Concurrent Algorithms for Emerging Hardware Platforms

by

Irina Calciu

B.Sc., Jacobs University Bremen; Bremen, Germany, 2009

M.Sc., Brown University; Providence, RI, 2011

A dissertation submitted in partial fulfillment of the

requirements for the degree of Doctor of Philosophy

in The Department of Computer Science at Brown University

PROVIDENCE, RHODE ISLAND

May 2015

This dissertation by Irina Calciu is accepted in its present form
by The Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date_____ _____

Maurice Herlihy, Ph.D., Advisor

Recommended to the Graduate Council

Date_____ _____

Justin Gottschlich, Ph.D., Co-advisor

Date_____ _____

Rodrigo Fonseca, Ph.D., Reader

Approved by the Graduate Council

Date_____ _____

Peter M. Weber, Dean of the Graduate School

Vita

Irina Calciu was born and raised in Bucharest, Romania. She became interested in Computer Science at an early age and participated in various programming competitions during high-school. In 2006 she moved to Bremen, Germany, to pursue a B.Sc. in Computer Science at Jacobs University Bremen. In the fall of 2008, she was an exchange student in the School of Computer Science at Carnegie Mellon University. Irina joined Brown University as a PhD student in 2009. She obtained a M.Sc. in Computer Science from Brown in 2011 and a PhD in Computer Science in 2015. Her research focuses on designing algorithms that leverage new architectural features of modern hardware to enable more software parallelism. In particular, she is interested in hybrid transactional memory and in non-uniform memory access (NUMA) algorithms. Irina has co-authored papers at top conferences and workshops, such as PACT, PPoPP, DISC, OPODIS, HotPar and TRANSACT, obtaining a Best Paper Award for work on software fallbacks for best-effort hardware transactional memory at TRANSACT 2014. She has been awarded a Kanellakis Fellowship in 2014 and a Dissertation Fellowship in 2013 and she has been a research intern at Microsoft Research, Intel Labs, Oracle Labs, Google and Mozilla.

Acknowledgements

"It's always easier to ask forgiveness than it is to get permission."

Grace Murray Hopper

I am incredibly lucky to have had amazing people accompany me through my PhD journey. This thesis exists because of their support and guidance.

I am very grateful to my advisor, Maurice Herlihy, who gave me independence before I even knew I wanted it. He inspired and motivated me. He found the time to listen, to help and to provide feedback whenever I needed it. Yet, he also gave me the confidence to fly on my own and to carve my own path. He taught me to be bold in my pursuits and not to ask for permission. It was a great experience to have him as an advisor!

I am forever indebted to my co-advisor, Justin Gottschlich. Justin was my mentor during an internship with Intel Labs and he remained my mentor even after the internship ended. He insisted that I can write better papers and patents and that I can make better visuals for my presentations. He trusted me more than I trusted myself. Today, I am a better writer, presenter, researcher and a better person because of his trust.

I am thankful to Rodrigo Fonseca, who was a member of my thesis committee, and the other faculty at Brown, in particular Sorin Istrail, Ugur Cetintemel and Shriram Krishnamurthi. They have all provided me invaluable guidance and advice in this journey and I do not have enough words to thank them for it.

I had the great pleasure to work with amazing researchers during my internships. I want to thank Mark Moir and the Scalable Synchronization group for teaching me about NUMA machines and systems research. I am grateful to the Programming Systems Lab at Intel for offering me the opportunity to work on Hybrid Transactional Memory on Haswell before anyone else could. I am also thankful to Konrad Lai and Andi Kleen. They spared much pain by providing support and tools to navigate the prototype hardware. It was a great honor to work with Marcos Aguilera, Mahesh Balakrishnan, Rama Ramasubramanian and Sid Sen during two internships at Microsoft Research. I am excited to continue working with them at VMware Research Group. I am also grateful to the interns with whom I shared many special moments in the Bay Area - Fangbo, Ilya, Jana, Mohsen, Rajiv, Tobias, and Tomas. Special thanks go to Yehuda Afek, who provided access to the machines used for many experiments in this thesis.

I wouldn't be here today without the love and support of my family back in Romania. They have always cheered for me and they have forgiven my absence from so many family holidays. I am grateful to my aunt and godmother, Rodica Ristea, who was the first to show me the world outside my country. I am also thankful to my aunts *tanti Miti* and *tanti Anda*, to my uncle *Nenicu*, and to my cousins, Catalina, Dani, Bogdan and Doru. Moreover, I was incredibly lucky to find a second family in Minnesota. They were kind to make me feel part of the family. Their encouragement during these last few years was unmatched. I am thankful to Theresa and Joe Berg, Rita and Kyle Johansen, Jenny Berg, and aunts Dorothy and Lisa.

My journey would have not been the same without my amazing family at Brown. Jenny and Nathan were my first officemates. They taught me to navigate the intricacies of the PhD and prepared me for every next step in the process. I shared the ups and downs of the PhD life with my good friends, Jeff, Shane, Yuri, Hammurabi

and Carrie, Marcelo and Yoko, Alex and Tanya, Steve and Alyssa, Oana and Igor. My friend Marquita has been my research and my gym buddy. Archita, Vikram and Zhiyu have been great academic siblings. Although we haven't had a chance to collaborate so far, I hope we will do so in the future. I have enjoyed countless cupcakes and life discussions with my GWICS friends - Alexandra, Betsy, Esha, Foteini, Gen, Hannah, Layla, Olya, Rebecca, Sasha, and Silvia. I have spent many late hours in the CIT with Deepak, Greg and Sunil early on during my PhD. I am thankful to Lauren Clarke for always having a solution to all problems and to Genie DeGouveia for always having a key to my office when I locked myself out before a deadline. I am grateful to my friends in Providence. Robin Feder provided a great introduction to Providence and the American culture. Charlie and Michael have been great neighbors and have organized unforgettable get-togethers.

Elena, Alin and Andrei have been very far, but also very close all these years. We have all experienced together the PhD adventure, even though from very different places. I am thankful to Elena for sharing all the joys and sorrows of the PhD life. Thank you for making all the roadblocks seem easier to navigate!

I owe all my achievements and success to my wonderful mother and closest confidant, Germina Ristea. Thank you for being my idol and inspiration! You magically managed to strike the perfect balance between giving me the freedom to explore the world while also keeping the bar high for me. You taught me to work hard and to see failures as a learning experience. You gave me all the skills I needed to succeed in a PhD program, perhaps without even knowing it.

My fiancé and best friend, Daniel Berg, has been my biggest supporter and my biggest critic during all these years. Thank you for believing in me and for not letting me settle for anything less than the best! Your passion and dedication to research continue to inspire me every day.

Abstract of " Concurrent Algorithms for Emerging Hardware Platforms " by Irina Calciu, Ph.D., Brown University, May 2015

Computer architecture has recently seen an explosion of innovation that has enabled more parallel execution, while parallel software systems have been making strides in providing more simplified programming models. The number of computing cores used in every area of the software ecosystem continues to increase, and parallelism within programs is now ubiquitous. Ideally, performance would scale linearly with the number of cores, but that is rarely the case in practice. Communication and synchronization between cores running the same application are often necessary, but usually come at a high cost. This results in reduced scalability and a significant drop in performance. In this context, parallel software needs to provide more simplified programming patterns and tools that enable a higher potential for parallelism without increasing the burden on the programmer.

This thesis discusses new techniques to simplify writing efficient parallel code by leveraging novel architectural features from many current systems. First, we describe various programming abstractions, such as delegation, elimination, combining and transactional memory, which improve scalability and performance of concurrent programs. Next, we show how to use and integrate these abstractions to write scalable concurrent algorithms, such as stacks and priority queues. Finally, we describe how to further improve these abstractions. In particular, we present new transactional memory algorithms that use Intel's new extension to the x86 instruction set architecture, called Restricted Transactional Memory, to simplify general synchronization. Developers can use all of these abstractions as building blocks to create efficient code that is able to scale on very diverse platforms, with minimal specialized knowledge of parallel programming.

Contents

List of Figures

Chapter 1

Introduction

This thesis discusses new techniques to simplify writing efficient parallel code that leverage architectural features of current systems. We focus on a few design patterns, such as elimination, delegation, combining and transactional memory. These techniques promise to simplify writing parallel code and improve scalability in scenarios with high contention. We describe how to use these techniques and integrate them to design scalable concurrent algorithms. First, we show how to use delegation and elimination to implement a scalable concurrent stack suitable for the Non-Uniform Memory Access (NUMA) architecture from a sequential stack. Next, we present the first elimination algorithm for a priority queue and describe how to integrate this algorithm with delegation and transactional memory to design a scalable concurrent priority queue. Finally, we describe two hybrid transactional memory (HyTM) algorithms that use Intel's Restricted Transactional Memory (RTM) to simplify general synchronization. These techniques make parallel programming simpler and more efficient and are suitable for the rapidly evolving hardware ecosystem. They represent a foundation for building large-scale concurrent systems that may be suitable to address wide-interest problems.

1.1 Motivation

The landscape of Computer Science is fundamentally changing. For a long time, Moore's law ensured that performance would increase with each new CPU iteration. But the "free ride" is over and the demand for faster computation is now satisfied through parallelism. A boom in hardware innovation is enabling more concurrency. Therefore, server machines with hundreds of cores are becoming ubiquitous. Ideally, performance would increase linearly with the number of cores, but that is rarely the case in practice. The culprits are communication and synchronization between cores running the same application. These are often necessary, and usually come at a high cost, causing a loss of scalability and reduced performance. In order to leverage the huge potential of these emerging hardware platforms, we need better synchronization methods and updated parallel programming abstractions.

Moreover, as computer architectures are changing and growing to accommodate more cores, the connecting bus is becoming the limiting factor to how many cores a system can accommodate. To circumvent these issues, machines are progressively adopting the non-uniform memory access (NUMA) model, where each processor (also called a node) has its own memory. Multiple cores are grouped on a node and share a last level cache. Although all the memory is shared, a thread running on a node can access local memory (on the same node) faster than it can access remote memory (on another node). Different access times and cache-to-cache traffic can significantly impact the performance of applications unaware of this non-uniformity. As these machines are becoming critical components in data centers, it is essential to provide software building blocks for developing efficient parallel applications on these platforms.

Meanwhile, software does not seem to leverage the potential for increased parallelism. Shared data needs to be protected from simultaneous access by multiple threads. The

primary mechanism to ensure mutually exclusive access to shared data currently in use is locking. Nevertheless, fine-grained locking is complex and prone to errors, while coarse-grained locking can impact scalability. Moreover, locks are not composable, which means that multiple critical sections cannot be combined together into one, which affects the code's modularity. Locks can also cause priority inversions and deadlocks, which are difficult to detect and recover from. For these reasons, locking is not an ideal solution for synchronization, especially on NUMA machines with hundreds of cores.

Transactional memory (TM) has been proposed to abstract away the complexity of lock-based mutual exclusion while providing benefits comparable to fine-grained locking. Moreover, transactional memory eliminates the negative side-effects of locking, such as deadlocks and priority inversions. Transactional Memory executes critical sections speculatively, as transactions, tracking all memory accesses and restarting or stalling one or more transactions if it detects a conflict. Transactional Memory can enable more parallelism by allowing critical sections to execute concurrently as long as there are no data conflicts between them.

For example, two threads that insert elements into different buckets of a hash table can execute in parallel as they do not have any data conflicts. If using coarse grained-locking, these threads would have to acquire a lock on the hash table before doing the insert. Therefore, they would not be able to execute in parallel. However, if these threads were using fine-grained locking, by each thread locking only its corresponding bucket, both threads could proceed in parallel.

Nevertheless, designing fine-grained synchronization is a bigger undertaking than using a coarse-grained lock and it is more prone to programming errors [44, 33]. Instead, one may be able to achieve the efficiency of fine-grained locking with the programming simplicity of coarse-grained locking by using transactional memory. In

the previous example, both threads can perform the inserts as transactions. If a conflict is detected, one of the transactions needs to roll back and retry.

Software transactional memory (STM) is implemented in software only. STM is most effective when used in applications with large, contended critical sections, where smart contention managers can efficiently manage transactions to obtain the best throughput. Unfortunately, keeping track of all memory accesses in software generally incurs a prohibitive overhead for short critical sections. For these, hardware transactional memory (HTM) has proven more feasible [17, 24, 66]. HTMs have recently become available in Intel's Haswell processor [48, 49] and IBM's Blue Gene/Q [89] and System z [51]. Practical HTMs, such as those provided by Haswell and Blue Gene/Q are best-effort, which means there are no forward progress guarantees. In particular, hardware transactions are restricted from using certain unsupported instructions and are limited in size. Therefore, a fallback is needed to ensure forward progress of hardware transactions.

In practice, a single global lock (SGL) is often used as a fallback to an aborted hardware transaction. The SGL is similar to Intel's Hardware Lock Elision (HLE) technique, used for legacy code, where existent locks are elided and the critical sections are executed as transactions. If a transaction aborts, the hardware acquires the locks that were previously elided and executes pessimistically. However, the SGL prevents any concurrency while the lock is being held.

Hybrid transactional memory (HyTM) [19] combines lightweight hardware transactions with the forward progress guarantees offered by software transactions, while also offering more flexibility for transaction and contention management. Therefore, HyTMs represent a complete solution for the problem of synchronization in shared memory. Although current consumer systems supporting HTM are limited to four cores (eight hardware threads using hyperthreading), we believe the next genera-

tion architectures may eventually offer support for HTM on machines with hundreds of cores, thus making Transactional Memory a viable and portable synchronization solution.

As more parallel architectures emerge, wide-scale adoption of parallel programming across multiple disciplines may be possible. However, the programming paradigm needs to be greatly simplified, as it is with Transactional Memory, or parallel programming is likely to remain a restricted "experts-only" domain.

1.2 Outline

In this thesis, we explore design patterns and abstractions that leverage novel hardware features to improve the scalability and performance of concurrent programs and to simplify writing parallel code.

In particular, we explore elimination [39], delegation [64], combining [38, 64, 54, 86, 6, 27, 74] and transactional memory [43, 84] and we propose new ways to use and integrate these abstractions to design new scalable concurrent algorithms. We present new designs for concurrent stacks and priority queues and analyze their performance and scalability benefits compared to prior work. Next, we propose new ways to further improve these abstractions. We focus on transactional memory and propose new fallback algorithms to be used in conjunction with Intel's new Restricted Transactional Memory instructions [49] to provide forward progress guarantees.

This thesis is organized as follows.

In Chapter 2, we describe related work. We focus on various abstractions that have been proposed in the concurrent computing community, such as elimination, delegation, combining and transactional memory. Elimination consists of canceling out inverse operations. For example, a thread that executes a push operation on a stack

can eliminate its operation with another thread executing a pop operation on the same stack. The delegation method consists of one dedicated thread, called a server, which is responsible for managing a sequential data structure and executing operations on behalf of other threads, called clients. Clients post synchronous (blocking) operation requests in dedicated memory locations, called slots, and the server loops through these slots, collects all operations and executes them on the data structure. The server is the only thread able to access the data structure, so it does not need any synchronization for the access. Combining is similar to delegation, but there is no dedicated server thread. Operations are performed by one of the clients, the one that acquires the combiner lock. Combining and delegation can reduce cache-to-cache traffic by allowing one thread to execute multiple operations. Moreover, some operations can be executed more efficiently as a batch, allowing the combiner or the server to achieve more throughput with less work. For example, removing multiple consecutive items in a sorted linked list can be executed at once with the cost of a single operation by a server or combiner thread, while it would take multiple operations if each operation was executed separately by the client threads. Transactional memory has been proposed as a general synchronization method and allows the critical sections to execute speculatively. In case of conflicts, where multiple transactions access the same data, one of the conflicting transactions needs to be stalled or aborted and retried at a later time.

