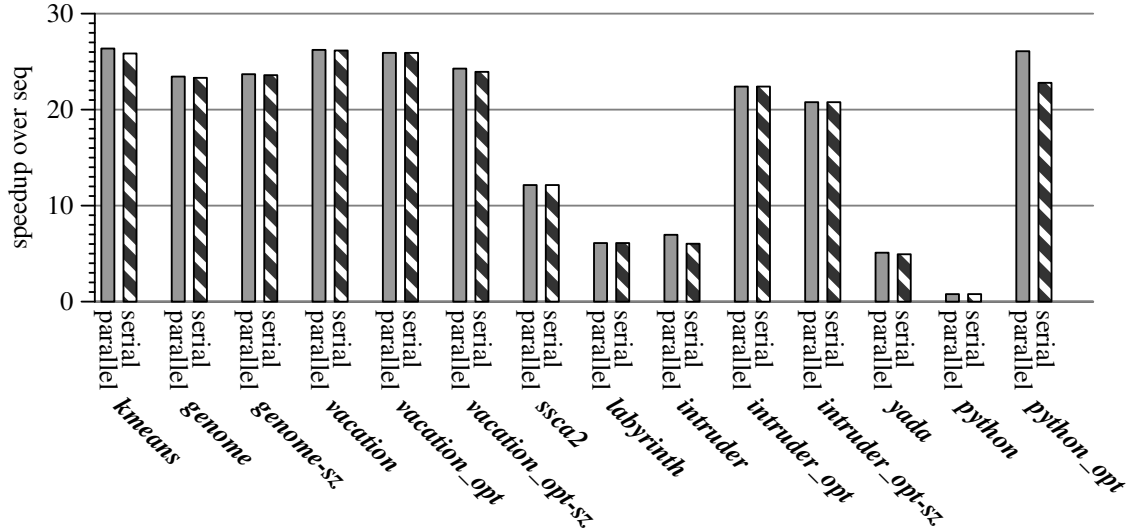


Figure 9.5: **Impact of serial reacquire at commit on RETCON performance.** "parallel" represents RETCON in its default configuration of reacquiring blocks in parallel at commit. "serial" represents a variant that reacquires blocks serially at commit.

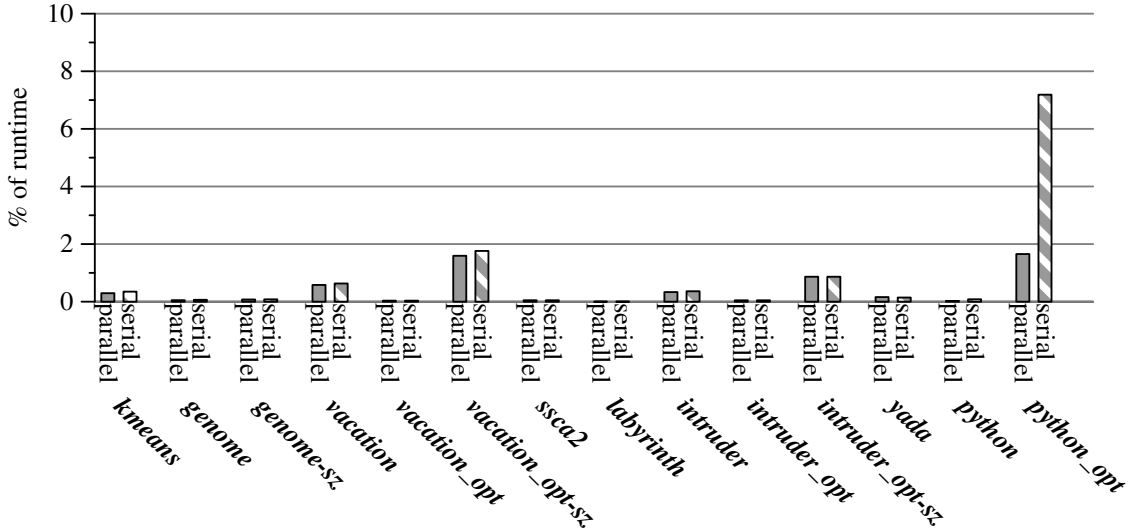


Figure 9.6: **Impact of serial reacquire on RETCON time in transaction commit.**

| Application | **blocks lost** | **blocks tracked** | **symbolic registers** | **private stores** | **constr. addrs.** | **slice size** |
|---|---|---|---|---|---|---|
| kmeans | 0.1 (2.0) | 0.6 (2.0) | 0.0 (0.0) | 1.0 (17.0) | 0.0 (0.0) | 0.0 (0.0) |
| genome | 0.0 (6.0) | 0.4 (34.0) | 0.0 (1.0) | 0.0 (14.0) | 0.4 (61.0) | 3.0 (3.0) |
| genome-sz | 0.0 (8.0) | 0.5 (34.0) | 0.1 (1.0) | 0.1 (14.0) | 0.4 (61.0) | 14.9 (120.0) |
| vacation | 0.7 (3.0) | 2.5 (7.0) | 0.0 (0.0) | 0.0 (4.0) | 3.0 (8.0) | 0.0 (0.0) |
| vacation_opt | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) |
| vacation_opt-sz | 0.6 (3.0) | 1.7 (3.0) | 0.9 (1.0) | 0.1 (5.0) | 0.1 (3.0) | 12.1 (30.0) |
| labyrinth | 0.0 (1.0) | 0.0 (1.0) | 0.0 (0.0) | 0.0 (1.0) | 0.0 (3.0) | 0.0 (0.0) |
| intruder | 0.4 (8.0) | 1.7 (17.0) | 0.0 (1.0) | 0.4 (17.0) | 1.3 (24.0) | 3.6 (5.0) |
| intruder_opt | 0.0 (1.0) | 0.1 (2.0) | 0.0 (1.0) | 0.0 (4.0) | 0.0 (4.0) | 0.0 (0.0) |
| intruder_opt-sz | 0.2 (1.0) | 0.4 (2.0) | 0.2 (2.0) | 0.4 (5.0) | 0.2 (4.0) | 7.5 (10.0) |
| yada | 0.4 (33.0) | 1.6 (37.0) | 0.2 (1.0) | 0.9 (71.0) | 1.5 (74.0) | 9.3 (42.0) |
| python | 10.9 (21.0) | 12.8 (22.0) | 0.0 (0.0) | 16.6 (37.0) | 29.5 (90.0) | 141.4 (230.0) |
| python_opt | 5.2 (9.0) | 5.2 (9.0) | 0.0 (0.0) | 6.0 (10.0) | 7.4 (13.0) | 185.3 (230.0) |

