Compiler Assisted Speculation for Multithreaded Systems

by

Lingxiang Xiang

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Michael L. Scott

Department of Computer Science
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2015
To my family.
Biographical Sketch

Lingxiang Xiang was born in Linhai, Zhejiang, China. In Fall 2003, he started his undergraduate study at Zhejiang University, where he obtained a Bachelor of Science degree, and then a Master of Science degree, both in Computer Science, in 2007 and 2010 respectively. After a short period of work as an engineer at Baidu Inc., Beijing, he came to the University of Rochester to pursue a doctoral degree, under the supervision of Professor Michael L. Scott. His research focuses on high performance synchronization and compiler support. He worked as a summer intern for IBM Canada in 2012.

The following publications were a result of work conducted during his doctoral study:


Acknowledgments

It is a long journey from my first day in the URCS to the completion of this dissertation. I'm very lucky to be guided through the journey by my advisor, Professor Michael L. Scott. His wisdom, patience, and enthusiasm made my graduate study and research enjoyable; his encouragement and trust helped me grow academically and intelligently in the past five years. I am deeply grateful to him. And I believe that things I’ve learned from him will remain far beyond this journey.

I would like to extend my gratitude to Professor Chen Ding, who revealed the true beauty of modern compiler to me, making me confident enough to start the entire dissertation with the word “compiler”. He and my other two committee members, Professor Sandhya Dwarkadas and Doctor Peng Wu, have also provided valuable suggestions on my research and helped me set realistic goals during six-month reviews.

Friends at the URCS who have contributed their time to my research merit my special gratitude. Especially, Li Lu—for him often being the first reviewer of my new ideas; Luke Dalessandro—for his insightful comments on my early work.

Furthermore, I would like to thank Marcel Mitran and his colleagues at IBM for making three powerful IBM machines, BG/Q, z, and Power8, accessible to me. They have always been available and very responsive to my questions regarding machine accesses and uses. Without their help, many exciting results would not
be able to appear in this dissertation. I’m also thankful to people at IBM Centers for Advanced Studies, who generously awarded me two fellowships.

This dissertation is dedicated to my parents, and my wife Xiaodong. Without their endless love and support all these years, this dissertation would never exist.
Abstract

Multithreaded programming requires synchronization, to coordinate access to shared resources. As more and more processor cores become available on a single machine, synchronization tends to be the performance bottleneck of many multithreaded programs. This thesis focuses on two important synchronization scenarios: lock-based critical sections and transactional atomic blocks.

Since speculation is a well-known means of increasing parallelism among concurrent executions that are usually but not always independent, this thesis first explores the manual addition of speculation to lock-based critical sections (in concurrent data structures). With simple language extensions accompanied by compiler assistance, a technique which we refer to as CSpec, the programmer can exploit high-level program knowledge to move speculative work out of lock-based critical sections, thereby improving scalability while still maintaining correctness.

Speculation is also a major approach to improved parallelism in transactional memory systems. Frequent failures of speculative execution, however, may render the technique unprofitable. In recently emerged best-effort hardware transactional memory, speculation fails mainly due to two reasons: hardware overflow and data conflicts. In this thesis we develop a programming technique and compiler support, ParT, to reduce the duration and memory footprint of hardware transactions, leading to lower abort rates while preserving deadlock-free composability. To reduce the incidence of conflict, we propose an automatic, high-level mechanism,
Staggered Transactions, that uses advisory locks to serialize (just) the portions of the transactions in which conflicting accesses occur.

In all proposed techniques, compiler assistance is essential to maintaining ease of programming. We thus conclude that compiler assistance can significantly improve speculation in multithreaded systems.
Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors Michael L. Scott, Chen Ding, and Sandhya Dwarkadas, of the Computer Science Department of the University of Rochester and Doctor Peng Wu of Huawei US Research Lab. Work presented in Chapters 3–5 is a result of extended discussions with Michael L. Scott. All other work conducted for the dissertation was completed by the student independently.

The material presented in this dissertation was supported in part by the U.S. National Science Foundation grants CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649; by equipment support from Oracle and IBM; and by two IBM Canada CAS Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of above named organizations.
Table of Contents

Biographical Sketch iii

Acknowledgments v

Abstract vii

Contributors and Funding Sources ix

List of Tables xv

List of Figures xvi

1 Introduction 1

1.1 Motivation and Goal . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.1.1 Embracing Speculation in Concurrent Data Structure Design 3

1.1.2 Improving Speculation of Hardware Transactional Memory 4

1.2 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Background 8

2.1 Concurrent Data Structures . . . . . . . . . . . . . . . . . . . . . 8

2.1.1 Mutual Exclusion . . . . . . . . . . . . . . . . . . . . . . . . 9
2.1.2 Improving Locking Schemes ........................................ 9
2.1.3 Accelerating Critical Sections ..................................... 11
2.1.4 Speculation .......................................................... 12
2.1.5 Lock Elision .......................................................... 13
2.2 Transactional Memory .................................................... 14
  2.2.1 Software Transactional Memory .................................. 15
  2.2.2 Hardware Transactional Memory ................................. 17
  2.2.3 Hybrid Transactional Memory Systems ......................... 19
  2.2.4 Conflict Detection and Management ............................ 20
2.3 Interaction Between Compiler and Speculation .................... 22

3 Compiler Aided Manual Speculation .................................. 23
  3.1 Introduction .................................................................. 23
  3.2 Motivating Example: Equivalence Sets ............................ 25
    3.2.1 Implementation Possibilities .................................. 26
    3.2.2 A Manual Speculative Implementation ....................... 27
    3.2.3 Performance Results .............................................. 30
    3.2.4 Limitations of Manual Speculation ............................ 33
  3.3 Compiler-Aided Manual Speculation ............................... 34
    3.3.1 Interface ............................................................ 34
    3.3.2 Implementation .................................................... 38
    3.3.3 Limitations .......................................................... 41
  3.4 Principles of Coding with CSpec .................................... 41
    3.4.1 Where do we place consume_lock directives? .............. 41
    3.4.2 How do we validate? .............................................. 43
5 Staggered Transactions

5.1 Introduction 100
5.2 Overview 103
5.3 Compiler Support 107
  5.3.1 A Data Structure Approach 107
  5.3.2 Building Local Anchor Tables 109
  5.3.3 Building Unified Anchor Tables 109
  5.3.4 Instrumentation 110
  5.3.5 Example 111
5.4 Hardware Requirements 112
5.5 Runtime Support 114
  5.5.1 Instrumentation 115
  5.5.2 Locking Policy 116
5.6 Experimental Results 118
  5.6.1 Instrumentation Overhead and Accuracy 121
  5.6.2 Parallel Performance 123
  5.6.3 Reduced Aborts and Wasted Cycles 126
5.7 Performance Model on POWER 8 126
  5.7.1 Setup 128
  5.7.2 Results 130
5.8 Related Work .......................................................... 132
5.9 Summary .............................................................. 134

6 Conclusion and Future Directions ............................... 136

6.1 Contributions ....................................................... 136
6.2 Future Directions ................................................... 138

Bibliography ............................................................. 142
List of Tables

3.1 A summary of the application of CSpec .................................................. 58

4.1 Summary of benchmarks ................................................................. 87

4.2 Comparison of HTM programming techniques ................................. 98

5.1 HTM contention in representative benchmarks ............................... 101

5.2 Configuration of the HTM simulator .............................................. 119

5.3 Benchmark characteristics ............................................................. 120

5.4 Static and dynamic statistics of instrumentation .............................. 122
List of Figures

3.1 Per-set lock implementation of concurrent equivalence sets . . . . . . 26
3.2 Manual speculative implementation of concurrent equivalence sets 28
3.3 Throughput of concurrent equivalence sets on Niagara 2 . . . . . . 31
3.4 Throughput of concurrent equivalence sets on Intel Xeon E5649 . 32
3.5 Concurrent equivalence sets using CSpec . . . . . . . . . . . . . . 36
3.6 Automatically generated code for the \texttt{Sum} method . . . . . . 37
3.7 CSpec tagged \texttt{Remove} method of cuckoo hash table . . . . . . 47
3.8 Throughput of cuckoo hash table on Intel Xeon E5649 . . . . . . 49
3.9 Throughput of cuckoo hash table on Niagara 2 . . . . . . . . . . . 50
3.10 Hardware profiling for cuckoo hash on Intel Xeon . . . . . . . . . 51
3.11 Throughput of \texttt{Blink}-tree methods on Intel Xeon E5649 . . . . 53
3.12 Hardware profiling on Intel Xeon . . . . . . . . . . . . . . . . . . 55
3.13 Throughput of growing trees on Niagara 2 . . . . . . . . . . . . 55
3.14 Throughput on workloads with growing trees on Intel Xeon E5649 55
3.15 Throughput of linear bitmap allocator . . . . . . . . . . . . . . . 57
4.1 A big transaction for account maintenance . . . . . . . . . . . . . 61
4.2 A conceptual partitioning for the transaction of Figure 4.1 . . . . . 64
4.3 The need for rule 3c in partitioned transactions ............... 70
4.4 Pseudocode of auxiliary functions in ParT library ............. 73
4.5 Partitioned insert method and validator red-black tree ....... 74
4.6 Compiler-generated ParT code for the transaction of Figure 4.1 . 81
4.7 Automatic partitioning of a transaction with nested calls ..... 82
4.8 Microbenchmark performance on the zEC12 ..................... 89
4.9 Microbenchmark performance on Haswell ...................... 91
4.10 Macrobenchmark performance on the zEC12 .................... 93
4.11 Aborts per transaction for TM macrobenchmarks on the zEC12 . 94
4.12 Macrobenchmark performance on Haswell .................... 95
4.13 Aborts per transaction on Haswell ......................... 96

5.1 Execution of 3 transactions on different HTMs ................ 104
5.2 An overview of Staggered Transactions ...................... 105
5.3 An atomic block in genome .................................. 111
5.4 ABContext structure ......................................... 114
5.5 ALPoint instrumentation function ............................... 115
5.6 Pseudocode of a simple locking policy ......................... 117
5.7 Performance comparison with 16 threads ...................... 124
5.8 Reduced aborts and wasted cycles ............................. 127
5.9 Code skeleton of test transactions on Power 8 ............... 128
5.10 Scalability of HTM, Lazysub, and Staggered on Power 8 .... 130
5.11 Aborts per commit of HTM, Lazysub, and Staggered on Power 8 131
1 Introduction

The most fundamental change in the computer industry over the last decade has been the transition from single-core processors to multi-core processors, a change that occurred when the growth of single-core clock frequency hit the heat dissipation and power wall [87]. Unlike simple frequency scaling or traditional micro-architectural level innovations, such as instruction parallelism [122], the move to multi-core processors, from the perspective of software, is not completely transparent. While a multiprogramming system may gain an immediate overall performance boost with appropriate operating systems support [7] for multi-core processors, individual applications need to be parallelized, explicitly [16] or implicitly [24, 57, 98], in order to take advantage of the multi-core hardware.

Shared-memory multithreading is one of the most popular parallel programming models. Its popularity is a result of two factors. First, most existing multi-core hardware has physical shared memory; multithreading is a seamless fit for the underlying hardware structure. Compared with other models [90], multithreading interacts directly with the memory system, incurring no extra software overhead. Therefore, multithreading is often considered essential when raw execution performance is a priority. Second, at the programming level, shared-memory multithreading provides the programmer with an intuitive and natural way to use shared variables.
An ideal expectation for multithreaded software systems running on multi-core hardware (including programs and possibly the underlying run-time systems) is that the performance, measured either as speedup or throughput, scales up with the number of CPU cores used. The actual scalability of multithreaded systems, however, is often below the ideal number. According to Amdahl’s Law, the limiting factor of scalability is the serial portion in a multithreaded program. Inherently, the serial portion is a result of the ordering in the underlying algorithm—the longest chain of dependences in the algorithm’s data flow. In practice, however, we may be unable to achieve the speedup implied by the inherent dependence chain because of synchronization—the overhead of enforcing ordering where needed—and data movement costs (in terms of actual communication and cache misses on a shared-memory multicore) [38, 56].

From the perspective of programmers, synchronization can be explicitly written as lock-protected critical sections, or as code blocks with high-level atomicity semantics, such as synchronized methods in Java, and more general atomic blocks in other languages. On the other hand, synchronization can also implicitly occur inside pre-designed thread-safe library code, such as data structure methods, in which the actual synchronization is isolated from the client code.

### 1.1 Motivation and Goal

This research focuses on two common and representative scenarios where synchronization tends to greatly affect the performance of existing and future multithreaded programs: lock-based concurrent data structures and transactional memory (hardware transactional memory in particular). We believe that the improvement of synchronization—avoiding over-synchronization of shared data access—in the above scenarios would increase the scalability of multithreaded programs.
1.1.1 Embracing Speculation in Concurrent Data Structure Design

Traditional implementations of concurrent data structures use locks to ensure the atomicity of concurrent method invocations. However, locks are often overly pessimistic: they prevent threads from executing at the same time even if their operations don’t actually conflict. Finer grain locking can reduce unnecessary serialization at the expense of additional acquire and release operations. For data that are often read but not written, reader-writer locks allow non-mutating methods to run in parallel. Even so, the typical concurrent data structure embodies an explicit compromise between, on the one hand, the overhead of acquiring and releasing extra locks and, on the other hand, the loss of potential concurrency when logically nonconflicting method calls acquire the same lock.

In a different vein, nonblocking concurrent data structures [47, 81, 82] are inherently optimistic. Typically, their dynamic method invocations include an instruction that constitutes their linearization point [55]. Everything prior to the linearization point is (speculative) preparation, and can be repeated without compromising the correctness of other invocations. Everything subsequent to the linearization point is “cleanup,” and can typically be performed by any thread.

The usual motivation for nonblocking data structures is to avoid performance anomalies when a lock-holding thread is preempted or stalled. For certain important (typically simple) data structures, average-case performance may also improve: in the absence of conflicts (and consequent misspeculation), the reduction in serial work may outweigh the increase in (non-serial) preparation and cleanup work. Unfortunately, for more complex data structures the tradeoff tends to go the other way and, in any event, the creation of efficient nonblocking algorithms is notoriously difficult.

But what about speculation? Automatic speculation is already used in many
Transactional memory systems to increase parallelism. With hardware support, TM may provide performance as good as or better than that of the best-tuned fine-grain locking [50, 94]. For application code, written by non-experts, even software TM (STM) may outperform coarse-grain locking. For concurrent data structures in libraries, however, STM seems unlikely ever to be fast enough to supplant lock-based code written by experts.

However, this does not mean speculation is an inappropriate method for concurrent data structure design. We argue that the problem of automatic speculation stems from its complete ignorance of data-structure-specific knowledge. With this knowledge incorporated, the overhead of speculation can be reduced. But how can we gain and utilize the data-structure-specific knowledge? An ideal solution would be an automatic inference mechanism that is able to obtain all required knowledge from the source code. But given that this knowledge is very high level and the design of high performance concurrent data structures requires the expert programmer’s intelligence, we do not see any hope in being able to do it automatically. Instead, a practical solution is to let the programmer, with minimal effort, explicitly express high-level speculation knowledge in source code, while leaving the more tedious jobs to the compiler.

1.1.2 Improving Speculation of Hardware Transactional Memory

Transactional Memory combines the convenience of atomic blocks as a programming idiom with the potential performance gains of speculation as an implementation. Twenty years after the initial proposal [50], hardware transactional memory (HTM) is becoming commonplace: early efforts by Azul [23] and Sun [34] have been joined by Intel [131] and three separate projects at IBM [17, 58, 123]. With the deployment of HTMs, we anticipate a “virtuous cycle” in which more and
more multithreaded programs employ TM, and the performance of the hardware becomes increasingly important.

Preliminary experience with HTM suggests that the raw performance, scalability, and energy consumption of transactional programs—especially those with medium- or large-scale transactions—are all limited by the hardware transaction abort rate [131]. Aborts happen for two main reasons: hardware overflow and conflicts with other transactions.

Overflow aborts are due to the limited hardware resources for transactional execution. Most existing HTMs monitor the read and write set of hardware transactions in L1/L2 cache or in a special buffer, in order to detect possible read-write conflicts. Once the hardware fails to track some cache line or buffer entry belonging to an ongoing transaction, the transaction has to be aborted.

While hardware overflow is an implementation issue, which we expect will be less serious in future HTMs with more or even effectively unbounded tracking capacity, conflict aborts are far more fundamental – they stem from actual data contention in transactional workloads. As speculative execution in HTMs is fully optimistic, a high contention rate means repeated aborts and retries. As a consequence, speculation may lead to worse performance than traditional pessimistic locking. Under this circumstance, reverting to pessimistic code (e.g., lock-based backup) to serialize conflicting transactions, of course, is a working solution, but doing so also destroys potential parallelism for the non-conflicting parts of these transactions.

Instead of designing more sophisticated hardware mechanisms to deal with the above problems, this research focuses on software solutions that are compatible with existing HTMs or require only minimal hardware modifications. In particular, to reduce overflow aborts, we propose a programming technique that partitions a large transaction into small hardware transactions that are more likely to fit in current HTM hardware, and maintains high-level (semantic) atomicity
for the overall operation. Meanwhile, at the programming level, the compiler support retains two compelling features of transactional memory—composibility and ease of programming. To reduce conflicts, we design an automatic, high-level mechanism that serializes just the portions of the transactions in which conflicting accesses occur. The compiler statically learns high-level program knowledge and passes it to the runtime. With that knowledge, the run-time system learns the abort pattern of workloads for precise serialization of conflicting transactions at execution time.

Given the above motivation and goal, my thesis statement is: Compiler assistance can significantly improve speculation and consequently performance in multithreaded systems.

1.2 Organization

The rest of the dissertation is organized as follows:

Chapter 2 briefly introduces the background knowledge of concurrent data structures and transactional memory, and provides a summary of previous work on the interaction between compiler and speculation techniques.

Chapter 3 describes a semi-automatic speculation technique (CSpec) to shrink lock-based critical sections of concurrent data structures. It begins with a design pattern for manual integration of speculation into high performance concurrent data structures, then shows how compiler assistance with simple language extensions can simplify the use of the design pattern in various data structures. This chapter tests CSpec’s performance on UltraSPARC and x86 platforms.

Chapter 4 presents a programming technique and compiler support that run large atomic blocks effectively on current best-effort HTMs by partitioning common operations into read-mostly and write-mostly operations, which then execute
separately. It also presents experimental results on IBM z-Series and Intel Haswell machines.

Chapter 5 presents “staggered transactions”, an automatic, high-level mechanism that reduces conflict aborts in HTMs. This chapter demonstrates the feasibility of this mechanism with fully developed compiler and runtime support, running on simulated hardware. A simple performance model of staggered transactions is also studied on real POWER 8 machine.

Chapter 6 concludes this dissertation by presenting a summary of our contributions, followed by a discussion of possible future directions for research in this area.
2 Background

As introduced in Chapter 1, the focus of this thesis is two representative scenarios of synchronization: concurrent data structures, where synchronization is beneath the methods interface, and transactional memory, where synchronization is a high-level programming idiom with flexible implementation choices. Fundamental knowledge and more comprehensive coverage of both topics can be found in Herlihy and Shavit’s book [51] and Scott’s book [106]. This chapter presents background material and recent literature, with an emphasis on how speculation techniques have been applied to the two topics. Each subsequent chapter may have extra dedicated section to introduce and compare directly related work.

2.1 Concurrent Data Structures

Concurrent data structures keep the same or a similar set of methods as in their sequential version, but these methods can be called simultaneously by multiple threads. Concurrent data structures play a crucial role in many multithreaded programs. For example, a key component of memcached, a popular in-memory key-value storage system, is a concurrent hashtable [119]. Due to their importance, numerous mechanisms have been proposed to increase the parallelism of concurrent data structures, without altering their basic properties.
2.1.1 Mutual Exclusion

All threads in a multithreaded program share their address space. In most multithreaded programming languages, one memory address can be accessed from different threads as long as these accesses are properly synchronized. Mutual exclusion is an essential approach to synchronization. It allows only one thread to access the shared data at a time. The period of mutually exclusive execution is called a critical section.

Mutually exclusive execution of critical sections is often too pessimistic. It’s conservatively assumed at coding time that relevant critical sections must be fully serialized in order to avoid potentially inconsistent views of memory due to conflicting accesses (i.e., read-write and write-write conflict). However, the occurrence of conflicting accesses actually depends on workloads. It’s possible at run time that critical sections contain no conflicting access at all. Another common problem with critical sections is over-serialization— they are larger than they need to be, and only a small portion of execution time is actually spent in protecting shared accesses. In both situations, more scalability can be gained by using optimistic synchronization techniques instead.

2.1.2 Improving Locking Schemes

Locking is a typical realization of mutual exclusion, thus a straightforward method to implement a concurrent data structure is using one or more locks to make each of its concurrent operations mutually exclusive. Coarse-grained locking, in which one or a few locks protect the whole data structure, is easy to implement, but suffers from poor scalability. By contrast, fine-grained locking boosts scalability by dividing one or few monolithic critical sections into smaller ones. However, fine-grained locking is prone to priority inversion and locking bugs (e.g., atomicity violations and deadlock), and is thus considered difficult to use in practice [41, 72].
One issue with respect to mutually exclusive locking is that simultaneous readers are forbidden in one critical section, although they do not cause conflicting access. Some read-biased lock variants and optimizations aim to address this issue. Scalable reader-writer locks [73, 79] are early example solutions. Sequence locks [63] is a writer-biased lock in which a sequence number is incremented by a writer on each acquire and release. So an odd value indicates the lock is held by a writer. Readers can process without acquiring the lock, but they must check the value of the sequence number at appropriate points to make sure the sequence number has never been modified by a writer.

More recently, Vallejo et al. [121] proposed Lock Control Unit (LCU), an architectural support for fair reader-writer locking. Each core is argumented with a Lock Control Unit to store distributed lock queues, which are managed by one Lock Reservation Table per-memory controller. To accelerate read-lock acquisition and reader detection in single-chip multicore machine, Dice and Shavit [32] proposed the TLRW lock. The entire lock occupies one cache line, where every byte of the lock indicates the existence of a reader. Unlike in a simple read-writer lock, TLRW avoids expensive CAS operation on the same location for readers. However, when used on multiple-chip machines, TLRW lock incurs significant inter-chip communication cost.

RCU (read-copy-update) [77] is a technique often used in operating system kernels to enable wait-free readers. Despite zero overhead in readers, a writer in RCU needs to create a new copy of its own working data, and uses atomic primitives such as CAS to update its own copy to the data structure. Thus, the application of RCU is limited to simple data structures [119] and read-dominated workloads.

In data structures such as queues, the concurrent methods (critical sections) protected by locks are simple and brief. The cost of lock acquisition and release, especially for highly contended workloads, may even outweigh the concurrent op-
erations themselves. Flat Combining [39, 48], a new synchronization paradigm, reduces the number of locking operations by combining the execution of multiple concurrent methods together. In the paradigm, if a thread fails to acquire the lock, instead of competing for the lock again and again, it publishes its request in a shared list and waits for the current owner of the lock—combiner—to accomplish the requested operation for it. Meanwhile, the combiner, after acquiring the lock, services all pending requests for other waiting threads. Therefore, multiple short requests can be served at the cost of single lock acquisition and release, and since the data are likely accessed by the same core (the combiner’s), Flat Combining benefits a lot from cache locality.

2.1.3 Accelerating Critical Sections

Both hardware and software solutions have been proposed to directly accelerate the execution of critical sections. For example, the Accelerated Critical Sections (ACS) [115] approach places the execution of selected critical sections in one or more large and powerful cores in an asymmetric chip multiprocessor. When a small core encounters a critical section, it signals the large core to take over the critical tasks, and waits for the tasks to be done by the big cores. Since the large cores provide higher IPC and better cache locality, the execution time spent on the critical path can be shortened. On the same architecture, the same group proposes a cooperative software-hardware method [60] to dynamically identify and schedule multiple critical sections of a multithreaded program.

Remote Core Locking (RCL), proposed by Lozi et al. [71], is a software analogue of ACS. It requires source code modification so that the actual execution of critical sections can be explicitly transferred to a remote core. Compared with ACS, RCL doesn’t need extra architectural support. However, when running on existing symmetric processors, the major benefit is due to the better locality of critical sections on the remote core. RCL also shares the high-level spirit with
Flat Combining, but unlike Flat Combining, RCL uses dedicated cores to execute critical sections.

Trancoso and Torrellas [118] use a compiler to improve cache locality of critical sections by inserting prefetch instructions for shared data that might be accessed in critical sections. In the same vein, the speculative phase of our proposed CSpec in Chapter 3 accesses shared data and thus warms up data cache before entering a critical section, so cache misses in the critical path can be reduced.