In Chapter 3, we describe how to use elimination and delegation to design a scalable NUMA-aware stack. In our design, clients use elimination before delegating their requests in order to reduce the burden on the server thread and to parallelize mixed workloads in which operations cancel each other out. Moreover, we experiment with placing the elimination layer locally, on each NUMA node and globally - where it is shared between NUMA nodes. We show that there is significant benefit from using local elimination and delegation together, by comparing to state-of-the-art

concurrent stack implementations and to global-elimination based stacks.

In Chapter 4, we describe a scalable concurrent priority queue design. We present the first elimination algorithm for a priority queue and show how to integrate this algorithm with delegation, combining and transactional memory to achieve a highly scalable design. Our algorithm is based on the observation that high-value *add()* operations should execute in parallel for high scalability, while *removeMin()* operations should be executed by a combiner to avoid contention on the smallest items in the priority queue. Moreover, we noticed that small value *add()* operations can either eliminate immediately, if their values are smaller than the priority queue minimum, or they can quickly become eligible for elimination, if they are likely to conflict with the *removeMin()* operation. Therefore, we allow these operations to attempt elimination before accessing the shared priority queue. Moreover, in order to reduce contention, we use a dedicated server thread that collects operations that failed to eliminate and executes them sequentially on the priority queue. We show that our design is more scalable and performs better than state-of-the-art priority queue implementations.

In Chapter 5, we describe improvements to the simple, but widely used, Single Global Lock (SGL) fallback mechanism to ensure forward progress for best-effort hardware transactional memory. First, we present Lazy Single Global Lock (L-SGL), a simple optimization that can achieve surprising benefits. Its simplicity, combined with high throughput on current Haswell machines, make L-SGL likely to be adopted by industry and implemented as a software library, or even in the compiler or hardware. Next, we describe how to refine conflict detection between multiple hardware transactions and a single software SGL transaction using Bloom Filters. Finally, we discuss how implementing these features in hardware would improve performance even further.

In Chapter 6, we present Invyswell, a novel hybrid transactional memory algorithm that uses Haswell RTM for hardware transactions. Invyswell, is more complex than L-SGL and uses InvalSTM [30] software transactions as a fallback mechanism for Haswell. This algorithm pays a penalty for its complexity at low thread counts, but we anticipate that it will be more scalable than L-SGL on machines with more cores. L-SGL and Invyswell are not meant to be compared. Rather, we believe they are complementary. L-SGL can be provided as an out-of-the-box hardware solution for simple application that do not utilize many hardware resources, while Invyswell can provide added benefit in software to highly parallel applications with tens or hundreds of threads.

Finally, Chapter 7 provides concluding remarks.

1.3 Contributions

Related papers published:

1. **Chapter 3.** *Using Elimination and Delegation to Implement a Scalable, NUMA-Friendly Stack*, **I. Calciu**, J. Gottschlich, M. Herlihy, HotPar 2013 [11].

2. **Chapter 4.** *The Adaptive Priority Queue with Elimination and Combining*, **I. Calciu**, H. Mendes, M. Herlihy, DISC 2014 [13].

3. **Chapter 5.** *Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory*, **I. Calciu**, T. Shpeisman, G. Pokam, M. Herlihy, Transact 2014 [14].

4. **Chapter 6.** *Invyswell, A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory*, **I. Calciu**, J. Gottschlich, T. Shpeisman, G. Pokam, M. Herlihy, PACT 2014 [12].

Other publications:

1. *Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores*, **I. Calciu**, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, M. Moir, OPODIS 2013 [9].

2. *NUMA-Aware Reader-Writer Locks*, **I. Calciu**, D. Dice, Y. Lev, V. Luchangco, V. Marathe, N. Shavit, PPoPP 2013 [10].

Chapter 2

Design Patterns and Abstractions for Concurrent Algorithms

There have been many proposals for general techniques to design and analyze concurrent algorithms [20, 21]. In addition, specific concurrent data structures designs have also been proposed, such as Stacks [65, 82], Queues [71, 37, 65], Deques [41, 55, 60], Trees [4, 34, 53, 57, 73, 72] and Priority Queues[87, 3, 40, 46, 50]. In this chapter we survey prior work related to the main techniques used in this thesis. First, we describe different notions related to concurrent data structures. Next, we present techniques such as elimination, delegation and combining. Finally, we survey the literature related to transactional memory.

2.1 Concurrent Data Structures

Concurrent data structures are quickly gaining importance, as multicore machines are becoming ubiquitous. The building blocks for designing concurrent data structures generally consist of locks and atomic primitives to ensure the safety of all

the shared data accesses. The most commonly used atomic primitive is *Compare-And-Swap (CAS)*, which is supported in most current processors. A CAS operation consists of atomically changing a memory location from a known old value to a new value, only if the memory location has not been updated in the meantime. CAS operations are often used to design lock-free algorithms. In addition, they are also used in designing efficient blocking synchronization techniques.

However, designing concurrent data structures is generally difficult [68, 83] because the many possible interleavings of different threads can cause different outcomes. Therefore, concurrent algorithms need to account for non-deterministic threads schedules and always produce the expected outcomes.

Linearizability [45] is the most commonly used correctness condition for data structures. Linearizability requires that all operations appear to take place instantaneously, at some moment in time between the invocation of the operation and the response. This means that all operations on the shared data structure can be ordered so that the result is equivalent to a sequential execution of these operations. In addition, linearizability enforces that this order reflects the real order of these operations. Therefore, concurrent operations, i.e. those operations whose executions overlap, can be re-ordered, but operations whose executions do not overlap cannot be re-ordered. The moment at which the operation appears to take place is called *linearization point*. Linearizability is *compsable*, which means that a data structure that is created out of multiple linearizable parts, is linearizable. In this thesis, we focus on linearizable designs of concurrent data structures.

2.2 Elimination

Stacks are generally seen as sequential data structures because all threads contend for access to the stack at its top location. However, prior work has shown that stacks can be parallelized using a technique called elimination [39]. This technique uses an additional data structure to allow threads performing push operations to meet threads performing pop operations and exchange their arguments. This is equivalent to the push being executed on the stack and immediately followed by a pop. The elimination data structure, generally implemented as an array, allows multiple such pairs to exchange arguments in parallel and decreases contention on the underlying lock-free stack. If one thread fails to find an inverse operation, then its elimination attempt times out and the thread accesses the stack directly.

This technique can be used as a backoff mechanism to a lock-free stack. A thread can first try to perform its operation on the lock free stack using a CAS operation and only use the elimination array if the CAS fails. Using elimination as a backoff mechanism allows the throughput to be significantly increased in high contention cases, without affecting latency of operations in cases where there is not much contention.

If the original stack design is linearizable, the resulting stack design using the elimination method is also linearizable. As described in section 2.1, concurrent operations can be reordered. Therefore, operations that perform elimination concurrently can be reordered to appear that each push operation was immediately followed by its eliminating pop operation.

The rendezvous method [2] improves the elimination algorithm by replacing the elimination array with a smarter structure for processing the elimination, consisting of an adaptive circular ring.

Elimination is generally used in the context of stacks, but an elimination algorithm for queues has also been proposed [67]. The main idea behind the queue elimination is to allow threads that fail to enqueue to linger for some time in the elimination array, until they become eligible to eliminate. This process is called *aging* the operation. The enqueue operation becomes eligible to eliminate with a dequeue operation when all the items that have been enqueued before the start of the enqueue operation have already been dequeued. The operation *aging* process is necessary for linearizability. If any enqueue operation would be allowed to eliminate, the First-In-First-Out property of the queue would not be satisfied. We use this idea as a basis for our priority queue elimination algorithm described in Chapter 4.

2.3 Combining and Delegation

The idea of one thread helping other threads to complete their work is a well-known concept [38, 64, 54, 86, 6, 27, 74, 40]. A recent example of this *helping* mechanism is called flat combining [38], in which a thread that acquires a lock for a data structure executes operations for itself and also for other threads that are waiting on the same lock. The global lock and the data remain in this thread's cache while it executes operations on behalf of other threads, thereby decreasing the number of cache misses and contention on the lock. Moreover, flat combining aligns well for data structures that are sequential, because only one thread is able to operate on it at a time, regardless.

Due to the increasing number of hardware threads in a system, the helper thread could be a dedicated thread (called a server thread) used only to service requests from other threads (client threads). This is especially useful on heterogeneous architectures, where some cores could be faster than others. An example of this approach

is CPHash [64], a partitioned hash table. Each partition has an associated server thread that receives add and remove requests from clients and sends back the responses obtained from performing the operations requested. Each client-server pair share a location where they exchange messages, called a communication channel. In [9], Calciu et al. investigate the tradeoffs between the traditional shared memory techniques and a message passing approach based on delegation.

2.4 Transactional Memory

Transactional memory (TM) systems [36] fall into three rough categories: *software* (STM), *hardware* (HTM), and *hybrid* (HyTM). Most of the research literature concerns itself with STM systems [84, 1, 25, 30, 42, 61, 76, 79]. In this thesis, we compare our HyTM design to NOrec [18], a state-of-the-art STM that uses value-based validation, deferred updates and lazy conflict detection.

Early HTM research was limited to simulation [35, 69]. Early implementations include Azul's Vega [16] and Sun's Rock [22], though neither became widely available.

Recently, however, Intel [49] and IBM [89, 51, 8] announced new processors with hardware support for transactions, and it seems likely that others will follow. Like Herlihy and Moss's original TM proposal [43], these systems rely on modified cache coherence protocols to achieve atomicity and isolation. Haswell also supports *hardware lock elision* [75, 48], a scheme where annotated lock-based critical sections are executed speculatively, but are retried pessimistically if speculation fails. We restrict the evaluation in this thesis to Intel's Haswell transactional memory instructions, called Restricted Transactional Memory.

HyTM schemes promise to provide the best of both worlds: the efficiency of HTM

with the scalability and forward progress guarantees of STM. The first papers to articulate this point are from Damron et al. [19] and Kumar et al. [56]. Later work in this area includes PhTM [58], intended for Sun's Rock architecture and Riegel et al.'s work [77] intended for AMD's proposed Advanced Synchronization Facility (ASF). Dalessandro et al. [17] proposed Hybrid NOrec, a HyTM based on NOrec STM. We compare our HyTM algorithm with Hybrid NOrec in Chapter 6. Matveev and Shavit [62, 63] describe a new type of HyTM based on *reduced hardware transactions*: HTM is used not only for the hardware transactions, but also for making the software transactions more efficient. More recently, Wang et al. [89] proposed a HyTM for IBM Blue Gene/Q's best-effort HTM, based on a Single-Global-Lock fallback.

Our HyTM algorithms, like any other TM, must address the problem of how to most efficiently resolve conflicts between transactions, a problem known as *contention management (CM)* [31, 80, 28, 85].

Chapter 3

A Concurrent NUMA-Aware Stack

Emerging cache-coherent non-uniform memory access (cc-NUMA) architectures provide cache coherence across hundreds of cores. These architectures change how applications perform: while local memory accesses can be fast, remote memory accesses suffer from high access times and increased interconnect contention. Because of these costs, performance of legacy code on NUMA systems is often worse than their uniform memory counterparts despite the potential for increased parallelism.

In this chapter, we explore these effects on prior implementations of concurrent stacks and propose the first NUMA-aware stack design that improves data locality and minimizes interconnect contention. We achieve this by using a dedicated server thread that performs all operations requested by the client threads. Data is kept in the cache local to the server thread thereby restricting cache-to-cache traffic to messages exchanged between the clients and the server. In addition, we match reciprocal operations (pushes and pops) by using the rendezvous algorithm [2], an improved elimination algorithm, before sending requests to the server. This has the

dual effect of avoiding unnecessary interconnect traffic and reducing the number of operations that change the data structure. The result of combining elimination and delegation is a scalable stack that outperforms all previous stack implementations on NUMA systems.

3.1 Background

The current trend in computer architecture is to increase system performance by adding more cores so that more work can be done simultaneously. In order to enable systems to scale to hundreds of cores, the main hardware vendors are switching to non-uniform memory access (NUMA) architectures. Recent examples include Intel's Nehalem family and the SPARC Niagara line.

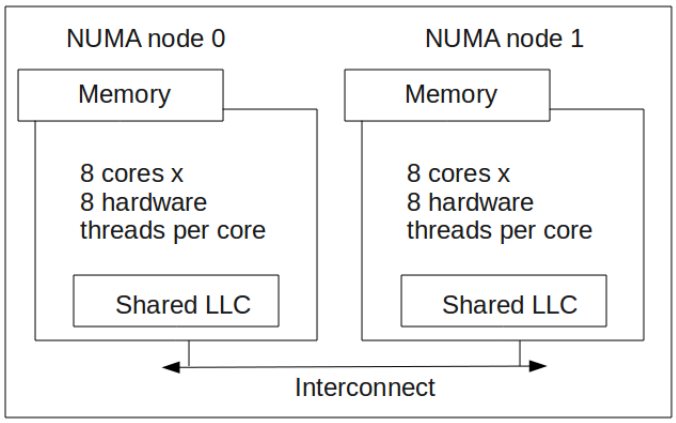


Figure 3.1: Example of a NUMA system with two nodes and 128 hardware threads.

NUMA systems contain multiple sockets connected by an interconnect. Each socket (also called a node) consists of multiple processing cores with a shared last level cache (LLC) and a local memory (as in Figure 3.1). A thread can quickly access the local memory on its own socket and it can access the memory on another socket using the interconnect, so the programming model is similar to uniform memory architectures. The NUMA design allows systems to scale to hundreds of cores and

provides inexpensive data sharing for cores on the same socket. However, remote cache invalidations and remote memory access can drastically degrade performance because of the interconnect's high latency and limited bandwidth. Therefore, in many cases, legacy code exhibits worse throughput when ported to NUMA machines than on non-NUMA ones.

Prior research addresses this by using a NUMA aware contention manager that migrates threads closer to the data they access [5]. However, migrating threads is a complex solution that, while feasible for operating systems, is not generally realistic for end-user applications. Alternatively, one could devise solutions in which the data are moved to the accessing threads. For example, cohort locks [23] and NUMA reader-writer locks [10] keep the data local to one cache as long as possible. This is implemented by transferring ownership of the locks from the threads finishing their critical sections to other threads on the same socket. Similarly, Metreveli et al. [64] minimize cache data transfers by partitioning a concurrent hash table and distributing operations for each partition to a specifically assigned thread. All threads wanting to access the hash table submit requests to these server threads through message passing implemented in shared memory. Essentially, the hash table resides in the caches of the accessing threads and the cache-to-cache traffic is limited to requests sent to and from the servers.

Making Data Structures NUMA-Aware. To maximize performance, Metreveli et al. [64] leverage the concurrency properties of hash tables in their partition implementation. Namely, hash tables are highly concurrent, easily partitionable data structures. However, many data structures do not have the inherent concurrency benefits of hash tables. This chapter focuses on a NUMA-aware implementation of a stack. Nevertheless, the method presented can be applied to other data structures as well.