Table 9.2: **Limit study of RETCON structure utilization.** RETCON structures are sized to 1024 entries. The columns, in order, show the average and maximum (in parentheses) number of *(a)* 64B blocks stolen away during a transaction, *(b)* initial value buffer entries, *(c)* symbolic registers repaired at commit, *(d)* symbolic stores performed at commit, *(e)* symbolic constraints to be checked at commit, and *(f)* instructions in a dependent slice (*i.e.*, the number of instructions that would need to be buffered and replayed in an instruction replay-based scheme).

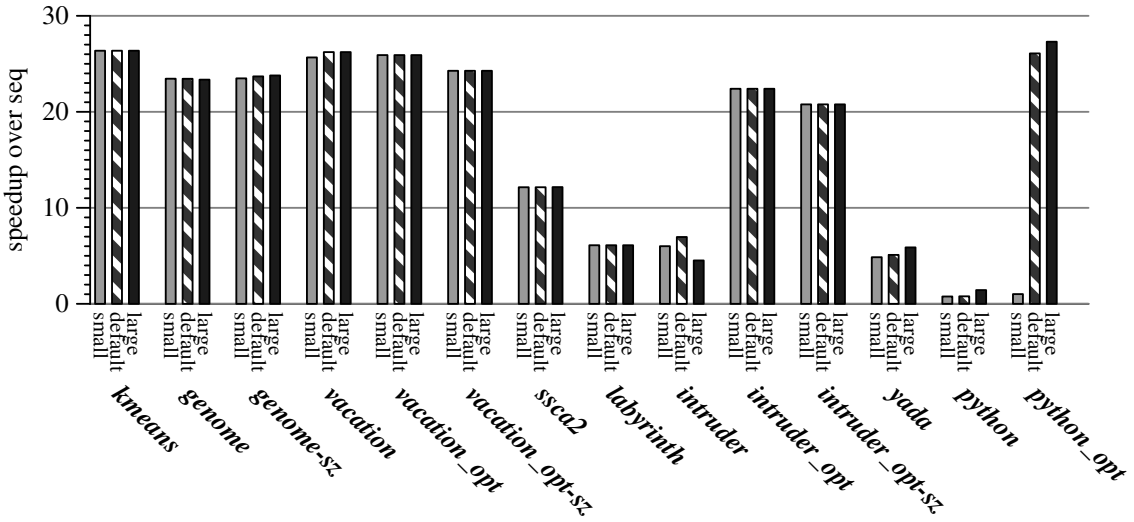


Figure 9.7: **Impact of RETCON structure sizes on performance.** "default" uses the default configuration of an 8-entry initial value buffer, an 8-entry constraint buffer, and the ability to make 8 requests in parallel at reacquire. "small" uses a 4-entry initial value buffer and constraint buffer and can make 4 requests in parallel at reacquire. "large" uses a 1024-entry initial value buffer and constraint buffer and can make 32 requests in parallel at reacquire.

| Application | **blocks lost** | **blocks tracked** | **symbolic registers** | **private stores** | **constr. addrs.** | **slice size** |
|---|---|---|---|---|---|---|
| kmeans | 0.1 (2.0) | 0.6 (2.0) | 0.0 (0.0) | 1.0 (17.0) | 0.0 (0.0) | 0.0 (0.0) |
| genome | 0.0 (7.0) | 0.2 (8.0) | 0.0 (1.0) | 0.0 (16.0) | 0.2 (8.0) | 3.0 (3.0) |
| genome-sz | 0.0 (7.0) | 0.3 (8.0) | 0.1 (1.0) | 0.1 (14.0) | 0.2 (8.0) | 16.1 (120.0) |
| vacation | 0.7 (3.0) | 2.5 (7.0) | 0.0 (0.0) | 0.0 (4.0) | 3.0 (8.0) | 0.0 (0.0) |
| vacation_opt | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) | 0.0 (0.0) |
| vacation_opt-sz | 0.6 (3.0) | 1.7 (3.0) | 0.9 (1.0) | 0.1 (5.0) | 0.1 (3.0) | 12.1 (30.0) |
| labyrinth | 0.0 (1.0) | 0.0 (1.0) | 0.0 (0.0) | 0.0 (1.0) | 0.0 (3.0) | 0.0 (0.0) |
| intruder | 0.4 (6.0) | 1.3 (8.0) | 0.0 (1.0) | 0.4 (14.0) | 0.9 (8.0) | 3.6 (5.0) |
| intruder_opt | 0.0 (1.0) | 0.1 (2.0) | 0.0 (1.0) | 0.0 (4.0) | 0.0 (4.0) | 0.0 (0.0) |
| intruder_opt-sz | 0.2 (1.0) | 0.4 (2.0) | 0.2 (2.0) | 0.4 (5.0) | 0.2 (4.0) | 7.5 (10.0) |
| yada | 0.3 (8.0) | 0.9 (8.0) | 0.1 (1.0) | 0.7 (22.0) | 0.7 (8.0) | 9.8 (42.0) |
| python | 6.0 (8.0) | 7.9 (8.0) | 0.0 (0.0) | 9.6 (14.0) | 8.0 (8.0) | 117.2 (230.0) |
| python_opt | 5.1 (8.0) | 5.2 (8.0) | 0.0 (0.0) | 6.0 (9.0) | 7.1 (8.0) | 192.6 (230.0) |