### 2.1.4 Speculation

Speculation, as an enhancement to mutual exclusion, is an optimistic technique that overlaps the execution of multiple critical sections so the latency of the critical path can be shortened if data conflicts are rare. Similar to the concept of speculation in database systems [62], speculative execution in multithreaded systems processes shared data without holding real locks. Since other threads are allowed to modify the shared data, validation is a necessary step at appropriate points to guarantee the correctness of speculation. Once the validation procedure discovers a conflict (shared data are modified by other threads), one of the conflicting speculative executions must abort and roll back for a retry. If no conflict happens, the speculation succeeds and all pending modifications are submitted.

The usefulness of speculation heavily depends on the data contention. With low data contention, aborts due to conflicts do not have a major impact on overall performance. However, when contention goes up, repeated aborts and retries will hurt performance instead. Therefore, there exists a tradeoff between speculation and conventional pessimistic techniques.

Regarding the usage of speculation, lock-free data structures use nonblocking primitives [42] (e.g., CAS and LL/SC) to get rid of locks. In the absence of conflicts (and consequent misspeculation), the reduction in serial work may outweigh the
increase in (non-serial) preparation and cleanup work, and nonblocking algorithms may improve average-case performance and reliability (deadlock/livelock immune) for certain important (typically simple) data structures, such as linked lists [47, 81], FIFO queues [82, 84, 85], hash tables [81], scalable stacks [48], and so on.

A general-purpose speculation mechanism for concurrent data structures has not been well explored in the literature. Work by Bronson et al. [14] demonstrates that hand-written, data-structure-specific speculation can provide a significant performance advantage over traditional pessimistic alternatives. Specifically, the authors describe a relaxed-balance speculative AVL tree that outperforms the Java ConcurrentSkipListMap. Their code employs a variety of highly clever optimizations. While fast, it is very complex, and provides little guidance for the construction of other hand-written speculative data structures.

2.1.5 Lock Elision

Speculative Lock Elision (SLE) [95] is an architectural technique to dynamically remove unnecessary serialization due to exclusive locks. It is based on the observation that simultaneous critical sections protected by the same lock do not always conflict with each other. This allows SLE to speculatively run a critical section without actually holding the lock. The modifications are buffered in a local store buffer until entire critical section successfully completes. If a data conflict is detected, all buffered results are discarded and the hardware rolls back to acquire the real lock. A similar technique, Hardware Lock Elision, has appeared in Intel’s Haswell processor [131].

Lock elision can also be implemented in software. SOLERO (Software Optimistic Lock Elision for Read-Only critical section) [88] is a lock implementation in which writes to lock variables can be elided for read-only critical sections in Java. A SOLERO lock works in a way very similar to a timestamp in many STM
algorithms. A writing critical section modifies the lock’s value, while a read-only critical section does not. Instead, a read-only critical section keeps a snapshot value for the lock at the beginning, and double-checks it at the end. Exceptions due to inconsistent data during speculative execution are handled by JVM. Similarly, for read-dominated workloads, Transactional Mutex Locks (TML) by Dalessandro et al. [26] delay the actual lock acquisition to the first possible write. A sequence lock contained in TML ensures data consistency of the read-only part.

2.2 Transactional Memory

Because of the difficulties in applying traditional synchronization techniques (e.g., fine-grained locking) in multithreaded programming, Transactional Memory [50] was proposed as a synchronization mechanism that places the emphasis on ease of programming. It follows the concept of transactions in database systems: memory operations in a transaction are either all-done or none-done. There is no half-done state. The programming interface in most TM systems is quite simple: the programmer just puts anything that needs synchronization into a single-entry, single-exit atomic code region, which is guaranteed to be atomically executed. The atomicity in TM is a high-level semantic property. The actual behavior of an atomic region depends on the underlying implementation.

Considered as a compelling alternative to traditional synchronization techniques, transactional memory has drawn a lot of attention from both academia and industry. At the programming language level, for example, a technical specification for transactional memory in C++ has recently been released [116]. It may appear in a future C++ standard as a language feature.
2.2.1 Software Transactional Memory

Software Transactional Memory (STM) supports TM semantics via pure software, commonly with a combination of library and compiler support. Although a few STM implementations [3] are nonblocking, most use locks. Almost all STMs are optimistic, though they often include a pessimistic (global-lock) fallback. Optimistic STM systems [28, 33, 110] exploit speculative execution of transactions to improve concurrency, so some key concepts of speculation such as validation, aborts and rollback, also apply to STMs. Data conflicts in STMs are detected through well-designed validation mechanisms (e.g., ownership-based [110] or value-based [28]) at different granularities. To support rollback after abort, STMs treat write operations specially. Some systems buffer writes until the transaction commits, while others make in-place writes but keep the original value in an undo log. Consequently, all shared reads and writes in STM should be properly instrumented either manually or by a compiler.

Here we introduce two representative STM designs.

TL2 [33] is an ownership record (orec) based STM. Reads and writes in a transaction are hashed to an array of ownership records according to their addresses. Each record is basically a sequence lock [63] whose value indicates the orec’s version. To perform a read, its orec version is read twice, immediately before and immediately after the actual read, in order to ensure the data consistency. A write locks the corresponding orec. Writes are buffered in a write set until commit. If the desired orec is already locked by other threads, the transaction aborts. Meanwhile, to ensure opacity (continuous consistency) [45], all threads share a global version number, which is incremented when a writer transaction commits. A transaction starts with keeping a value of the global version (beginning version). If any orec accessed during the execution has a version greater than the beginning version, the transaction aborts. The main benefit of TL2 is that it acquires locks
at very fine granularity so there is no serialization bottleneck.

NOrec [28], as its name suggests, is an STM without ownership records. Rather than version numbers, it uses value-based validation, where all read pairs of a transaction (memory address and a snapshot of corresponding value at the time of read) are kept in a thread-local read set. During validation, every pair in the set is checked by comparing the snapshot and latest value at the specific address. Any mismatch indicates an inconsistent view of memory and the current transaction will abort. Like TL2, NOrec uses a global timestamp to check if any other writing transactions have committed. Due to the lack of ownership records, there is no way to tell if a specific memory address is being written by other threads. Transactions in NOrec have to commit their write sets sequentially. The use of value-based validation in NOrec guarantees privatization safe [76], meaning that the behavior that a thread privatizes shared data transactionally and then uses the privatized data nontransactionally will not jeopardize other transactions working on the same shared data. However, the serial commit policy limits NOrec’s scalability.

Some STM systems add special semantics in order to take advantage of algorithm knowledge. For instance, Early Release [53] is a feature that allows STM to drop a part or entire read history at any point within a transaction. Any conflict in the dropped read history can be safely ignored and will not abort the transaction. Therefore, it benefits data structures containing long search operation and the search result only depends on the most recently shared read rather than the entire read set. SpecTM [37] is a lightweight STM system that trades some of the convenience of standard TM interface for the performance. It provides expert programmers with a group of TM-style auxiliary functions that can be used to write small transactions with less overhead.

Though some STM has good scalability as a result of its optimistic design, the validation cost, instrumentation cost and other overhead together make STM
slower than HTM. However, one advantage of STM over HTM is that STM is not bounded by hardware capacity, thus STM is sometimes considered as a backup of HTM in a hybrid TM system for cases in which hardware resources cannot handle a large transaction.

### 2.2.2 Hardware Transactional Memory

Hardware Transactional Memory (HTM) implements the transactional execution in hardware. Since all work (speculation, conflict detection, and conflict resolution) is done by the hardware, it may provide performance that is as good as or better than that of the best-tuned fine-grain locking. Numerous HTM designs have been proposed in the literature.

These proposals can be broadly categorized into two kinds, according to the way read and write sets are tracked for conflict detection. One kind of hardware, TCC [46], TokenTM [11], Intel’s Haswell [131], and IBM’s zEC12 [58] for example, tracks transactional reads and writes in private L1 and/or L2 cache. Commonly, extra transactional bits are added to a cache tag to track the ownership of a cache line. Bounded by cache capacity, however, cache-based HTMs are not friendly to transactions with a big footprint, as tracking information will be lost once a line is evicted from the cache. In order to support “unbounded” transactions, other proposals, such as LogTM-SE [130] and FlexTM [108], decouple the read and write set tracking from L1 cache tags and use Bloom-filter based signatures instead, at the cost of lower accuracy of conflict detection.

Although “unboundedness” appears to be an attractive feature in the literature [4, 96], real HTM designs adopt a more conservative and realistic approach. All of the existing commercial HTM processors, including one of Azul’s processors [23], Sun’s Rock [19], IBM’s BG/Q [124], IBM’s zEC12 [58], IBM’s POWER 8 [17], and Intel’s Haswell [131], are best-effort HTMs, which commonly
do not provide progress guarantees (zEC12 provides a progress guarantee for small, special “constrained” transactions). In addition to bounded hardware resources for transactional execution, they also abort a transaction on conditions such as prohibited instructions and interrupts.

**Example Design: HTM in IBM zEnterprise EC12**

The IBM zEnterprise EC12 [58] is a multi-socket machine with HTM capability. At the ISA level, six new instructions were introduced as the Transactional Execution Facility. The hardware guarantees strong isolation for a transaction marked by **TBEGIN** and **TEND** instructions. A transaction may abort for various reasons, including interrupts, restricted instructions, excessive nesting depth, capacity overflow, conflicting accesses, and so on. Upon an abort, **TBEGIN** sets the condition code (CC), and execution resumes with the instruction immediately following **TBEGIN**. This instruction is typically a conditional branch, which either commences transactional work (if CC = 0) or jumps to an abort handler (if CC ≠ 0). The hardware characterizes each abort as persistent (CC = 3) or nonpersistent (CC = 2). Nonpersistent aborts (e.g., fetch/store conflicts) may be worth retrying; persistent aborts (e.g., overflows) may not. A transaction can also be explicitly aborted by using a **TABORT** instruction with a user-defined abort reason code. Additional information about the cause of an abort can be obtained by enabling a transaction diagnostic block (TBD), but this significantly increases overhead.

One zEC12 chip contains six out-of-order superscalar cores with a clock speed of 5.5 GHz. Each core has a private 96 KB 6-way associative L1 data cache and a private 1 MB 8-way associative L2 data cache, with 256 B cache lines. Cores on the same chip share a 48 MB L3 cache. To track the read set of a transaction, every tag in the L1 cache directory includes a a tx-read bit. When a transactional read line is evicted from the L1 cache, it is tracked by a LRU-extension vector.
instead. LRU-extension vectors extend the read set capability to the L2 cache size and associativity. Since the L1 and L2 are both write-through, proper handling of transactional writes requires a store cache, a circular queue of 64 half-lines, which buffers and merges stores until the end of the transaction, at which point it sends them on to the L3 cache. Transactional write set size is thus limited by the store cache size and the L2 cache size and associativity.

2.2.3 Hybrid Transactional Memory Systems

Since current commercial HTMs are bounded—typically limited by cache capacity—large transactions may never complete as a hardware transaction on these machines. Hybrid TM addresses this issue by taking advantage of HTM’s native speed and STM’s flexibility and generality.

HyTM [30] puts emphasis on accelerating STM with HTM support, as long as HTM is available and helpful. The STM part in HyTM is a typical orec-based system, which can independently work in the absence of HTM support. For each transaction, the HyTM compiler generates both STM and HTM paths. Shared loads/stores in the HTM path are also instrumented in order to detect the presence of an STM path. Before accessing actual shared data in the HTM path, an instrumented load or store checks its corresponding orec first. If the orec is already owned or modified by a software transaction, current HTM path aborts. One disadvantage of HyTM is that the instrumentation of the HTM path reduces the speed benefit of HTM.

Unlike HyTM, in which HTM accelerates STM, Hybrid NOrec [29] favors HTM by running as many transactions on HTM as possible. When HTM fails, the system reverts to the STM path as a backup. The most significant advantage of Hybrid NOrec over HyTM is that the HTM transaction does not need instrumentation, so in most cases Hybrid NOrec runs transactions at HTM’s native
speed. HTM and STM communicate with each other via the global timestamp and one or more HTM counters, thus the timestamp and counters tend to be major contributors to HTM aborts.

Rather than simply mixing HTM and STM paths together, Phased Transactional Memory (PhTM), proposed by Lev et al. [69], is a coarse grain TM system, which divides the execution of a program into different phases. The program can execute in either HTM or STM, but not both, in each phase. By observing the behavior of current mode, PhTM decides whether to switch to the other mode. However, HTM is the favorite mode; consequently, most transactions run on fast HTM.

Generally speaking, the profit of speculation in TM systems decreases when data contention increases. Under high contention, TM may perform worse than conventional synchronization techniques. A few hybrid systems avoid this issue by reverting to other synchronization techniques once high contention is observed. For example, Adaptive Locks [120] combine transactional memory and conventional locking for atomic execution. An atomic region can switch between locking mode and TM mode. During execution, one mode is dynamically chosen according to data contention, which is measured as abort rate (in TM mode) or the number of waiters on the lock (in locking mode). Similarly, Dice et al. [36] propose Adaptive Lock Elision (ALE) to allow a critical section to execute in one of three modes: Lock, software optimistic execution, and HTM. The choice of execution mode is made at run time based on statistical information.

### 2.2.4 Conflict Detection and Management

A conflict detection policy describes when a TM system checks for possible conflicts among ongoing transactions. An eager policy detects conflict as early as possible, in the hope of minimizing wasted work. A lazy policy delays the conflict
detection as long as possibly, usually to commit time. There is no single winner between them, as different applications favor different detection policies [111, 117]. Some TM systems, such as EazyHTM [117] and FlexTM [108], also support mixed policies.

A conflict resolution policy determines what to do if a conflict is discovered. Two simple resolution policies are requester-wins, in which the requester transaction to a conflicting resource aborts the owner, and requester-loses, in which the requester is aborted. More sophisticated conflict resolution strategies have been explored in STM [111].

Another purpose of conflict management is to reduce or avoid conflict aborts, usually in HTMs with eager conflict detection. For example, LogTM-SE [130] stalls one or more conflicting transactions upon conflicts. Potential deadlock is detected using coherence message. Similarly, IBM’s zEC12 stalls a requester briefly (using a NAK that was already available in the coherence protocol) instead of aborting the owner immediately, in the hope that the owner would submit during the stall period. However, as suggested by Shriraman and Dwarkadas [107], even a simple conflict manager creates significant implementation challenges because of the possible protocol extension and validation cost.

More sophisticated hardware solutions have been proposed for eager HTMs. In Ramadan et al.’s dependence-aware transactions (DATM) [97], dependent data (mainly cyclic dependences) are forwarded between on-going transactions. Jafri et al. proposed Wait-n-GoTM [59], which is based on TokenTM [11], to serialize cyclic dependencies. In this approach, a hardware predictor learns the set of cache lines that are likely to contribute to cyclic conflicts as well as the length of cycles. Hardware exceptions are generated to inform the software handler to wait before entering cyclic dependencies in case other transactions are already in the cycle. Most recently, Qian et al. proposed OmniOrder [93] to support cycle detection and conflict serialization in a directory-based coherence environment.
RETCON [10], which targets lazy HTMs, tries to “rescue” conflicting transactions by re-executing the conflicted code slice at commit time. FlexTM allows a flexible choice of policies (including requester-wins, committer-wins, age priority, and so on) by offloading both conflict resolution and management onto the software side.

2.3 Interaction Between Compiler and Speculation

There are relatively few previous projects focusing on compiler optimization for speculation. Two of them improve lock assignment of STM. Riegel et al. [101] use data structure analysis to detect data partitions and map data to partition placeholders statically. At run time, the partition-aware STM assigns different STM algorithms to different partitions. Also, the system chooses appropriate locking granularities based on the characteristics of the data structure.

Ownership record based STMs—TL2 for example—grab locks at very fine granularity (at the word level or even at the byte level), taking no actual data access pattern into consideration. This results in significant validation cost for transactions with a large footprint. Mannarswamy et al. [74] improve TL2’s lock assignment by statically inferring conflict information for shared data. Consider a transaction that accesses a complex data structure with many fields: the original TL2 algorithm will dynamically assign one lock to each data structure field. By contrast, if the runtime can gain the knowledge of the data structure from the compiler, it will acquire only one lock for the entire structure so that locking overhead can be dramatically reduced.
3 Compiler Aided Manual Speculation

3.1 Introduction

Concurrent data structures play a key role in multithreaded programming. Typical implementations use locks to ensure the atomicity of method invocations, but as discussed in Section 2.1, locks are often overly pessimistic. By contrary, nonblocking concurrent data structures are inherently optimistic, but the creation of efficient nonblocking algorithms is notoriously difficult.

So what about speculation? Work by Bronson et al. [14] demonstrates that hand-written, data structure-specific speculation can provide a significant performance advantage over traditional pessimistic alternatives. Specifically, the authors describe a relaxed-balance speculative AVL tree that outperforms the java.util.concurrent.ConcurrentSkipListMap by 32–39%. Their code employs a variety of highly clever optimizations. While fast, it is very complex, and provides little guidance for the construction of other hand-written speculative data structures.

Our work attempts to regularize the notion of manual speculation (MSpec). Specifically, we characterize MSpec as a design pattern that transforms traditional, pessimistic code into optimistic, speculative code by addressing three key
questions: (1) Which work should be moved out of a critical section and executed speculatively? (2) How does the remaining critical section validate that the speculative work was correct? (3) How do we avoid erroneous behavior when speculative code sees inconsistent data?

Using MSpec, we have constructed speculative versions of four example data structures: equivalence sets, cuckoo hash tables, Bink-trees, and a linear bitmap allocator. Our implementations outperform not only STM, but also pessimistic code with similar or finer granularity locking. The advantage typically comes both from reducing the overall number of atomic (read-modify-write) instructions and from moving instructions and cache misses out of critical sections (i.e., off the critical path) and into speculative computation. We note that unlike STM, MSpec does not require locking and validation to be performed at identical granularities.

In developing MSpec, our intent was to provide a useful tool for the construction (by experts) of concurrent data structures. We never expected it to be easy to use, but we expected the principal challenges to revolve around understanding the (data structure-specific) performance bottlenecks and speculation opportunities. As it turned out, some of the more tedious, mechanical aspects of MSpec were equally or more problematic. First, the partitioning of code into speculative and nonspeculative parts, and the mix of validation code and normal code in the speculative part, breaks natural control flows, making the source code difficult to read, understand, and debug. Second, since the original locking code must generally be preserved as a fallback when speculation fails, the programmer must maintain two versions, applying updates and bug fixes to both. Third, because naive speculation introduces data races, the programmer must deeply understand the memory model, and pepper the code with atomic annotations and/or memory fences.

To address these “mechanical” challenges, we have developed a set of program annotations and a source-to-source translator, CSpec, that generates speculative
code automatically from annotated source. CSpec allows the programmer to continue to work on the (lightly annotated) original code. It eliminates the possibility of version drift between speculative and nonspeculative versions, and automates the insertion of fences to eliminate data races.

As a motivating example, we present our hand-constructed (MSpec) version of equivalence sets in Section 3.2, with performance results and implementation experience. We then turn to the CSpec language extensions and source-to-source translator in Section 3.3, and to guidelines for using these in Section 3.4. Additional case studies appear in Section 3.5, with performance results for both MSpec and CSpec versions. Our principal conclusion, discussed in Section 3.6, is that compiler-assisted manual speculation can be highly attractive option. While more difficult to use than STM or coarse-grain locking, it is substantially easier than either fully manual speculation or the creation of ad-hoc, data structure-specific nonblocking or fine-grain locking alternatives.

### 3.2 Motivating Example: Equivalence Sets

An instance of the data structure for equivalence sets partitions some universe of elements into a collection of disjoint sets, where the elements of a given set have some property in common. For illustrative purposes, the code of Figure 3.1 envisions sets of integers, each represented by a sorted doubly-linked list. Two methods are shown. The `Sum` method iterates over a specified set and returns some aggregate result. The `Move` method moves an integer from one set to another. We have included a `set` field in each element to support a constant-time `MemberOf` method (not shown).
struct Element {
    int key;
    Element *next, *prev;
    Set *set;
};

struct Set {
    Element head;
    Lock lck;
};

int ESets::Sum(Set *s) {
    int sum = 0;
    s->lck.lock();
    Element *pnext = s->head->next;
    while (pnext != s->head) {
        sum += pnext->key;
        pnext = pnext->next;
    }
    s->lck.unlock();
    return sum;
}

void ESets::Move(Element *e, Set *s) {
    Set *oset = e->set;
    grab_unordered_locks(oset->lck, s->lck);
    // find e's next element in s
    Element *pprev = s->head;
    Element *pnext = pprev->next;
    while (pnext->key < e->key) {
        pprev = pnext;
        pnext = pprev->next;
    }
    // remove e from its original set
    e->prev->next = e->next;
    e->next->prev = e->prev;
    // insert e before pnext
    e->next = pnext;
    e->prev = pprev;
    e->set = s;
    pprev->next = e;
    pnext->prev = e;
    release_locks(oset->lck, s->lck);
}

Figure 3.1: Per-set lock implementation of concurrent equivalence sets. The key field of each set's head is initialized to $+\infty$ to avoid loop bound checks in lines 16. For code simplicity, it's assumed that for any element, there is at most one thread calling Move on it at any time.

### 3.2.1 Implementation Possibilities

A conventional lock-based implementation of equivalence sets is intuitive and straightforward: each method comprises a single critical section. Code with a single lock per set (lines 1–43 in Figure 3.1) provides a good balance between coding efficiency and performance. It is only slightly more complicated than a scheme in which all sets share a single global lock—the critical section in Move must acquire two locks (in canonical order, to avoid deadlock at line 25)—but scalability and throughput are significantly better. Given the use of doubly-linked lists, we
do not see any hope for an efficient nonblocking implementation on machines with only single-word (CAS or LL/SC) atomic primitives.

STM can of course be used to create an optimistic version of the code, simply by replacing critical sections with transactions. The resulting performance (with the GCC default STM system—see Figures 3.3 and 3.4) is sometimes better than with a global lock, but not dramatically or uniformly so. Even a specialized STM with “elastic transactions” [40], which provide optimization for the search phase in Move, still lags behind per-set locking. STM incurs nontrivial overhead to initialize a transaction and to inspect and modify metadata on each shared memory access. Additional overheads stem from two major lost opportunities to exploit application semantics. First, to avoid proceeding on the basis on an inconsistent view of memory, STM systems typically validate that view on every shared memory read, even when the programmer may know that inconsistency is harmless. And while heuristics may reduce the cost of validation in many cases, the fallback typically takes time proportional to the number of locations read. Second, STM systems for unmanaged languages typically perform locking based on hash-based “slices” of memory [33], which almost never coincide with program-level objects. In our equivalence set example, the commit code for GCC STM must validate all head and next pointers read in the loop in Sum, and acquires 6 locks (one per pointer) for write back in Move. Hand-written code can do much better.

### 3.2.2 A Manual Speculative Implementation

Our MSpec version of Sum (line 57–69 in Figure 3.2) exploits the fact that the method is read only, that it traverses only one set, and that nodes encountered in that traversal will never have garbage next pointers, even if moved to another set. These observations allow us to write an obstruction-free [52] MSpecSum that validates using a single version number in each set. Because locking and validation are performed at the same granularity, we combine the lock and version number
// speculative implementation
// for TSO machines
struct Element {
  int key;
  atomic< Element*> next, prev;
  atomic< Set*> set;
};

struct Set {
  Element head;
  SeqLock lck;
};

int ESets::MSpecSum(Set* s) {
  again:
  int sum = 0;
  int v = s->lck.v;
  if (v & 1) goto again;
  Element* pnext = s->head;
  while (pnext!=s->head && v==s->lck.v) {
    sum += pnext->value;
    pnext = pnext->next;
  }
  if (v != s->lck.v) goto again;
  return sum;
}

void ESets::MSpecMove(Element* e, Set* s) {
  Set* oset = e->set;
  again:
  Element* pprev = s->head;
  Element* pnext = pprev->next;
  while (pnext->value < e->value) {
    pprev = next;
    pnext = pprev->next;
    if (pnext->set != s)
      goto again;
  }
  grab_unordered_locks(oset->lck, s->lck);
  if (AL(pnext->set, MO_relaxed) != s ||
      AL(pnext->prev, MO_relaxed) != pprev) {
    release_locks(oset->lck, s->lck);
    goto again;
  }
  AS(AL(e->prev, MO_relaxed)->next,
      AL(e->next, MO_relaxed), MO_relaxed);
  AS(AL(e->next, MO_relaxed)->prev,
      AL(e->prev, MO_relaxed), MO_relaxed);
  AS(e->next, pnext, MO_relaxed);
  AS(e->set, s, MO_relaxed);
  AS(pprev->next, e, MO_relaxed);
  AS(pnext->prev, e, MO_relaxed);
  release_locks(oset->lck, s->lck);
}

Figure 3.2: Manual speculative implementation of concurrent equivalence sets for TSO machines. AL=atomic_load, AS=atomic_store, MO_=memory_order_*.