Stacks have a broad range of uses: from evaluating expressions in calculators to parsing the syntax of expressions and program blocks in compilers. In addition, stacks can easily be used to implement unfair thread pools and any containers without ordering guarantees. An example of this is the Java unfair synchronous queue [81].

Unfortunately, stacks cannot be easily partitioned without forfeiting their last-in-first-out (LIFO) property. Because of this, multiple threads often contend on the single entry point providing access into the stack. It is primarily for this reason that stacks seem to be inherently sequential. However, prior work has shown that stacks can benefit from parallelism under balanced workloads (i.e., a similar number of push and pop operations) using elimination [39, 2]. This is implemented by canceling concurrent inverse operations from different threads even before they reach the stack. Elimination is not specific to stacks. Moir et al. [67] have shown how to use elimination with queues. Although this method significantly improves scalability of stacks, it does not address the primary concern of this chapter: i.e., remote cache invalidations on NUMA systems.

The goal is to reduce cache traffic and maintain data locality while using the properties of the underlying data structure to enable parallelism. The result is a scalable and highly parallel stack that outperforms all previous stack implementations on NUMA systems.

3.2 Algorithm Design

This section describes the use of delegation to implement a NUMA-aware stack. At the highest level, the design provides efficiencies in increased cache locality and reduced interconnect contention. After discussing the design, this section shows

how to employ the rendezvous elimination algorithm [2] to make this stack scalable. Moreover, this section presents the difference between global elimination, which is implemented using one rendezvous structure shared by all threads, and local elimination, which contains an elimination structure for each NUMA node.

3.2.1 Delegation

We use the delegation approach to implement a NUMA-aware stack. In particular, we use one dedicated server thread that accepts push and pop requests from many client threads. Figure 3.2 shows the overall interaction between the server and the clients. The communication is implemented in shared memory, using one location (which we call a slot) for each client. The server loops through all the slots, collecting and processing requests and writing the responses back to the slots. The clients post requests to their private slots and spin-wait on that slot until a response is provided by the server. Figure 3.3a provides a high-level overview of this communication protocol.

We note that only the pop operations need to spin-wait until a response is provided. The push operations could return as soon as the server notices their requests. This optimization improves throughput, but we decided not to use it in our experiments, for a more fair comparison with the other methods.

A weakness of this design is that using a reserved slot for each client can result in wasted space if the clients' workloads are not evenly distributed. Furthermore, the server must loop through all slots, even those not in use, when looking for requests. These two drawbacks can result in increased space and time complexity. To overcome these limitations, we statically assign several threads to the same slot

by thread id. [1] To synchronize the access of multiple threads to the same slot, we introduce an additional spinlock for each slot. Figure 3.3b reflects these changes to the communication protocol.

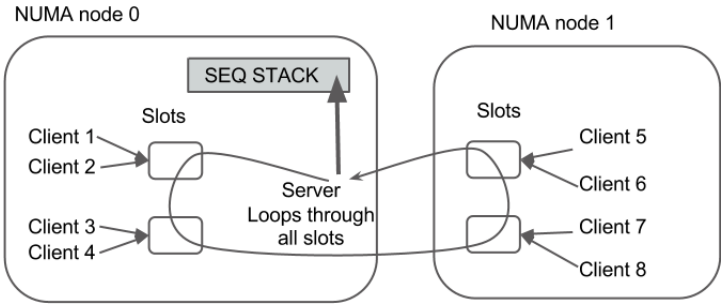


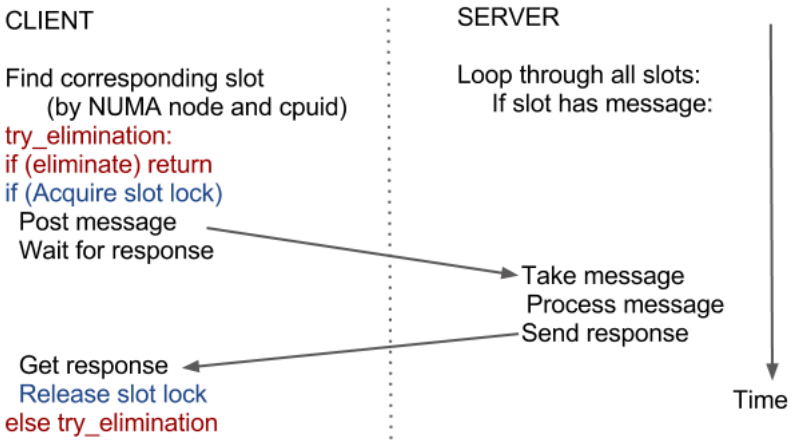
Figure 3.2: Delegation: clients post their requests in shared local slots and wait for the server to process them. The server loops through all the slots, processes requests as it finds them and immediately posts back the response in the same slot. The sequential stack is only accessed by the server thread; therefore, the top part of the stack remains in the server's cache all throughout execution.

3.2.2 Elimination

Elimination generally works best when the number of inverse operations are roughly equivalent. For inequivalent, unbalanced workloads, many operations cannot be eliminated, thereby requiring a thread to access the data structure directly. This creates contention and cache-to-cache traffic because these operations could originate from different NUMA nodes. In order to solve these problems, we augment the delegation stack presented in the previous section with a rendezvous elimination layer. Threads first try to eliminate and, if they time out, they delegate their operation to the server thread. Delegation ensures that the data remains in the server's cache, while elimination enables parallelism, thus making the NUMA-aware stack more scalable. Moreover, threads can continue to try to eliminate while they wait for the spinlock of their slot to be released. The complete algorithm is described in

[1] It is important to note that all threads using the same slot need to be on the same NUMA node in order to maintain the slot's locality.

Figure 3.3c.



(a) (Black) Single thread per slot: each thread posts requests in its private slot, without any synchronization.

(b) (Blue) Multiple threads per slot: threads share slots, so they need to acquire the slot's spinlock before writing the request.

(c) (Red) Elimination: Threads first try to eliminate; if they fail they then try to acquire the slot spinlock and submit a request, but if the lock is already taken, they go back to the elimination structure; they continue this loop until either they eliminate, or they acquire the spinlock.

Figure 3.3: Communication protocol between a client thread and the server thread using slots.

Local vs. Global Elimination. For the rendezvous stack, threads first try elimination and, in the case of failure, they then directly access the stack. Our NUMA-aware stack is an improvement over this design, because it increases locality and reduces contention on the stack by replacing direct access to the stack with delegation. However, the initial stage of elimination can still cause a number of invalidations between different NUMA nodes' caches because each thread accesses the same shared structure when performing elimination. To overcome this bottleneck, our NUMA-aware stack splits the single elimination data structure into several local structures, equal to the number of NUMA nodes. To minimize interconnect contention, we limit elimination to occur only between those threads located on the same socket.

3.2.3 Advantages and Limitations

Our stack design is optimized for the NUMA architecture. Local elimination and delegation both contribute to removing the contention on the interconnect and on the stack. Moreover, delegation makes the inter-node communication explicit and reduces it to the messages exchanged between the server and the clients. The stack remains local to the server's cache and requires no synchronization, because only the server thread can access it directly. In contrast, state-of-the-art synchronization methods, such as locking, allow all threads to access the shared data, causing more cache-to-cache transfers than used by delegation. In addition, these methods also require communication for achieving synchronization.

One potential drawback of this approach is that the access to the stack is serialized by using only the server thread. However, the direct access of multiple threads to a stack would also be serialized by a lock to keep the stack's integrity. Moreover, we enable parallelism by using elimination, which compensates for accessing the stack sequentially.

Another drawback is the potential for additional communication overhead between the clients and the server. For example, if the stack is only rarely accessed, then direct access to it would likely be more efficient. However, the overhead of elimination and delegation is eclipsed by their benefits when there are many threads contending for access to the stack.

Finally, our description assumes one server thread for each shared stack. In order to maintain high throughput, this thread must always be available to handle queries. Therefore, each server thread is assigned a hardware thread and runs at high priority. Unfortunately, we might not have enough hardware threads if an application uses multiple shared data structures, so some of the structures might have to share a server. If the application uses many shared data structures, the server threads could

become a performance bottleneck. However, we believe this scenario does not happen often in practice.

3.3 Evaluation

We conducted our experiments on an Oracle SPARC T5240 machine with two Niagara T2+ processors running at 1.165GHz. Each chip has 8 cores and each core has 8 hardware threads for a total of 128 hardware threads (64 per chip). We implemented our NUMA stack algorithm in C++ and we compared it to previous stack implementations using the same microbenchmark as [2]: a rendezvous stack, a flat combining stack and a lock-free stack. The benchmark has flexible parameters, allowing us to measure throughput under different percentages of push and pop operations. The results we present were obtained using threads with fixed roles (e.g. push-only threads and pop-only threads). We allow the scheduler to assign our software threads to NUMA nodes and then pin them to their respective processors. [2] The server thread is created with increased priority compared to the client threads, to decrease the likelihood of being swapped out by the scheduler.

For our experiments, we started by comparing our local elimination and delegation NUMA stack (nstack_el) with a lock-free stack (lfstack) [88], which has been the basis for other stack implementations such as rendezvous [2] and flat-combining [38]. Then, we compared our stack to the flat combining stack (fcstack) [38], which outperforms the rendezvous stack when there is no significant potential for elimination (i.e., in unbalanced workloads).

The scalable performance of the lock-free stack begins to degrade around 16 threads. The flat-combining stack, however, is unaffected by the type of workload and achieves

[2]We also experimented with unbounded and variable role threads, but the results were too similar to warrant inclusion in this thesis.

relatively stable scalability across different thread counts. However, the elimination based NUMA stack outperforms both of them by a large margin. These results can be observed in Figures 3.4, 3.5 and 3.6.

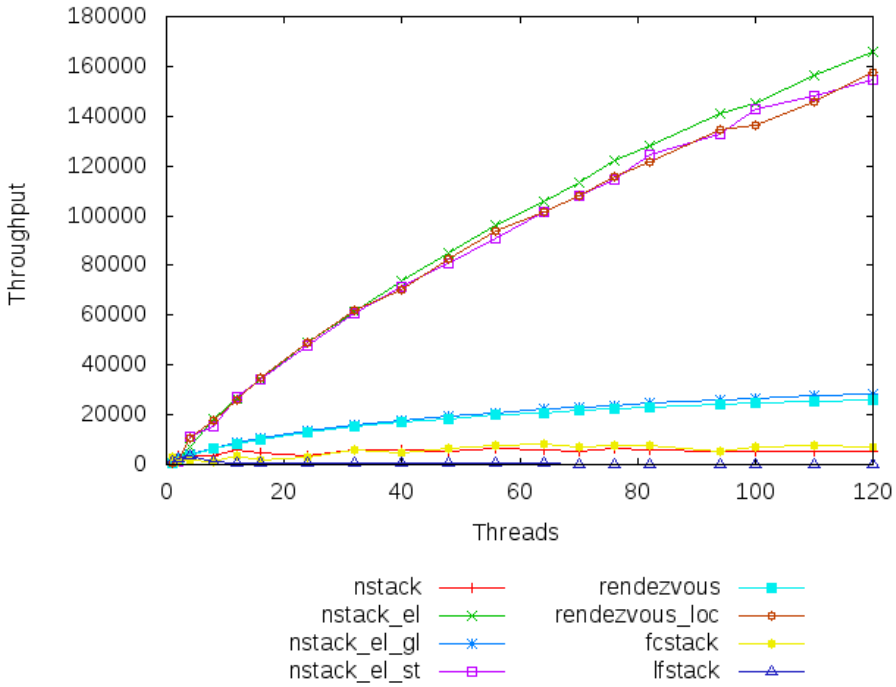


Figure 3.4: Results for 50% pushes and 50% pops

Effect of elimination. To judge the effect of the local elimination structures used in our implementation, we compared our NUMA stack (nstack_el) against two other versions; one without elimination (nstack) and one with global elimination (nstack_el_gl). As expected, the global elimination algorithm outperforms the algorithm without elimination, while both perform worse than local elimination. From Figures 3.4, 3.5 and 3.6, we conclude that local elimination is crucial for the scalability of our algorithm because it achieves locality for most of the operations. Our experiments were performed on a 2-node NUMA system, but we expect that these results generalize to systems with more nodes, as long as the push and pop operations are distributed uniformly across all the nodes.

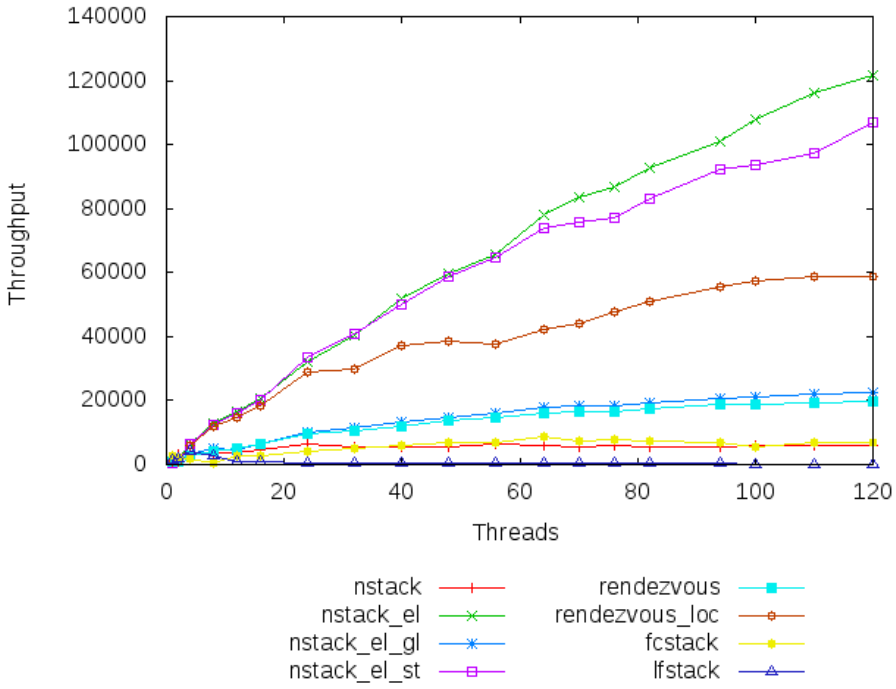


Figure 3.5: Results for 70% pushes and 30% pops

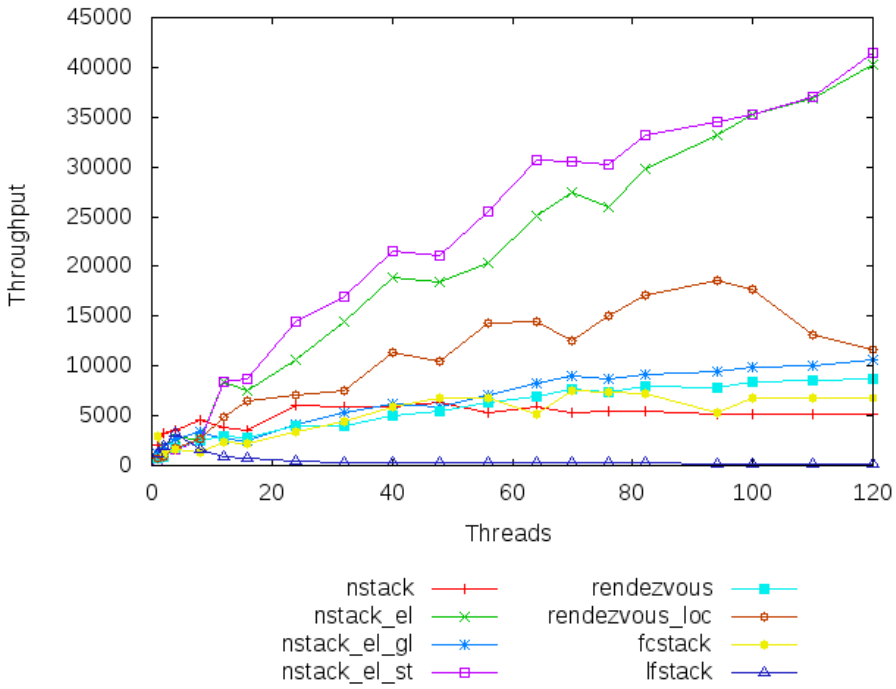


Figure 3.6: Results for 90% pushes and 10% pops

Effect of delegation. To better understand and characterize the impact of delegation, and because elimination has such a strong influence on performance, we compare our stack against two variations of the rendezvous stack: one uses local elimination and the other uses global elimination. The rendezvous stack (rendezvous) consists of global elimination and direct access. To provide a more fair comparison, we modified the rendezvous stack to perform elimination locally on each NUMA node (rendezvous_loc). Threads that fail to eliminate on each node must access the data structure directly. This local version of the rendezvous stack improves the scalability of the rendezvous stack for NUMA systems. However, our NUMA stack performs even better, indicating there is an observable performance benefit using delegation under high contention, for both balanced and unbalanced workloads (Figures 3.4, 3.5 and 3.6) due to reduced cache-to-cache traffic. We believe the benefit of delegation would become more apparent on a NUMA system with more sockets, because the penalty of inter-node communication is higher on such systems. Although the latency of an individual operation could increase because the server needs to inspect slots on more nodes, cache and memory locality would play an even more significant role than they do on a 2-node system, so the benefit given by delegation would increase. We leave evaluation on such a system as future work.