Table 9.3: **RETCON structure utilization.** The columns, in order, show the average and maximum (in parentheses) number of *(a)* 64B blocks stolen away during a transaction, *(b)* initial value buffer entries, *(c)* symbolic registers repaired at commit, *(d)* symbolic stores performed at commit, *(e)* symbolic constraints to be checked at commit, and *(f)* instructions in a dependent slice (*i.e.*, the number of instructions that would need to be buffered and replayed in an instruction replay-based scheme).

---

Figure 9.7 on page 155 shows the performance of these variant configurations. The smaller configuration has performance loss only on `python_opt`. On that workload, however, the performance loss is disastrous. The larger configuration results in small performance increases on several workloads (*e.g.*, `python_opt`).

## 9.7 Sensitivity of RETCON to Predictor Configuration

In this section, we examine the sensitivity of RETCON to the configuration of the predictor that is used to determine whether to track a block symbolically. We vary the predictor along three dimensions: predictor size, number of bits in the saturating counter, and the training ratio.

We first study the sensitivity of RETCON performance to predictor size. Figure 9.8 on page 157 presents the performance of RETCON variants with an 8-entry predictor, a 16-entry predictor (the default), a 32-entry predictor, and a 1024-entry predictor. All of these predictors are 4-way set-associative. The results indicate that the 16-entry predictor is able to capture the performance of the larger predictors, while the 8-entry predictor has only minor performance degradation (*e.g.*, on `genome`).

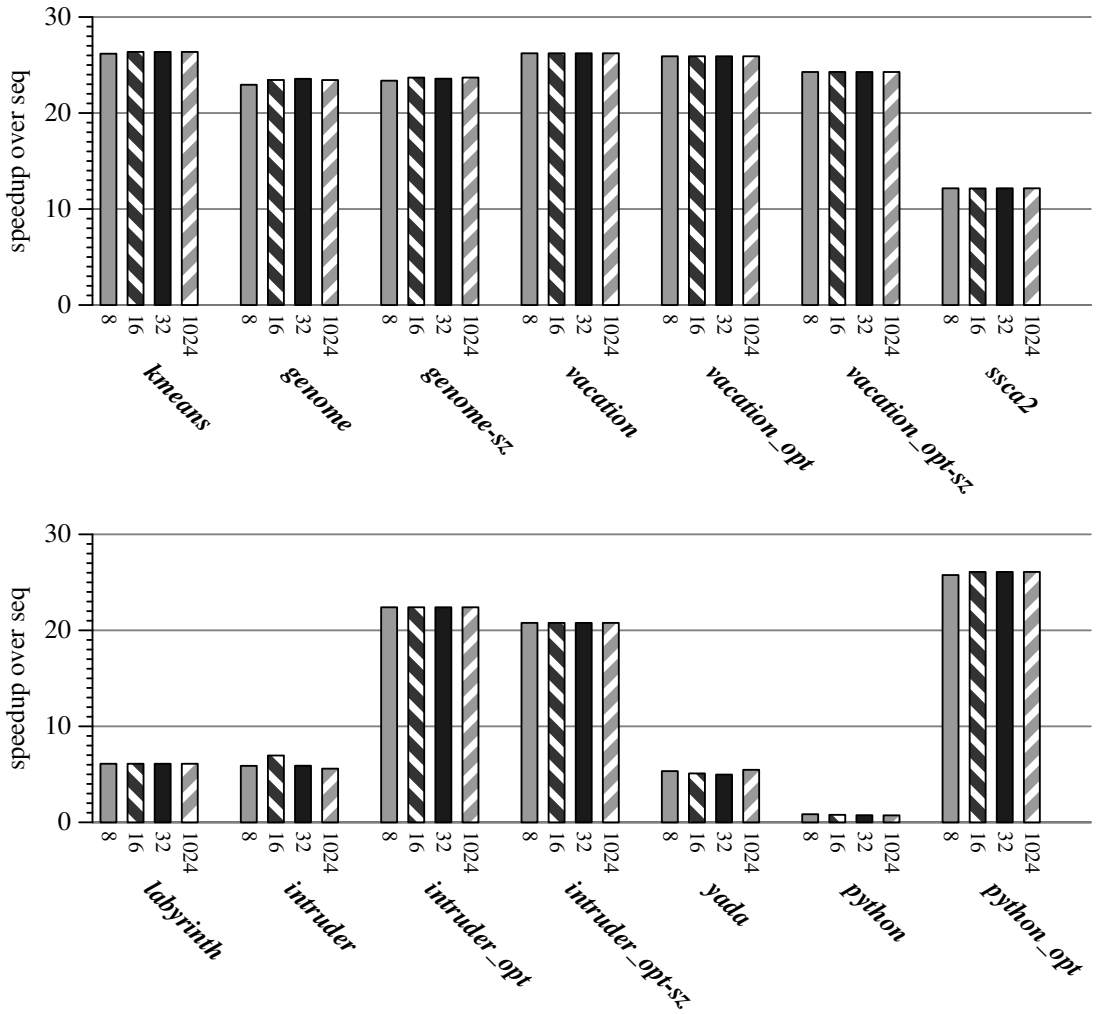


Figure 9.8: **Impact of varying size of RETCON predictor.** "8" represents an 8-entry predictor, "16" a 16-entry predictor (*i.e.*, the default configuration), "32" a 32-entry predictor, and "1024" a 1024-entry predictor.