Atomic loads in the speculative part (lines 73–80) use the default memory_order_seq_cst, which incurs no extra costs on TSO machines.

into a single sequence lock field (lck.v) [63]. If we iterate over the entire set without observing a change in this field, we know that we have seen a consistent snapshot. As in most STM systems, failed validation aborts the loop and starts over (line 67). In manual speculation, however, the programmer is responsible for
any state that must be restored (in this case, none).

The baseline critical section in Move begins by finding an appropriate (sorted) position in the target list s (lines 27–32 in Figure 3.1). The element e is then removed from its original list (lines 34–35) and inserted into s at the chosen position (lines 37–41). The position-finding part of Move is read only, so it can be performed speculatively in MSpecMove—that is, before entering the critical section. The validation at line 82 ensures that the pnext element remains in the same set and no new element has been inserted between pprev and pnext since line 80, so that it is correct to insert e before pnext. The other validation, at line 78, ensures that the while loop continues to traverse the same set, and will therefore terminate. A set’s version number is increased both before and after a modification. The low bit functions as a lock: an odd version number indicates that an update is in process, preventing other threads from starting new speculations. Version number increments occur implicitly in lock acquisition and release.

Under the C++11 memory model [13], any class field that is read during speculation and written in a critical section (in our case, three fields of Element) must be declared as an atomic variable to avoid data races. For reads and writes to atomics inside the critical section, since locks already guarantee exclusive modification, we can specify relaxed memory order for best performance. For sequence-lock based speculative reads, depending on the hardware, different methods for tagging memory orders add different overhead [12]. Since our experiments employ total store order (TSO) machines [1] like the x86 and SPARC, in which loads are not reordered by the processor, atomic loads incur no extra costs under the sequential consistency memory order. So we simply skip tagging speculative reads and let them use memory_order_seq_cst by default in the speculative part (lines 73–80).
3.2.3 Performance Results

We tested our code on an Oracle (Sun) Niagara 2 and an Intel Xeon E5649. The Niagara machine has two UltraSPARC T2+ chips, each with 8 in-order, 1.2 GHz, dual-issue cores, and 8 hardware threads per core (4 threads per pipeline). Each core (8 logical cores) has 24KB L1 cache, and cores on one chip share 4 MB integrated L2 cache. The Xeon machine also has two chips, each with 6 out-of-order, 2.53 GHz, hyper-threaded cores, for a total of 24 hardware thread contexts. Each core has a 256 KB L2 cache, and 6 cores in a single chip share a 12 MB unified L3 cache. Code was compiled with gcc 4.7.1 (-O3).

To measure throughput, we arrange for worker threads to repeatedly call randomly chosen methods for a fixed period of time. We bind each thread to a logical core to eliminate thread migration, and fill all thread contexts on a given chip before employing multiple chips. Our mutex locks, where not otherwise specified, use test-and-test\_and\_set with exponential back-off, tuned individually for the two machines.

Test Configurations

We compare six different implementations of equivalence sets. FGL and SpecFGL are the versions of Figure 3.1, with set-granularity locking. (Code for the Move operation includes an extra check to make sure that e is still in the same set after line 25.) CGL and SpecCGL are analogous versions that use a single global lock. Gnu-STM uses the default STM system that ships with gcc 4.7.1; ε-STM employs elastic transactions [40], which are optimized for search structures. Loads and stores of shared locations were hand-annotated in ε-STM.
Performance and Scalability

Performance results appear in Figures 3.3 and 3.4. We use 50 equivalence sets in all cases, with either 500 or 5000 elements in the universe (10 or 100 per set, on average). We use 100% Move operations to simulate a write-dominated workload, and a 50/50 mix of Move and Sum operations to simulate a mixed but higher-contention workload. The number of elements per set determines the amount of work that can be moved out of the critical section in SpecMove.

Figure 3.3 illustrates scalability on Niagara 2. As expected, FGL outperforms CGL in all tests, and SpecFGL outperforms SpecCGL. Since an invocation of the Move method holds 2 locks simultaneously, FGL reaches its peak throughput when
the thread count is around 32. The sharper performance drop after 64 threads is
due to cross-chip communication costs.

Manual speculation improves scalability and throughput for both CGL and
FGL. In the 5000-element, 50/50 sum/move case, SpecCGL even outperforms
FGL, out to 80 threads. This suggests that simple coarse-grained locking with
speculation could be an attractive alternative to fine-grained locking for workloads
with significant work amenable to speculation. The baseline overhead of SpecFGL,
measured by comparing to FGL on a single thread, is less than 10%. Therefore,
even without contention, SpecFGL can deliver competitive performance. By con-
trast, both gnu-STM and ε-STM have significant baseline overhead (2–4× slower
than CGL on a single thread), and can outperform only CGL.
Results on the Xeon machine resemble those on Niagara 2, with the interesting exception that single-thread performance is significantly higher in the 500-element case, where the data set can almost fit in the 32KB L1 data cache.

3.2.4 Limitations of Manual Speculation

The above results show that hand-written speculation can limit the cost of validation and yield significant performance and scalability improvements not only over STM, but also over lock-based alternatives. Other examples, summaries of which appear in Section 3.5, confirm that these results are not limited to a single data structure. The presence of general principles suggests, in fact, that manual speculation be thought of as a first-class design pattern for concurrent data structures. At the same time, our experience suggests that this pattern is not very easy to use.

Some of the challenges are expected, and are discussed in more detail in Section 3.4. To extract concurrency, one must understand the workload characteristics and concurrency bottlenecks that suggest an opportunity for speculation. To control the cost of validation, one must identify any data-structure-specific indications of consistency. To maintain correctness, one must insert validation before any operation that might produce anomalous results in the wake of mutually inconsistent reads.

Several challenges, however, are more “mechanical,” and suggest the need for automation. First, the changed code layout and the mix of validation code and normal code break natural control flows, making the source code difficult to read, understand and debug. Second, if the original locking code co-exists with its speculative version (either as a backup when speculation fails or as an alternative when speculation is unprofitable), the programmer will have to maintain two separate versions of the code, and make sure that they track each other in the face of up-
dates and bug fixes. Third, manual speculation requires a deep understanding of
the underlying memory model. Speculative loads, almost by definition, constitute
data races with stores in the critical sections of other threads. In C++ 11, where
data races are illegal, the programmer must identify and annotate all places where
naive speculation would otherwise introduce a race. As discussed by Boehm [12],
even the choice of annotation is nontrivial, and potentially architecture-dependent
(two correct annotations—a minimal annotation and an unnecessarily conserva-
tive annotation—may have a big performance difference on some machines). In
the equivalence sets example, the default sequentially consistent ordering is zero-
cost for TSO machines, but has unacceptable overhead on PowerPC machines.

3.3 Compiler-Aided Manual Speculation

We propose to overcome the difficulties in applying manual speculation with the
aid of a compiler. The key idea is to automatically generate speculative code from
an annotated version of the lock-based code, allowing the programmer to focus
on higher-level issues of what to do speculatively, and how and when to validate.
This semi-automatic approach to speculation, which we call CSpec (compiler-
aided manual speculation), consists of (1) a simple but flexible set of annotations
to specify the speculation method, and (2) a source-to-source compiler that trans-
forms the annotated source into an explicitly speculative version.

3.3.1 Interface

The language interface is designed to be as simple as possible. It comprises the
following directives:

#pragma spec: tells the compiler to generate a speculative version for the
following critical section. This directive should appear immediately before
a lock acquisition statement, which must be the single entry point of its critical section.

`#pragma spec consume_lock(lock0, lock1, ...)`: instructs the compiler to “consume” one or more locks (these should be a subset of the critical section’s lock set) at a particular place in the code. Here, “consume” means that the lock is truly needed (to protect writes or to achieve a linearization point), and that speculation on data protected by the lock must end. Statements that cannot be reached from any `consume_lock` form the speculative part of a critical section. Multiple `consume_lock` directives are allowed in a critical section. It is also legal to consume the same lock more than once on a given code path: the compiler is responsible for acquiring each lock at the first `consume_lock` that names it.

`#pragma spec set_checkpoint(id)`: marks a checkpoint with a specific integer id (id>0). A checkpoint is a place to which execution can roll back after a failed validation. During rollback, all local variables modified beyond the checkpoint will be reset to their snapshots at the checkpoint. The default checkpoint (id=0) is located at the critical section’s entry point.

`#pragma spec validate_ver(ver0, ver1, ...[, cp_id])`: does a version number-based validation. If any version number has changed since checkpoint `cp_id`, roll back to that checkpoint.

`#pragma spec validate_val(val0, val1, ...[, cp_id])`: is similar to `validate_ver`, but does value-based validation.

`#pragma spec validate_cond(cond_expr[, cp_id])`: evaluates the expression `cond_expr` and rolls back to checkpoint `cp_id` if it is false.

`#pragma spec waive_rollback(cp_id, var0, var1, ...)`: waives value rollback for specified variables at checkpoint `cp_id`.
```c
int ESets::Sum(Set *s) {
    int sum = 0;
    #pragma spec
    s->lck.lock();
    Element *pnext = s->head->next;
    while (pnext != s->head) {
        sum += pnext->key;
        pnext = pnext->next;
        #pragma spec validate Ver(s->lck.v)
    }
    #pragma spec validate Ver(s->lck.v)
    s->lck.unlock();
    return sum;
}

void ESets::Move(Element *e, Set *s) {
    Set *oset = e->set;
    #pragma spec
    grab_unordered_locks(oset->lck, s->lck);
    Element *pprev = s->head;
    Element *pnext = pprev->next;
    while (pnext->key < e->key) {
        pprev = pnext;
        pnext = pprev->next;
        #pragma spec validate Cond(pnext->set==s)
    }
    #pragma spec consume_lock(oset->lck, s->lck)
    #pragma spec validate Cond(pnext->set==s && pnext->prev==pprev)
    e->prev->next = e->next;
    ...... // same as L35–41 of Figure 1
    release_locks(oset->lck, s->lck);
}
```

Figure 3.5: Concurrent equivalence sets using CSpec.

Figure 3.5 presents a re-implementation of the equivalence set data structure using CSpec annotations in Figure 3.1. This new version is almost identical to the original lock-based code, except for seven embedded directives. Compiler output for the `Sum` operation appears in Figure 3.6. It includes both the original locking version and a speculative version. The speculative version includes appropriate
int ESets::Sum(Set *s) {
    .... // same as L12–20 of Figure 1
}

int ESets::SpecSum(Set *s) {
    int sum = 0;
    int SPEC_tmp_var_0 = sum;
    SPEC_label_1:
    sum = SPEC_tmp_var_0;
    int SPEC_ver_0 = spec::wait_ver(s->lck.v);
    Element *pnext = AL(s->head->next, MO_seq_cst);
    while (pnext != s->head) {
        sum += pnext->key;
        pnext = AL(pnext->next, MO_seq_cst);
        if (AL(s->lck.v, MO_seq_cst) != SPEC_ver_0)
            goto SPEC_label_1;
    }
    if (AL(s->lck.v, MO_seq_cst) != SPEC_ver_0)
        goto SPEC_label_1;
    return sum;
}

Figure 3.6: Automatically generated code for the Sum method.

tags on all atomic accesses, optimized for the target machine.

Compared with the pure manual speculation code in Figure 3.1, the CSpec implementation has the following advantages: (1) The language interface is concise. Usually, only a small number of directives is needed to describe the speculation mechanism; often, the original control flow can be retained, with no code motion required. The resulting code remains clear and readable. (2) Code maintenance is straightforward: there is no need to manage separate speculative and nonspeculative versions of the source. Any updates will propagate to both versions in the compiler’s output. (3) The programmer is no longer exposed to low-level details of the memory model, as the compiler will handle them properly.
3.3.2 Implementation

Our compiler support is implemented as a clang [22] front-end plugin. The plugin takes user-annotated source code as input and does source-to-source transformation to create a speculative version. A nonspeculative version is also generated, simply by discarding all embedded directives. Alternative implementations, such as IR-based transformation or static and run-time hybrid support would also be possible; we leave these for future exploration.

We define a lock’s live range as the set of all statements that are reachable from its acquisition statement but not reachable from its release statement. A speculative code region is defined as the union of all live ranges of the locks that appear in a critical section’s entry point—the lock acquisition call immediately following #pragma spec.

Lockset Inference. A key part of the CSpec translation algorithm is to identify the lock set that each consume_lock should actually acquire. The intuition behind the analysis is straightforward: we want every critical section, at run time, to acquire locks exactly once (this marks the end of speculation), and to acquire all locks named in any consume_lock directive that may subsequently be encountered on any path to the end of the critical section. Conservatively, we could always acquire the entire lockset at the first dynamically encountered consume_lock directive. To the extent possible with static analysis, we wish to reduce this set when possible, so that locks that are never consumed are also never acquired. Pseudocode of the algorithm we use to accomplish this goal appears in Algorithm 3.1.

The algorithm works in two phases. The first phase (lines 3–7) calculates the LockSet of each statement in code region R. These are the locks that must be acquired before the statement executes at run time. The second phase first infers the lock set for each consume_lock (lines 9–10) so that the set will con-
Algorithm 3.1: Translating consume_lock directives

Input: a speculative code region \( R \)

1. // GetLiveRange\((L, S)\): get \( L \)'s live range assuming \( S \) is the acquisition statement;
2. \( C \leftarrow \) all consume_lock directives in \( R \);
3. foreach consume_lock \( cl \) in \( C \) do
4.     foreach Lock \( l \) in \( cl \)’s LockArgList do
5.         \( cl \)’s LiveRange \( \leftarrow \) \( cl \)’s LiveRange \( \cup \) GetLiveRange\((l, cl)\);
6.         foreach Statement \( s \) in GetLiveRange\((l, cl)\) do
7.             \( s \)’s LockSet \( \leftarrow \) \( s \)’s LockSet \( \cup \) \( cl \)’s LockArgList;
8.     endforeach
9.     foreach Statement \( s \) in \( cl \)’s LiveRange do
10.        \( cl \)’s LockSet \( \leftarrow \) \( cl \)’s LockSet \( \cup \) \( s \)’s LockSet;
11.    if \( cl \) is unreachable from any other consume_lock in \( C \) then
12.        replace \( cl \) with LockStmt\((R, cl \)’s LockSet\);
13.    else if \( cl \) is not dominated by another consume_lock in \( C \) then
14.        remove \( cl \);
15.    endif
16.    foreach BasicBlock \( b \) in GetBasicBlock\((cl)\)’s Parents do
17.        if a path from \( R \)’s entry to \( cl \) goes through \( b \) and the path does not contain any other consume_lock then
18.          emit LockStmt\((R, cl \)’s LockSet\) at the end of \( b \);
19.        endif
20.    endforeach
21. else // \( cl \) is dominated by other consume_lock
22.    remove \( cl \);
23. endif

tain all locks required by the consume_lock’s live range (all code paths between the consume_lock statement and the exit of the critical section). Then for each consume_lock, the algorithm decides how to handle it according to its reachability (lines 11–19). The generated lock acquisition calls use the same function name as the original call (the entry point of \( R \)) and the lock parameters appear in the same order as in the original call’s parameter list. Thus, the output code is deadlock free if the original code was.

As an alternative to static lockset inference, we could allow locks to be acquired more than once at run time, and release each the appropriate number of times at the end of the critical section. This strategy, however, would be incompatible with source code using non-reentrant locks.
**Checkpoint.** A checkpoint serves as a possible rollback position. There are three kinds of variables a checkpoint may snapshot: (1) Live-in local variables. For a local variable that is declared before a checkpoint and may be modified after the point, our source transformation tool creates a local mirror for that variable and copies its value back when speculation fails (see `sum` in Figure 3.6). Nonlocal variables are assumed to be shared; if they need to be restored, the programmer will have to do it manually. (2) Version numbers. The tool finds `validate_vers` that may roll back to this checkpoint and, for each, inserts a `wait_ver` call to wait until the number is unblocked (line 10 in Figure 3.6). The values read (e.g., `SPEC_ver_0`) are kept for validation. (3) Validation values. `Validate_vals` are handled in the same way as `validate_vers`, except that no `wait_ver` is inserted.

**Tagging Atomic Variables.** After code transformation, our tool detects the class fields that are both written in write-back mode (after consuming a lock) and accessed in speculative mode, and re-declares them as atomic variables. Accesses to these variables are grouped into three categories: (1) accesses in write-back mode; (2) reads in validation; (3) other reads in a speculative phase. The tool selects an appropriate (target machine-specific) memory ordering scheme (including a fence if necessary) for each access category.

**Switching Between Locking and Speculative Versions.** The generation of two versions of a data structure opens the possibility of dynamically switching between them for the best performance. Since speculation doesn’t win in all cases, it might sometimes be better to revert to the original locking code. The switch could be driven by user-specified conditions such as workload size or current number of threads. Switch conditions might also be automatically produced by the compiler based on profiling results, or on run-time information like the abort rate of speculation. These possibilities are all subjects for future work. In the experiments reported here, we use only the speculative version of the code.
3.3.3 Limitations

Our source-to-source tool currently faces three limitations, all of which suggest additional subjects for future work. First, the critical section must have a single entry point. In source programs with multiple entry points, it may be possible to merge these entry points by manually rearranging code structure. Second, we do not support nested speculation due to the complexity of nested rollbacks; inner directives will simply be discarded. Third, the analysis of locksets is static, and thus cannot handle complicated data flow and re-assignment of lock variables. We argue that these are uncommon in concurrent data structures, and could be addressed through manual source code changes.

3.4 Principles of Coding with CSpec

Sections 3.2 and 3.3 presented an example of manual speculation and a compiler-based tool to assist in the process. In the current section we generalize on the example, focusing on three key questions that the user of CSpec must consider.

3.4.1 Where do we place consume_lock directives?

This question amounts to “what should be done in speculation?” because consume_locks mark the division between the speculative and nonspeculative parts of the original critical section. Generally, consume_lock(L) is placed right before the first statement that modifies the shared data protected by L on a given code path. Sometimes a modification to shared data A can only be performed under the guarantee that shared data B won’t be changed; in this case B’s lock should also be consumed. Since the speculative parts come from the original critical section, it may be profitable to rearrange the code to delay the occurrence of consume_locks. The principal caveat is that too large an increase in total work—
e.g., due to misspeculation or extra validation—may change the critical path, so that the remaining critical section is no longer the application bottleneck.

In general, a to-be-atomic method may consist of several logical steps. These steps may have different probabilities of conflicting with the critical sections of other method invocations. The overall conflict rate (and hence abort rate) for the speculative phase of a method is bounded below by the abort rate of the most conflict-prone step. Steps with a high conflict rate may therefore best be left after \texttt{consume_lock}.

There are several common code patterns in concurrent data structures. Collection classes, for example, typically provide \texttt{lookup}, \texttt{insert}, and \texttt{remove} methods. \texttt{Lookup} is typically read-only, so there’s no need to place any \texttt{consume_lock} in it. \texttt{Insert} and \texttt{remove} typically start with a search to see whether the desired key is present; we can make this speculative as well, by inserting \texttt{consume_locks} before the actual insertion/deletion. In resource managers, an \texttt{allocate} method typically searches for free resources in a shared pool before actually performing allocation. In other data structures, time-consuming logical or mathematical computations, such as compression and encryption, are also good candidates for speculation.

At least three factors at the hardware level can account for a reduction in execution time when speculation is successful. First, speculation may lead to a smaller number of instructions on the program’s critical path, assuming this consisted largely of critical sections. Second, since the speculative phase and the following critical section usually work on similar data sets, speculation can serve as a data prefetcher, effectively moving cache misses off the critical path. This can improve performance even when the total number of cache misses per method invocation stays the same (or even goes up). Within the limits of cache capacity, the prefetching effect increases with larger working sets. Third, in algorithms with fine-grain locks, speculation may reduce the number of locks that must be acquired, and locks are quite expensive on many machines. We will return to
these issues in more detail in Section 3.5.

3.4.2 How do we validate?

Validation is the most challenging and flexible part of CSpec programming. Most STM systems validate after every shared-memory load, to guarantee opacity (mutual consistency of everything read so far) [45]. Heuristics such as a global commit counter [109] or per-location timestamps [33, 100] may allow many validation operations to complete in constant time, but the worst-case cost is typically linear in the number of shared locations read so far. (Also: per-location timestamps aren’t privatization safe [76].) As an alternative to opacity, an STM system may sandbox inconsistent transactions by performing validation immediately before any “dangerous” instruction, rather than after every load [27], but for safety in the general case, a very large number of validations may still be required.

Using the three validate.* directives provided by compiler-aided manual speculation, we can exploit data-structure-specific programmer knowledge to minimize both the number of validations and their cost. Determining when a validation is necessary is a tricky affair; we consider it further in the following subsection. To minimize the cost of individual validations, we can identify at least two broadly useful idioms:

Version Numbers (Timestamps): While STM systems typically associate version numbers with individual objects or ownership records, designers of concurrent data structures know that they can be used at various granularities [14, 54]. Regardless of granularity, the idea is the same: if an update to location l is always preceded by an update to the associated version number, then a reader who verifies that a version number has not changed can be sure that all reads in between were consistent. The validation_ver directive serves this purpose in CSpec.
It is worth emphasizing that while STM systems often conflate version numbers and locks (to minimize the number of metadata updates a writer must perform), versioning and locking serve different purposes and may fruitfully be performed at different granularities. In particular, we have found that the number of locks required to avoid over-serialization of critical sections is sometimes smaller than the number of version numbers required to avoid unnecessary aborts. The Spec-CGL code of Section 3.2, for example, uses a single global lock, but puts a version number on every set. With a significant number of long-running readers (the lower-right graphs in Figures 3.3 and 3.4), fine-grain locking provides little additional throughput at modest thread counts, but a single global version number would be disastrous. For read-mostly workloads (not shown), the effect is even more pronounced: fine-grain locking can actually hurt performance, but fine-grain validation is essential.

**In-place Validation:** In methods with a search component, the “right” spot to look up, insert, or remove an element is self-evident once discovered: how it was discovered is then immaterial—even if it involved an inconsistent view of memory. Mechanisms like “early release” [53] and elastic transactions [40] in STM systems exploit this observation. In manual speculation, we can choose to validate simply by checking the local context. An example appears at line 28 of Figure 3.5, where `pnext→set` and `pnext→prev` are checked to ensure that the two key nodes are still in the same set, and adjacent to one another. When it can be used, in-place validation has low overhead, a low chance of aborts, and zero additional space overhead. In CSpec, it is realized as value-based validation (`validate_val`) or condition-based validation (`validate_cond`).
3.4.3 What can go wrong and how do we handle it?

In general, our approach to safety is based on sandboxing rather than opacity. It requires that we identify “dangerous” operations and prevent them from doing any harm. Potentially dangerous operations include the use of incorrect data values, incorrect or stale data pointers, and incorrect indirect branches. Incorrect data can lead to faults (e.g., divide-by-zero) or to control-flow decisions that head into an infinite loop or down the wrong code path. Incorrect data pointers can lead to additional faults or, in the case of stores, to the accidental update of nonspeculative data. Incorrect indirect branches (e.g., through a function pointer or the vtable of a dynamically chosen object) may lead to arbitrary (incorrect) code.

An STM compiler, lacking programmer knowledge, must be prepared to validate before every dangerous instruction—or at least before those that operate on values “tainted” by speculative access to shared data [25]. In a few cases (e.g., prior to a division instruction or an array access) the compiler may be able to perform a value-based sanity check that delays the need for validation. In CSpec, by contrast, we can be much more aggressive about reasoning that the “bad cases” can never arise (e.g., based on understanding of the possible range of values stored to shared locations by other threads). We can also employ sanity checks more often, if these are cheaper than validation. Both optimizations may be facilitated by using a type-preserving allocator, which ensures that deallocated memory is never reused for something of a different type [82].

3.5 Additional Case Studies

This section outlines the use of CSpec in three additional concurrent data structures, and summarizes performance results.
3.5.1 Cuckoo Hash Table

Cuckoo hashing [91] is an open-addressed hashing scheme that uses multiple hash functions to reduce the frequency of collisions. With two functions, each key has two hash values and thus two possible bucket locations. To insert a new element, we examine both possible slots. If both are already occupied, one of the prior elements is displaced and then relocated into its alternative slot. This process repeats until a free slot is found.

Concurrent cuckoo hashing was proposed by Herlihy and Shavit [51]. It splits the single table in two, with each having its own hash function. In addition, each table becomes an array of probe sets instead of elements. A probe set is used to store elements with the same hash value. To guarantee constant time operation, the number of elements in a probe set is limited to a small constant \textit{CAPACITY}. One variant of the data structure (a \textit{striped cuckoo hash table}) uses a constant number of locks, and the number of buckets covered by a lock increases if the table is resized. In an alternative variant, (a \textit{refinable cuckoo hash table}) the number of locks increases with resizing, so that each probe set retains an exclusive lock. The refinable variant avoids unnecessary serialization, but its lock protocol is much more complex.