Balanced workloads. We experimented with different percentages of push and pop operations. Elimination works best when the number of pushes is very similar to the number of pops. In the balanced workload case, we use 50% push threads and 50% pop threads. Experimental results are shown in Figure 3.4. For this setting, elimination plays a significant role, as most operations will manage to eliminate. There is some benefit from delegation, as we can see when we compare to the local rendezvous algorithm, but not that significant.

Unbalanced workloads. For unbalanced workloads, elimination plays a much smaller role in reducing the number of operations. We present results for 70% pushes, 30% pops in Figure 3.5 and 90% pushes, 10% pops in Figure 3.6. In both cases, there is some elimination, but not as significant as in the balanced workload case. However, delegation plays a much more important role for these workloads, as more operations fail to eliminate and need to access the stack. Results show that preserving cache locality through delegation works much better than direct access to the stack.

Number of slots. Finally, we want to measure the impact of the synchronization introduced with sharing slots by different threads. We compared the implementation of the NUMA stack using shared slots (nstack_el) with the implementation using one slot per client thread, which does not require any synchronization to access the slots (nstack_el_st - nstack elimination single thread per slot). The results indicate that there is no clear winner in this case, which can be explained by the fact that the server has to loop through all the slots to service requests. Each request might have to wait a linear time in the number of slots to be found by the server. If the server finds too many of the slots empty, then much of the work performed by the server is wasted. However, if the server finds requests in most of the slots, then the algorithm can benefit from more slots because of the lack of synchronization. Our results seem to support this claim: the single thread (ST) per slot version outperforms the multiple threads per slot version (MT) for very unbalanced workloads as in Figure 3.6, while MT outperforms ST for more balanced workloads, as in Figures 3.4 and 3.5. This is due to the elimination algorithm significantly reducing the number of requests sent to the server for balanced workloads, while for unbalanced workloads there is less elimination and more requests sent to the server.

In our experiments, we assumed that we know the maximum number of client threads in the system and always check all the slots, even when running with fewer threads.

This could be improved using an adaptive way of determining the number of slots, but we leave that as future work.

3.4 Summary

Hardware's shift towards NUMA systems urges a compatible software redesign. Basic data structures are not optimized for these architectures. We propose the first NUMA-aware design of a stack, using local elimination and delegation. Combining these two methods is favorable across a number of scenarios: elimination works best when the number of pushes and pops is roughly the same, while delegation significantly reduces contention in the cases in which there is not enough potential for elimination because the workload is not very balanced. Our NUMA-aware stack outperforms prior stack implementations across different scenarios from completely balanced workloads to the more unbalanced ones.

However, this is just the first step in transitioning to NUMA systems. There are vast and exciting opportunities for exploring the design of other NUMA-aware data structures. We presented one technique and showed that it works well for a stack. The same technique could be applied to other data structures, such as queues and lists, which also admit inverse operations. In contrast, other data structures might not be suitable for elimination or might suffer from the serialized access of the server thread. For these data structures, we need to find new tools that allow us to redesign them for the NUMA space.

Chapter 4

A Concurrent Priority Queue

Priority queues are fundamental abstract data structures, often used to manage limited resources in parallel programming. Several proposed parallel priority queue implementations are based on skiplists, harnessing the potential for parallelism of the `add()` operations. In addition, methods such as Flat Combining [38] have been proposed to reduce contention, batching together multiple operations to be executed by a single thread. While this technique can decrease lock-switching overhead and the number of pointer changes required by the `removeMin()` operations in the priority queue, it can also create a sequential bottleneck and limit parallelism, especially for non-conflicting `add()` operations.

In this chapter, we describe a novel priority queue design, harnessing the scalability of parallel insertions in conjunction with the efficiency of batched removals. Moreover, we present a new elimination algorithm suitable for a priority queue, which further increases concurrency on balanced workloads with similar numbers of `add()` and `removeMin()` operations. We implement and evaluate our design using a variety of techniques including locking, atomic operations, hardware transactional memory, as well as employing adaptive heuristics given the workload.

4.1 Background

A priority queue is a fundamental abstract data structure that stores a set of keys (or a set of key-value pairs), where keys represent priorities. It usually exports two main operations: `add()`, to insert a new item in the priority queue, and `removeMin()`, to remove the first item (the one with the highest priority). Parallel priority queues are often used in discrete event simulations and resource management, such as operating systems schedulers. Therefore, it is important to carefully design these data structures in order to limit contention and improve scalability. Prior work in concurrent priority queues exploited parallelism by using either a heap [47] or a skiplist [59] as the underlying data structures. In the skiplist-based implementation of Lotan and Shavit [59], each node has a "deleted" flag, and processors contend to mark such "deleted" flags concurrently, in the beginning of the list. When a thread logically deletes a node, it tries to remove it from the skiplist using the standard removal algorithm. A lock-free skiplist implementation is presented in [87].

However, these methods may incur limited scalability at high thread counts due to contention on shared memory accesses. Hendler et al. [38] introduced Flat Combining, a method for batching together multiple operations to be performed by only one thread, thus reducing the contention on the data structure. This idea has also been explored in subsequent work on delegation [64, 9], where a dedicated thread called a *server* performs work on behalf of other threads, called *clients*. Unfortunately, the server thread could become a sequential bottleneck. A method of combining delegation with elimination has been proposed to alleviate this problem for a stack data structure [11]. Elimination [39] is a method of matching concurrent inverse operations so that they don't access the shared data structure, thus significantly reducing contention and increasing parallelism for otherwise sequential structures, such as stacks. An elimination algorithm has also been proposed in the context of a

queue [67], where the authors introduce the notion of *aging operations* - operations that wait until they become suitable for elimination.

In this chapter, we describe, to the best of our knowledge, the first elimination algorithm for a priority queue. Only `add()` operations with values smaller than the priority queue minimum value are allowed to eliminate. However, we use the idea of aging operations introduced in the queue algorithm [67] to allow `add()` values that are *small enough* to participate in the elimination protocol, in the hope that they will soon become eligible for elimination. We implement the priority queue using a skiplist and we exploit the skiplist's capability for both operations-batching and disjoint-access parallelism. `RemoveMin()` requests can be batched and executed by a server thread using the combining/delegation paradigm. `Add()` requests with high keys will most likely not become eligible for elimination, but need to be inserted in the skiplist, requiring expensive traversals towards the end of the data structure. These operations would represent a bottleneck for the server and a missed opportunity for parallelism if executed sequentially. Therefore, we split the underlying skiplist into two parts: a *sequential* part, managed by the server thread and a *parallel* part, where high-valued `add()` operations can insert their arguments in parallel. Our design reduces contention by performing batched sequential `removeMin()` and small-value `add()` operations, while also leveraging parallelism opportunities through elimination and parallel high-value `add()` operations. We show that our priority queue outperforms prior algorithms in high contention workloads on a SPARC Niagara II machine. Finally, we explore whether the use of hardware transactions could simplify our design and improve throughput. Unfortunately, machines that support hardware transactional memory (HTM) are only available for up to four cores (eight hardware threads), which is not enough to measure scalability of our design in high contention scenarios. Nevertheless, we showed that a transactional version of our algorithm is better than a non-transactional version on a Haswell four-core machine. We believe

that these preliminary results will generalize to machines with more threads with support for HTM, once they become available.

4.2 Algorithm Design

Our priority queue exports two operations: `add()` and `removeMin()` and is implemented using an underlying skiplist. The elements of the skiplist are buckets associated with keys. For a bucket b, the field b.key denotes the associated key. We split the skiplist in two distinct parts. The *sequential part*, in the beginning of the skiplist, is likely to serve forthcoming `removeMin()` operations of the priority queue (`PQ::removeMin()` for short) as well as `add(`v`)` operations of the priority queue (`PQ:add()` for short) with v small enough (hence expected to be removed soon). The *parallel part*, which complements the sequential part, is likely to serve `PQ::add(`v`)` operations where v is large enough (hence not expected to be removed soon). Either the sequential or the parallel part may become empty. Both lists are complete skiplists, with (dummy) head buckets called `headSeq` and `headPar`, respectively, with key $-\infty$. Both lists also contain (dummy) `tail` buckets, with key $+\infty$. We call the last non-dummy bucket of the sequential part `lastSeq`, which is the logical divider between parts. Figure 4.1 shows the design.

When a thread performs a `PQ::add(`v`)`, either (1) $v >$ `lastSeq.key`, and the thread inserts the value concurrently in the parallel part of the skiplist, calling the `SL::addPar()` skiplist operation; or (2) $v \leq$ `lastSeq.key`, and the thread tries to perform elimination with a `PQ::removeMin()` using an elimination array. A `PQ::add(`v`)` with v less than the smallest value in the priority queue can immediately eliminate with a `PQ::removeMin()`, if one is available. A `PQ::add(`v`)` operation with v bigger than `minValue` (the current minimal key) but smaller than `lastSeq.key` lingers in the

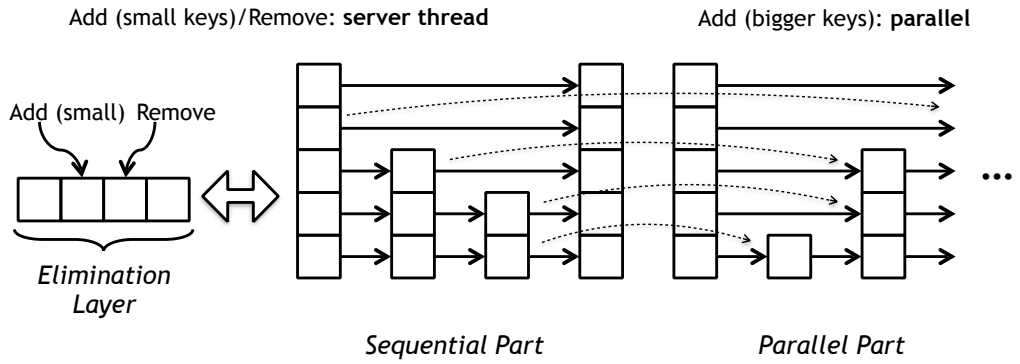


Figure 4.1: Skiplist design. An elimination array is used for `removeMin()`s and `add()`s with small keys. A dedicated server thread collects the operations that do not eliminate and executes them on the sequential part of the skiplist. Concurrent threads operate on the parallel part, performing `add()`s with bigger keys. The dotted lines show pointers that would be established if the single skiplist was not divided in two parts.

elimination array for some time, waiting to become eligible for elimination or time-out. A server thread executes sequentially all operations that fail to eliminate.

This mechanism describes the first elimination algorithm for a priority queue, well integrated with delegation/combining, presented in more detail in Section 4.2.2. Specifically: (1) The scheme harnesses the parallelism of the priority queue `add()` operations, letting those `add()` operations with keys physically distant and large enough (bigger than `lastSeq.key`) execute in parallel. (2) At the same time, we batch concurrent priority queue `add()` with small keys and `removeMin()` operations that timed out in the elimination array, serving such requests quickly through the server thread – this latter operation simply consumes elements from the sequential part by navigating through elements in its bottom level, merely decreasing counters and moving pointers in the most common situation. While detaching a sequential part is non-negligible cost-wise, a sequential part has the potential to serve multiple removals.

4.2.1 Concurrent Skiplist

Our underlying skiplist is operated by the server thread in the sequential part and by concurrently inserting threads with bigger keys in the parallel part.

Sequential part. The server calls the skiplist function `SL::moveHead()` to extract a new sequential part from the parallel part if some `PQ::removeMin()` operation was requested and the sequential part was empty. Conversely, it calls the skiplist function `SL::chopHead()` to relink the sequential and the parallel parts, forming a completely parallel skiplist, if no `PQ::removeMin()` operations are being requested for some time. In `SL::moveHead()`, we initially determine the elements to be moved to the sequential part. If no elements are found, the server clears the sequential part, otherwise separating the sequential part from the rest of the list, which becomes the parallel part. The number of elements that `SL::moveHead()` tries to detach to the sequential part adaptively varies between 8 and 65,536. Our policy is simple: if more than N insertions (e.g. N = 1000) occurred in the sequential part since the last `SL::moveHead()`, we halve the number of elements moved; otherwise, if less than M insertions (e.g. M = 100) were made, we double this number. After `SL::moveHead()` executes, a pointer called `currSeq` indicates the first bucket in the sequential part, and another called `lastSeq` indicates the final bucket. The server uses `SL::addSeq()` and `SL::removeSeq()` within the sequential part to remove elements or insert elements with small keys (i.e., belonging to the sequential part) that failed to eliminate. Buckets are not deleted at this time; they are deleted lazily when the whole sequential part gets consumed. A new sequential part can be created by calling `SL::moveHead()` again.