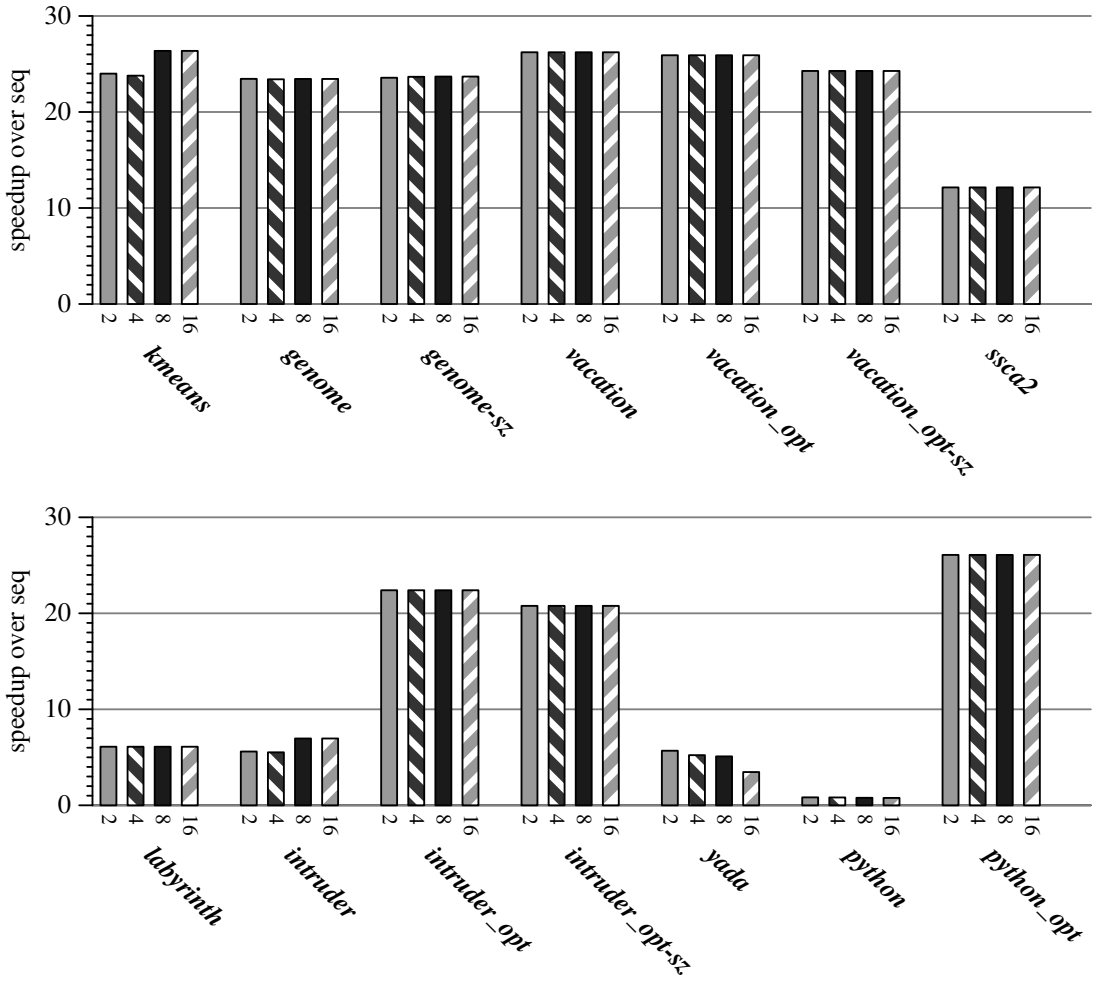


Figure 9.9: **Impact of varying the size of the saturating counters in the RETCON predictor.** "2" represents a 2-bit counter, "4" a 4-bit counter, "8" an 8-bit counter (*i.e.*, the default configuration), and "16" a 16-bit counter.

We next study the sensitivity of RETCON performance to number of bits in the saturating counters of the predictor entries. Figure 9.9 on page 158 presents the performance of RETCON variants with 2-bit saturating counters, 4-bit saturating counters, 8-bit saturating counters (the default), and 16-bit saturating counters. These results show that RETCON is generally insensitive to this parameter. However, on `kmeans`, the 2-bit and 4-bit counter variants suffer performance degradation relative to the default 8-bit variant.

Finally, we study the sensitivity of RETCON performance to the training ratio of the predictor. Figure 9.10 on page 159 presents the performance of RETCON variants with a 1:1 up/down training