Since an element \textit{E} may appear in either of two probe sets—call them \textit{A} and \textit{B}—an atomic operation in the concurrent cuckoo hash table has to hold two locks simultaneously. Specifically, when performing a \textit{lookup} or \textit{remove}, the locks for both \textit{A} and \textit{B} are acquired before entering the critical section. In the critical section of the \textit{insert} method, if both \textit{A} and \textit{B} have already reached \textit{CAPACITY}, then a resize operation must be done. Otherwise, \textit{E} is inserted into one probe set. If that set contains more than \textit{THRESHOLD} < \textit{CAPACITY} elements, then after the critical section, elements will be relocated to their alternative probe sets to keep the set’s size below \textit{THRESHOLD}. 
```cpp
bool CuckooHashMap::Remove(const KeyT& key) {
    std::pair<Lock*, Lock*> lcks = map_locks(key)
    #pragma spec
    lock(lcks.first, lcks.second);
    Bucket &setA = bucket0(key), &setB = bucket1(key)
    #pragma spec set_check_point(1)
    #pragma spec validate_ver(this→ver)
    int idx = search(setA, key);
    if (idx≥0) { // 1 key is in setA
        #pragma spec consume_lock(lcks.first)
        #pragma spec validate_ver(setA.ver, 1)
        setA.remove(idx);
        unlock(lcks.first, lcks.second);
    } else if ((idx = search(setB, key)) ≥0) { // 2 key is in setB
        #pragma spec consume_lock(lcks.first, lcks.second)
        #pragma spec validate_ver(setB.ver, 1)
        setB.remove(idx);
        unlock(lcks.first, lcks.second);
    } else { // 3 key is not found
        #pragma spec validate_ver(setA.ver, setB.ver, 1)
        #pragma spec validate_ver(this→ver)
        unlock(lcks.first, lcks.second);
    }
    return idx≥0;
}
```

**Figure 3.7:** CSpec tagged Remove method of cuckoo hash table.

**Speculation:** Speculation makes lookup obstruction-free. We place consume_locks only before a modification to a probe set. This moves the presence/absence check in insert/remove out of the critical section. We illustrate the CSpec implementation of remove in Figure 3.7. If the element to remove is speculatively found in probe set A, remove needs to consume only A’s lock instead of both A’s and B’s (case 1).

**Validation:** A version number is added to each probe set to enable validate_ver. In remove (and similarly in lookup and insert), to validate the presence of an element in a set (cases 1, 2), we only need to validate that set’s version (lines 11,
16) after its lock is consumed. We don’t do any validation in the search method, because linear search in a limited-sized (< \text{CAPACITY}) probe set will terminate in \text{CAPACITY} steps regardless of any change in set elements. To validate the absence of an element (case 3), both probe sets’ versions should be checked (line 20). Though the two sets may be checked at different times, their version numbers ensure that the two histories (in each of which the element is not in the corresponding set) overlap, so there exists a linearization point [55] in the overlapped region when the element was in neither set. To support concurrent \text{resize}, a resize version number is associated with the whole data structure. At the checkpoint, that number is validated (line 7) to detect a \text{resize} which breaks the mapping between \text{key} and probe sets (line 5).

**Performance:** Our experiments (Figures 3.8 and 3.9) employ a direct (by-hand) C++ translation of the Java code given by Herlihy and Shavit[51]. We use a \text{CAPACITY} of 8 and a \text{THRESHOLD} of 4; this means a probe set usually holds no more than 4 elements. We ran our tests with two different data set sizes: the smaller (~500 elements) can fit completely in the shared on-chip cache of either machine; the larger (~200K elements) is cache averse. For the striped version of the table, we use 128 locks. For the refinable version, the number of locks grows to 64K. Before each throughput test, a warm-up phase inserts an appropriate number of elements into the table, randomly selected from a double-sized range (e.g., [0, 1000) for small tables). Keys used in the timing test are randomly selected from the same range.

For striped tables with 128 locks, CSpec code is 10%–20% faster than the baseline with 64 threads on the Niagara 2, and even better with 120 threads. The gap is significantly larger on the Xeon. Scalability in the baseline suffers from lock conflicts with increasing numbers of threads. CSpec overcomes this problem with fine-grain speculation, a shorter critical path, and fewer lock operations.
Figure 3.8: Throughput of cuckoo hash table on Intel Xeon E5649, for different data-set sizes and method ratios. The 128-* curves use striped locking; the 26k-* curves are refinable. *-MSpec are pure manual speculation implementations.

For the same reason, CSpec is also useful for refinable tables in all configurations ("*-CSpec" vs "*-base"). For small data sets (upper graphs in Figures 3.8 and 3.9), refinable locking offers no advantage: there are only 128 buckets. For large data sets (lower graphs), baseline refinable tables ("64k-base") outperform baseline striped tables ("128-base") as expected. Surprisingly, CSpec striped tables ("128-CSpec") outperform baseline refinable tables and are comparable to CSpec refinable tables ("64k-CSpec"), because the larger lock tables induce additional cache misses (compare the four "B-*" bars in Figure 3.10b).

This example clearly shows that fine-grained locking is not necessarily best. The extra time spent to design, implement and debug a fine-grained locking algorithm does not always yield superior performance. Sometimes, a simpler coarse-
grained algorithm with CSpec can be a better choice.

A best-effort manual speculation version (MSpec) is also included in our experiments. For easy rollback, a critical section in MSpec is divided as a monotonic speculative phase and a nonspeculative phase by reorganizing the code. Also, when a speculation fails, only one probe set has been changed in most cases, and we can skip the unchanged set in the next try. MSpec code, which adds 15% more lines to the baseline code, is considerably more complex than CSpec code. However, CSpec’s performance matches MSpec’s on Xeon (“*-CSpec” vs “*-MSpec”), and is even faster than the latter on Niagara 2, due to its simpler control flow and fewer instructions.

Figure 3.9: Throughput of cuckoo hash table on Niagara 2.
3.5.2 $B^{\text{link}}$-tree

$B^{\text{link}}$-trees [67, 104] are a concurrent enhancement of $B^+$-trees, an ordered data structure widely used in database and file systems. The main difference between a $B^+$-tree and a $B^{\text{link}}$-tree is the addition of two fields in each node: a *high key* representing the largest key among this node and its descendants, and a *right pointer* linking the node to its immediate right sibling. A node’s high key is always smaller than any key of the right sibling or its descendants, allowing fast determination of a node’s key range. The right pointer facilitates concurrent operations.

The original disk-based implementation of a $B^{\text{link}}$-tree uses the atomicity of file operations to avoid the need for locking. Srinivasan and Carey describe an in-memory version with a reader-writer lock in every node [112]. To perform a lookup, a reader descends from the root to a leaf node, then checks the node’s high key to see if the desired key is in that node’s key range. If not (in the case that the node has been split by another writer), the reader follows right pointers until an appropriate leaf is found. During this process, the reader holds only one
reader lock at a time. When moving to the next node, it releases the previous node’s lock before acquiring the new one. In an insert/remove operation, a writer acts like a reader to locate a correct leaf node, then releases that leaf’s reader lock and acquires the same leaf’s writer lock. Because a node split may occur during the lock switch, the writer starts another round of search for the proper leaf using writer locks.

A full node $A$ is split in three steps. First, a new node $B$ is allocated with its right pointer linking to $A$’s right sibling, and half the elements from $A$ are moved to $B$. Second, $A$’s right pointer is redirected to $B$, and $A$’s writer lock is released. Third, $A$’s new high key is inserted into its parent node. For simplicity, we employ the remove algorithm of Lehman and Yao [67], which does not merge underflowed leaf nodes; this means there is no node deallocation in our code.

**Speculation:** The $B^{link}$-tree algorithm already uses fine-grained locking. Its lookup, insert and remove operations contain two kinds of critical sections: (1) Critical sections protected by reader locks check a node’s key range for a potential right move, or search for a key within a node. Not all of them can be transformed using CSpec, because the movement from one node to its right sibling or its child needs to hold two reader locks. CSpec cannot translate the overlapped critical section formed by the two locks. So, we manually convert the move step to speculation. (2) Critical sections protected by writer locks perform actual insertion and removal. If a node is full, a split occurs in the critical section. CSpec is able to handle these critical sections. The result of the transformation is that lookup becomes entirely speculative, and insert and remove start with a speculative search to check the presence/absence of the key to be inserted/removed. By performing searches in speculative mode, speculation eliminates the need for reader-writer locks. Simpler and cheaper mutex locks suffice for updates, and lookup operations become nonblocking.
Validation: Validation in a $\text{B}^{\text{link}}$-tree is relatively easy. Every speculation works on a single node, to which we add a version number. If (type-preserving) node deallocation were added to remove, we would use one bit of the version number to indicate whether the corresponding node is in use. By setting the bit, deallocation would force any in-progress speculation to fail its validation and go back to the saved parent (not the beginning of the method) to retry.

Performance: Figure 3.11 compares the original and CSpec versions of $\text{B}^{\text{link}}$-tree on the Xeon machine. Results on the Niagara 2 are qualitatively similar. The locking code uses a simple, fair reader-writer lock [78]. To avoid experimental bias, the CSpec code uses the same lock’s writer side. Each node contains a maximum of 32 keys in both algorithms, and occupies about 4 cache lines. To
avoid a performance bottleneck at the top of the tree, the original algorithm uses speculation at the root node (only).

We ran the code with two different sizes of trees and two different mixes of methods. Small trees (10K elements) are more cache friendly than larger trees (1M elements), but suffer higher contention. The 90% lookup, 5% insert and 5% remove method mix simulates read-dominated workloads, and 0%:50%:50% simulates write-dominated workloads. As in the cuckoo hash experiments, we warm up all trees before beginning timing.

CSpec provides both greater throughput and greater scalability in all experiments, even with speculation at the root node in baseline locking code. CSpec scales very well even when running across chips (>12 threads). Comparing the left-hand and right-hand graphs in Figure 3.11, we can see that the advantage of speculation increases with higher contention (smaller trees). In separate experiments (not shown) we began with originally-empty trees, and ran until they reached a given size. This, too, increased the advantage of CSpec, as the larger number of node splits led to disproportionately longer critical sections in the baseline runs.

Figure 3.12 presents hardware profiling results to help understand the behavior of speculation. Clearly, the critical path is shorter in MSpec code. As a consequence, lock contention is significantly reduced.¹

The trees in the above experiment are stable because of the almost equal numbers of inserts and removes. In another typical category of workloads, trees grows gradually as keys are frequently added but occasionally removed. In such trees, internal nodes are more likely to be changed, and node splits occur quite often.

¹We modified the OS kernel so that hardware performance counters could be read in user mode with low cost. Due to profiling overhead (20-30% in B\text{link}-tree), the throughput of profiled code (Figure 3.12) does not exactly match the un-profiled case (Figure 3.11).
Figure 3.12: Hardware profiling of CPU cycles and L2 load misses per operation for “50% Insert/Remove” on Intel Xeon.

Figure 3.13: Throughput on workloads with growing trees on Niagara 2.

Figure 3.14: Throughput on workloads with growing trees on Intel Xeon E5649.

To capture this sort of workload, we start with an empty tree and two operation ratios (70%:20%:10% and 90%:9%:1%). Figure 3.13 shows the results, in which
the margin between MSpec trees and non-MSpec trees is more significant. As contents of nodes are frequently modified, non-MSpec trees consume more time in critical sections moving elements and splitting nodes, especially in the top few levels of the tree. MSpec trees replace such waiting with speculation, and scale well especially when there is no inter-chip communication.

3.5.3 Bitmap Allocator

Bitmaps are widely used in memory management [126] and file systems. They are very space efficient: only one bit is required to indicate the use of a resource. A bitmap allocator may use a single flat bitmap or a hierarchical collection of bitmaps of different sizes. We consider the simplest case, where a next-fit algorithm is used to search for available slots. In the baseline algorithm, an array of locks protects the structure (one lock protects one segment of the bitmap). The data structure supports concurrent allocate and deallocate operations. To find the desired number of adjacent bits, allocate performs a linear search in its critical section, then sets all the bits to indicate they are used. Deallocate takes a starting position and size as parameters, and resets the corresponding bits in its critical section.

Speculation: Most of execution time is spent searching the bitmap for free bits. Since only one lock in the lock array is held at a time, we simply place a consume_lock directive before flipping the free bits.

Validation: After the lock is consumed, a validate_cond checks that the bits found during speculation are still available. No safety issues arise, as the bitmap always exists.
Performance: Our experiments (Figure 3.15) employ an array of 256K bits with a time-limited scenario in which a 70% allocate / 30% deallocate mix consumes the bitmap gradually so it gets harder and harder to find a free slot. Allocation requests have the distribution 20% 1 bit, 30% 2 bits, 30% 4 bits, 15% 8 bits, and 5% 16 bits. Two lock array sizes, 1 (global lock) and 32, are used.

On Niagara 2, CSpec-generated code (“*-CSpec”) is significantly faster than the baseline as a result of a much shorter critical section in Allocate. Again, we see that coarse-grained locking with speculation (“1-CSpec”) beats nonspeculative finer-grained locking (“32-base”). However, the benefit of CSpec is more modest on the Xeon machine. This is because the Xeon CPU can execute bit operations much faster than the simpler cores of the Niagara 2, leaving less work available to be moved to speculation. We also test a manual speculative version (not shown). Surprisingly, it is slightly slower than the CSpec version because its reorganized code contains more branch instructions. On both machines, the STM implementation is only faster than the single-lock baseline.
Table 3.1: A summary of the application of CSpec. CG/FG = coarse-/fine-grained.

### 3.6 Summary

While fully automatic speculation, as provided by transactional memory, has the advantage of simplicity, we believe that manual speculation has a valuable role to play, particularly in the construction of concurrent data structure libraries. In support of this contention, we have presented four different structures—equivalence sets, a B^{link}-tree, a cuckoo hash table, and a bitmap allocator—in which speculation yields significant performance improvements. The principal challenges in their construction, as suggested in Section 3.4, were to identify the work that could profitably be moved to speculation, and to determine how and when to validate. To eliminate other, more mechanical challenges, we developed a set of compiler directives and a translation algorithm that partitions critical sections; identifies covering lock sets; and automates checkpointing, rollback, and the access tagging required to avoid data races and preserve sequential consistency.

Table 3.1 summarizes our four example data structures, comparing the baseline locking policies, the number of CSpec regions, the total number of added directives (excluding `#pragma spec`), and other changes to the source code. It clearly supports the claim that speculation can easily be added to existing lock-based code, with a small number of CSpec directives and few or no additional adjustments.

In the following chapter, we extend of the idea of manual speculation to atomic
blocks, and explore its application on hardware transactions memory.
4 Partitioned Transactions

4.1 Introduction

All existing HTMs—and all that are likely to emerge in the near future—are best effort implementations: in the general case, a transaction may abort and retry not only because of an actual data conflict with some concurrent transaction, but also because of various hardware limitations—notably, the size or associativity of the space used to buffer speculative reads and writes.

One of the most appealing aspects of transactional memory—and a major advantage over locks—is the ability to compose larger transactions out of smaller ones without sacrificing performance or risking deadlock. As TM becomes more widely used, we can expect that many transactions will incorporate smaller, pre-existing operations (typically library calls). Unfortunately, because of their increased size, composed transactions tend to be precisely the ones that place the highest pressure on hardware TM limits [131].

Consider the example transaction in Figure 4.1, which includes query and update operations on two separate data structures, with data dependences and nonlinear control flow among these operations. This code may have difficulty completing as a pure hardware transaction: its spatial footprint may be too large.
atomic { // users: a shared hashtable; accounts: a shared rb-tree
User* u = 0;
if ((u = htableFind(users, streetAddress) != 0) {
    for (int i=0; i<u→numAccounts; i++) {
        Account *acct = rbtreeFind(accounts, u→accts[i]);
        sum += acct→val;
        acct→lastAccessedTime = timeStamp;
    }
} else {
    u = (User*)malloc(sizeof(User));
    htableInsert(users, streetAddress, u);
    initUser(u);
}
}

Figure 4.1: A big transaction for account maintenance.

for hardware buffers, and its temporal duration may increase the likelihood of conflict aborts, particularly given that conflict in any constituent operation will abort the entire transaction.

One way to improve the odds of completion for large transactions on best-effort HTM is to pull read-only preliminary (“planning”) work out of the transaction, thereby reducing both the memory footprint and the temporal duration of the work that remains. Afek et al. [2] explored this approach in the context of software TM; they call it “consistency oblivious programming,” because the work that is removed from the transaction must be able to tolerate an inconsistent view of memory. We pursued a similar partitioning of lock-based critical sections, to shorten critical path length [127, 129]; we subsequently suggested that partitioning might reduce the abort rates of hardware transactions [128], a suggestion that was echoed by Avni and Kuszmaul [5]. In all this work, the key observation is that a transaction (or critical section) can often check the continued validity of a plan computed in advance more quickly—and with fewer memory references—than it could regenerate the plan from scratch.
Unfortunately, while the partitioning of small transactional operations is often straightforward (at least for the experts who write libraries), its naive application is incompatible with composability. In Figure 4.1, for example, suppose we have access to a partitioned version of the `rbtreeFind` operation. We cannot simply take the planning part of the lookup at line 5 and hoist it out of the parent transaction without knowing what was (will be?) returned by the hash table lookup at line 3. Avni and Suissa have suggested [6] that one suspend an active transaction and perform planning in non-speculative mode, but while this may reduce the transaction’s spatial footprint, it does nothing for temporal duration. The suspension mechanism, moreover, is supported on only one current hardware architecture [17], where it is quite expensive.

We assume that the partitioning of operations like `rbtreeFind` will continue to be performed by hand. Our contribution is to automate composition, with no need for special hardware. We call our approach partitioned transactions (ParT). For purposes of exposition, we distinguish between operations, which are partitioned by hand, and transactions, which are partitioned automatically.

Given a library of partitioned operations, each of which follows a set of well-defined structural and semantic rules, we allow the programmer to write—and compose—transactions that call these operations in their original unpartitioned form. Automatic compiler support then extracts the planning portions of the embedded operations and hoists them out of the outermost parent transaction, along with sufficient “glue code” to preserve all inter-operation control and data dependences.

We call the halves of a partitioned operation its planning operation and its completion operation. The completion operation is always performed by (or subsumed in) a hardware transaction; the planning operation may be a hardware transaction, or it may run in ordinary code. The halves of a partitioned transaction are its planning phase and its completion phase. The completion phase is
always a hardware transaction; the planning phase runs in ordinary code, possibly peppered with smaller hardware transactions used by planning operations. A summary of the plan for each individual operation (including expected return values and side effects) is carried through the planning phase and into the completion phase, automatically, by means of validator objects held in thread-local storage.

Returning to Figure 4.1, suppose we have a partitioned implementation of htableInsert. Planning operation htableInsertP figures out where to insert a key in the table and saves this position in an appropriate validator object, hidden in thread-local storage. Completion operation htableInsertC accesses V and validates its plan. It then either performs any necessary updates (if the plan is still valid) or performs the whole operation from the beginning (if the plan is no longer valid). Given similar partitioned implementations of rbtreeFind, htableFind, and malloc, our compiler will generate code along the lines of Figure 4.2. The planning phase (lines 1–13) tries to call as many planning operations as possible (note that initUser has been removed). Unlike regular code, which must run to completion, the planning phase may safely stop at any time—e.g., because it encounters inconsistent data, or because the savings from further planning is unlikely to be worth the cost.

The completion phase (lines 15–28) is almost identical to the original transaction, except that completion operations are called instead of the original unpartitioned versions. So where is the magic? Inside the partitioned operations and in the validator objects that carry information from each planning operation to the corresponding completion operation (and to any subsequent planning operations that may need to see its side effects).

Assuming our code is for an outermost transaction, the completion phase may abort for any of the usual reasons, at which point there are several options. If we choose to repeat the completion phase, we can shortcut any completion operations
Figure 4.2: A conceptual partitioning for the transaction of Figure 4.1.

whose previous plans still validate successfully. As a last resort, a transaction that aborts repeatedly can—as in current run-time systems for HTM—retry with a global lock.

ParT provides several simultaneous benefits to hardware transactions. By shrinking memory footprint, partitioning reduces the likelihood that hardware buffering limits will be exceeded. By shrinking both footprint and duration, it reduces the likelihood of conflicts with concurrent transactions. By re-executing only those planning operations whose plans are no longer valid, it achieves the
effect of partial rollback when the completion phase aborts. Finally, to the extent that the planning and completion phases access common locations, the former serves to warm up the cache for the latter, further reducing the duration of the completion transaction. For locations that are to be modified, the planning phase may even choose to prefetch in exclusive mode.

4.2 ParT Execution Model

Given the non-atomicity of planning, a partitioned transaction needs to satisfy two basic requirements to be equivalent to the original transaction: (1) the planning phase should be logically side-effect free: its execution should be invisible to other threads, it should impact the completion phase only through the precomputation of useful hints, and it should never exhibit erroneous or undefined behavior; (2) the completion phase, given a correct planning phase, should produce the same results as the original transaction. This section formalizes these requirements, and presents the rules that must be followed when writing partitioned operations. Note that programmers who simply use partitioned operations do not have to learn these rules.

4.2.1 Histories

Following standard practice, we model a computation as a history of events, including reads, writes, and transaction begin (\texttt{tbegin}) and end (\texttt{tend}). We assume that histories are well formed and that the TM implementation is correct—in particular, that \texttt{tbegin} and \texttt{tend} events are paired and properly nested in each thread subhistory, and all the events of any outermost transaction are contiguous in the overall history (that is, reflecting atomicity, the events of a transaction do not interleave with events in other threads). For simplicity, we assume a static partition between thread-local and shared data (the model could easily be extended to
accommodate privatization and publication if desired). We ignore events within aborted transactions, since they have no visible effect. We also ignore reads and writes of transaction-local state, since these have no impact outside the transaction.

For the sake of the formalism, we assume the availability of per-thread checkpoint and restore primitives, to modify the behavior of both shared and thread-local memory. (In our compiler, these are implemented using a cache of recent changes, which is consulted before each access in the planning phase, and discarded at the end.) In a well-formed history, checkpoint and restore occur in un-nested pairs, outside of any transaction. Their purpose is to change the mapping from reads to writes. Specifically, if a read of location $l$ in thread $t$ occurs within a checkpoint-restore pair in $t$’s subhistory, the read will return the most recent write to $l$ by $t$ within the checkpoint-restore pair; if there is no such write, the read will return the most recent write to $l$ in the overall history. Finally, we assume the availability of per-thread scratchpad memory, which is not affected by checkpoint and restore.

### 4.2.2 Partitioned Operations

A contiguous subsequence of the events within a transaction may correspond to an operation (the invocation of a method) on some abstract object $O$. As suggested informally in Section 4.1, we allow the library programmer to rewrite a method $m$ of $O$ as a planning method $m^p$ and a completion method $m^c$. We impose a variety of restrictions on this rewriting.

1. **Interface.** Methods $m^p$ and $m^c$ must take the same arguments, and produce the same type of result(s), as the original (“ordinary”) method $m$. All three must be written in a strict object-oriented style: they must touch no memory other than their own parameters and the state of their common object.
Methods $m^p$ and $m^c$ may, however, read and write scratchpad memory, which is not accessed outside of operations.

2. **Behavior.** Method $m^c$, when invoked on a given abstract state of $O$, must produce the same return value(s), and must make the same abstract changes to $O$’s state, as $m$ would have made. Hopefully, $m^c$ will execute more quickly than $m$, utilizing information gathered by a previous execution of $m^p$.

We assume, in any well-formed history, that $m$ is called only inside of transactions. Our compiler will ensure this for $m^c$ as well; thus, both $m$ and $m^c$ can comprise straightforward sequential code. By contrast, we expect calls to $m^p$ to occur outside of any program-level transaction; therefore we have extra restrictions on its behavior:

3. **Safety.** Method $m^p$ must:

   (a) make no change to the abstract state of $O$.

   (b) be written in such a way that its execution will linearize with arbitrary concurrent executions of original, completion, and planning methods of the same object. The simplest way to ensure this linearizability is to place $m^p$ within its own small hardware transaction; alternatively, the code may be written in a nonblocking or hybrid style.

   (c) “see” the object state corresponding to recent planning operations in the same thread. More precisely, an execution of $m^p$, performed by thread $t$, must produce the same return value(s) that $m$ would have produced, if every planning operation performed by $t$ since the last ordinary or completion operation had been replaced with the corresponding ordinary operation. In practice, this implies that $m^p$ must memoize (in scratchpad memory) the changes to $O$’s abstract state that would have been performed by $m$. 
(d) export, to the surrounding planning phase, sufficient information to
determine whether the value returned by \( m^p \) is still valid. This infor-
mation can be conservative: “no” is always an acceptable answer.