Parallel part. The skiplist function `SL::addPar()` inserts elements into the parallel part, and is called by concurrent threads performing `PQ::add()`. While these insertions are concurrent, the skiplist still relies on a Single-Writer Multi-Readers

lock with writer preference for the following purpose. Multiple `SL::addPar()` operations acquire the lock for reading (executing concurrently), while `SL::moveHead()` and `SL::chopHead()` operations acquire the lock for writing. This way, we avoid that `SL::addPar()` operates on buckets that are currently being moved to the sequential part by `SL::moveHead()`, or interferes with `SL::chopHead()`. Despite the lock, `SL::addPar()` is not mutually exclusive with the *head-moving operations* (`SL::moveHead()` and `SL::chopHead()`). Only the pointer updates (for new buckets) or the counter increment (for existing buckets) must be done in the parallel part (and not have been moved to the sequential part) after we determine the locations of these changes. Hence, in the `SL::addPar()` operation, we first try to get a *clean* `SL::find()`: a find operation followed by lock acquisition for reading, with no intervening head-moving operations. We can tell whether no head-moving operation took place since our lock operations always increases a timestamp variable, checked in the critical section. After a clean `SL::find()`, therefore now holding the lock, if a bucket corresponding to the key is found, we insert the element in the bucket (incrementing a counter). Otherwise, a new bucket is created, and inserted level by level using `CAS()` operations. If a `CAS()` fails in a certain level, we release the lock and retry a clean `SL::find()`.

Our algorithm differs from the traditional concurrent skiplist insertion algorithms in two ways: (1) we hold a lock to avoid head-moving operations to take place after a clean `SL::find()`; and (2) if the new bucket is moved out of the parallel section while we insert the element in the upper levels, we stop `SL::addPar()`, leaving this element with a capped level. This bucket is likely to be soon consumed by a `SL::removeSeq()` operation, resulting from a `PQ::removeMin()` operation.

Pseudo-code

We present the pseudo-code for the concurrent skiplist algorithm. The skiplist contains a Single-Writer-Multi-Readers lock with writer preference, called simply `lock`. In terms of notation, `lock.acquireR()` acquires the lock for reads, and `lock.acquireW()` acquires the lock for writes. The `SL::removeSeq()` skiplist procedure is described in Algorithm 1.

Algorithm 1 SL::removeSeq()

```
 1: if minValue = MaxInt then
 2:     return MaxInt
 3: if currSeq = ⊥ then
 4:     moveHead()
 5: key ← currSeq.key
 6: currSeq.counter ← currSeq.counter - 1
 7: if currSeq.counter = 0 then
 8:     while currSeq ≠ lastSeq do
 9:         currSeq ← currSeq.next[0]
10:         if currSeq.counter > 0 then
11:             minValue = currSeq.key
12:             return key
13:     moveHead()
14: return key
```

The variable `lock.timestmap` contains the timestamp associated with the lock (and hence with the head-moving operations). Algorithm 2 returns a pair of elements (b, r): b is a bucket found using the skiplist `SL::find()` operation, and r is a boolean defined as follows. If a head-moving operation happened anywhere between Lines 1 and 4, the timestamp moved and r will be false.

The `SL::addPar()` skiplist procedure is described in Algorithm 3. It uses the clean find protocol above. It performs a clean find, followed by mutable operations (either increasing a counter or inserting a bucket), executed with `lock` acquired for reading.

Algorithm 2 cleanFind(v, preds, succs)

```
1: t ← lock.timestamp
2: b ← find(headPar, v, preds, succs)
3: lock.acquireR()
4: if t < lock.timestamp then
5:     lock.release()
6:     return (⊥, false)
7: return (b, true)
```

Algorithm 3 SL::addPar(v)

```
1: if v ≤ lastSeq.key then
2:     return false
3: (b, r) ← cleanFind(v, preds, succs)
4: if r = false then
5:     restart at line 3
6: if b ≠ ⊥ then
7:     Atomically increment b.counter
8:     lock.release()
9:     return true
10: b ← newNode(v)
11: for i: 1 → b.topLevel do
12:     b.next[i] ← succs[i]
13: if not CAS(preds[0].next[0]: succs[0] → b) then
14:     lock.release()
15:     restart at line 3
16: repeat
17:     m ← minValue
18: until m ≤ v or CAS(minValue: m → v)
19: for i: 1 → b.topLevel do
20:     b.next[i] ← succs[i]
21:     if CAS(preds[i].next[i]: succs[i] → b) then
22:         continue
23:     lock.release()
24:     repeat
25:         (b, r) ← cleanFind(v, preds, succs)
26:     until r = true
27:     if b = ⊥ then
28:         lock.release()
29:         return true
30: return true
```

The `SL::moveHead()` skiplist procedure is described in Algorithm 4. Line 19 creates the sequential part starting from where the parallel part used to be, and the operations starting at Line 21 separate the skiplist in two parts. Note how `SL::find()` is used to locate the pointers that will change in order to separate the skiplist.

Algorithm 4 SL::moveHead()

```
 1: n is determined dynamically (see text)
 2: lock.acquireW()
 3: currSeq ← ⊥
 4: pred ← headPar
 5: curr ← headPar.next[0]
 6: i = 0
 7: while i < n and curr ≠ tail do
 8:     i ← i + curr.counter
 9:     if currSeq = ⊥ then
10:         currSeq ← curr; minValue ← curr.key
11:     pred ← curr; curr ← curr.next[0]
12: if i = 0 then
13:     for i : MaxLvl → 0 do
14:         headPar[i], headSeq[i] ← tail
15:     lastSeq ← headPar, minValue ← MaxLvl
16:     lock.release()
17:     return false
18: lastSeq ← pred
19: for i : MaxLvl → 0 do
20:     headSeq[i] ← headPar[i]
21: find(headSeq, lastSeq + 1, preds, succs)
22: for i : MaxLvl → 0 do
23:     preds[i].next[i] ← tail
24:     headPar.next[i] ← succs[i]
25: lock.release()
26: return true
```

Finally, the `SL::chopHead()` skiplist procedure is described in Algorithm 5. Note that all the `SL::find()` operations are executed outside the critical section. These operations identify the pointers that will change in order to relink the skiplist.

Algorithm 5 SL::chopHead()

```
1: if currSeq = ⊥ then
2:     return false
3: find(headSeq, lastSeq.key + 1, preds, ⊥)
4: find(headSeq, currSeq.key, ⊥, succs)
5: lock.acquireW()
6: for i : MaxLvl → 0 do preds[i].next[i] ← headPar.next[i]
7: lastSeq ← headPar, currSeq ← ⊥
8: for i : MaxLvl → 0 do
9:     headPar.next[i] ← succs[i] if succs[i] ≠ tail
10: lock.release()
11: return true
```

4.2.2 Elimination and Combining

Elimination allows matching operations to complete without accessing the shared data structure, thus increasing parallelism and scalability. In a priority queue, any `SL::removeMin()` operation can be eliminated, but only `SL::add()` operations with values smaller or equal to the current minimum value can be so. If the priority queue is empty, any `SL::add()` value can be eliminated. We used an elimination array similar to the one in the stack elimination algorithm [39]. Each slot uses 64 bits to pack together a 32-bit value that represents either an opcode or a value to be inserted in the priority queue and a stamp that is unique for each operation. The opcodes are: EMPTY, REMREQ, TAKEN and INPROG. These are special values that cannot be used in the priority queue. All other values are admissible. In our implementation, each thread has a local count of how many operations it performed. This count is combined with the thread ID to obtain a unique stamp for each operation. Overflow was not an issue in our experiments, but if it becomes a problem a different algorithm for associating unique stamps to each operation could be used. The unique stamp is used to ensure linearizability, as explained in Section 4.3. All slots are initially empty, marked with the special value EMPTY, and the stamp value is zero.

A `PQ::removeMin()` thread loops through the elimination array until it finds a request to eliminate with or it finds an empty slot in the array, as described in Algorithm 6. If it finds a value in the slot, then it must ensure that the stamp is positive, otherwise the value was posted as a response to another thread. The value it finds must be smaller than the current priority queue minimum value. Then, the `PQ::removeMin()` thread can `CAS` the slot, which contains both the value and the stamp, and replace it with an indicator that the value was taken (TAKEN, with stamp zero). The thread returns the value found. If instead, the `PQ::remove()` thread finds an empty slot, it posts a *remove request* (REMREQ), with a unique stamp generated as above. The thread waits until the slot is changed by another thread, having a value with stamp zero. The `PQ::removeMin()` thread can then return that value.

Algorithm 6 PQ::removeMin()

```
1:  while true do
2:      pos ← (id + 1)% ELIM_SIZE; (value, stamp) ← elim[pos]
3:      if IsValue(value) and (stamp > 0) and (value ≤ skiplist.minValue)) then
4:          if CAS(elim[pos], (value, stamp), (TAKEN, 0)) then
5:              return value
6:      if value = EMPTY then
7:          if CAS(elim[pos], (value, stamp), (REMREQ, uniqueStamp())) then
8:              repeat
9:                  (value, stamp) ← elim[pos]
10:             until value ≠ REMREQ and value ≠ INPROG
11:             elim[pos] ← (EMPTY, 0); return value
12:     inc(pos)
```

A `PQ::add()` thread initially tries to use `SL::addPar()` to add its key concurrently in the parallel part of the skiplist. A failed attempt indicates that the value should try to eliminate or should be inserted in the sequential part instead. The `PQ::add()` thread tries to eliminate by checking through the elimination array for REMREQ indicators. If it finds a remove request, and its value is smaller than the priority queue `minValue`, it can `CAS` its value with stamp zero, effectively handing it to

another thread. If multiple such attempts fail, the thread changes its behavior: it still tries to perform elimination as above, but as soon as an empty slot is found, it uses a `CAS` to insert its own value and the current stamp in the slot, waiting for another thread to match the operation (and change the opcode to TAKEN) returning the corresponding value.

The `PQ::add()` and `PQ::removeMin()` threads that post a request in an empty slot of the elimination array wait for a matching thread to perform elimination. However, elimination could fail because no matching thread shows up or because the `PQ::add()` value is never smaller than the priority queue `minValue`. To ensure that all threads make progress, we use a dedicated server thread that collects add and remove requests that fail to eliminate. The server thread executes the operations sequentially on the skiplist, calling `SL::addSeq()` and `SL::removeSeq()` operations. To ensure linearizability, the server marks a slot that contains an operation it is about to execute as *in progress* (INPROG). Subsequently, it executes the sequential skiplist operation and writes back the response in the elimination slot for the other thread to find it. A state machine showing the possible transitions of a slot in the elimination array is shown in Figure 4.2, and the algorithm is described in Algorithm 7.

Algorithm 7 Server::execute()

```
1: while true do
2:     for i: 1 → ELIM_SIZE do
3:         (value, stamp) ← elim[i]
4:         if value = REMREQ then
5:             if CAS(elim[i], (value, stamp), (INPROG, 0)) then
6:                 min ← skiplist.removeSeq(); elim[i] ← (min, 0)
7:         if IsValue(value) and (stamp > 0) then
8:             if CAS(elim[i], (value, stamp), (INPROG, 0)) then
9:                 skiplist.addSeq(value); elim[i] ← (TAKEN, 0)
```

The priority queue insertion algorithm is shown in Algorithm 8. If the value being inserted is not suitable for the parallel part (`PQ::addPar()` returns false), the request

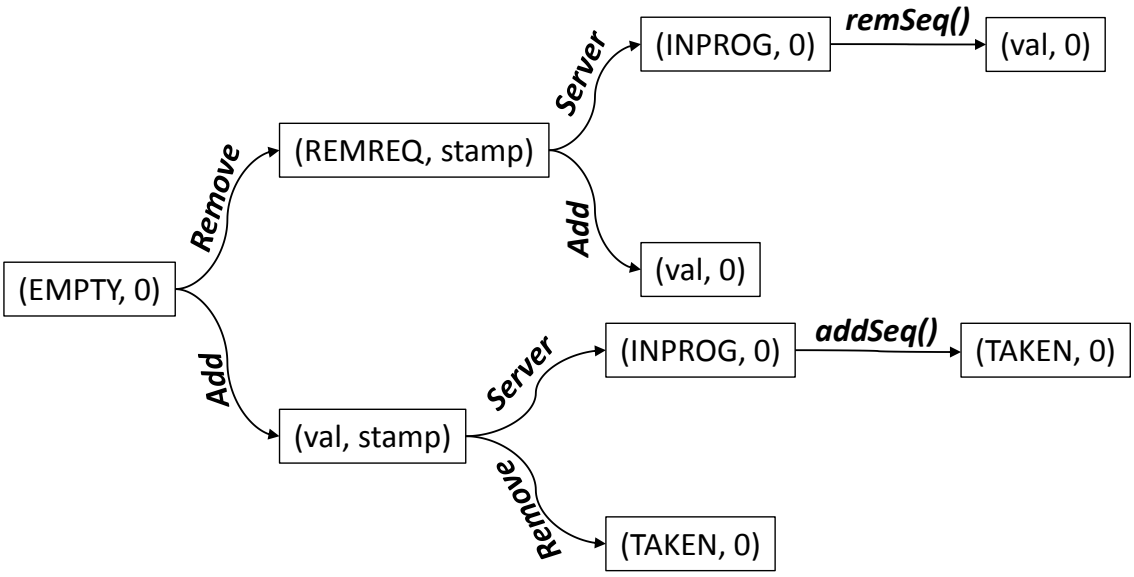


Figure 4.2: Transitions of a slot in the elimination array.

is posted in the elimination array, until eliminated with a suitable `PQ::removeMin()` or consumed by the server thread.

4.3 Linearizability

Our design provides a linearizable [45] priority queue algorithm. Some operations have multiple possible linearization points by design, requiring careful analysis and implementation.

Skiplist. A successful `SL::addPar(`v`)` (respectively, `SL::addSeq(`v`)`) usually linearizes when it inserts the element in the bottom level of the skip list with a `CAS` (respectively, with a store), or when the bucket for key v has its counter incremented with a `CAS` (respectively, with a store). However, a thread inserting a minimal bucket, whenever $v <$ `minValue`, is required to update `minValue`. When the sequential part is not empty, only the server can update `minValue` (without synchronization). When the sequential part is empty, a parallel add with minimal value needs to update `minValue`. The adding thread loops until a `CAS` decreasing `minValue` succeeds

Algorithm 8 PQ::add(inValue)

```
 1: if inValue ≤ skiplist.minValue then
 2:     rep ← MAX_ELIM_MIN
 3: else
 4:     if skiplist.addPar(inValue) then
 5:         return true
 6:     rep = MAX_ELIM
 7: while rep > 0 do
 8:     pos ← (id + 1)% ELIM_SIZE; (value, stamp) ← elim[pos]
 9:     if value = REMREQ and (inValue ≤ skiplist.minValue)) then
10:         if CAS(elim[pos], (value, stamp), (inValue, 0)) then
11:             return true
12:     rep ← rep −1; inc(pos)
13: if skiplist.addPar(inValue) then
14:     return true
15: while true do
16:     (value, stamp) ← elim[pos]
17:     if value = REMREQ and (inValue ≤ skiplist.minValue)) then
18:         if CAS(elim[pos], (value, stamp), (inValue, 0)) then
19:             return true
20:     if value = EMPTY then
21:         if CAS(elim[pos], (value, stamp), (inValue, uniqueStamp())) then
22:             repeat
23:                 (value, stamp) ← elim[pos]
24:             until value = TAKEN
25:             elim[pos] ← (EMPTY, 0); return true
26:     inc(pos)
```

or another thread inserts a bucket with key smaller than v. Note that no head-moving operation can execute concurrently because the `SL::addPar()` threads hold the `lock`. Threads that succeed changing `minValue` linearize their operation at the point of the successful `CAS`.