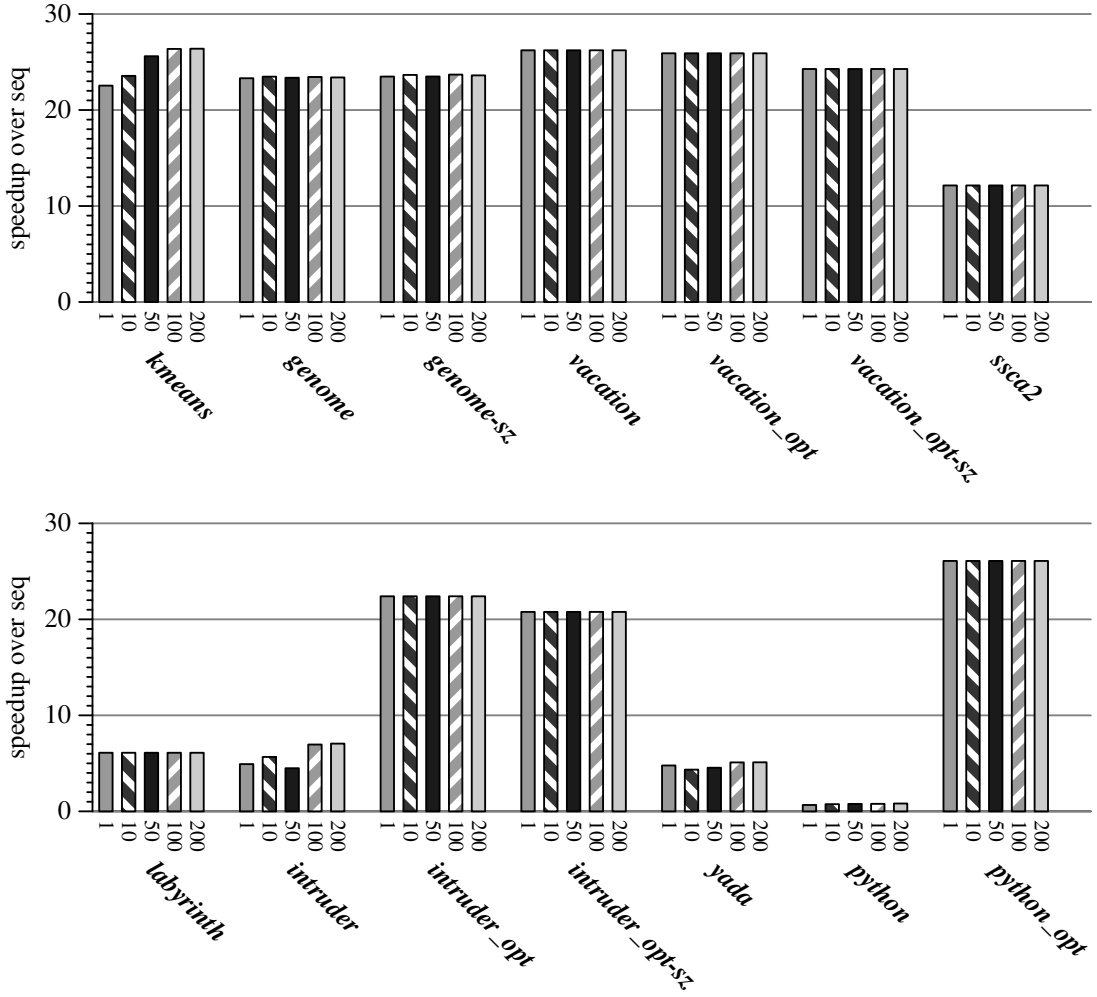


Figure 9.10: **Impact of varying training ratio of RETCON predictor.** "1" represents a 1:1 up/down training ratio, "10" a 1:10 ratio, "50" a 1:50 ratio, "100" a 1:100 ratio (*i.e.*, the default configuration), and "200" a 1:200 ratio.

ratio, a 1:10 ratio, a 1:50 ratio, a 1:100 ratio (the default), and a 1:200 ratio. Once again, RETCON is generally insensitive to this parameter. However, on `kmeans` the variants with 1:1 and 1:10 training ratios suffer performance degradation.

For completeness, we also studied the impact of varying the training ratio on RETCON configured with the other counter sizes from Figure 9.9 on page 158. Results (not shown) were qualitatively similar to those presented in Figure 9.10 on page 159. In particular, none of the variant training ratios enabled the 2-bit and 4-bit counter variants to match the performance of the default RETCON configuration on `kmeans`.

## 9.8 Discussion of the Power Implications of RETCON

The structures employed by RETCON have several potential implications on power consumption. The initial value buffer is written on the first load of a symbolically-tracked block within a transaction. It is subsequently read by local loads as detailed in Figure 8.3 on page 132. To limit the power required by these accesses, the initial value buffer could be placed behind the L1 cache (*i.e.*, it is accessed only when the block is not found in the L1 cache). The symbolic store buffer is written by symbolic stores and read by all local loads (Figure 8.3 on page 132), similar to the processor's regular store buffer. The symbolic store buffer could potentially be integrated with the regular store buffer. Finally, the constraint buffer is accessed on conditionals with symbolically-tracked values as inputs. These structures could be completely powered down when empty.

Figure 9.11 on page 161 presents the percentage of execution time that the various RETCON structures are active (*i.e.*, non-empty). For several workloads RETCON is active for a majority of execution time. Many (but not all) of these workloads are the ones on which RETCON increases performance significantly. On these workloads, the energy expended by RETCON is likely to be worth the performance gains, especially considering that these structures are organized as RAM-based structures (as opposed to CAM's) and are small.