4.2.3 Partitioned Transactions

Consider a history \( H \) containing a transaction \( T \). Suppose that for every operation \( o_i \) in some prefix of \( T \), we have available partitioned implementations \( o_i^p \) and \( o_i^c \). Suppose further that we are able to replace the sequence

\[
\text{tbegin } s_0 \ o_1 \ s_1 \ o_2 \ s_2 \ldots \ o_k \ s_k \ldots
\]

where the \( s_i \) are sequences of non-method-call events, with

\[
\text{checkpoint } s_0 \ o_1^p \ s_1 \ o_2^p \ s_2 \ldots \ o_k^p \ s_k \text{ restore}
\]

\[
\text{tbegin } s_0 \ o_1^c \ s_1 \ o_2^c \ s_2 \ldots \ o_k^c \ s_k \ldots
\]

The first line of this new sequence is the planning phase; the second is (a prefix of) the completion phase. If the two sequences remain jointly contiguous in \( H \), we claim they will have the same behavior as the sequence they replaced. The proof is straightforward: the \( o_i^p \) operations have no impact on the abstract states of their objects; the \( o_i^c \) operations have the same impact as the \( o_i \) operations they replace; and any impacts of the \( o_i^p \) operations or \( s_i \) sequences on thread-local or (other) shared state except those saved in the scratchpad memory are invisible to other threads, and will be reversed by the \text{restore} operation.

Consistency   In practice, of course, the new planning phase has no guarantee of contiguity: it no longer lies within a hardware transaction. Unlike conventional transactions, which require full data consistency, the planning phase of a partitioned transaction needs only \text{partial} consistency: only the locations exported
under rule 3d (together with locations read by the $s_i$) must remain consistent between checkpoint and restore. Planning phases also differ from conventional transactions in the way they handle conflicts: a transaction will retry until it commits, but a planning phase can simply stop, by calling restore and moving immediately to the completion phase. If a conflicting operation in another thread has intervened—say after $o_j^p$ ($j < k$) in our example—we can safely truncate the planning phase as of the end of the consistent prefix:

```
checkpoint $s_0$ $o_1^p$ $s_1$ $o_2^p$ $s_2$ ... $o_j^p$ restore
```

Assuming there was no earlier conflict before $o_j^p$, this truncated phase is perfectly legal. As we shall see in Section 4.4, our compiler algorithm arranges for appropriate truncation by generating code (assisted by planning methods and their validators) to double-check consistency after every shared-memory read or planning operation.

**Dependences**  Rule 3c specifies that if a transaction includes multiple operations on the same shared object, each successive operation must see the changes envisioned by its predecessors. The need for this rule can be seen in the transaction of Figure 4.3a. Line 5 should never be executed, because line 4 always returns `false`. In the planning phase of the ParT transaction (Figure 4.3b), if the dependence were not respected (i.e., if `setA.contains_P` were unable to see the results of `setA.insert_P`), line 10 would return `true` and behave incorrectly.

## 4.3 Toward A Library of Partitioned Operations

Building on the framework of Section 4.2, we now turn to the practical details of writing partitioned operations. These will in turn support the compiler framework of Section 4.4. As noted in Section 4.1, previous projects have explored how to
create partitioned operations [2, 5, 127–129]. We review the idea here, casting it in a form amenable to our composition efforts.

As a general rule, it makes sense to partition an operation if much of its work can be off-loaded to the planning operation, and if validation of the plan is comparatively cheap. Perhaps the most obvious examples fit a search-then-modify pattern, where the search is time consuming (and may have a large memory footprint), but the continued validity of a found location can quickly be checked via local inspection. Other examples fit a compute-then-modify pattern: in transactions containing some variant of \( y = \text{expensive.pure.function}(x) \), the expensive function can be precomputed in a planning operation; later, if the value of \( x \) has not changed, the completion operation can use the precomputed \( y \). Additional examples appear in Section 4.3.3.

### 4.3.1 The Basic Partitioning Template

To partition method `foo` of a concurrent object, the library designer creates two new functions `foo_P` and `foo_C`. Both should have the same argument types and return type as `foo`. Function `foo_C` will always be called within an `atomic` block, and, like `foo`, may be written without regard to synchronization. For `foo_P`, which will be called outside the main transaction, the library designer must devise a linearizable nonblocking implementation. Transactions can of course be used to...
simplify this task. In the limit, the entire planning operation can be placed in a transaction.

Informally, foo_P “figures out what to do” (and what it depends upon) and embeds this plan in a validator object, which may be accessed by foo_C. As noted in rule 3c of Section 4.2.2, the validator also serves to carry dependences from one planning operation (of a given object O) to the next—it captures the (abstract) changes that would have been made to O if foo had been called directly.

In Chapter 3, we explored several specific ways to build a validator. The details are up to the library designer: our composition mechanism (Section 4.4) does not access validators directly. To assist designers, we provide what amounts to a simple key-value store in thread-local memory. A planning operation and its corresponding completion operation need only agree on a key (often the id of their mutual object) in order to pass information between them. Multiple operations on the same object will typically share a validator.

If validation succeeds (and the planning phase was consistent), foo_C can execute “fast path” code that effects the changes to foo’s object captured by the validator’s plan. Otherwise, it must abandon the plan and switch to fallback code, which is commonly the original operation foo. It is worth emphasizing that planning operations and validators are simply optimizations: in the degenerate case, we can always fall back on the original code.

The consistency requirement of rule 3d can be realized by adding to the read set of the surrounding planning phase enough locations to guarantee that if the operation’s return value is no longer valid, at least one of the added locations will have changed. We provide library designers with a Part_readset_put operation that serves this purpose. Note that successful validation is a weaker requirement than consistency: in the wake of a consistency violation that stops a planning phase, a validator may still indicate that foo_P’s plan is still usable. As a simple example, insertion at a remembered location a sorted linked list may still be
appropriate even if the values of the adjacent elements have changed, so long as the to-be-inserted element has an intermediate value. In principle, one could use validators instead of exported read sets to determine when to stop planning. We decided against this option because it would increase the cost of run-time instrumentation.

4.3.2 Extended Example: Red-black Tree

A red-black balanced tree supports log-time insert, remove, and find operations with an attractively small constant factor. Each operation begins with a tree search, which can be moved into an explicitly speculative planning operation [5, 128]. To make this planning amenable to composition, we must structure it according to rules 1–3.

The code skeleton for partitioned insert is shown in Figure 4.5 (lines 28–55), as it would be written by a library designer. To ensure correct execution, we use a type-preserving allocator [82] for nodes, and increment a node’s version number, ver, when the node is deallocated, reallocated, or modified. Like the structure described by Avni and Kuszmaul [5], the tree is threaded with predecessor (pred) and successor (succ) pointers.

The internal lookup operation serves to initialize a validator. To support arbitrary key types and to avoid the possibility that a speculative search will find itself on the wrong branch of the tree due to a rotation, we call lookup within a small transaction, delimited by ParT_plan_hmt_begin and ParT_plan_hmt_end (Figure 4.4). These primitives differ from the usual htm_begin and htm_end in that repeated aborts cause fallback not to a software lock, but to a ParT_plan_stop routine (corresponding to the restore primitive of Section 4.2) that truncates the planning operation and any planning phase in which it appears, transferring control immediately to the completion operation or phase. All rb-tree validators
Figure 4.4: Pseudocode of auxiliary functions in ParT library.

are kept in thread-local storage, indexed by the address of the tree and a key (line 31). To minimize allocation cost, a fixed number of these validators are pre-allocated; if the supply is exhausted, newRBValidator will call ParT_plan_stop internally. If a desired key doesn’t exist, memory space is reserved for a new node (line 40), which will be claimed in the completion phase (line 49), again using thread-local storage.

A version-based implementation of the validator is shown in lines 1–10. During planning, we initialize it with the last accessed node (curr), a snapshot of its version number (ver), and an indication as to whether the desired key was found (exist). In a subsequent completion operation (e.g., RBTree::insert_C, which is called inside the completion phase of the parent transaction), the isValid method will succeed only if (1) the key has been searched for by at least one planning operation (curr!=0), and (2) curr has not subsequently been modified (curr→ver==version). Note that isValid will never return a false positive. It may return a false negative, but this will simply trigger execution of the fallback path (line 54).
struct RBValidator {
    // last accessed node
    RBTree::Node *curr;
    // snapshot of curr's version
    int ver;
    bool exist;
    bool localExist;

    bool isValid() {
        return curr && curr->ver == ver;
    }
};

void RBTree::lookup(RBValidator *v, KeyT &k) {
    // root is a dummy node
    Node *p = root->right;
    v->curr = root;
    v->ver = root->ver;
    while (p) {
        v->curr = p;
        v->ver = p->ver;
        if (k == p->key) {
            v->exist = true;
            break;
        }
        p = k < p->key ? p->left : p->right;
    }
}

#pragma plan for RBTree::insert
bool RBTree::insert_P(KeyT &k) {
    ParT_plan_hm_begin();
    ParT_readset_validate();
    RBValidator *v = getRBValidator(this, k);
    if (v == 0) { // first access to (this, k)
        v = newRBValidator(this, k);
        lookup(v, k);
        ParT_readset_put(&v->curr->ver);
    }
    ParT_plan_hm_end();
    bool ret = !v->localExist;
    v->localExist = true;
    if (!v->exist) allocNode_P();
    return ret;
}

#pragma complete for RBTree::insert
bool RBTree::insert_C(KeyT &k) {
    RBValidator *v = getRBValidator(this, k);
    if (v && v->isValid()) {
        if (!v->exist) {
            Node *n = allocNode_C();
            ... // insert n as v->curr's neighbor and rebalance the tree
        }
    }
    return !v->exist;
}

Figure 4.5: Partitioned insert method (lines 27–55) and validator (lines 1–10) for red-black tree.

It’s easy to see that this partitioning obeys rules 1, 2, and 3a. Since the lookup is done inside a hardware transaction, the linearization point (rule 3b) is at line 37. Meanwhile, the dependence chain through insert_P, remove_P, and find_P (rule 3c) is tracked by the localExist flag of the (key-specific) validator. Assuming setA in Figure 4.3b is a red-black tree, setA.insert_P(5) will set validator(setA, 5).localExist = true at line 39 of Figure 4.5. Therefore, after checking the same flag, setA.contains_P(5) will pretend key 5 was already
in the tree and properly return. Consistency of the parent transaction’s planning phase (rule 3d and Section 4.2.3) is ensured by adding appropriate version numbers to the read set used by that transaction (line 35) and by performing explicit calls to \texttt{ParTreadset.validate} (e.g., line 30) when consistency is needed within a planning operation.

### 4.3.3 Additional Examples

As transactional memory becomes more widely adopted, we envision a library of partitioned operations that can be used, transparently, to reduce transaction duration, memory footprint, and consequent abort rates. In all cases, manually partitioned operations, constructed once by an expert, can be called at arbitrary points within a larger transaction, using the same syntax that would have been used for the unpartitioned operation (as explained in detail in Section 4.4, the compiler hides all details of separating out the planning phase).

**Collections:** Ordered and unordered sets, mappings, and buffers are among the most common shared abstractions. Operations often start with a search component that is easily moved to a planning operation, and verified quickly in the completion operation. In some cases (e.g., the red-black tree of Section 4.3.2), the planning phase will use its own hardware transaction to ensure a consistent view of memory. In other cases (e.g., a sorted linked list), the planning operation can (with a bit of care) be written to tolerate inconsistency, and run without HTM protection.

**Commutative operations:** Operations that commute with one another need not necessarily execute in the serialization order of the transactions in which they appear. Operations such as random number generation, unique id generation, or lookup in a memoization table can be performed entirely in a planning phase; the validator simply encapsulates the result.
Memory allocation: Depending on the choice of allocator, calls to malloc and its relatives may be a significant component of transaction run time and a major source of conflicts. A custom malloc\_P can allocate space for operations during the planning phase. The corresponding malloc\_C confirms that the space was indeed pre-allocated, and simply uses it. An additional “clean up” hook (not described in detail here) may be used at the end of the transaction to reverse any allocations that were mis-speculated, and not actually needed in the completion phase. As we can see, malloc\_C breaks rule 3a, but it’s safe in this case, because malloc can be seen as a special commutative operation whose object has no abstract state. In contrast to naive, manual pre-allocation, which allocates the maximum amount of space that might be needed in any execution of the transaction, our compiler-supported partitioning mechanism will reflect conditional branches within the transaction, avoiding unnecessary memory churn.

Object initialization: Object constructors (initializers) are often called immediately after allocation. A constructor is easily partitioned if—as required by rule 1—it modifies only the state of the object. The planning phase performs all the writes; the completion phase confirms that the constructor arguments have not changed since the planning phase. Writes in a mis-speculated plan are unnecessary, but semantically harmless.

4.4 Automatic Composition

Transactions in real applications often comprise multiple simpler operations. Given a library of partitioned operations written by experts, a programmer could manually transform a simple composition of these operations, like the one in Figure 4.1, into its ParT equivalent for improved performance. Unfortunately, manual composition hurts the readability of source code, and dramatically increases programming complexity. Transactions in real applications may have complex control flow,
making the construction of “glue code” (the $s_i$ sequences of Section 4.2.3, which must log their writes and consult the log on reads) tedious and error-prone. Moreover, when partitioned operations are reached via long chains of function calls, each intermediate call must be cloned for use in planning, completion, and (potentially) nontransactional contexts. Manual management of such cloning would be a code maintenance nightmare. In contrast, our compiler support allows automatic composition: the programmer writes transactions in the conventional way, as in Figure 4.1, and the compiler turns these transactions into partitioned versions like the one we will see in Figure 4.6.

### 4.4.1 Language Interface

To enable automatic composition, we provide the following compiler directives:

- **#pragma part** is placed before the code of an atomic block to instruct the compiler to transform the transaction to its ParT form. In Figure 4.1, the directive would be inserted before line 1, prompting the compiler to automatically create the desired composed partition. Given a ParT library that covers many operations, a TM programmer can easily request partitioning with minimal changes to the source code.

- **#pragma plan_for func** and **#pragma complete_for func** allow a ParT library designer (expert) to link planning and completion methods to an original method with name `func`, so the compiler knows how to partition it. In Figure 4.5, these directives have been used at lines 27 and 44. As required by rule 1, planning and completion methods take the same parameters as the original method.

- **#pragma stop** tells the compiler that planning should stop at this point. This directive allows the programmer to fine-tune performance by precomputing only the initial portion of a transaction.
4.4.2 Automatic Partitioning

Starting with annotated source code, our compiler synthesizes a planning method for each identified top-level atomic block. The algorithm is recursive, so atomic blocks can be nested: every function and atomic block that calls a partitionable operation (and that may itself be called, directly or indirectly, from a to-be-partitioned transaction) is partitioned into planning and completion phases; these can then be called by higher-level functions. The compiler rewrites the original atomic block as the code of the completion phase.

Synthesizing Planning Code

The goal of synthesis is to generate a minimal and safe planning phase that covers as many planning operations as possible. The synthesis algorithm begins by cloning and extracting the code of a ParT transaction as a separate function. It then passes this function to SynthesizePlan (Algorithm 4.1) to generate the composed planning function.

All functions called directly or indirectly from a ParT transaction are categorized as one of the following:

**Partitionable:** These have pre-defined planning functions (identified by `#pragma plan_for`), so there is no need to synthesize them.

**Unsafe:** These include functions with no source code available, library functions (unless specified as partitionable operations), OS APIs, I/O operations, and various other special cases. They preclude the use of planning for the remainder of the transaction.

**Others:** These may require (recursive) synthesis to generate their planning functions. They include the function created for the outermost ParT transaction.
Algorithm 4.1: SynthesizePlan(func, fTable)

Input: a function func, a function information table fTable

1 if func is marked as a partitionable operation then
2     fTable[func].stopBefore ← false;
3     fTable[func].plan ← GetPragmaPlanFor (func);
4 else if func is unsafe then
5     fTable[func].stopBefore ← true;
6     fTable[func].plan ← null;
7 else
8     planFunc ← CloneFunction (func);
9     foreach #pragma stop in planFunc do
10        replace #pragma stop with function call to Part_plan_stop
11     foreach function call C in planFunc do
12        ... // code for checking recursion of func is omitted;
13        f ← C.calledFunction;
14        if f not in fTable then
15           SynthesizePlan(f, fTable);
16        if not fTable[f].stopBefore then
17           replace C with function call to fTable[f].plan;
18        else
19           insert function call to Part_plan_stop before C;
20     PrunePlan (planFunc);
21     InstrumentPlan (planFunc);
22     fTable[func].stopBefore ← false;
23     fTable[func].plan ← planFunc;

An “Other” function will typically comprise two kinds of code: calls to other functions and the skeleton or glue code that connects these function calls.

SynthesizePlan inspects the source code of each “Other” function. It inserts Part_plan_stop before any call to an unsafe operation (line 19) and replaces calls to “Partitionable” and “Other” functions with calls to their planning functions (line 17).

PrunePlan (line 20) reduces the size of the generated planning function. First, any code strictly dominated by a call to Part_plan_stop is removed. Second, if planFunc is a top-level transaction, we perform backward slicing on each call to
a partitioned operation. Instructions that do not belong to any of these slices are removed, leaving only the “glue” instructions necessary to invoke the subsidiary planning functions.

**InstrumentPlan** (line 21) deals with data consistency in the glue code. **Load** and **Store** instructions are instrumented if they may read or write shared memory. The instrumentation redirects them to the read and write logs of the planning phase, and validates all previous reads when a new location is read. More details about data consistency are given in Section 23.

Function **atomic1.P** in Figure 4.6 is the compiler-synthesized planning function for the transaction of Figure 4.1, assuming no partitioned operation inside **initUser**. Lines 6 and 12 of Figure 4.1 have been removed by **PrunePlan**, since they do not belong to any backward slicing of partitionable operation calls. More advanced alias analysis (e.g., data structure analysis[64]) could further remove line 7 of Figure 4.1 (line 8 of Figure 4.6).

Figure 4.7 contains the skeleton of a more extensive compiler partitioning, as required for a transaction with a nested partitioned operation (**op**) and an unsafe call (**unsafe_call**). This example illustrates how difficult it would be to manually partition transactions with nested calls, which are not uncommon in real applications.

**Ensuring Consistency**

As noted in Section 4.2.3, the planning phase of a partitioned operation, which is not automatically atomic, must be designed to ensure its own internal consistency, much like the implementation of an atomic block in a software TM system. Outside planning operations, which are responsible for their own consistency, function **InstrumentPlan** in Algorithm 4.1 employs STM-like instrumentation to buffer reads and writes in thread-private logs. The read log allows us to validate, in the wake of each new read, that all previously read locations still hold their original
```c
void atomic1_P(User* user, const char* strAddress, HashTable* users, RBTree* accounts) {
    User* u = 0;
    if ((u = htableFind_P(users, strAddress) != 0) {
        int tmp0 = ParT_read_32(&u→accountNum);
        for (int i=0; i< tmp0; i++) {
            int tmp1 = ParT_read_32(&u→accts[i]);
            Account* acct = rbtreeFind_P(accounts, tmp1);
            ParT_write_32(&acct→lastAccessedTime, timeStamp);
        }
    } else {
        u = (User*)malloc_P(sizeof(User));
        htableInsert_P(users, strAddress, u);
    }
}

// ParT transaction
if (setjmp(ParTjmpbuf)==0) // planning phase
    atomic1_P(user, strAddress, users, accounts);
ParT_plan_stop();
htm_begin(); // completion phase
User* u = 0;
if ((u = htableFind_C(users, strAddress) != 0) {
    for (int i=0; i<u→accountNum; i++) {
        Account* acct = rbtreeFind_C(accounts, u→accts[i]);
        sum += acct→val;
        acct→lastAccessedTime = timeStamp;
    }
} else {
    u = (User*)malloc_C(sizeof(User));
    htableInsert_C(users, strAddress, u);
    initUser(u, strAddress);
}
htm_end();
```

Figure 4.6: Compiler-generated ParT code (source-level equivalent) for the transaction of Figure 4.1.
void foo() {
    op(); // a ParT op
}
void bar() {
    foo();
    unsafe_call();
    ...
}
void fun() {
    #pragma part
    atomic {
        bar();
    }
}

(a) original code

Figure 4.7: Automatic partitioning of a transaction with nested calls.

own writes.” In contrast to STM, the write log is discarded by ParT.plan_stop when the planning phase naturally ends or an inconsistency occurs; all we need going forward are the validators created by the constituent planning operations.

Synthesizing Completion Code

The composed completion phase is comparatively easy to construct: it is almost the same as the code of the original transaction/function, except that each function call that was replaced by Algorithm 4.1 in the planning phase will be replaced by its completion function in the corresponding position in the completion phase. The generated ParT equivalent of the transaction in Figure 4.1 appears in Figure 4.6 (lines 17–33), where the use of setjmp allows ParT.plan_stop to escape any nested context (line 18–23 of Figure 4.4).

When validation of the plan for operation \( O \) fails within the completion phase of composed transaction \( T \), fallback code will re-execute only operation \( O \), au-
tomatically salvaging everything else in the transaction and continuing the completion phase. If the outer, hardware transaction of the completion phase aborts and retries, the fact that plans are constructed outside the transaction means that we will similarly salvage every plan whose validator still returns true. These properties essentially constitute a partial rollback mechanism, likely resulting in shorter turn-around time and higher throughput than would be available without partitioning.

A simple example can be seen in transactions that end with a reduction (e.g., the update of a global sum). Unlike a monolithic composed transaction, a ParT completion phase that aborts due to conflict on a reduction variable can generally salvage the planning phases of all constituent partitioned operations.

### 4.4.3 Run-time Support and Optimizations

The ParT run-time system manages the execution of composed planning. As described in previous sections, we maintain data consistency by keeping read and write logs for each planning phase, and performing incremental validation each time a new address is read and each time a planning operation requires consistency. The validation is value-based and employs a small hardware transaction, so no ownership records or locks are needed.

As in an STM system, read/write logging and validation impose nontrivial overheads. Read and write logs tend to be considerably smaller than in STM, however. Moreover, as discussed in Section 4.4, the planning phase, unlike a software transaction, does not have to execute to its end. These observations enable several optimizations:

**Limiting the size of read/write logs.** The cost of consistency checking and instrumentation goes up with the size of the read/write logs. Assuming most shared reads/writes happen inside (uninstrumented) partitioned operations,
the read/write set of the planning phase is likely to be small. To guard against pathological cases, we stop planning if a predefined limit is exceeded. Small size allows us to employ a fast and simple structure for the logs.

**Merging partitioned operations.** The more partitioned operations a planning phase contains, the more instrumentation and consistency checking is required in the intervening glue code. We provide an interface for advanced programmers to reduce these overheads by merging the execution of several planning operations into one hardware transaction.

**Switching between ParT and non-ParT transactions.** Some transactions are not worth partitioning. A transaction with low contention and a small footprint, for example, is likely to succeed in HTM with few retries. At the opposite extreme, a transaction whose planning never works should always execute the monolithic alternative, even if HTM will fail and must fall back to a global lock. Deciding which version is better—ParT or non-ParT—is difficult at coding time, as the answer may depend on the input, and may vary across execution phases. A dynamic switching strategy may make a better choice at run time. One possible strategy is to use a short period of time to profile non-ParT transaction abort rates, and then chose the atomic blocks for which to employ the partitioned code over some longer span of time. As ParT and non-ParT code can safely run together, the switching policy has little overhead. We implemented this adaptive policy in our compiler, but did not employ it in the experiments of Section 4.5. Further experiments with adaptation are a subject for future work.

### 4.4.4 Limitations

While we are able to automate the composition of partitioned operations (and transactions that contain them), we require that the partitioned operations them-
selves already exist. The most significant limitation of our work is therefore the effort involved in constructing such operations. In future work we hope to develop tools to assist the library designer.

Because we end a planning phase when we encounter an unsafe function (Section 4.4.2), ParT will have little or no benefit when a call to such a function appears early in a transaction. Given the overhead of instrumentation on planning reads and writes, it is also possible for the cost of partitioning to outweigh the concurrency benefits, especially when contention is low. Likewise, if the work that remains in the completion phase still takes too long, or consumes too much memory, to fit in a hardware transaction, it is still possible that threads will serialize. (Even then, planning may help, if it is able to move work off of the serial path.) Despite these limitations, we have found ParT to be an effective means of enhancing HTM, as described in the following section.

4.5 Experimental Evaluation

Our experiments were conducted on two different HTM-capable machines—an IBM zEnterprise EC12 mainframe server (in a virtual machine with 64 dedicated cores) and a 4-core Intel Haswell Core i7-4470 machine. Both machines implement best-effect HTM and provide a similar ISA interface to the software. (The zEC12 also supports special “constrained” transactions that are guaranteed to complete successfully in hardware; these are not used in our experiments.) The transactional region is marked by a pair of begin/end instructions. A transaction may abort for various reasons, including interrupts, prohibited instructions, excessive nesting depth, capacity overflow, and conflicting accesses.