The head-moving operations `SL::moveHead()` and `SL::chopHead()` execute while holding the `lock` for writing, which effectively linearizes the operation at the `lock.release()` instant because: (1) no `SL::addPar()` is running; (2) no `SL::addSeq()` or `SL::removeSeq()` are running, as the server thread is the single thread performing those operations. The head-moving operations do not change `minValue`. In fact, they preclude any

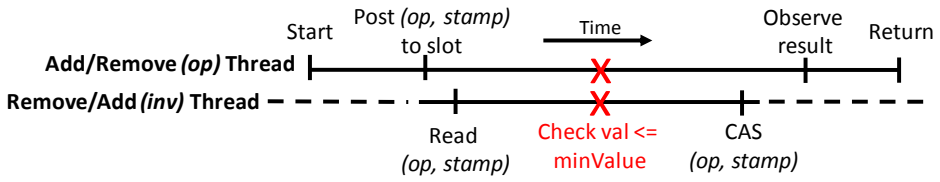


Figure 4.3: Concurrent execution of an *op* thread posting its request to an empty slot, and an *inv* thread, executing a matching operation. The operation by the *inv* thread could begin any time before the *Read* and finish any time after the *CAS*. The linearization point is marked with a red X.

changes to it. During these operations, however, threads may still perform elimination, which we discuss next.

Elimination. A unique stamp is used in each request posted in the array entries to avoid the "ABA" problem. Each elimination slot is a 64-bit value that contains 32 bits for the posted value (for `PQ::add()`) or a special opcode (for `PQ::removeMin()`) and 32 bits for the unique stamp. In our implementation, the unique stamp is obtained by combining the thread id with the number of operations performed by each thread. Each thread, either adding or removing, that finds the inverse operation in the elimination array must verify that the exchanged value is smaller than `minValue`. If so, the thread can `CAS` the elimination slot, exchanging arguments with the waiting thread. It is possible that the priority queue minimum value is changed by a concurrent `PQ::add()`. In that case, the linearization point for both threads engaged in elimination is at the point where the value was observed to be smaller than the priority queue minimum. See Fig. 4.3.

The thread performing the `CAS` first reads the stamp of the thread that posted the request in the array and verifies that it is allowed to eliminate. Only then it performs a `CAS` on both the value and the stamp, guaranteeing that the thread waiting did not change in the meantime. Because both threads were running at the time of the verification, they can be linearized at that point. Without the unique stamp, the eliminating thread could perform a `CAS` on an identical request (i.e., identical

operation and value) posted in the array by a different thread. The `CAS` would incorrectly succeed, but the operations would not be linearizable because the new thread was not executing while the suitable minimum was observed.

The linearizability of the combining operation results from the linearizability of the skiplist. The threads post their operation in the elimination array and wait for the server to process it. The server first marks the operation as *in progress* by `CAS`ing INPROG into the slot. Then it performs the sequential operation on the skiplist and writes the results back in the slot, releasing the waiting thread. The waiting thread observes the new value and returns it. The linearization point of the operation happens during the sequential operation on the skiplist, as discussed above. See Fig. 4.4.

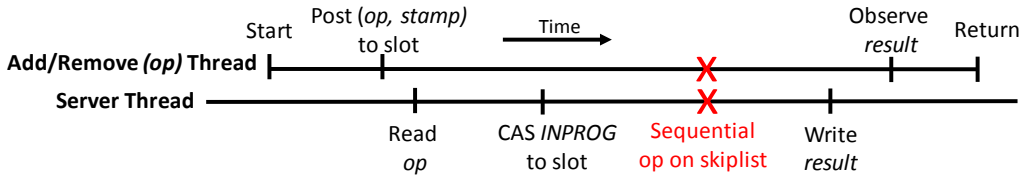


Figure 4.4: Concurrent execution of a client thread and the server thread. The client posts its operation *op* to an empty slot and waits for the server to collect the operation and execute it sequentially on the skiplist. The linearization point occurs in the sequential operation and is marked with a red X.

4.4 Evaluation

In this section, we discuss results on a Sun SPARC T5240, which contains two UltraSPARC T2 Plus chips with 8 cores each, running at 1.165 GHz. Each core has 8 hardware strands, for a total of 64 hardware threads per chip. A core has a 8KB L1 data cache and shares an 4MB L2 data cache with the other cores on a chip. We restrict the evaluation to cores within one chip to avoid cache traffic and memory effects. Each experiment was performed five times and we report the

median. Variance was very low for all experiments. Each test was run for ten seconds to measure throughput. We used the same benchmark as flat combining [38]. A thread randomly flips a coin with probability p to be an `PQ::add()` and $1-p$ to be a `PQ::removeMin()`. We started a run after inserting 2000 elements in the priority queue for stable state results.

Our priority queue algorithm (*pqe*) uses combining and elimination, and leverages the parallelism of `PQ::add()`. We performed experiments to compare against previous priority queues using combining methods, such as flat combining skiplist (*fcskiplist*) and flat combining pairing heap (*fcpairheap*). We also compared against previous priority queues using skiplists with parallel operations, such as a lock free skiplist (*lfskiplist*) and a lazy skiplist (*lazyskiplist*). The flat combining methods are very fast at performing `PQ::removeMin()` operations, which then get combined and executed together. However, performing the `PQ::add()` operations sequentially is a bottleneck for these methods. Conversely, the *lfskiplist* and *lazyskiplist* algorithms are very fast at performing the parallel adds, but get significantly slowed down by having `PQ::removeMin()` operations in the mix, due to the synchronization overhead involved. Our *pqe* design tries to address these limitations through our *dual* (sequential and parallel parts), *adaptive* implementation that can be beneficial in the different scenarios.

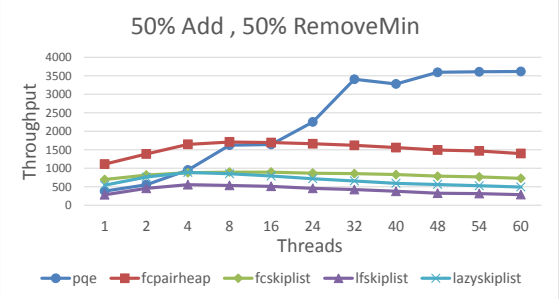


Figure 4.5: Priority queue performance with 50% `add()`s, 50% `removeMin()`s.

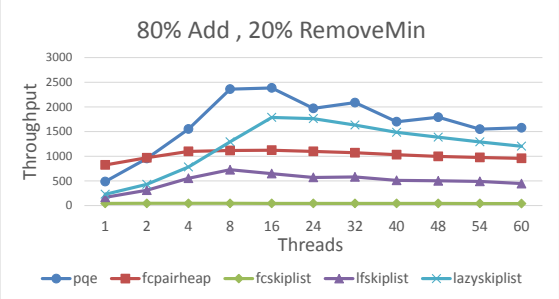


Figure 4.6: Priority queue performance with 80% `add()`s, 20% `removeMin()`s.

We considered different percentages of `PQ::add()` and `PQ::removeMin()` in our tests.

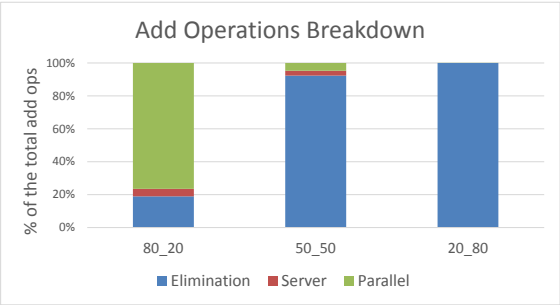


Figure 4.7: `add()` work breakdown.

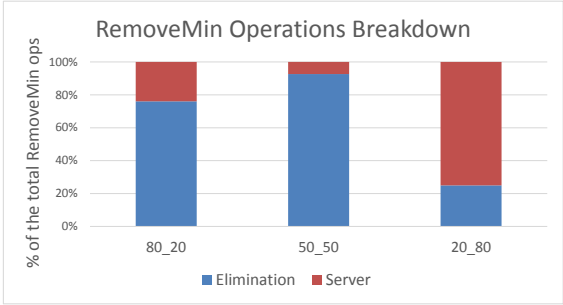


Figure 4.8: `removeMin()` work breakdown.

When the operations are roughly the same number, *pqe* can fully take advantage of both elimination and parallel adds, so it has peak performance. Figure 4.5 shows how for 50% `PQ::add()` and 50% `PQ::removeMin()`, *pqe* is much more scalable and can be up to 2.3 times faster than all other methods. When there are more `PQ::add()` than `PQ::removeMin()`, as in Figure 4.6 with 80% `PQ::add()` and 20% `PQ::removeMin()`, *pqe* behavior approaches the other methods, but it is still 70% faster than all other methods at high thread counts. In this specific case there is only little potential for elimination, but having parallel insertion operations makes our algorithm outperform the flat combining methods. The *lazyskiplist* algorithm also performs better than other methods, as it also takes advantage of parallel insertions. However, *pqe* uses the limited elimination and the combining methods to reduce contention, making it faster than the *lazyskiplist*. For more `PQ::removeMin()` operations than `PQ::add()` operations, the *pqe*'s potential for elimination and parallel adds are both limited, thus other methods can be faster. *Pqe* is designed for high contention scenarios, in which elimination and combining thrive. Therefore, it can incur a penalty at lower thread counts, where there is not enough contention to justify the overhead of the indirection caused by the elimination array and the server thread.

To better understand when each of the optimizations used is more beneficial, we analyzed the breakdown of the `PQ::add()` and `PQ::removeMin()` operations for different

`PQ::add()` percentages. When we have 80% `PQ::add()`, most of them are likely to be inserted in parallel (75%), with a smaller percentage being able to eliminate and an even smaller percentage being executed by the server, as shown in Fig. 4.7. In the same scenario, 75% of `removeMin()` operations eliminate, while the rest is executed by the server, as seen in Fig. 4.8. For balanced workloads (50% − 50%), most operations eliminate and a few `PQ::add()` operations are inserted in parallel. When the workload is dominated by `PQ::removeMin()`, most `PQ::add()` eliminate, but most `PQ::removeMin()` are still left to be executed by the server thread, thus introducing a sequential bottleneck. Eventually the priority queue would become empty, not being able to satisfy `PQ::removeMin()` requests with an actual value anymore. In this case, any `add()` operation can eliminate, allowing full parallelism. We do not present results for this case because it is an unlikely scenario that unrealistically favors elimination.

4.4.1 `PQ::moveHead()` and `PQ::chopHead()`

Maintaining separate skiplists for the sequential and the parallel part of the priority queue is beneficial for the overall throughput, but adds some overhead, which we quantify in this section. The number of elements that become part of the sequential skiplist changes dynamically based on the observed mix of operations. This adaptive behavior helps reduce the number of `moveHead()` and `chopHead()` operations required. Table 4.1 shows the percentage of the number of head-moving operations out of the total number of `PQ::removeMin()` operations for different mixes of `PQ::add()` and `PQ::removeMin()` operations. The head-moving operations are rarely called due to the priority queue's adaptive behavior.

Add() percentages	% moveHead()	% chopHead()
80	0.24%	0.03%
50	0.32%	0.01%
20	0.00%	0.00%

Table 4.1: The number of head-moving operations as a percentage of the total number of `PQ::removeMin()` operations, considering different `add()` and `removeMin()` mixes.

4.5 Hardware Transactions

Transactional memory [43] is an optimistic mechanism to synchronize threads accessing shared data. Threads are allowed to execute critical sections speculatively in parallel, but, if there is a data conflict, one of them has to roll back and retry its critical section. Recently, IBM and Intel added HTM instructions to their processors [89, 49]. In our priority queue implementation, we used Intel's Transactional Synchronization Extensions (TSX) [49] to simplify the implementation and reduce the overhead caused by the synchronization necessary to manage a sequential and a parallel skiplist. We evaluate our results on an Intel Haswell four core processor, Core i7-4770, with hardware transactions enabled (restricted transactional memory - RTM), running at 3.4GHz. There are 8GB of RAM shared across the machine and each core has a 32KB L1 cache. Hyperthreading was enabled on our machine so we collected results using all 8 hardware threads. Hyperthreading causes resource sharing between the hyperthreads, including L1 cache sharing, when running with more than 4 threads, thus it can negatively impact results, especially for hardware transactions. We did not notice a hyperthreading effect in our experiments. We used the GCC 4.8 compiler with support for RTM and optimizations enabled (-O3).

4.5.1 Skiplist

The Single-Writer-Multi-Readers lock used to synchronize the sequential and the parallel skiplists complicates the priority queue design and adds overhead. In this section, we explore an alternative design using hardware transactions. The naive approach of making all operations transactional causes too many aborts. Instead, the server increments a timestamp whenever a head-moving operation - `SL::moveHead()` or `SL::chopHead()` - starts or finishes. A `SL::addPar()` operation first reads the timestamp and executes a nontransactional `SL::find()` and then starts a transaction for the actual insertion, adding the server's timestamp to its read set and aborting if it is different from the initially recorded value. Moreover, if the timestamp changes after starting the transaction, indicating a head-moving operation, the transaction will be aborted due to the timestamp conflict. If the timestamp is valid, `SL::find()` must have recorded the predecessors and successors of the new bucket at each level i in `preds[i]` and `succs[i]`, respectively. If a bucket already exists, the counter is incremented inside the transaction and the operation completes. If the bucket does not exist, the operation proceeds to check if `preds[i]` points to `succs[i]` for all levels $0 \leq i \leq$ MaxLvl. If so, the pointers have not changed before starting the transaction and the new bucket can be correctly inserted between `preds[i]` and `succs[i]`. Otherwise, we commit the (innocuous) transaction, yet restart the operation.

Figures 4.9 and 4.10 compare the performance of the lock-based implementation and the implementation based on hardware transactions for two different percentages of `PQ::add()`s and `PQ::removeMin()`s. When fewer `PQ::removeMin()` operations are present, the timestamp changes less frequently and the `PQ::add()` transactions are aborted fewer times, which increases performance in the 80%-20% insertion-removal mix. In the 50%-50% mix, we obtain results comparable to the *pqe* algorithm using

the lock-based approach, albeit with a much simpler implementation.

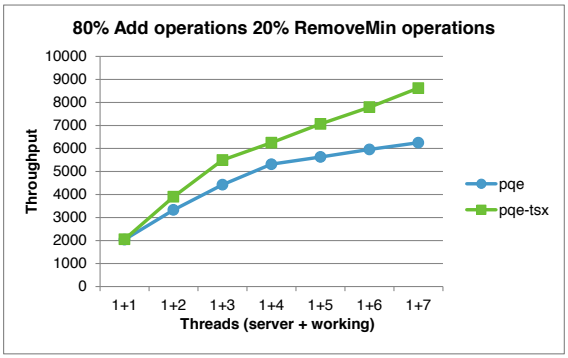


Figure 4.9: Priority queue performance when we use a transaction-based dual skiplist; 80% `add()`s, 20% `removeMin()`s.