## 9.9 Summary and Remaining Challenges

In this chapter we analyzed the performance impact of RETCON on the workloads that we use. We found that RETCON was able to eliminate the performance impact of auxiliary data conflicts on

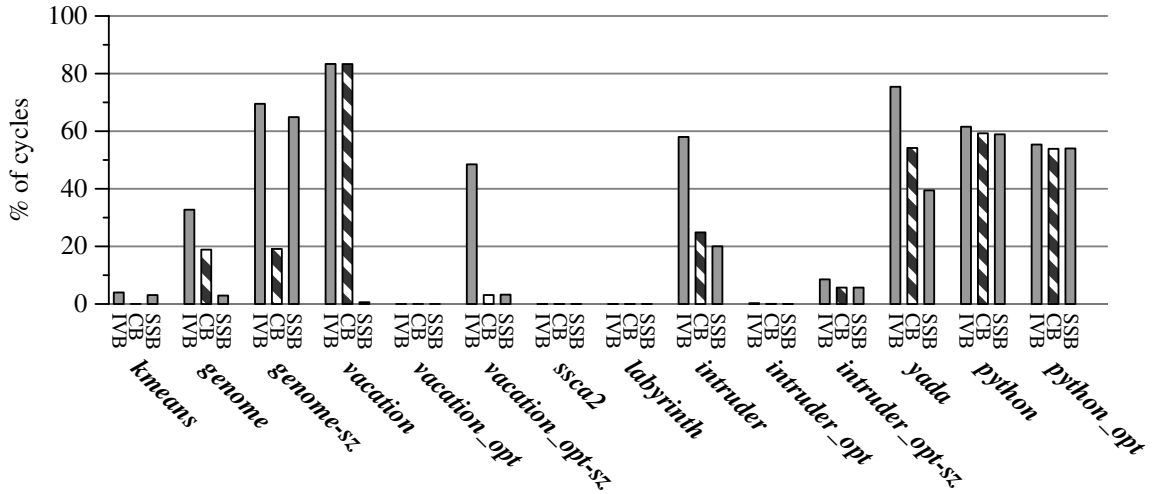


Figure 9.11: **Percentage of time that RETCON structures are non-empty.** "IVB" represents the initial value buffer, "CB" represents the constraint buffer, and "SSB" represents the symbolic store buffer.

these workloads, resulting in significant speedups on several workloads. While RETCON's incorporation of laziness and value-based conflict detection provided small performance benefits, RETCON's ability to repair the effects of remote modifications was most responsible for its performance gains.

We also analyzed several important parameters of RETCON. We found that the inexact representation of constraints that RETCON employs did not degrade performance from a less space-efficient exact representation. We also found that RETCON could be configured to reacquire blocks serially rather than in parallel at commit with little performance loss. We performed an analysis on the impact of RETCON structure size on performance, finding that the default configuration of an 8-entry initial value buffer and constraint buffer approached the performance of an essentially-unbounded size configuration, but that a 4-entry initial value buffer and constraint buffer was insufficient to repair the effects of conflicts in `python_opt`. Finally, we analyzed the sensitivity of RETCON to the configuration of the predictor used to determine whether to track blocks symbolically, finding that RETCON is generally insensitive to the exact configuration of this predictor.

As Figure 9.1 on page 150 and Figure 9.2 on page 150 show, there are three workloads with significant conflicts that RETCON does not greatly affect: the unmodified variants of `intruder`, `yada`, and `python`. The nominal reason that RETCON cannot help on these workloads is that

each of these workloads experiences conflicts due to multiple threads simultaneously enqueueing and dequeueing from shared lists. As the variables on which there is contention are used to index into memory (*e.g.*, the pointer to the list head), RETCON is not able to repair these conflicts.

However, these workloads also serve as examples that a repair-based approach is not always the right one to take. In each of these cases, the data elements being operated on are central to the dataflow of the entire transaction. Therefore, a repair that involves selecting a different list element at commit than one previously selected during execution would likely involve redoing most of the work of the transaction, resulting in little savings over a full abort. In such cases, an approach based on forwarding speculative values (*e.g.*, the pointer to the head of the queue in `intruder`) such as dependence-aware transactional memory (DATM) [92] may be more useful. We discuss the potential to integrate such a technique into RETCON as well as other avenues of future work in Section 10.2.

# Chapter 10

# Conclusions

In this chapter, we first summarize this dissertation in Section 10.1. We next outline directions for further research on our proposals in Section 10.2. Finally, I offer reflections on transactional memory in Section 10.3.

## 10.1 Dissertation Summary

In this dissertation, we have addressed two challenges to the applicability of hardware transactional memory as a general-purpose synchronization primitive: first, supporting transactions that are unbounded in space and time with high performance and low complexity, and second, maintaining robust performance in the presence of conflicts on auxiliary data. We outlined our baseline hardware support for bounded transactions. We described the usage of the permissions-only cache to extend the range of this bounded hardware transactional memory and proposed OneTM as a mechanism for handling the case of overflows of the bounded HTM. Via experimental evaluation, we showed that the combination of the permissions-only cache and OneTM can provide similar performance to that of an idealized hardware transactional memory that supports unbounded transactions with full concurrency and no overheads.

After removing overflows as a performance bottleneck, we found that data conflicts formed the primary remaining source of performance loss on our workloads. We proposed a novel form of conflict detection that tracks relationships between inputs and outputs symbolically during execution and uses these relationships to account for changed inputs at commit. We described the architecture

and operation of RETCON, our instantiation of this approach that is tailored to the needs of auxiliary data. Our experimental evaluation showed that RETCON mitigates the performance impact of a class of auxiliary data conflicts in our workloads.

## 10.2 Future Work

The primary avenues of future work that we see stemming from this dissertation focus on increasing the robustness of hardware transactional memory to conflicts. In this regard, we see two potential extensions of RETCON: first, extending RETCON to repair the effects of conflicts that change symbolically-tracked memory addresses, and second, integrating RETCON with techniques that communicate speculative data.

Repairing state after conflicts that change symbolically-tracked memory addresses is a complex problem. The primary difficulty is that changing memory addresses can potentially also change dataflow, *i.e.*, which loads forward from which stores. In effect, it could change the dependent slice of the conflicting address. While this complication can be handled (*e.g.*, [48, 103]), it is challenging.

One potentially fruitful avenue of tackling this problem in the context of RETCON is to seek to repair only conflicts that do not change the dependent slice. Although this would not capture all conflicts, it would still be sufficient to repair some of the conflicts in the workloads that we have studied (*e.g.*, conflicts on freelists in `python`). How much this restriction simplifies the problem is an open question.