The IBM zEnterprise EC12 [58] is a multi-socket machine. Each processor chip contains six single-threaded, out-of-order superscalar cores with a clock speed of 5.5 GHz. A summary of its HTM implementation can be found at Section 2.2.2.
Our Intel Core i7-4740 machine has a single processor with 4 SMT cores (8 hardware threads). Each core has a private 32 KB, 8-way associative L1 data cache and a private 256 KB, 8-way associative L2 cache, with 64 B cache lines. The 4 cores share an 8 MB L3 cache. The HTM system is implemented on top of an existing cache design [131]. Transactional data are tracked in the L1 data cache, at cache-line granularity. If a written line is evicted from the cache, the transaction will abort. Evicted reads are tracked in a secondary structure that supports a larger read set, at the risk of a higher rate of false conflicts.

4.5.1 TM Compiler and Runtime

On Haswell, we implemented ParT as an LLVM 3.3 optimization pass, taking the bitcode of the entire program as its input. On the zEC12, where we did not have access to production-quality LLVM support, benchmarks were hand-modified to mimic the LLVM output. (With a bit more effort, we could have engineered support into one of the z compilers: there is nothing inherently LLVM-specific about the partitioning algorithm.)

For software fallback, the TM run-time library uses a global test-and-test-and-set lock with exponential backoff. If a completion phase aborts for a non-persistent reason, we retry the transaction up to MAX_RETRIES times before switching to the lock. Otherwise, we retry up to MAX_PERS_RETRIES times, rather than immediately reverting to the lock, because the “persistent” diagnostic flag is only a hint, and a retry may succeed despite it. To avoid repeated aborts on the same conflict, we delay briefly before restarting a transaction. Hardware transactions in planning operations are handled in the same way, except that the whole parent planning phase will stop after a fixed number of retries.

We set MAX_RETRIES and MAX_PERS_RETRIES to 8 and 5, respectively, on the zEC12, and to 10 and 3 on Haswell: these values delivered the best overall perfor-
Table 4.1: Summary of benchmarks. The “#” column is the static number of ParT transactions, “Comp.” indicates if transactions comprise multiple partitioned operations.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>Description</th>
<th>#</th>
<th>Comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>rb-tree</td>
<td>[33]</td>
<td>insert/delete elements</td>
<td>3</td>
<td>N</td>
</tr>
<tr>
<td>equiv. sets</td>
<td>[127]</td>
<td>move elements between sets</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>account</td>
<td>Fig 4.1</td>
<td>account management</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>genome</td>
<td>STAMP</td>
<td>gene sequencing</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>intruder</td>
<td>STAMP</td>
<td>network intrusion detector</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>vacation</td>
<td>STAMP</td>
<td>online travel reservation system</td>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>UtilityMine</td>
<td>RMS-TM</td>
<td>utilization-based item sets mining</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>memcached</td>
<td>ver1.4.9</td>
<td>in-memory key value store</td>
<td>4</td>
<td>Y</td>
</tr>
</tbody>
</table>

Performance for the HTM baseline. Read and write logs were sized at 32 and 16 entries, respectively.

4.5.2 Benchmark Suite

We use three microbenchmarks and five larger applications to evaluate ParT, and to compare it to other techniques. Table 4.1 lists these benchmarks, including their provenance, a description of their major behaviors, and the number of ParT transactions in the source.

The microbenchmarks in the top half of the table all contain operations with a nontrivial search phase. In our tests, operations on the data structure are invoked as often as possible for a period of 1 second; our performance graphs plot throughput in operations per microsecond. In all microbenchmarks, we pre-populate thread-local free lists with enough data nodes to eliminate the impact of memory allocation.

The macrobenchmarks in the bottom half of Table 4.1, from the STAMP [83] and RMS-TM [61] benchmark suites, were chosen because their transactions are
complex and good candidates for ParT optimization. All four were run with the recommended non-simulation inputs. The `memcached` macrobenchmark is a slightly modified version of memcached 1.4.9. Critical sections protected by three major locks (`cache_lock`, `slabs_lock`, and `stat_lock`) were replaced with hardware transactions. Several global data structures were padded to avoid false sharing in cache lines. To stress the server, we bypass networking and inject input (dumped by `memslap`) directly into relevant functions. For each application, we report speedup over the sequential code. The STAMP benchmarks, as distributed, use `malloc/free` for the sequential version but a customized thread-local allocator for the TM version. We found the thread-local version to be significantly faster; for the sake of fair comparisons, we re-configured the sequential code to use this version instead.

Source code on the zEC12 was compiled with IBM’s XL C/C++ compiler with `-O3 -qstrict` flags; on Haswell, we used LLVM 3.3 with `-O3`. Reported results are the average of 5 runs each, though no significant performance variation was observed.

### 4.5.3 Microbenchmark Results

**Red-black tree** The partition is described in Section 4.3.2. Two different data set sizes are used in the tests: in the smaller, keys are randomly selected from [0, 1K); in the larger, keys are randomly selected from [0, 1M). Throughput results appear in Figures 4.8a and 4.9a. In the smaller tree, HTM can finish most transactions without aborting, so the overhead of planning and validation is essentially wasted in ParT. In the larger tree, relative performance is reversed: pure HTM experiences many aborts, while partitioning shrinks the footprint of completion transactions enough for them to succeed much more often.
Equivalent Sets The equivalence set data structure comprises a collection of sorted linked lists, which partition a universe of elements; each operation moves a specified element from its current set to a given, specified set. Insertion in the new set requires a search of the list, which is done in the list’s `insert_P` method in the ParT implementation.

Throughput for multiple implementations is shown in Figures 4.8b and 4.9b. The “Lock” curve uses per-set fine-grained locking. “ParLock” is a variant of ParT that uses similar locking (instead of a transaction) to protect the completion phase. All five curves dip at 2 cores, due to the cost of coherence misses. ParT scales better than fine-grained locking, which in turn outperforms the baseline HTM in this experiment: in the absence of partitioning, we have a “stress-test”
workload, where speculation is rarely successful.

**Account Management**  This synthetic benchmark updates account information stored in two shared data structures. The transaction for account insertion and update appears in Figure 4.1, where user information is indexed by addresses. The transaction for account deletion (not shown) removes a user and all linked accounts.

In our tests, address strings are 35 characters long and each user has 4 accounts on average. Figures 4.8c and 4.9c plot throughput for two data sets: the smaller one (more conflicts) contains 1K unique addresses; the larger one (more overflows) contains 1M unique addresses. In both cases, ParT significantly reduces both conflicts and overflows (by more than 50% in every workload) and thus results in better scalability, though pure HTM provides higher throughput when transactions are small and contention is low.

### 4.5.4 Macrobenchmark Results

Three of the standard STAMP benchmarks [83] make heavy use of shared container objects. We built a ParT library containing partitioned versions of sorted linked list, hashtable, red-black tree, and memory allocation methods to enable automatic partitioning. Other than the addition of **pragmas**, no modifications were made to the source code of transactions.

**Genome**  The first phase of the application removes duplicate gene segments. Segments are divided into chunks, and every chunk is transactionally inserted into a hash set. The hash set is non-resizable, so some buckets may contain hundreds of segments. In the original transaction, insertion of a segment could involve a long search phase to find the proper position in a sorted linked list. A data conflict in the insertion would invalidate all active insertions in the same
transaction. ParT optimizes the transaction by moving all planning operations to a compound planning phase. By eliminating almost all load/store overflows and reducing load/store conflicts (Figures 4.11a and 4.13a), ParT leads to significantly better scalability on both machines (Figures 4.10a and 4.12a).

**Intruder** This application processes network packets in parallel—in capture, reassembly, and detection phases. Reassembly is the most complex phase. It uses a red-black tree to map each session id to a list of unassembled packets belonging to that session. If all packets from a session are present in the list, the list is removed from the tree and its packets are assembled as a complete stream and inserted back to a global queue, which is the principal locus of conflicts. The entire

**Figure 4.9:** Microbenchmark performance on Haswell. The Y axis indicates throughput.
reassembly phase is enclosed in a transaction, which we use a \texttt{#pragma part} to optimize.

The planning operations for both tree search and list insertion have a low failure rate (2.2\% and 2.6\%, respectively, at 16 threads on the zEC12), meaning that if a conflict occurs on the global queue, in the next re-execution, ParT can skip tree and list search. This dramatically reduces the duration and footprint of re-executed transactions. ParT almost doubles the throughput of the program (Figure 4.10b) on the zEC12, though scalability is still constrained by the global queues used to connect the three program phases. On Haswell, ParT starts to outperform the baseline when hyperthreading is used.

\textbf{Vacation} This application manages reservations for cars, flights, and rooms in a database implemented as a set of red-black trees. Each transaction creates, cancels, or updates a reservation in the database. Creation consumes the largest share of execution time. ParT optimization is straightforward, but the compiler must process multiple levels of plan functions to reach the tree operations.

On the zEC12 (Figures 4.10c and 4.10d), ParT lags slightly behind the original implementation on 1 or 2 threads. It outperforms the original on 3 or more threads, however. One improvement comes from the elimination of overflows, which often happen near the end of a big transaction and are quite expensive. Interestingly, as shown in Figures 4.11c and 4.11d, ParT increases load conflicts. At the same time, because the completion transactions are so much shorter than in the original code, and most of the planning can be preserved on retry, less work is wasted by each abort, and overall throughput still improves. On Haswell, ParT eliminates most aborts of the composed complete phase (Figures 4.13c and 4.13d) and therefore brings significant performance improvement.
UtilityMine  This application spends most of its time in a transaction that updates the “utility” of items according to data read from a database file. The main transaction scans a utility array, whose size is input dependent, to locate a specific item. If the item is found, its utility is increased; otherwise, a new item is inserted. We optimize this transaction by replacing the most common path, in which the item is found, with a function call, and using \texttt{#pragma plan\_for} and \texttt{#pragma complete\_for} to build an ad-hoc partitioned operation. In the planning function, the position of the found item is saved in a validator, whose \texttt{is\_valid} method confirms that the array has not been deallocated, and the item is still in the same position. In addition to reducing transaction duration and footprint, partitioning allows us to issue prefetch instructions on the zEC12; these avoid a “promote-to-exclusive” (PEX) case in the coherence protocol, which can
Figure 4.11: Aborts per transaction for TM macrobenchmarks (completion transactions only) on the zEC12.

sometimes lead to spurious aborts.

Only 1, 2, 4, and 8-thread configurations are available for this benchmark. Executing with the standard input sets, transactions rarely overflow, so the benefit of ParT, as shown in Figure 4.11e, comes mainly from a reduction in the low but still significant number of PEX aborts (note the modest scale of the y axis). Running with larger input sets, data overflow could emerge as a major issue in the baseline case, while ParT would still be ok. In general, ParT can reduce the chance of sudden performance anomalies with changes in program input.

Memcached This widely used server application stores key/value pairs in a global hash table and in an auxiliary table used for resizing operations. In the presence of concurrency, the table structure itself does not incur many conflicts,
but transactions also access the table’s size field and global statistics information, increasing the chance of contention. We partitioned three major table access functions (assoc_find, assoc_insert, and assoc_delete) using version numbers for validation [127]. Transactions that call these functions, directly or indirectly, are then partitioned by the compiler.

Another second major source of aborts arises in the function do_item_alloc, which begins with a relatively long search for a possibly expired item in a list of slabs, in the hope of performing fast space reuse instead of slow allocation. As this particular search is commutative, we do it in the function’s planning operation, transactionally. As shown in Figure 4.13f, by pulling table and list searches out of transactions, ParT significantly reduces the incidence of overflow aborts. Also, as transactions become shorter, the conflict window is narrowed, a phenomenon
that is further enhanced by ParT’s partial rollback. As a result, ParT improves performance by roughly a third at 8 threads on Haswell (Figure 4.12f). On zEC12, although some shared structures are padded, the much larger size of cache lines still causes false sharing problem, which restricts the overall scalability.

### 4.6 Related Work

ParT draws inspiration from our previous lock-based MSpec/CSpec work [127]. There the goal was to reduce the size of critical sections by removing work that could be executed speculatively in advance—ideally with the aid of a compiler that deals with mechanical issues of code cloning and data race prevention. In MSpec/CSpec, pre-speculation served to shorten the application’s critical path;
with partitioned transactions it serves to reduce transaction conflicts. Both systems share the benefit of cache warmup. ParT adds the benefits of reduced cache footprint, composability, and partial rollback.

The manual partitioning of operations in ParT also resembles the consistency oblivious programming (COP) of Avni et al. [2, 5], in which a search-then-modify operation is divided into a non-atomic search followed by atomic validation and modification. Avni and Suissa have extended COP to accommodate composition [6], but mainly for software TM. In an HTM system, their technique would require special instructions to suspend and resume a transaction. This would seem to preclude the use of additional transactions during planning—something that ParT uses freely. Planning during suspension would also do nothing to shorten the temporal duration of the main transaction.

For search-based operations, partitioned transactions bear a certain resemblance to *early release* [53] and *elastic transactions* [40], both of which allow a (software) transaction to remove no-longer-needed locations from its read set. From the perspective of usability, however, it seems likely that the writers of partitioned operations will find it easier to specify what they *do* need (as a validator) than what they don’t (as early releases).

In comparison to all these alternatives, ParT has the advantage of working with existing HTM; early release, elastic transactions, and composed COP would all require new hardware instructions, and would introduce thorny issues of false sharing within cache lines. Finally, only ParT allows partial rollback of an aborted transaction: in the present of contention, valid planning work can be leveraged during retry. Table 4.2 summarizes these comparisons.

In mechanism—if not in purpose—partitioned transactions also resemble the *split hardware transactions* of Lev and Maessen [68]. In that work, a transaction is divided into two or more segments, each of which logs its reads and writes (in software) and passes the logs to the following segment. A segment that needs to
<table>
<thead>
<tr>
<th></th>
<th>ParT</th>
<th>CSpec</th>
<th>ElasT</th>
<th>COP</th>
<th>COP-c</th>
</tr>
</thead>
<tbody>
<tr>
<td>support composition?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>require special HTM features?</td>
<td>No</td>
<td>No</td>
<td>early release</td>
<td>No</td>
<td>suspend/resume</td>
</tr>
<tr>
<td>benefits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>smaller footprint</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>shorter duration</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>partial rollback</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of HTM programming techniques

be consistent with its predecessors begins by re-reading the locations in their read logs. The final segment re-reads everything, and then performs the updates in the write log. While log maintenance incurs a nontrivial penalty, performance is still better than with STM, because conflict detection remains in hardware.

The principal goal of split hardware transactions is to support true closed and open nesting, in which an inner transaction can abort without aborting (the prefix of) its parent transaction, or commit even if the parent aborts. Lev and Maessen suggest that the mechanism could also be used for debugging and ordered speculation (loop parallelization). In contrast, the principal goal of partitioned transactions is to increase scalability by exploiting programmer knowledge of high-level program semantics. Rather than mechanically transfer read and write logs from one segment to the next, we transfer only as much information as we need to validate the planning phase and perform the completion phase. Low-level partitioned operations must of course be implemented manually by the library designer, but once they exist, the compiler can automate the rest, and we can expect decreases in abort rate that are unlikely with split transactions, where read and write logs are likely to induce a net increase in cache footprint.

Other connections are more tenuous. Transactional boosting [49], transactional predication [15], and transaction-safe nonblocking data structures [76] all exploit high-level semantic information to reduce the cost of nested operations in a software TM system, but we see no way to employ them with HTM. The
Foresight mechanism [44] facilitates composition, but for conservative, lock-based systems. True closed nesting [86] offers the possibility of partial rollback, but again it is not supported by current HTM.

4.7 Summary

As hardware transactional memory becomes more widely used, programmers will need techniques to enhance its performance. We have presented one such technique: partitioned hardware transactions (ParT). The key idea is to extract the read-mostly planning portions of common operations and to execute them—either in ordinary software or in smaller transactions—before executing the remaining completion transaction. To ensure atomicity, a validator object carries information across planning operations and into the corresponding completion operation, allowing the latter to confirm, quickly, that the planning work is still valid. Automatic compiler support allows partitioned operations—and transactions that include them—to compose as easily and safely as traditional monolithic transactions, with no special hardware support.

We tested ParT on both the IBM zEnterprise EC12 (currently the most scalable HTM-capable architecture) and a smaller Intel Haswell machine. Using a variety of examples, including three micro- and five macrobenchmarks, we demonstrated that ParT can yield dramatic performance improvements—often making the difference between scalable and nonscalable behavior. We conclude that ParT is a valuable addition to the “TM programmer’s toolkit.”
5 Staggered Transactions

5.1 Introduction

As discussed in Section 1.1.2, the hardware transaction abort rate dramatically affects the raw performance, scalability, and energy consumption of transactional programs. Aborts happen for two main reasons: hardware overflow and conflicts with other transactions. We have addressed the overflow case in Chapter 4; we focus here on conflicts.

Conflicts among transactions lead to poor scalability and energy waste. Table 5.1 shows several representative TM programs running on a typical 16-core eager HTM system (the experimental methodology is described in Section 5.6). Repeated aborts limit speedup (column “S”) and force many transactions to acquire a global lock and revert to irrevocable mode (column “%I”) in order to make forward progress. A high ratio of cycles spent in aborted transactions relative to committed transactions (“W/U” column) correlates strongly with energy waste. As shown in the final, vacation row, wasted work can still be significant even in programs with reasonable speedup.

Several previous projects have proposed hardware mechanisms to reduce the incidence of conflict aborts. Examples include DATM [97], RETCON [10], Wait-
Table 5.1: HTM contention in representative benchmarks. S: speedup with 16 threads over sequential run. %I: % of txns forced into irrevocable mode. W/U: ratio of wasted to useful cycles in transactions. LA: locality of contention addresses. LP: locality of contention PCs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>S</th>
<th>%I</th>
<th>W/U</th>
<th>Contention Source</th>
<th>LA</th>
<th>LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>list-hi</td>
<td>1.0</td>
<td>27%</td>
<td>4.92</td>
<td>linked-list</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>tsp</td>
<td>3.6</td>
<td>10%</td>
<td>1.53</td>
<td>priority queue</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>memcached</td>
<td>2.6</td>
<td>25%</td>
<td>3.11</td>
<td>statistics information</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>intruder</td>
<td>3.2</td>
<td>32%</td>
<td>4.02</td>
<td>task queue</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>kmeans</td>
<td>4.6</td>
<td>35%</td>
<td>3.57</td>
<td>arrays</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>vacation</td>
<td>9.7</td>
<td>1%</td>
<td>0.34</td>
<td>red-black trees</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

n-GoTM [59], and OmniOrder [93]. While these proposals achieve nontrivial improvements in abort rates and system throughput, they suffer from several limitations. First, most entail significant new hardware complexity—e.g., for the transactional-cycle detector of Wait-n-GoTM or the 8 new coherence protocol states of DATM. Second, they tend to target specific patterns of contention—e.g., the simple conflicting code slices of RETCON or the circular dependences of DATM and Wait-n-GOTM; they may miss other opportunities. Third, they are often specific to particular styles of HTM—e.g., lazy for RETCON, eager for others—and may not apply across the rest of the HTM design spectrum.

As an alternative to all-hardware mechanisms, we introduce the notion of Staggered Transactions, an automatic technique to reduce the frequency of aborts. Staggered Transactions serialize the execution of conflict-heavy portions of transactions by means of advisory (optional) locks, implemented using nontransactional loads and stores. Nonconflicting code continues to execute speculatively in parallel, thereby maintaining scalability. Because correctness remains the responsibility of the underlying TM system, Staggered Transactions function correctly even if some transactions neglect to obey the locking protocol. Moreover, the contention reduction achieved is largely independent of other HTM implementation details; in particular, it should be compatible with most conflict resolution techniques.
Effective implementation of Staggered Transactions requires a combination of compile-time and run-time mechanisms to choose whether to acquire a lock and, if so, which lock to acquire and where to acquire it. In our (fully automated) system, the compiler uses Data Structure Analysis [64] to identify and instrument a small subset of the accesses in each transaction, which it determines, statically, are likely to constitute initial accesses to shared locations. These accesses constitute *advisory locking points*. At run time, a locking policy decides which of these locking points to activate, and which lock to acquire at each, based on prior execution history. As we will shown in Section 5.6, a careful fusion of compile-time and run-time information allows us to avoid not only simple, repetitive contention but also more complex cyclic dependences, in which each transaction in the cycle accesses a (typically dynamically chosen) location on which the next transaction has already performed a conflicting access.

To escape isolation when acquiring locks, Staggered Transactions require non-transactional loads and stores within transactions. They also benefit from a hardware mechanism to identify the PC at which a conflicting location was first accessed. Neither of these features is common on existing machines, but both appear orthogonal to most other HTM features, and neither seems prohibitively difficult to add.

Our principal contributions, discussed in the following sections, include

- A hybrid optimistic / pessimistic execution model for hardware transactions, in which the most commonly conflicting portions of transactions are serialized by advisory locks, built with nontransactional loads and stores.
- Compiler techniques to insert required instrumentation with negligible impact on run-time overhead.
- Run-time techniques to detect contention and enable advisory locks that avoid it, using high-level program knowledge learned by the compiler.
• Experimental results, collected on a variant of AMD's ASF simulator [21], that show significant contention reduction on a 16-core eager HTM, leading to an average of 24% improvement in performance for a collection of 9 benchmarks.

5.2 Overview

Unlike conventional TM, Staggered Transactions force portions of transactions to serialize; the remaining portions run concurrently. Specifically, when conflicts tend to arise in the middle of a given atomic block, our compiler and runtime arrange to acquire an advisory lock at the end of the contention-free prefix, and to hold it until commit time.

Figure 5.1 illustrates the execution of three transactions with mutually conflicting accesses, shown with a diamond symbol. On a typical eager (Figure 5.1a) or lazy (Figure 5.1b) HTM system, only one of the transactions will be able to commit if all three execute concurrently. In a Staggered Transactions system (Figure 5.1c), execution of the portion of a transaction beginning with the conflicting access is preceded by an advisory locking point (ALP). In the diagram, t1, t2, and t3 all attempt to acquire the same advisory lock. Transaction t1 acquires it first; the others wait their turn, and all are able to commit. We say that the conflicting portions of the transactions have been staggered. While a transaction could, in principle, acquire multiple advisory locks, we acquire only one per transaction.

Accurate identification of all and only the contention-prone portions of transactions requires careful coordination of compile-time and run-time techniques. Figure 5.2 illustrates how the pieces fit together in our system.

The main goal of the compile-time steps (1–3) is to statically insert ALPs in the source code, so that once a frequently conflicting access is found during execution, the runtime can activate the nearest ALP ahead of that access, preventing
concurrent accesses to the same object in other transactions. A dedicated compiler pass inspects the IR (intermediate representation) of all atomic blocks (1) and considers load and store instructions as anchors—accesses in front of which to insert an ALP (2). To minimize the number of ALPs (and thus the run-time overhead of instrumentation), the compiler uses Data Structure Analysis [65] to select as anchors only those instructions that are likely to be initial accesses to shared objects (data structure nodes) inside a transaction. To help the runtime locate ALPs quickly, the compiler builds an anchor table that maps PC ranges to the closest prior ALP in each atomic block (3).

The run-time steps (4–8) focus on two decisions concerning advisory locks: (1) whether the runtime should acquire an advisory lock for the current transaction, and if so, (2) which advisory lock to acquire, at which ALP. The first decision
is made by tracking abort history for every atomic block. When an atomic block runs on HTM (1) and a contention abort occurs (5), the hardware-triggered handler receives an indication of the conflicting address and (ideally) the program counter at which that address was first accessed (6). The runtime appends this information to a per-thread abort history table for the current atomic block. Based on the frequency of contention aborts, a software locking policy makes decision (1).

Decision (2) is harder. As shown in Table 5.1 ("LA" and "LP" columns), the program counters associated with initial accesses to conflicting locations are often the same across dynamic transactions, but the accessed data locations often differ: sometimes a common datum is responsible for most aborts; other times, it differs from one transaction instance to another.

The locking policy augments its understanding of the conflict pattern each time a contention abort occurs (7). Once a pattern is found, the runtime consults the anchor table to identify an ALP and activate it for future instances of the transaction (8). For simplicity, we currently allow only one active ALP for a given atomic block. We also employ a fairly simple policy (more complex possibilities are a subject of future work). Specifically, we activate an ALP only if it corresponds
to a PC that has frequently performed the initial access to data that subsequently proved to be the source of conflict. If the addresses of the data in these conflicts were usually the same, the ALP is activated in *precise mode*: it acquires a lock only if the data address in the current transaction instance matches the address of past conflicts. If data addresses have varied in past conflicts, the ALP is activated in *coarse-grain* mode: it acquires a lock regardless of the current data address. In either case, the choice of which lock to acquire is based on the current data address: this will always be the same in precise mode; it may vary in coarse-grain mode.