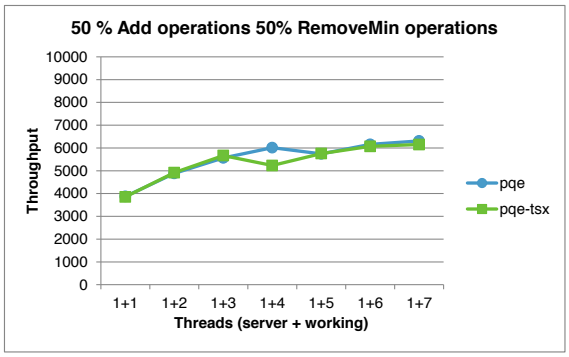


Figure 4.10: Priority queue performance when we use a transaction-based dual skiplist; 50% `add()`s, 50% `removeMin()`s.

4.5.2 Aborted Transactions

The impact of aborted transactions is reported in Tables 4.2 and 4.3. As the number of threads increases, the number of transactions per successful operation also increases, as does the percentage of operations that need more than 10 retries to succeed. Note that the innocuous transactions that find inconsistent pointers, changed between the `SL::find()` and the start of the transaction are not included in the measurement. After 10 retries, threads give up on retrying the transactional path and the server executes the operations on their behalf, either in the sequential part, using sequential operations, or in the parallel part, using `CAS()` for the pointer changes, but without holding the readers lock. The server does not need to acquire the readers lock because no other thread will try to acquire the writer lock.

The number of transactions per successful operation is at most 3.92, but 3.22 in the $50\% - 50\%$ case. The percentage of operations that get executed by the server (after aborting 10 times) is at most 10% of the total number of operations, but between 1.73% and 2.01% for the $50\% - 50\%$ case.

Working Threads	Transactions per successful operation	Fallbacks per total operations
1	1.01	0.00%
2	2.34	0.51%
3	3.21	1.73%
4	3.31	2.12%
5	3.46	2.74%
6	3.46	2.67%
7	3.61	3.25%

Table 4.2: Transaction stats for varying # of threads, with 50% `PQ::add()`s and 50% `PQ::removeMin()`s

Add percentage	Transactions per successful operation	Fallbacks per total operations
100	1.32	0.00%
80	1.77	0.01%
60	2.37	0.29%
50	3.22	2.01%
40	3.64	5.24%
20	3.92	10.34%
0	1.09	0.00%

Table 4.3: Transaction stats for varying mixes, with 1 server thread and 3 working threads.

4.5.3 Combining and Elimination

In this section, we describe our experience using Intel TSX to simplify combining and elimination. Adapting the elimination algorithm to use transactions was straightforward, by just replacing the pessimistic synchronization with transactions. We note that a unique stamp as described in Section 4.2.2 is not necessary for linearizability of elimination if the operations are performed inside hardware transactions. If a thread finds a matching operation and ensures in a transaction that the value is smaller than the minimum, then elimination is safe. If a change in the matching operation had occurred, the transaction would have aborted. We retry each transaction N times (e.g. $N = 3$ in our implementation). If a thread's transaction is aborted too many times during elimination, the thread moves on to other slots without retrying the failed slot in a fallback path. However, if the transaction fails while trying to insert an `PQ::add()` or `PQ::removeMin()` operation in an empty slot to be collected by the server thread, the original pessimistic algorithm is used as a software fallback path in order to guarantee forward progress. Unfortunately, the unique stamp needs to be used to ensure linearizability of the operations executed on the fallback path.

Using transactions in the server thread implementation required including `SL::addSeq()`

and `SL::removeSeq()` inside a transaction, which in turn caused too many aborts. Therefore, we designed an alternative combining algorithm that executes these operations outside the critical section. The complete algorithm is presented in Algorithm 9. It is based on the observation that, as long as there is a sequential part in the skiplist, the `SL::removeSeq()` and the `SL::addSeq()` operations can be executed lazily. The server can use the skiplist's *minValue* to return a value to a remove request and only execute the sequential operation after, without the remove thread waiting for it. Note that the skiplist's `minValue` could, in the meantime, return a value that is outdated. However, this value is always smaller or equal to the actual minimum in the skiplist, because it can only lag behind one sequential remove. This function is used by the `PQ::add()` operations to determine if they can eliminate or not. Therefore, estimating a minimum smaller than the actual minimum can affect performance, but will not impact correctness of our algorithm. Moreover, the server performs the `PQ::removeMin()` operation immediately after writing the minimum, thus cleaning up the sequential part and updating the minimum estimate. The `PQ::add()` case is similar too. If there is a sequential part to the skiplist, the server can update the skiplist lazily, after it releases the waiting thread. There is one difference. If the value inserted is smaller than *minValue*, then this needs to be updated before releasing the waiting thread.

Using these changes in the combining algorithm allowed a straightforward implementation using hardware transactions. However, our experiments indicated that certain particularities of the best-effort HTM design make it unsuitable for this scenario. First of all, because of its best-effort nature, a fallback is necessary in order to make progress. Therefore, the algorithm might be simplified on the common case, but it is still as complex as the fallback. Moreover, changes are often needed to adapt algorithms for an implementation using hardware transactions. Because these changes involve decreasing the sizes of the critical sections and decreasing the

number of potential conflicts, these changes could be beneficial to the original algorithm too. Finally, it seems that communications paradigms, such as elimination and combining, are best implemented using pessimistic methods. Intel TSX has no means of implementing non-transactional operations inside transactions (also called escape actions) and no polite spinning mechanism to allow a thread to wait for a change that is going to be performed in a transaction. The spinning thread could often abort the thread that it is waiting for. We used the PAUSE instruction in the spinning thread to alleviate this issue, but better hardware support for implementing communication paradigms using hardware transactions is necessary. For our elimination and combining algorithms, we concluded that pessimistic synchronization works better.

Algorithm 9 Server::execute()

```
1:  while true do
2:      for i: 1 → ELIM_SIZE do
3:          (value, stamp) ← elim[i]
4:          if value = REMREQ then
5:              if skiplist.currSeq = ⊥ then
6:                  skiplist.moveHead()
7:              if skiplist.currSeq ≠ ⊥ then
8:                  if CAS(elim[i], (value, stamp), (skiplist.minValue, 0)) then
9:                      skiplist.removeSeq()
10:             else
11:                 if CAS(elim[i], (value, stamp), (INPROG, 0)) then
12:                     min ← skiplist.removeSeq(); elim[i] ← (min, 0)
13:         if IsValue(value) and (stamp > 0) then
14:             if skiplist.currSeq ≠ ⊥ then
15:                 if CAS(elim[i], (value, stamp), (INPROG, 0)) then
16:                     if value < skiplist.minValue then
17:                         skiplist.minValue ← value
18:                     elim[i] ← (TAKEN, 0); skiplist.addSeq(value)
19:             else
20:                 if CAS(elim[i], (value, stamp), (INPROG, 0)) then
21:                     skiplist.addSeq(value); elim[i] ← (TAKEN, 0)
```

4.6 Summary

In this chapter, we describe a technique to implement a scalable, linearizable priority queue based on a skiplist, divided into a sequential and a parallel part. Our scheme simultaneously enables parallel `PQ::add()` operations as well as sequential batched `PQ::removeMin()` operations. The sequential part is beneficial for batched removals, which are performed by a special server thread. While detaching the sequential part from the parallel part is non-negligible cost-wise, the sequential part has the potential to serve multiple subsequent removals at a small constant cost. The parallel part is beneficial for concurrent insertions of elements with bigger keys (smaller priority), not likely to be removed soon. In other words, we integrate the flat combining/delegation paradigm introduced in prior work with disjoint-access parallelism.

In addition, we present a novel priority queue elimination algorithm, where `PQ::add()` operations with keys smaller than the priority queue minimum can eliminate with `PQ::removeMin()` operations. We permit `PQ::add()` operations, with keys small enough, to linger in the elimination array, waiting to become eligible for elimination. If the elimination is not possible, the operation is delegated to the server thread. Batched removals (combining) by the server thread is well-integrated with both: (1) parallelism of `add()` operations with bigger keys; and (2) the elimination algorithm, that possibly delegates failed elimination attempts (of elements with smaller keys) to the server thread in a natural manner. Our priority queue integrates delegation, combining, and elimination, while still leveraging the parallelism potential of insertions.

Chapter 5

Software Fallbacks for Best-effort Hardware Transactional Memory

Intel's Haswell and IBM's Blue Gene/Q and System Z are the first commercially available systems to include hardware transactional memory (HTM). However, they are all best-effort, meaning that every hardware transaction must have an alternative software fallback path that guarantees forward progress. The simplest and most widely used software fallback is a single global lock (SGL), in which aborted hardware transactions acquire the SGL before they are re-executed in software. Other hardware transactions need to subscribe to this lock and abort as soon as it is acquired. This approach, however, causes many hardware transactions to abort unnecessarily, determining even more transactions to fail and resort to the SGL.

In this chapter we suggest improvements to the simple SGL fallback. First, we use lazy subscription to reduce the rate of SGL acquisitions. Next, we propose fine-grained conflict detection mechanisms between hardware transactions and a software

SGL transaction. Finally, we describe how our findings can be used to improve future generations of HTMs.

5.1 Background

Parallel programming has gained significant importance due to the rise of commodity multicore computer systems. Unfortunately, writing correct and efficient software that effectively utilizes the resources of multicore systems remains an obstacle for a wider spread adoption of parallel programming. Locks, the current state-of-the-art solution for synchronizing shared memory access in parallel programs, are notoriously challenging, even for expert programmers [44].

Transactional memory (TM) [43] aims to simplify writing correct and efficient parallel software while avoiding the pitfalls of locks. Threads can speculatively execute transactions, maintaining read and write sets to track conflicts. If a conflict is detected between two transactions, one is usually aborted and rolled back so the other can commit. Transactional memory generally promises all-or-nothing semantics, where critical sections appear as if they executed atomically or not at all. Unfortunately, the overhead associated with these designs is generally high.

Hardware transactional memory (HTM), on the other hand, promises a faster performing, lower overhead alternative to STM. Yet, practical HTMs are *best-effort*: they do not guarantee forward progress. Furthermore, practical HTMs are bounded in size and support a restricted set of operations. It is for these reasons that an HTM alone is an insufficient TM solution.

In short, ensuring forward progress requires a software fallback. A simple and attractive solution to use for many applications is a *single global lock* (SGL) mechanism [89, 90], where all transactions that access a particular data structure synchro-

nize through a single lock[1]. Perhaps the most visible example of an SGL fallback scheme is Haswell's *hardware lock elision* (HLE) [48], which supports a lock fallback directly through the instruction set architecture. SGL schemes are attractive because they can easily be retrofitted to legacy code, and they do not require code duplication.

In both HLE, and HTM with SGL fallback, each hardware transaction starts by reading the lock's state, called *subscribing* to the lock. Subscription ensure that any software transaction that subsequently acquires that lock will provoke a data conflict, ensuring correctness by forcing any active subscribing hardware transactions to abort. The duration of a lock subscription represents a "window of vulnerability" during which the arrival of a software transaction will prevent any subscribing hardware transactions from executing.

In this chapter we present novel optimizations to the simple SGL fallback approach. We show that one can significantly improve performance by performing lock subscription in a *lazy* manner: optimistically postponing reading the lock state for as long as possible (usually the very end of the transaction). Lazy subscription was first proposed in the context of Hybrid NOrec [17] to allow concurrent execution of multiple hardware transactions with the committing phase of a speculative software transaction. Here, lazy subscription allows concurrent execution of multiple hardware transactions with a single non-speculative SGL transaction. The resulting mechanism maintains the simplicity and correctness of the original SGL fallback, but reduces its costs. We evaluate this design using Haswell's *restricted transactional memory* (RTM) running the STAMP benchmark suite, and compare it to several alternatives: a non-speculative SGL implementation, a speculative implementation with the usual SGL fallback, the hardware-only HLE, and to an STM

[1] "Global" here could mean a single lock per data structure, not necessarily system-wide if composability is not an issue.

(TL2). We also show how to improve conflict detection with the SGL transaction and we propose several novel hardware extensions.

5.2 SGL Fallback (E-SGL)

As noted, hardware transactional memory (HTM) has become a commercial reality, but HTM provided by processors such as Intel's Haswell and IBM's Power ISA offer no progress guarantees, implying that some form of software fallback is needed. In the single global lock (SGL) approach, each shared data structure has an associated lock. When it starts, a hardware transaction immediately reads the lock state, an action known as *eager subscription*. When a repeatedly failed hardware transaction restarts in software, it acquires exclusive access to the lock, forcing any subscribed hardware transactions to abort.

SGL fallback is attractive because it is simple, requiring no memory access annotation, and no code duplication between alternative paths. Nevertheless, an inherent limitation of current SGL fallbacks schemes is that hardware and software transactions that share a global lock cannot execute concurrently. Figure 5.1 shows the four ways in which hardware and software transaction can overlap. In cases 2 and 3, the hardware transaction is aborted as soon as it checks the lock, while in cases 1 and 4 the hardware transaction is aborted when the software transaction acquires the lock. With eager subscription, it makes sense for a thread starting a hardware transaction to wait until the SGL becomes free.

In this chapter, we describe how to improve conflict detection to allow some concurrency between the hardware and software transaction that share a lock. In Section 5.3, we describe a *lazy subscription* mechanism that permits concurrent hardware and software transactions to share the same SGL and intuitively show its correctness.

We evaluate this scheme's performance in Section 5.4. We describe finer-grained conflict detection mechanisms in Section 5.5. In Section 5.6, we describe how these observations might improve future hardware.

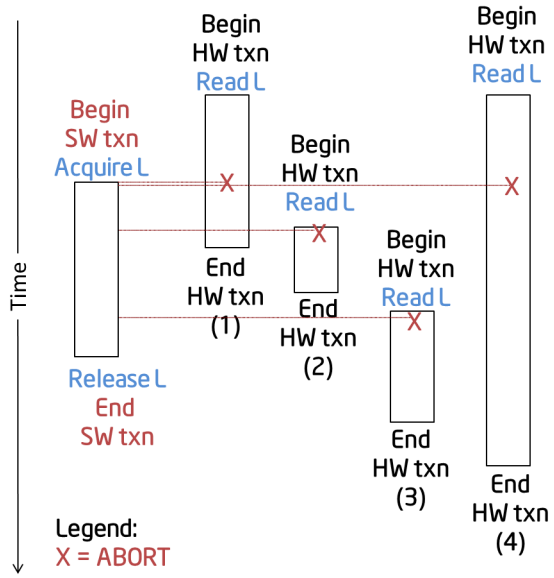


Figure 5.1: Obvious SGL Fallback implementation (E-SGL).

5.3 Lazy SGL (L-SGL)

In a naïve SGL implementation (E-SGL), a hardware transaction immediately adds the lock to its read set, ensuring the transaction will be aborted if that lock is acquired by a software transaction. Hardware and software transactions cannot overlap (Figure 5.1).

Lazy subscription can improve the chances of success of a hardware transaction by allowing some overlap with a software transaction. In Figure 5.2, L-SGL allows transactions (3) and (4) to commit, while E-SGL would abort them.

Software and hardware transactions are treated differently in L-SGL. Each software transaction must acquire the SGL. Hardware transactions do not acquire the SGL,

but they must check its status. With some exceptions described later, L-SGL hardware transactions read the lock only at the *end*, right before committing. If the lock is held by a software transaction, the hardware transaction explicitly aborts. This check is necessary because the hardware transaction may have observed an inconsistent state. If the lock is free, then no software transaction is in progress, and the hardware transaction can commit.