We also note that some conflicts are simply not amenable to a repair-based approach. For example, consider transactions that dequeue data from a shared workqueue and then compute on this data (as `intruder` does). Repairing the effects of a conflict on the workqueue amounts to essentially redoing the entire computation of the transaction. Thus, even if we were able to extend RETCON to repair the effects of more general classes of conflicts, there would be conflicts for which this repair-based process does not increase performance.

Researchers have proposed an alternate approach that employs speculative values, for example by forwarding values from in-progress transanctions [92] or by predicting values [26, 83, 84, 105]. RETCON does not currently incorporate speculative value forwarding, but it is well-suited to do so. By generating constraints throughout execution and checking that the current architectural value

satisfies all generated constraints at commit, RETCON ensures correctness regardless of the source of the value used during execution (*e.g.*, the current architectural value, a forwarded value from an in-progress transaction, or a predicted value). This technique could help increase the robustness of RETCON to conflicts that are not suitable to a repair-based approach (*e.g.*, transactions dequeueing tasks from a shared workqueue and processing these tasks). We see unifying RETCON with such speculative value forwarding techniques as a promising area of future work.

Finally, we note that creating a concurrent version of the Python interpreter via transactional memory is not a solved problem. By increasing the robustness of HTM to auxiliary data conflicts, this dissertation took an initial step toward enabling such a concurrent interpreter. However, conflicts on reference counts are not the only ones plaguing the interpreter. As we outlined in Chapter 3, `python` also experiences conflicts on shared data structures such as freelists. More complex workloads are likely to exercise other conflicts in the interpreter as well. Just as our initial investigations on transactionalizing the Python interpreter led us to the problem of auxiliary data conflicts, we expect that further investigations will also inspire innovation in transactional memory design that is more broadly applicable — along the lines mentioned above or others.

## 10.3 Reflections on Transactional Memory

This section offers my reflections on transactional memory.[1] I have developed these reflections over the course of five years spent working on this dissertation. They have also been informed by my work on deep speculation in other contexts [12], experience gained from a fruitful summer spent at IBM Research, and insights gained from the work of others (an incomplete list includes [18, 19, 33, 36, 41, 50, 52, 72, 73, 79, 88, 89, 91, 92, 101, 105, 115]).

**Locks are unlikely to go away in the near future.** Expert programmers will always see a benefit to putting in more programming effort in return for finer-grained control over the robustness of the performance of synchronization. As a result, transactional memory must factor in interaction with locks as a first-class concern. While there have been promising efforts in this direction (*e.g.*, [114]), this problem is a challenging one with no single right solution.

[1] As these thoughts and opinions are my own and not necessarily shared by my co-authors, I use the singular pronoun throughout this section.

**Transactional memory is dead; long live speculative concurrency.** It is not clear whether transactional memory will become generally adopted as a programming interface. However, the idea of speculative parallel execution of critical sections with the semantics of isolation will continue to be relevant regardless of whether the interface presented to the programmer is transactional or is lock-based ([73, 88]).

**The primary challenge of speculative concurrency is supporting common programming paradigms with high performance.** There is a compelling argument for the implementation of synchronization using speculation. However, microarchitecture designers must make such speculation robust to common programming paradigms for it to be truly useful to programmers other than expert parallel programmers (to whom the task of conventional lock-based programming is probably a manageable one). This dissertation has aimed to take steps in that direction, but the challenge of course extends far beyond the problem of auxiliary data updates. One example is the paradigm of using lists to implement an unordered set interface (*e.g.*, freelists).

**Deep speculation will be a useful mechanism for multiprocessors regardless of the benefits of speculative concurrency.** Mechanisms of deep speculation such as those presented in Chapter 2 can reduce communication costs in multiprocessors via load latency tolerance [52] and store latency tolerance [12, 18, 115]. As such, the utility of these mechanisms is not entirely dependent on there being a perceived benefit for using them in the context of synchronization.

The last point mentioned above is particularly important, as it increases the likelihood that deep speculation mechanisms will appear in future multiprocessors. The appearance of these mechanisms will then provide an avenue for researchers to address the above-described challenge of further increasing their utility as a means of enabling high-performance synchronization. With respect to this challenge, I conclude this dissertation with a statement from Mark Moir that was originally made in the context of high-performance non-blocking software transactional memory but applies equally well here: "*Of course* it's hard! That's what they pay us for!"

# Bibliography


[1] The FeS2 simulator. URL `http://fes2.cs.uiuc.edu/acknowledgements.html`.

[2] The Unladen-Swallow benchmark suite. URL `http://code.google.com/p/unladen-swallow/wiki/Benchmarks`.

[3] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[4] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[5] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, Feb. 2005.

[6] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[7] L. Baugh and C. Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2008.

[8] A. Bensoussan, C. Clingen, and R. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, 1972.

[9] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[10] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE TCCA Computer Architecture Letters*, 5(2), Nov. 2006.

[11] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr. 2006.

[12] C. Blundell, M. M. K. Martin, and T. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[13] C. Blundell, A. Raghavan, and M. M. K. Martin. RetCon: Transactional Repair without Replay. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010.

[14] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[15] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[16] B. D. Carlstrom, A. MacDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[17] C. Cascaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why is it Only a Research Toy? *Communications of the ACM*, 51 (11), 2008.

[18] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[19] L. Ceze, J. M. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[20] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.