If conflicts continue to be common despite the activation of a coarse-grain ALP, the runtime uses information gathered by the compiler to activate the *parent* of the ALP instead. The notion of parents again leverages Data Structure Analysis. In a linked data structure, if node $B$ is reached via a pointer from node $A$, we say that the ALP associated with the initial access to $A$ is the parent of the ALP associated with the initial access to $B$. In code that traverses a linked list, for example, each node other than the first is accessed through its predecessor; the first is accessed through the head node. In typical traversal code, nodes within the list will share an ALP (embedded in a loop). The parent of that ALP will be the ALP of the head node. Interested readers may consult Lattner’s thesis for details [65].

As a simple example, suppose in Figure 5.2 that $q \rightarrow \text{head}$ is a frequent source of conflicts. After a few aborts, the locking policy will realize that most conflicts happen on the data address $q \rightarrow \text{head}$, whose initial access in the transaction is usually at the same instruction address, say $\text{Addr}$. The policy categorizes the conflict pattern as *precise* and the runtime activates the ALP right before $\text{Addr}$.

Like the advisory locks of database and file systems, Staggered Transaction advisory locks are purely a performance optimization. Correctness is ensured by the underlying TM system. If the runtime fails to instrument a transaction that
participates in a conflict, the only consequence is continued aborts. Likewise, when a transaction does attempt to acquire a lock, it need not wait forever—to avoid blocking on a very long transaction (or, on hardware that supports it, a transaction that has been preempted), an ALP can specify a timeout for its acquire operation, and simply proceed when the timeout expires.

5.3 Compiler Support

Our Staggered Transactions system uses a compiler pass for static insertion of advisory locking points (ALPs). Using Data Structure Analysis [65], the compiler identifies and instruments only those loads and stores (anchors) that are likely to constitute initial accesses to shared locations. It also generates an anchor table for each atomic block to describe the anchors and non-anchors (uninstrumented loads and stores) and the relationships among them. Anchor tables are subsequently consulted by the runtime to make locking decisions and to locate desired advisory locking points.

Compared to naive instrumentation of every load and store, our technique significantly reduces the number of ALPs, thereby minimizing execution overhead. Moreover, by capturing information about the hierarchical structure of program data, the generated anchor tables allow the runtime to make better locking decisions.

5.3.1 A Data Structure Approach

We consider program-level objects (data structure nodes) to be an appropriate granularity for the insertion of advisory locking points, based on two observations: First, fields of the same object frequently fit together in just a few cache lines. Since HTM systems typically detect conflict at cache-line granularity, if there
is contention on the object, the contended code region will usually start at the first instruction that accesses the data structure node. Second, assuming modular program structure, instructions that access the same object are often concentrated in a short code segment. This implies that the initial access to an object in a given function or a method is likely to be a good anchor candidate. Once these candidates are instrumented, we can skip a number of following loads and stores that access the same objects, without losing the ability to trace back from an abort to an appropriate ALP.

A field-sensitive Data Structure Analysis can be used to identify the objects associated with loads and stores and their alias relationships. We base our compiler pass on Lattner’s DSA [65], which we use essentially as a black box. DSA distinguishes pointer-to sets according to the data structure type and field to which they point. It has previously been used for improved heap allocation [64] and type safety checking. A complete DSA pass has three stages: (1) A local stage creates a data structure node (DSNode) for each unique pointer target in a function, and links each pointer access to a DSNode. All pointers linked to the same DSNode may point to the same instance of a data structure. If a pointer field in a data structure points to another data structure (or itself), there will be an outgoing edge from this DSNode to the target DSNode. All DSNodes of a function are organized together as a data structure graph (DSGraph). (2) A bottom-up stage merges callees’ local DSGraphs into those of their callers. (3) A top-down stage merges callers’ DSGraphs into those of their callees. We utilize only the result from stage 2, which is more context sensitive than that of stage 3 (to get more accurate alias results, which we do not need, the latter may collapse too many DSNodes).

Given DSA results, our algorithm works in three stages of its own, which we describe in Sections 5.3.2–5.3.4.
5.3.2 Building Local Anchor Tables

A local anchor table keeps information for all loads and stores of a function directly or indirectly called in an atomic block (static transaction). Each load/store is described by a table entry, \textit{ATEntry}, a 4-field tuple: \texttt{(instr, isAnchor, parent, pioneer)}. Field \texttt{instr} indicates the location of the load/store instruction. Field \texttt{isAnchor} is a Boolean flag. We call a load/store an \textit{anchor} if it is the initial access to a DSNode in a possible execution path. (ALPs are placed before anchors in a later stage.) Field \texttt{parent} points to another anchor through which a pointer to the current node was loaded. For example, in the code sequence \{\texttt{B = A→child; \ldots = B→size;}\}, if the loads of \texttt{A→child} and \texttt{B→size} are anchors, then the first one is the second one's parent. For a non-anchor access, field \texttt{pioneer} points to the anchor that accesses the same DSNode. For example, in \{\texttt{n = queuePtr→head; \ldots; queuePtr→tail = m;}\}, the \texttt{pioneer} of the store to \texttt{queuePtr→tail} is the load of \texttt{queuePtr→head} (assuming that load is an anchor). The reason for keeping non-anchor loads and stores in the table, even though they are not instrumented at a later stage, is that if a conflict actually occurs on a non-anchor access, the \texttt{pioneer} information still enables us to trace back to the closest ALP.

Algorithm 5.1 shows how the compiler constructs an anchor table for a given function. The first stage (lines 3–14) categorizes load and store instructions as anchors or non-anchors through a depth-first traversal of the function’s dominator tree. The second stage (lines 15–19) sets up the parent relationship among anchors, using the result of DSA’s field-sensitive analysis.

5.3.3 Building Unified Anchor Tables

Walking from callers to callees, a top-down stage creates one unified anchor table for each atomic block. This stage doesn’t change the local tables. Instead, it clones and merges them, taking account of the DSNode mapping from caller to callee at
Algorithm 5.1: BuildLocalAnchorTable(func)

Input: a function func
Output: a local anchor table aTable

1. $aTable \leftarrow \emptyset$
2. $domTree \leftarrow \text{GetDominateTree}(func)$
3. foreach BasicBlock $b$ in DepthFirstVisit($domTree$) do
   4. foreach LoadStoreInst $inst$ in $b$ do
      5. $dsNode \leftarrow \text{GetDSNode}(inst$.pointerOperand$);
      6. $entry \leftarrow \text{new ATEntry}$;
      7. $entry$.inst $\leftarrow inst$;
      8. if $\exists m \in aTable[dsNode]: m$.inst dominates $inst$ then
         9. $entry$.isAnchor $\leftarrow$ false;
         10. $entry$.pioneer $\leftarrow m$.inst;
      else
         11. $entry$.isAnchor $\leftarrow$ true;
         12. $entry$.parent $\leftarrow$ nil;
      13. $aTable[dsNode]$.push($entry$);
5. foreach DSNode $n$ in $aTable$ do
   6. foreach Edge $e$ in $n$.edges do
      7. foreach ATEntry $m$ in $aTable[e.toNode]$ do
         8. if $m$.isAnchor then
            9. $m$.parent $\leftarrow n$;

function call sites. Missing parent information for certain anchors, if passed via function arguments, is filled in at this stage. From the construction procedure, we can see that unified anchor tables are context-sensitive: anchors originating from the same instruction may have different parents in different unified anchor tables.

5.3.4 Instrumentation

Once per-function local anchor tables and per-atomic-block unified anchor tables are available, the insertion of advisory locking points is straightforward: the compiler iterates over all local tables and inserts calls to ALPoint right before each anchor. Every ALP is assigned a unique ID so that the run-time locking policy can locate and activate it. After the binary code has been generated, the com-
TM_BEGIN(); // genome/sequencer.c:292
...
for (ii = i; ii < ii_stop; ii++) {
    void* segment = vector_at(segmentsContentsPtr, ii);
    TM_hashtable_insert(uniqueSegmentsPtr, segment, segment);
}
TM_END();

void* vector_at (vector_t* vectorPtr, long i){ //lib/vector.c:164
    if ((i < 0) || (i >= vectorPtr->size))  // A 51: Parent 0
        return NULL;
    return (vectorPtr->elements[i]);  // 53: Pioneer 51
?
// lib/hashtable.c:171
bool_t TM_hashtable_insert (hashtable_t* hashtablePtr, ...){
    long numBucket = hashtablePtr->numBucket;

    ...
    ... = TMlist_find(hashtablePtr->buckets[i],  // A 42: Parent 0
                     &findPair);

void* TMlist_find (list_t* listPtr, ...){ //lib/list.c:588
    list_node_t* nodePtr;
    list_node_t* prevPtr = &(listPtr->head);
    for (nodePtr=(list_node_t*)prevPtr->nextPtr;  // A 35: Parent 42
         nodePtr != NULL;
         nodePtr = (list_node_t*)nodePtr->nextPtr)  // 38: Pioneer 35
        ...
Figure 5.3: An atomic block in genome. ► marks an entry with its ID and parent/pioneer field in the unified anchor table.

piler knows the real PC of each instruction. It makes the unified anchor tables indexable by PC address, and then emits all unified tables.

5.3.5 Example

Figure 5.3 presents a portion of the generated anchor table for an atomic block in the STAMP genome benchmark. The compiler pass first runs DSA for functions called inside the atomic block (vector_at, TM_hashtable_insert, and TMlist_find). Function TMlist_find, for example, contains a single DSNode (for nodePtr and prevPtr) with two loads on it. The first load is marked as an anchor (A 35),
according to Algorithm 5.1; the second is a non-anchor (38) whose pioneer is A
35. A 35’s parent field is not filled in until the unified anchor table is constructed.

As we can see, the parent chain between anchors (from hashtablePtr to
listPtr) is preserved in the unified anchor table, through which the runtime
can make advanced locking decisions. For this code snippet, the compiler will fi-
nally instrument three loads (vectorPtr→size, hashtablePtr→numBucket, and
prePtr→nextPtr) as advisory locking points.

5.4 Hardware Requirements

Staggered Transactions require—or can benefit from—several hardware features. First, they must be able to acquire an advisory lock from within an active hard-
ware transaction. This operation violates the usual isolation requirement, but
is compatible with several real and proposed HTM architectures. Second, they
must be able to identify the data address that has been the source of a conflict
leading to abort. This capability is supported by all current HTM designs. Third,
they can benefit from a mechanism to identify the address of the instruction that
first accessed an address that was the source of a data conflict. While no such
mechanism is available on current hardware, it seems relatively straightforward to
provide. Moreover, unlike the hardware features required by systems like DATM
[97] and Wait-n-GoTM [59], both intra-transaction lock acquisition and initial
access recording appear to be independent of other details of HTM design.

Advisory lock acquisition is most easily performed using nontransactional
loads and (immediate) stores. Nontransactional loads have appeared in Sun’s
Rock processor [19] and AMD’s ASF proposal [21]. They allow a transaction to
see writes that have been performed (by other threads) since the beginning of the
current transaction. They do not modify the transaction’s speculative state: that
is, a store by another thread to a nontransactionally-read location will not cause
the transaction to abort (unless the location has also been accessed transaction-
ally). Nontransactional stores similarly allow a transaction to update locations
even in the event of an abort, and (in the absence of transactional loads or stores
to the same location) will not lead to aborts due to conflicting accesses in other
threads. To be of use in acquiring an advisory lock, a nontransactional store
must (as in ASF) take place immediately; it cannot (as in Rock or IBM’s z series
machines [58]) be delayed until the end of the transaction.

On ASF, Rock, and z, nontransactional accesses are performed by special load
and store instructions. On IBM’s POWER 8 [17], a transaction can suspend and
resume execution. While suspended, its loads and stores are nontransactional,
and its stores immediately visible. Significantly, nontransactional accesses have a
variety of other uses, including hybrid TM [29, 102]; debugging, diagnostics, and
statistics gathering [43]; and ordered speculation (e.g., for speculative paralleliza-
tion of loops).

**Conflict information** of some sort is provided on abort by all current HTM
designs. All, in particular, provide the address of the location (or cache line) on
which a conflict has occurred. For Staggered Transactions, a machine would also,
ideally, provide the address of the instruction at which the conflicting datum was
initially accessed in the transaction. We call this address the conflicting PC. Note
that it is generally not the current PC at the time the conflict is discovered.

In an HTM system implemented in the L1 cache, conflicting PC information
could be maintained by adding a PC tag to each cache line, in addition to transac-
tion status bits. This tag would be set whenever a line transitions into speculative
mode. Because the tag is inspected only when the line is the source of a data con-
fusion, it need not ever be cleared. As we shall see in Section 5.6, one can in fact
get by with just a subset of the PC (e.g., the 12 low-order bits). This suffices to
keep the space overhead under 2.4%.
Software Alternatives to Conflicting PC  While commercial HTMs have 
begun to support nontransactional loads and stores, hardware recording of the 
conflicting PC is still a missing feature. Without it, we need a software alterna-
tive to map a conflicting data address to the appropriate anchor in instruction 
space. A relatively cheap solution is to keep a map $M$ for each thread, indexed 
by cache line address. At every ALP (with anchor ID $I$, preceding a load/store 
of some data address $A$), the runtime can use nontransactional loads and stores 
to set $M(A)$ to $I$, if $A$ was previously absent. If a conflict subsequently occurs 
on address $A$, $M(A)$ can be used to identify the ALP that should, perhaps, be 
activated. While the overhead of this method is nontrivial, it can sometimes be 
acceptable. Section 5.6.2 compares this method with hardware-supported con-
flicting PC tracking.

5.5 Runtime Support

Our compiler assigns a unique ID to each source code atomic block. For each of 
these, the runtime maintains an ABContext data structure, as shown in Figure 5.4, 
for each executing thread. A pointer to the ABContext for the current atomic block 
and thread is loaded into a local variable at the beginning of each transaction, 
and accessed at each ALP. If the transaction aborts, the ABContext may also be 
accessed by the policy for ALP activation.
```c
void ALPoint (ABContext *c, int myID, void* addr) {
    if (c->activeAnchor == myID &&
        IsAddressMatched(c->blockAddress, addr) {
        c->activeAnchor = 0;
        AcquireLockFor(addr);
    }
}
```

**Figure 5.5:** ALPoint instrumentation function.

### 5.5.1 Instrumentation

In our current implementation, each ALP comprises a call to the ALPoint function, shown in Figure 5.5. The function takes three arguments—a pointer to the appropriate ABContext, the ID of the anchor, and the data address accessed in the following load or store instruction.

The ALPoint function acquires an advisory lock if the current anchor is active (line 2) and either the cache line of the data address matches that of the address in the ABContext, or the ALP is in coarse-grain mode (indicated by c->blockAddress==0). This disjunction is checked by function IsAddressMatched (line 3). When a lock is acquired, c->activeAnchor is cleared to avoid additional locking attempts within the current transaction (line 4). The advisory lock is released when the transaction commits or aborts. The activeAnchor field is restored the next time the thread begins a transaction for the same atomic block.

The actual blocking/waiting is performed in function AcquireLockFor. In our implementation, this function uses a hash of the data address to choose one of a static set of pre-allocated locks, which it then accesses using nontransactional loads and stores. In a system that allows a transaction to remain active when its thread is preempted or blocked, it is important not to force all other transactions to wait if the stalled one holds a lock. With appropriate OS support, the runtime could register the location of any advisory lock it acquires, and the kernel could free this lock when descheduling the thread. Alternatively, a transaction that waits
“too long” for an advisory lock could simply time out and proceed without it.

5.5.2 Locking Policy

The locking policy serves to predict, based on past behavior, which instructions are likely to constitute the first access within a transaction to a location that is likely to be the source of a conflict. Based on this prediction, the runtime activates an appropriate ALP. Many policies are possible. We describe our current choice, which is simple and seems to perform well in practice.

Table 5.1 suggests that the conflicting data address alone may not be a predictor of contention, due to its poor locality in certain access patterns. Likewise, PC alone is an overpredictor: while the same instruction is very often the initial access to a conflicting location, there are often cases in which that instruction accesses a location that is not a source of conflict. These observations suggest that the combination of PC and data address might make an effective predictor. A policy based on this idea appears in Figure 5.6. The policy works on a per-thread, per-atomic block basis. Function ActivateALPoint is called on an abort. Depending on the frequency with which the current conflicting data address (line 6) and initially-accessing PC (line 7) have appeared in the recent past, the policy chooses one of four behaviors:

Precise Mode. Both the conflicting PC and the data address appear multiple times in the history. This is typical of statistics and bookkeeping information, or of cyclic dependences. In this mode, the appropriate anchor is activated (line 9) with the conflicting address as the target (line 10).

Coarse-grain Mode. In this case the conflicting PC is recurrent, but the data address keeps changing. This is typical of pointer-based structures like lists (Figure 4.7) and trees, whose nodes are scattered across cache lines. In this mode we activate the anchor with a “wild card” data address (line 14). In the next instance
```c
1 void ActivateALPoint(ABContext *c, AbortInfo *abt) {
2     AEntry *en = SearchByPC(c->anchorTable, abt->confPC);
3     if (!en->isAnchor) // always begin with an anchor
4         en = en->pioneer;
5     AbortInfo *history = c->abtHistory;
6     bool a = CountAddr(history, abt->confAddr) > ADDR_THR;
7     bool p = CountPC(history, en->PC) > PC_THR;
8     if (p && a) {  // case 1: precise mode
9         c->activeAnchor = en->ID;
10        c->blockAddress = abt->confAddr;
11    } else if (p && !a) {  // case 2: coarse grain
12        if (retries < PROM_THR) {
13            c->activeID = en->ID;
14            c->blockAddr = 0;
15        } else {  // case 3: locking promotion
16            c->activeID = en->parent;
17            c->blockAddr = 0;
18        }
19    } else {  // !p  // case 4: training mode
20        c->activeID = 0;
21        c->blockAddr = 0;
22    }
23    AppendToHistory(history, en->PC, abt->confAddr);
24 }
```

**Figure 5.6:** Pseudocode of a simple locking policy.

of the transaction, the first accessed DSNode of the structure (usually the root or head node—i.e., the whole data structure) will be locked.

**Locking Promotion.** If contention persists in coarse-grain mode, the lock is promoted to the parent anchor (line 16) in the hope of avoiding contention there.

**Training Mode.** When no pattern has (yet) emerged, the policy simply continues to gather statistics.

When a transaction commits while holding an advisory lock, but there was no contention on that lock, an empty entry can be appended to the abort history to shift out the previous records, avoiding over-locking in the case of low contention.

Coarse-grain locking and locking promotion serve to break cycles of conflict
among transactions that occur on separate locations. In pointer-based data structures in particular, conflicting addresses may vary across both time and threads. Location-based hardware techniques for conflict avoidance (e.g., Wait-n-GoTM [59]) are generally unable to avoid such conflicts. With the advantage of information from the compiler’s Data Structure Analysis, Staggered Transactions handle these conflicts by acquiring locks at a coarser granularity or a higher (more abstract) level of the data structure. Consider, for example, the transaction in Figure 4.7, which consumes significant time in genome. The transaction inserts segments into several lists in a shared table. A cycle of conflict may easily arise among threads—thread 1 inserts segments to lists A, B, and D; thread 2 to D and C; thread 3 to C and A. With a frequently conflicting PC (Anchor 35) and unstable conflicting addresses, the locking policy will eventually reach the advisory lock for the whole table (Anchor 42), breaking the conflict cycle.

5.6 Experimental Results

Hardware. Since no existing HTM provides all the hardware features required by Staggered Transactions, we conducted our experiments on MARSSx86 [92], a full-system cycle-accurate x86 simulator with high fidelity HTM support [20]. The HTM simulation is based on a variant of AMD’s Advanced Synchronization Facility (ASF) proposal [20, 114]. The ISA uses speculate/commit instructions to mark a transaction region. Read and write sets are maintained in the L1 cache by adding two bits (tx read and tx write) to each cache line. An eager requester-wins conflict resolution policy is implemented on top of a modified MOESI coherence protocol [21, 114]. This HTM, designed to “incur the fewest modifications to the existing cache coherence and core designs” [131], is similar to those employed in Intel’s Haswell [131] and IBM’s zEC12 [58], with the notable addition of nontransactional loads and stores within transactions. We made the following modifications
### Table 5.2: Configuration of the HTM simulator.

<table>
<thead>
<tr>
<th>CPU cores</th>
<th>2.5GHz, 4-wide out-of-order issue/commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>private, 64K D + 64K I, 8-way, write-back, 64-byte line, 2-cycle</td>
</tr>
<tr>
<td>L2 cache</td>
<td>private, 1M, 8-way, write-back, 10-cycle</td>
</tr>
<tr>
<td>L3 cache</td>
<td>shared, 8M, 8-way, write-back, 30-cycle</td>
</tr>
<tr>
<td>Coherence</td>
<td>MOESI</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB, 50ns, 2 memory channels</td>
</tr>
<tr>
<td>HTM</td>
<td>2-bit (r/w) per L1 cache line</td>
</tr>
<tr>
<td></td>
<td>eager requester-wins policy</td>
</tr>
<tr>
<td>Stag. Trans.</td>
<td>12-bit PC tag per L1 cache line</td>
</tr>
</tbody>
</table>

- The original ASF design [21] treated only annotated load/store instructions \(\text{lock \text{mov}}\) as transactional operations, and normal load/store as nontransactional. In keeping with later changes in the proposal, we reversed this behavior to match that of other HTM systems.

- As described in Section 5.4, we added a 12-bit PC tag to every L1 cache line to record bits of the conflicting PC. The space overhead in the L1 cache is less than 2.4%.

- On a contention abort, the hardware places the low 12 bits of the conflicting PC and the low 52 bits of the conflicting data address into the \%rbx register.

We model a 16-core machine with the configuration shown in Table 5.2.

**Compiler and HTM Runtime.** Our compiler support is realized in LLVM 3.4 as an optimization pass, using an existing DSA implementation.

The locking policy keeps 8 recent abort records in each \text{ABContext}, with PC\_THR=2 and ADDR\_THR=2. The HTM runtime tries each hardware transaction up to 10 times; it then enters irrevocable mode by acquiring a global lock. Hardware transactions add the global lock to their read set immediately before attempting to commit. Prior to a retry, the runtime spins for an amount of time whose mean
<table>
<thead>
<tr>
<th>Program</th>
<th>Source</th>
<th>Description and input</th>
<th>ABs</th>
<th>%TM</th>
<th>S</th>
<th>Abts/C</th>
<th>Contention</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>STAMP</td>
<td>-g1024 -s16 -n16384</td>
<td>5</td>
<td>61%</td>
<td>6.0</td>
<td>0.25</td>
<td>low</td>
</tr>
<tr>
<td>intruder</td>
<td></td>
<td>-a10 -l4 -n2038 -s1</td>
<td>3</td>
<td>98%</td>
<td>3.2</td>
<td>5.28</td>
<td>high</td>
</tr>
<tr>
<td>kmeans</td>
<td>STAMP</td>
<td>-m15 -n15 -t0.05 -i random-n2048-d16-c16</td>
<td>3</td>
<td>42%</td>
<td>4.6</td>
<td>4.74</td>
<td>high</td>
</tr>
<tr>
<td>labyrinth</td>
<td>STAMP</td>
<td>-i random-x16-y16-z3-n64, w/ early release</td>
<td>3</td>
<td>91%</td>
<td>1.9</td>
<td>3.47</td>
<td>high</td>
</tr>
<tr>
<td>sscac2</td>
<td>IntSet[103]</td>
<td>-s13 -i1.0 -u1.0 -l3 -p3</td>
<td>10</td>
<td>16%</td>
<td>4.8</td>
<td>0.02</td>
<td>low</td>
</tr>
<tr>
<td>vacation</td>
<td>IntSet[103]</td>
<td>-n4 -q40 -u90 -r16387 -t4096</td>
<td>3</td>
<td>87%</td>
<td>9.7</td>
<td>0.49</td>
<td>med</td>
</tr>
<tr>
<td>list-lo</td>
<td>IntSet[103]</td>
<td>64 nodes, 90%/5%/5% lookup/insert/delete</td>
<td>4</td>
<td>86%</td>
<td>3.6</td>
<td>1.11</td>
<td>med</td>
</tr>
<tr>
<td>list-hi</td>
<td>IntSet[103]</td>
<td>64 nodes, 60%/20%/20% lookup/insert/delete</td>
<td>4</td>
<td>83%</td>
<td>1.0</td>
<td>4.05</td>
<td>high</td>
</tr>
<tr>
<td>tsp</td>
<td>[8]</td>
<td>travel salesman problem solver, 17 cities</td>
<td>3</td>
<td>90%</td>
<td>3.6</td>
<td>1.74</td>
<td>med</td>
</tr>
<tr>
<td>memcached</td>
<td>[80]</td>
<td>in-memory key-value storage</td>
<td>17</td>
<td>85%</td>
<td>2.6</td>
<td>4.77</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 5.3: Benchmark characteristics. ABs: number of atomic blocks in the source code. %TM: percentage of execution time spent in transactional mode. S: speedup with 16 threads over sequential run on the baseline HTM. Abts/C: aborts per commit on the baseline with 16 threads.
value is proportional to the number of retires (as in the “Polite” policy of Scherer & Scott [105]).