[21] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[22] Y. Chou, L. Spracklen, and S. G. Abraham. Store Memory-Level Parallelism Optimizations for Commercial Applications. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[23] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded Page-Based Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[24] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in Transactional Memory Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[25] M. Cintra, J. Martinez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Systems. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[26] M. Cintra and J. Torellas. Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In *Proceedings of the Eighth Symposium on High-Performance Computer Architecture*, Feb. 2002.

[27] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[28] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2004.

[29] D. E. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[30] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[31] R. Desikan, S. Sethumadhavan, D. Burger, and S. W. Keckler. Scalable Selective Reexecution for EDGE Architectures. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[32] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing Processor Performance Through Early Register Release. In *Proceedings of the International Conference on Computer Design*, Oct. 2004.

[33] M. Franklin and G. S. Sohi. The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[34] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. Efficient ECC-Based Directory Implementations for Scalable Multiprocessors. In *Proceedings of the 12th Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2000)*, Oct. 2000.

[35] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1991.

[36] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[37] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1st edition, 1993.

[38] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, Oct. 1998.

[39] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[40] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[41] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[42] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2003.

[43] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, June 2005.

[44] T. Harris and S. Stipic. Abstract Nested Transactions. In *Proceedings of the Second ACM SIGPLAN Workshop on Transactional Computing*, Aug. 2007.

[45] M. Herlihy and E. Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2008.

[46] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, July 2003.

[47] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[48] A. D. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating All-Level Cache Misses in In-Order Processors. In *Proceedings of the 14th Symposium on High-Performance Computer Architecture*, Feb. 2008.

[49] A. D. Hilton and A. Roth. Ginger: Control Independence Using Tag Rewriting. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[50] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum Benefit from a Minimal HTM. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[51] T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third Generation 64-Bit Performance. *IEEE Micro*, 19(3):73–85, May/June 1999.

[52] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence Decoupling: Making Use of Incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[53] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[54] T. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, 1986.

[55] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Mar. 2006.

[56] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[57] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2007.

[58] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[59] R. B. Lee. Precision Architecture. *IEEE Computer*, 22(1):78–91, Jan. 1989.

[60] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[61] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[62] K. M. Lepak and M. H. Lipasti. Temporally Silent Stores. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[63] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the Second ACM SIGPLAN Workshop on Transactional Computing*, Aug. 2007.

[64] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[65] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, May 1996.

[66] P. S. Magnusson *et al.* SimICS/sun4m: A Virtual Workstation. In *Proceedings of Usenix Annual Technical Conference*, June 1998.

[67] P. S. Magnusson *et al.* Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2): 50–58, Feb. 2002.

[68] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Sept. 2005.

[69] M. M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin, 2003.

[70] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[71] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, 2005.

[72] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.

[73] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[74] C. J. Mauer, M. D. Hill, and D. A. Wood. Full System Timing-First Simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, June 2002.

[75] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[76] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Sept. 2008.

[77] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[78] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[79] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[80] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2007.

[81] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, and M. Parkin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1995.

[82] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary Rewriting Approach to Software Transactional Memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[83] S. M. Pant and G. T. Byrd. Limited Early Value Communication to Improve Performance of Transactional Memory. In *Proceedings of the 23rd International Conference on Supercomputing*, June 2009.

[84] S. M. Pant and G. T. Byrd. A Study of Conflicting Data in TM Programs and Methods to Increase Concurrency Using Value Prediction. In *Proceedings of the Sixth ACM Conference on Computing Frontiers*, May 2009.

[85] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[86] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating System Transactions. In *Proceedings of the 22st ACM Symposium on Operating Systems Principles*, Oct. 2009.

[87] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, July 2001.

[88] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[89] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[90] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[91] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional Memory for an Operating System. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[92] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2008.

[93] N. Riley and C. Zilles. Hardware Transactional Memory Support for Lightweight Dynamic Language Evolution. In *Proceedings of the 21st SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2006.

[94] M. F. Ringenburg and D. Grossman. AtomCaml: First-Class Atomicity via Rollback. In *Proceedings of the 10th ACM Internation Conference on Functional Programming*, Sept. 2006.

[95] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006.

[96] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[97] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Aug. 1995.

[98] A. Shriraman and S. Dwarkadas. Refereeing Conflicts in Hardware Transactional Memory Systems. In *Proceedings of the 23rd International Conference on Supercomputing*, June 2009.

[99] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, June 2008.

[100] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[101] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[102] F. G. Soltis. *Inside the AS/400*. Duke Press, 2nd edition, 1997.

[103] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[104] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[105] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the Eighth Symposium on High-Performance Computer Architecture*, Feb. 2002.

[106] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[107] P. Sweazey and A. J. Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986.

[108] F. Tabba. Adding Concurrency in Python Using a Commercial Processors Hardware Transactional Memory Support. *Computer Architecture News*, 38(4), Sept. 2010.

[109] F. Tabba, A. W. Hay, and J. R. Goodman. Transactional Value Prediction. In *Proceedings of the Fourth ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[110] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[111] R. Titos, M. E. Acacio, and J. M. Garcia. Speculation-Based Conflict Resolution in Hardware Transactional Memory. In *Proceedings of the International Parallel and Distributed Processing Symposium Symposium*, May 2009.

[112] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-Lazy Hardware Transactional Memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2009.

[113] G. van Rossum. *The Python Language Reference: Release 2.6.4*. Python Software Foundation, 2009.

[114] H. Volos, N. Goyal, and M. M. Swift. Pathological Interaction of Locks with Transactional Memory. In *Proceedings of the Third ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[115] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[116] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[117] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.

[118] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*, Apr 2007.

[119] C. Zilles and R. Rajwar. Transactional Memory and the Birthday Paradox. In *Proceedings of the Nineteenth ACM Symposium on Parallel Algorithms and Architectures*, June 2007.