**Benchmarks.** We use the STAMP suite [83] and three other representative TM programs as benchmarks, as summarized in Table 5.3. STAMP’s yada and bayes are excluded because yada has overflow issues and bayes has unstable execution time. The list-hi microbenchmark is drawn from the RSTM test suite [103]. It comprises a set of threads that search and update a single shared, sorted list. The tsp benchmark is our own C++ implementation of a branch-and-bound TSP solver. All candidate tasks are kept in a B+ tree-based [8] priority queue, which supports \(O(1)\) pop and \(O(n \log n)\) push operations. We eliminated the tree’s size field, which tends to be highly contended. The memcached benchmark is a modified version of memcached 1.4.9. The network code is elided in order to speed up simulation and to increase the number of working threads. We obtain the input data from memslap and inject them directly into the application’s command processing functions.

All binaries were compiled with -O2 optimizations, running on Debian 7.0 with a Linux 3.2 kernel. To avoid the potential contention bottleneck in the default glibc memory allocator, we use the Lockless Memory Allocator [70] instead. To reduce the impact of the OS scheduler, we pin every worker thread to a specific CPU core during program initialization. Each run was repeated 5 times; the average number is reported.

5.6.1 Instrumentation Overhead and Accuracy

The “Static Stats” section of Table 5.4 shows the number of loads and stores analyzed by the compiler and the number of these that were instrumented as anchors (“anchs”) at compile time. On average, 13% of loads and stores are instrumented.
<table>
<thead>
<tr>
<th>Program</th>
<th>Static Stats</th>
<th>Dynamic Stats (1 thread)</th>
<th>Accuracy (16 thds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld/st instrs</td>
<td>anchs</td>
<td>u-ops per txn</td>
</tr>
<tr>
<td>genome</td>
<td>82</td>
<td>19</td>
<td>957</td>
</tr>
<tr>
<td>intruder</td>
<td>410</td>
<td>56</td>
<td>351</td>
</tr>
<tr>
<td>kmeans</td>
<td>13</td>
<td>6</td>
<td>261</td>
</tr>
<tr>
<td>labyrinth</td>
<td>418</td>
<td>18</td>
<td>16968</td>
</tr>
<tr>
<td>ssca2</td>
<td>33</td>
<td>7</td>
<td>86</td>
</tr>
<tr>
<td>vacation</td>
<td>442</td>
<td>76</td>
<td>4621</td>
</tr>
<tr>
<td>list-hi</td>
<td>43</td>
<td>5</td>
<td>391</td>
</tr>
<tr>
<td>tsp</td>
<td>737</td>
<td>75</td>
<td>2348</td>
</tr>
<tr>
<td>memcached</td>
<td>405</td>
<td>54</td>
<td>2520</td>
</tr>
</tbody>
</table>

**Table 5.4:** Static and dynamic statistics of instrumentation.

The “Dynamic Stats” section reflects the behavior of instrumented code in single-threaded runs. Since the number of anchors executed in each transaction is small compared to the total number of µ-ops, and an inactive ALP is simply a test and a non-taken branch, the execution time change is negligible in most benchmarks. The principal exception is the list microbenchmark, in which the anchors appear in tight loops. Further optimization of such loops may be a fruitful direction for future work.

For comparison, we also constructed a naive implementation in which every load and store was instrumented. This lead to slowdowns in excess of 10% for six of the benchmark programs (labyrinth, kmeans, vacation, list, tsp, and memcached).

The “Accuracy” section of the table shows the percentage of dynamic aborts for which our runtime was able to correctly identify the anchor associated with the initial access of the contended datum. All are above 95%; six out of nine are above 98%.
5.6.2 Parallel Performance

In the “S” column of Table 5.3, we list the speedup of benchmarks running on the baseline eager HTM at 16 threads. These benchmarks show low to high contention, as indicated in the final column. The worst is list-hi, which stops scaling after 4 threads.

Comparative performance on 16 threads appears in Figure 5.7, which plots speedup relative to the baseline (“HTM”) for “StaggeredTM” (with hardware Conflict PC support) and “StaggerTM w/o CPC” (with software-based anchor tracking, as described in Section 5.4). Also plotted is a much simpler scheme, “AddrOnly,” which places one fixed ALP at the beginning of each atomic block and uses only precise mode to trigger lock acquisition.

**Result 1:** Staggered Transactions improve the performance of high-contention applications without slowing down application with low contention.

We see substantial performance improvement (>30%) in intruder, kmeans, list-hi, tsp, and memcached. The improvement in intruder comes from serializing the modifications to a global queue, especially an enqueue that occurs near the end of a long transaction (TMdecoder.process). In kmeans, most conflicts take place when updating an array of pointers representing the centers of data clusters. Due to the good locality of conflicting PC and data addresses, Staggered Transactions are able to acquire advisory locks on a per-cluster basis (close to what fine-grain locking could achieve). In list-hi, Staggered Transactions avoid repetitive aborts among several conflicting transactions by locking the entire list (case 2 in Figure 5.6). Note that locking is triggered only when contention actually arises, and transactions that do not contribute to that contention are not blocked. In tsp, Staggered Transactions successfully discover that the head of the priority queue (left-most node of the tree) is the most contended object. Transactions that perform insertions on the same leaf node are also serialized if they repeatedly
Figure 5.7: Performance comparison with 16 threads for 4 different implementations. The Y axis is the performance normalized to Eager HTM.
abort each other. Most conflicts in **memcached** are due to access to global shared statistics, accessed in the middle of transactions. Staggered Transactions introduce significant serialization, but still allow more concurrency than the baseline with fallback to a global lock.

Moderate performance improvements (6–24%) are obtained in **genome**, **list-lo**, and **labyrinth**. In **genome**, the most time-consuming transaction inserts a few elements into a fixed-sized hash table, which ends up overloaded and prone to contention, particularly when a conflict chain is established among several transactions. Although the conflicting PC is associated with the list-traversal code used to access buckets of the table, the Staggered Transactions policy can serialize at the level of the table as a whole via locking promotion (Section 5.5.2), thereby avoiding aborts even in the presence of conflict chains. Two benchmarks (**ssca2** and **vacation**) see no significant improvement in execution time. Even in these, however, Staggered Transactions reduce the frequency of aborts, as shown in Section 5.6.3.

The harmonic mean of performance improvements across all benchmarks is 24%.

**Result 2:** *Staggered Transactions benefit from both partial overlap and a flexible blocking policy.*

In **intruder**, **tsp**, and **memcached**, conflicting data addresses are stable across transaction instances. Here the performance improvements stem simply from serializing the conflicting portions of those transactions, and allowing the remainder to execute in parallel. In the other benchmarks, conflicting data addresses vary greatly; for these, coarse-grain locking and locking promotion are essential to conflict reduction.
5.6.3 Reduced Aborts and Wasted Cycles

**Result 3:** Staggered Transactions reduce contention and wasted CPU cycles for most applications.

For each of our benchmark applications, we compare Staggered Transactions to the baseline HTM with regard to (1) the ratio of aborted to committed transactions and (2) the ratio of wasted to committed work (cycles) (Figure 5.8). Staggered Transactions eliminate up to 89% of the aborts (in intruder) and an average of 64% across the benchmark set (excluding ssc2, which has too few aborts for the numbers to be meaningful). This results in an average savings of 43% of the wasted CPU cycles (a lower number than the savings in abort rate, because aborted transactions typically stop when only part-way through). Assuming that cycles that would have been wasted on aborted transactions are instead devoted either to useful work or to waiting (at relatively low power consumption) for advisory locks, it seems reasonable to expect Staggered Transactions to achieve a significant reduction in energy as well.

5.7 Performance Model on Power 8

As a further step towards real machines, we conducted an analysis of the Staggered Transactions model on IBM’s Power 8 [17, 66]. Although the required nontransactional loads and stores (Section 5.4) can be indirectly implemented using the transaction `suspend` and `resume` mechanism on POWER 8, two factors unfortunately prevent us from repeating the experiment of Section 5.6: (1) the suspend and resume mechanism is too expensive to be considered a realistic implementation of nontransactional loads and stores. A test on our IBM Power System S822L server indicates that an empty `suspend/resume` pair takes 120 ns (~500 cycles), compared to the cost of 35 ns for executing an empty transaction; (2) hardware
Figure 5.8: Reduced aborts and wasted cycles. (a) aborts per commit and (b) ratio of wasted CPU cycles over useful cycles with 16 threads.
buffering capacity for transactional execution is too limited for many benchmarks in Section 5.6. More specifically, each POWER 8 core has only 64 cache lines of speculative state (shared by 8 hardware threads) with which to track transactional reads and writes [66, 89]. Therefore, we use a simplified model, instead of real applications, to study the behavior of Staggered Transactions on POWER 8.

### 5.7.1 Setup

**Benchmark** Due to the above hardware limitations, we model Staggered Transactions via a simple transaction, as shown in Figure 5.9a, where `globalCnt` is a globally shared variable and `delay` is a set of register operations. When the same transaction is running on different cores simultaneously, overlapped execution of the portion after line 4 will lead to conflicts, while the portion before line 4 is conflict-free. By changing the values of `NCONF_LEN` in line 2 and `CONF_LEN` in line 5, we can adjust the length of both portions.

In the corresponding staggered transactions shown in Figure 5.9b, an advisory lock for `globalCnt` is always acquired at line 13, immediately before the initial access of `globalCnt`, so conflicting portions will be serialized at run time.
AcquireLockFor is internally implemented using POWER 8’s suspend and resume mechanism.

We test three different versions of the code:

- **HTM** is a baseline HTM implementation of Figure 5.9a. The runtime retries a transaction up to 12 times, then reverts to a global lock protected software path (irrevocable mode). The runtime performs *early subscription*, reading the global lock at the beginning of each hardware transaction.

- **Lazysub** is the same as HTM except that it preforms *lazy subscription*, reading the global lock only at commit time. As pointed by Dice et al. [35], lazy subscription should be considered unsafe unless necessary hardware support is present. It’s used here as a “future” baseline.

- **Staggered** is the implementation of Figure 5.9b, with early subscription.

For each code version, we use four NCONF_LEN and CONF_LEN combinations such that (1) the length of transaction (i.e., the sum of NCONF_LEN and CONF_LEN) does not change in each combination and (2) the conflicting portion comprises the final 5%, 20%, 50%, and 75% of execution time, respectively. To balance the high overhead of suspend/resume instructions, we choose a relatively large sum of NCONF_LEN and CONF_LEN, so that line 13 takes approximately 5% of the execution time of Figure 5.9b.

**Hardware and Test Configurations** We ran tests on an IBM Power System S822L server, which has 20 cores with 8 hardware threads per core, running at 4.1GHz. Every 5 cores are grouped as a NUMA node.

The code is compiled using gcc 4.9.2 with -O2 optimizations, running on Ubuntu 15.04 Linux with a little-endian ppc64le kernel. Each test starts with a specific number of threads which keep executing the test transaction repeatedly.
for a period of one second. As in Section 5.6, we pin every worker thread to a specific CPU core during program initialization. We use up to 80 threads so cross-NUMA communication occurs with more than 40 threads. Each run was repeated 5 times; the average number is reported.

5.7.2 Results

Scalability results appear in Figure 5.10. For all workloads, Staggered approaches its peak performance much faster than the others, and the number of threads...
under the peak point is close to the theoretical number. For example, in Figure 5.10b, *Staggered* reaches its peak performance with 8 threads (the theoretical number is 5). Moreover, it provides the best performance among the three in most configurations. This result strongly indicates that Staggered Transactions can be a significant aid to good performance on real hardware.

It’s also observable that *Staggered*’s performance, after reaching two NUMA nodes with more than 40 threads, gradually drops as the number of threads increases, especially in workloads with a short conflicting portion. *Staggered* is outperformed by *Lazysub* in Figure 5.10a with more than ~20 threads, and in Figure 5.10b with more than 40 threads. The reason is that all threads in *Stag-
gered compete for the same advisory lock at line 13 of Figure 5.9b, which is currently implemented as a *partitioned ticket lock* [31] with proportional backoff. This suggests that a more advanced design of advisory locks could further improve *Staggered’s* performance under high contention. However, we believe this kind of workload (more than 16 threads waiting at the same advisory locking point, serializing a short region of code) is likely to be rare in real applications.

Figure 5.11 displays the average number of aborts per commit. Obviously, *Staggered* successfully serializes conflicting portions, eliminating almost all conflict aborts in all configurations. This echoes previous results in Section 5.6.3. As a comparison, *Lazysub*’s abort rate increases with the number of threads. By comparing to *Lazysub* across the four sub-figures, we can see that the earlier a conflict appears in a transaction, the less benefit the lazy subscription can provide. It’s also interesting to note in Figure 5.10 and Figure 5.11 that the performance and abort rate of baseline *HTM* often collapse at the point where the thread number is the reciprocal of the length percentage of the conflicting portion of the transaction.

### 5.8 Related Work

HTM systems can be broadly categorized as *eager* or *lazy*, depending on whether conflicts are discovered “as they occur” or only at commit time. A few systems, such as FlexTM [108] and EazyHTM [117], support mixed policies. Most current commercial systems employ a “requester wins” policy in order to avoid changes to the coherence protocol (IBM’s z series machines leverage existing NAK messages to delay aborts, in an attempt to give the victim a chance to commit [58]). A few designs are more sophisticated: LogTM-SE [130], for example, will stall some transactions on conflict; potential deadlock is detected using timestamps in coherence messages. Even a simple conflict manager, however, introduces signifi-
cant implementation challenges because of the necessary protocol extensions and validation cost [107].

Nonetheless, several ambitious solutions have been proposed for eager HTM. In the dependence-aware transactions (DATM) of Ramadan et al. [97], speculative data may be forwarded from transaction $A$ to transaction $B$ if prior accesses have already dictated that $A$ must commit before $B$, and $B$ attempts to read something $A$ has written. In the Wait-n-GoTM of Jafri et al. [59], hardware may generate an exception that prompts the runtime to delay a transaction, if prior experience indicates that upcoming instructions are likely to introduce a circular dependence with some other active transaction. Most recently, Qian et al. proposed Omni-Order [93] to support cycle detection and conflict serialization in a directory-based coherence protocol. RETCON [10], which targets lazy HTMs, tries to “rescue” conflicting transactions by re-executing the conflicting code slice at commit time.

While these hardware proposals may achieve significant reductions in conflict rate, most are specific to a particular class of HTM (e.g., eager or lazy), or are applicable only to certain conflict patterns. FlexTM and DATM, for example, require changes to the cache coherence protocol. RETCON can resolve only simple conflicts such as counter increment. Wait-n-GoTM requires the underlying TM to be version-based, and the centralized predictor tends to be a bottleneck.

Staggered Transactions share the “stall before encountering contention” philosophy of systems like LogTM-SE and Wait-n-GoTM. Because Staggered Transactions are implemented principally in software, however, they are not bound to any particular style of HTM or conflict resolution strategy. The required support, we believe, could be added easily to existing hardware. More significantly, Staggered Transactions’ use of high-level program knowledge allows them to resolve contention patterns that are unlikely to be captured by a pure hardware solution (e.g., conflicts in a data structure with no stable set of conflicting data addresses).

Contention management has also been a subject of active research in STM sys-
tems, where the flexibility of software and the high baseline overhead of instrumentation can justify even very complex policies. While much early work [105, 111] served mainly to recover from contention once it happened, several projects have aimed to avoid contention proactively. Multi-version STM, pioneered by Riegel et al. [99], significantly reduces contention by allowing a transaction to “commit in the past” if it has not written any location that was read by a subsequent transaction. Later work by the same authors [101] uses Data Structure Analysis [64] to partition shared data and choose a potentially different STM algorithm or locking policy for each partition. Chakrabarti et al. [18] use a profiling-based abort information graph to identify data dependences and optimize STM policy. Given their reliance on software instrumentation, none of these techniques are compatible with existing HTM, and all would be difficult to integrate into future hardware designs.

In a manner less dependent on TM system details, contention can sometimes be avoided by carefully scheduling the threads that run conflicting transactions. Proactive Transaction Scheduling [9] learns from repeated aborts and predicts future contention. The scheduler uses the prediction to serialize entire transactions when they are likely to conflict with one another. In comparison to such techniques, Staggered Transactions avoid the overhead of scheduling decisions, thereby avoiding any negative impact on the performance of short transactions. Also, by serializing only the conflicting portions of transactions, Staggered Transactions can achieve more parallelism.

5.9 Summary

We have presented an automatic mechanism, Staggered Transactions, to serialize the conflicting portions of hardware transactions, thereby reducing aborts. Our technique employs compile-time Data Structure Analysis to understand program
data, allowing us to accommodate a wide variety of conflict patterns. While the choice of which lock to acquire is always based on the address of the data being accessed, the decision as to whether to acquire a lock, and at what instruction address, is made adaptively at run time.

From the hardware, Staggered Transactions require the ability to acquire an advisory lock from within an active transaction; they also benefit from a mechanism to recall the program counter of the initial speculative access to a given conflicting location. Experiments on the MARSSx86 ASF simulator demonstrate speedups averaging 24% on a collection of 9 TM applications.


## 6 Conclusion and Future Directions

With the maturity of multicore hardware, the language and runtime support for high performance multithreaded programming has drawn a lot of attention from academia and industry. In this thesis, we focus on two scenarios where the scalability of multithreaded programs is likely limited by synchronization performance: 1) lock-based concurrent data structures; 2) transactional memory. We demonstrate that in both scenarios, compiler-assisted speculation can significantly improve the performance of synchronization.

The major contributions of this work are summarized in the following section. A discussion of future directions appears in Section 6.2.

### 6.1 Contributions

In Chapter 3, using several different concurrent data structures as examples, we showed that manual addition of speculation to traditional lock-based code can lead to significant performance improvements. Successful speculation requires careful consideration of profitability, and of how and when to validate consistency. Unfortunately, it also requires substantial modifications to code structure and a deep understanding of the memory model. These latter requirements make it difficult
to use in its purely manual form, even for expert programmers. To simplify the process, we present a compiler tool, CSpec, that automatically generates speculative code from baseline lock-based code with user annotations. Compiler-aided manual speculation keeps the original code structure for better readability and maintenance, while providing the flexibility to chose speculation and validation strategies.

In Chapter 4, we described a programming technique and compiler support to reduce both overflow and conflict rates of HTM by partitioning common operations into read-mostly (planning) and write-mostly (completion) operations, which then execute separately. The completion operation remains transactional; planning can often occur in ordinary code. High-level (semantic) atomicity for the overall operation is ensured by passing an application-specific validator object between planning and completion. Transparent composition of partitioned operations is made possible through fully-automated compiler support, which migrates all planning operations out of the parent transaction while respecting all program data flow and dependences. For both micro- and macro-benchmarks, experiments on IBM z-Series and Intel Haswell machines demonstrate that partitioning can lead to dramatically lower abort rates and higher scalability.

In Chapter 5, we developed a novel hybrid optimistic/pessimistic execution model to reduce the incidence of conflict in HTM. The model uses advisory locks to serialize (just) the portions of the transactions in which conflicting accesses occur. We demonstrated the feasibility of this mechanism with fully developed compiler and runtime support, running on simulated hardware. Our compiler identifies and instruments a small subset of the accesses in each transaction, which it determines, statically, are likely to constitute initial accesses to shared locations. At run time, the instrumentation acquires an advisory lock on the accessed datum, if (and only if) prior execution history suggests that the datum—or locations “downstream” of it—are indeed a likely source of conflict. Policy to drive the
decision requires one hardware feature not generally found in current commercial offerings: nontransactional loads and stores within transactions. It can also benefit from a mechanism to record the program counter at which a cache line was first accessed in a transaction.

6.2 Future Directions

CSpec as An Uniform Interface of Speculation  In Chapter3, CSpec was primarily considered as an alternative to locking. In fact, the introduction of CSpec enables at least four different implementation options of a concurrent method in concurrent data structures: pure lock-based method (no speculation), CSpec with lock-based modification phase, CSpec with HTM-based modification phase, and pure HTM method. Because of the variety of data layouts and the workload diversity, it is difficult to figure out, at coding time, whether speculation will be profitable or not, and which of the four implementations will deliver the best performance.

Fortunately, CSpec provides a uniform language interface of all options. Therefore, we suggest to use it as a language-level speculation mechanism— lower-level issues such as static code generation for different implementations and adaptive switches between them can be left to the compiler.

Annotations in CSpec carry useful semantic hints about a concurrent operation, such as the boundary of speculative work, and the locking and validation granularities. These hints, along with knowledge that could be learned through other analysis techniques (e.g., points-to set analysis [113]), can help the compiler better understand the speculation behavior, and make a preliminary strategy decision at compilation time. For example, the \texttt{Sum} method of equivalence sets (Figure 3.5) validates with per-set version numbers. The compiler thus knows the speculation works at the same granularity (per-set) as the \texttt{Sum} method itself. In
the \texttt{Move} method, the speculation also works at set granularity but the validation occurs at a finer per-node granularity. Therefore, CSpec is not necessarily better than HTM for \texttt{Sum}, but could be better than HTM during the speculative phase of \texttt{Move}. Similarly, the critical section of the CSpec \texttt{Move} method acquires a set-level lock, but only four elements are modified in the critical section. So the compiler can infer that HTM may be better than locking for CSpec’s modification phase. Static knowledge not only lets the compiler skip the code generation of options that are unlikely to be better than the others, but also helps to construct a highly efficient run-time switch policy for a specific data structure.

\textbf{Automatic Verification and Creation of Speculation Code} CSpec, as a design pattern, can of course be enhanced by a richer set of annotations and more sophisticated translation mechanisms, to support new features such as nested speculation. But it is still a semi-automatic method: notably, the most intelligent and challenging part is done by the programmer. Therefore, in Chapter 3, we limited its use to a rather narrow scope—concurrent data structures—to simply the analysis and to maximize the potential profit. To extend CSpec’s application to more general situations, we believe the following two directions are worth exploring.

First, it remains a manual task to reason about whether a speculative algorithm specified by annotations is correct or not. As CSpec annotations already carry some high-level program information, we hope an advanced compiler analysis, by absorbing this knowledge and possibly with the help of a few extra “verification” annotations, could automatically verify the correctness of a CSpec speculation algorithm. Again, critical sections in concurrent data structures seem to be a reasonable starting point of the analysis technique due to their relatively simple code logic.

Second, the technical boundary can be further pushed towards full compiler
automation: the compiler could independently figure out the speculation region and create validation code. A major challenge, however, seems to be the lack of precise alias analysis and data flow analysis.

Similar verification and automatic generation techniques may also be applicable to ParT in Chapter 4, particularly in the creation of a ParT library. For example, the compiler could check a specific partitioning against the correctness rules in Section 4.2.2.

**Better Handling of Hardware Speculation Failure** Speculation may fail due to data conflict. The handling of speculation failure has a great impact on performance. Instead of simply re-executing the entire transaction, partial rollback might try to redo only the part that has been affected by the conflict, and thus minimize redundant computation.

ParT in Chapter 4 supports partial rollback on existing HTMs. Currently, the completion phase of a ParT transaction, after an abort, will fully re-execute the completion operations of those failed speculative operations inside a hardware transaction. This increases the chance of overflow for that hardware transaction, especially when more than one speculative operation fails at the same time. Research on better rollback policies may further improve ParT’s performance under high data contention.

**Compiler Optimization of Transactional Code** Other than better STM instrumentations [25, 74, 75, 101] and uses in this thesis, the compiler can also apply special optimizations to transactional code.

One example optimization is code reorganization that minimizes the length of the “vulnerability window”, during which a transaction is likely to conflict with others. Different data accesses of a transaction may have different chances to trigger conflicts. As noticed by Blundell et al. [10] and as shown in Chapter 5,
many data conflicts happen on just few memory locations. We call these “risky accesses”: at the source code level, they are often global variables and statistics information. General speaking, the position where risky accesses appear defines the length of the vulnerability window, which further determines the scalability. Ideally, risky accesses should be positioned at the end of a transaction, for both STM and HTM, as suggested by the results in Section 5.7.

Given an operation that is possibly called inside a transaction, the programmer can of course intentionally rearrange the code in favor of a shorter vulnerability window. However, the composition of operations will render the manual optimization useless. On the other hand, the compiler always sees the whole transaction. So it can globally optimize a transaction by clustering and then delaying risky accesses and their forward slices [125], as long as it knows where they are. Such knowledge can be gained through profiling runs. Moreover, the optimization can be performed even at run time if JIT compilation is available.
Bibliography


Dependable Multicore and Transactional Memory Systems, DMTM, Vienna Austria, January 2014.


[95] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference...*


[102] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing Hybrid Transactional Memory: the Importance of Non-


[117] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-


[129] Lingxiang Xiang and Michael L. Scott. MSpec: A Design Pattern for Con-