Lazy subscription has been proposed to improve HyTM performance [17], but its use for SGL fallback is new. HyTMs typically use sophisticated techniques to allow concurrency between multiple hardware and software transactions, but SGLs' simplicity makes them attractive in practice [89, 90]. The lazy SGL (L-SGL) approach described here improves a popular HTM fallback mechanism by allowing multiple hardware transactions to run concurrently with one software transaction.

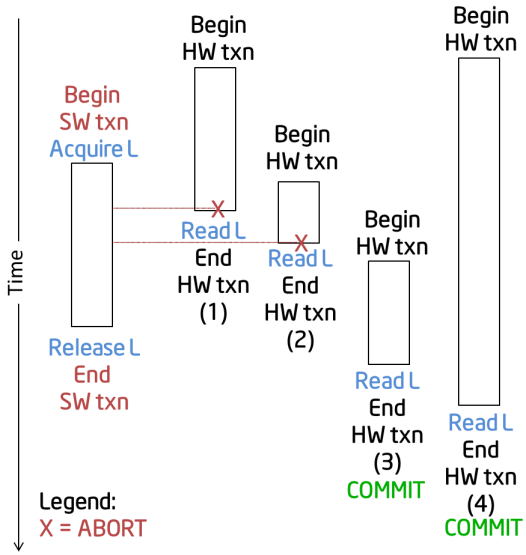


Figure 5.2: Lazy SGL (L-SGL).

Haswell RTM provides an abort status code that offers limited information about why a hardware transaction aborted. L-SGL makes it easier to collect diagnostic information about failed hardware transactions from this abort status code. When an E-SGL hardware transaction is about to start, it makes sense to wait until the

SGL is free. As a result, eager subscription rarely aborts hardware transactions explicitly at the time of subscription, so transactions are much more likely to be aborted automatically in-flight. Therefore, the abort status code will report this abort as a conflict. By contrast, L-SGL's lazy subscription mechanism makes it more likely that transactions will be aborted explicitly on subscription, allowing the programmer to obtain more detailed diagnostic information because, in this case, the abort status code can indicate precisely that the abort was caused by the lock.

L-SGL is similar to E-SGL in that it does not require read or write annotations, it permits transactions to be arbitrarily nested, but does not permit explicit transaction aborts in user code.

A software transaction waiting to acquire the SGL uses a combination of backoff and sleeping to reduce cache line contention. It starts by inserting an exponentially increasing number of null operations (NOPs) between successive lock attempts. When the number of NOPs reaches a threshold, T, the transaction calls the sleep function to release the processor for a brief duration before trying again. We found that sleeping right away is generally too slow for benchmarks where transactions are small and fast, but works well for larger and slower running transactions. Overall, we found that exponential waiting followed by sleeping works best across the range of benchmarks we considered.

Before a thread starts a hardware transaction, it reads the SGL to prefetch the lock into the cache. If no software transaction tries to acquire that lock, the lock is likely to be cached at commit time, which our experiments have observed to speed commit.

5.3.1 Correctness

STM designers often go to great efforts to ensure that all transactions see a consistent state, even after synchronization conflicts have occurred, a property called *opacity* [32]. The L-SGL design is simplified because hardware transactions do not need opacity. Instead, the L-SGL design relies on two guarantees. First, Haswell's hardware sandboxing mechanism ensures that any hardware transaction that raises an exception or enters an infinite loop because of an inconsistent state is aborted and rolled back without affecting any other transactions. Second, the L-SGL design ensures that no hardware transaction can commit while a software transaction is in progress. There is one exception, explained in the next section.

Fig. 5.3 illustrates why opacity is unnecessary: variables X and Y are linked by the invariant $Y = X + 1$. Now suppose a hardware transaction reads X and Y after a software transaction has incremented X, but before it has incremented Y, resulting in the inconsistent view $X = Y$. This hardware transaction will never commit, but it may encounter a division by zero when it evaluates $1/(Y - X)$. The Haswell hardware sandboxing mechanism will suppress the exception and roll back the transaction, ensuring that no other transaction is affected.

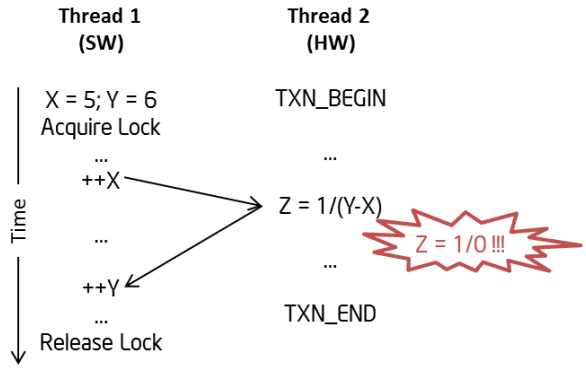


Figure 5.3: Inconsistent reads.

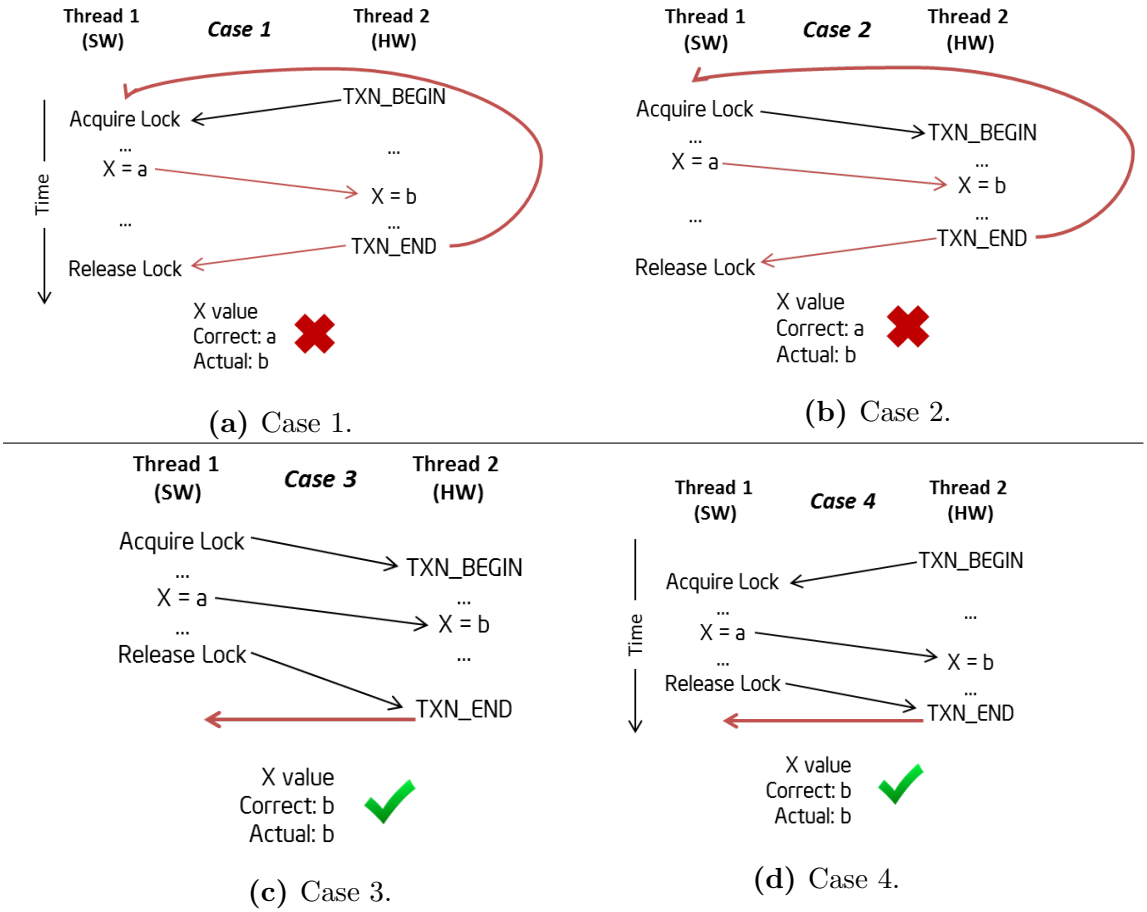
Fig. 5.4 outlines possible orderings between hardware and software transactions.

We order transactions by their commit time. Because software transactions cannot abort, any conflicting operation a software transaction executes after a hardware transaction has committed must be ordered after the hardware transaction. Moreover, because TSX provides no "escape actions" a hardware transaction cannot wait for a software transaction to commit.

In cases 1 (Fig. 5.4a) and 2 (Fig. 5.4b), the hardware transaction ends before the software transaction ends, and finds the lock taken when it tries to commit. In these two cases, the hardware transaction must be serialized before the software transaction. If a software transaction performs an operation that conflicts with a concurrently executing hardware transaction while the hardware transaction is still in-flight, the hardware transaction is aborted by the Haswell HTM conflict detection mechanism. If, on the other hand, the conflicting operation is performed by the hardware transaction, the conflict would not be detected. If both transactions were permitted to commit, the value of the conflicting location would be incorrect because the hardware overwrote the software transaction's write (see Fig. 5.4). Here, we must abort the hardware transaction, because software transactions cannot be aborted. It does not matter when the hardware transaction is aborted, so it is sufficient to check for conflicts as the final step of the hardware transaction before it commits. In L-SGL, such conflicts are detected by inspecting the state of the lock.

In cases 3 (Fig. 5.4c) and 4 (Fig. 5.4d), the hardware transaction begins its commit after the concurrent software transaction has committed. If the lock is free at the time of the hardware commit, then the hardware transaction can commit even though it might have overlapped one or more software transactions. Because the hardware transaction commits after any concurrently executing software transaction, it will be ordered after any such overlapping software transaction. Therefore the correct value for any conflicting location is the value written by the hardware transaction.

If the last value written to a location that conflicts with the hardware transaction belonged to the software transaction, then the hardware transaction would have aborted, because Haswell's HTM conflict detection system would have identified such a conflict and aborted the hardware transaction. Moreover, a software transaction observes only old values until the hardware transaction commits, so the software reads are serialized before the hardware writes.



(a) Case 1. **(b)** Case 2. **(c)** Case 3. **(d)** Case 4.

Figure 5.4: Correctness: Cases 1-4. Arrows denote the "happens-before" relation.

5.3.2 Sandboxing

Hardware sandboxing prevents faults that occur inside a hardware transaction from propagating outside of the transaction. Spurious writes and faults caused by reading

inconsistent state from the SGL transaction are not visible to other threads. There is, however, one unlikely situation when inconsistent reads can cause a hardware transaction to commit prematurely. In principle, inconsistent reads could cause a spurious write to a location that is later used by the same transaction as the target of an indirect jump. If the target of the incorrect jump is is an xend (commit) instruction, or data that looks like one, then the hardware transaction might commit incorrectly, without checking the lock. Note that the inconsistent transaction cannot actually change the program code and insert spurious xend instructions, as the code area is protected and accessing it would cause the transaction to abort.

To address this hazard, lazy subscription must be performed before any indirect jump executed inside a hardware transaction that has written to memory. A read-only transaction, or one that is read-only before the indirect jump is not subject to this hazard. Moreover, if a transaction makes multiple indirect jumps, it is sufficient to check the lock only before the first jump, because the lock remains in the transaction's read set.

In the results presented in Section 5.4, we use L-SGL with early subscription on the first indirect jump that occurs after a shared memory write. We found that this restriction did not affect performance.

In general, this problem is similar to security concerns caused by buffer overflows. There is a trend towards compiler support to help with this issue, which might also be used to protect hardware transactions from incorrect premature commits. For example, the latest GCC supports security functionality to check vtable integrity. Moreover, for optimizations levels higher than -O2, GCC uses devirtualization and inlining for the most likely target in indirect jumps. A transactional compiler could use similar techniques to generate multiple likely targets and use the early lock check only in the unlikely case that none of the pre-established targets are chosen.

5.4 Evaluation

Our experimental evaluation was performed on an Intel Haswell processor (Core i7-4770) with RTM and HLE enabled, running at 3.40 GHz. The machine has a total of 8GB of RAM shared across four cores, each having a 32 KB L1 cache. For our experiments, hyper-threading was enabled, giving us a total of eight hardware threads. However, we notice that hyper-threading negatively impacts performance at 8 threads due to L1 cache sharing. In practice, this results in more hardware transactions being aborted because of overflow. To show this effect, we performed a simple experiment in which we measured the rate of aborts due to overflow for one, two, four and eight threads for all STAMP benchmarks. The rate of overflow for 1 thread is indicative of the percentage of transactions that cannot succeed in hardware because of cache size or associativity limitations. As we increase the number of threads, the rate of overflow decreases, as more and more transactions abort because of conflicts with other transactions. However, for 8 threads, the rate of overflow significantly increases, showing the negative effects of hyper-threading, as can be seen for the Vacation High benchmark in Fig. 5.5. Results were similar for all other STAMP benchmarks, except for the Labyrinth benchmark, where most of the aborts are caused by unsupported instructions; we omitted these graphs due to space constraints.

We used GCC 4.8.2 compiler with -O3 optimization enabled and gcc intrinsics [49]. We used the STAMP benchmarks [15] to compare L-SGL's speedup relative to a single-threaded sequential execution with software only approaches - a state-of-the-art STM (TL2) and a single global lock (spinlock) without any transactional execution (SGL) - and with a hardware only solution (Haswell HLE). For HLE, we used a single global spin lock prefixed with HLE-Acquire and HLE-Release instructions to suggest that the critical section should be executed speculatively. If speculation

fails, the critical section is retried non-speculatively, according to a hardware policy. We also compared to the naïve SGL implementation with eager subscription (E-SGL). We ran all methods five times and presented the median of the results. Variance was generally low. We also compared L-SGL's rate of transactional success with that of HLE and E-SGL, by measuring the percentage of transactions executed non-speculatively for both methods.

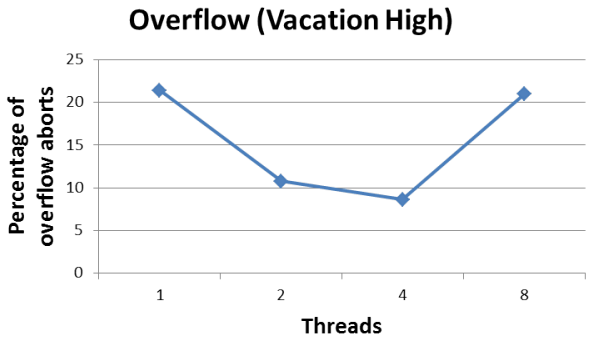
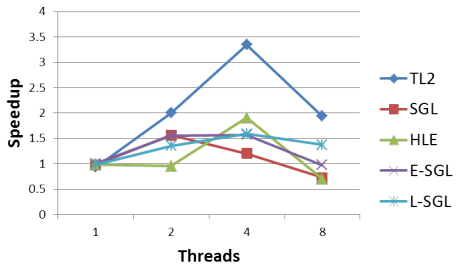


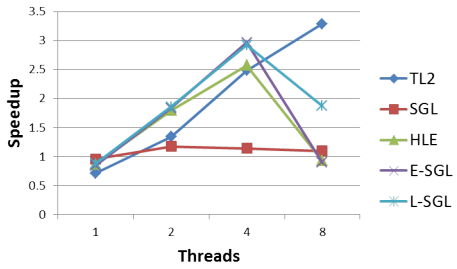
Figure 5.5: Example of overflow due to hyper-threading (vacation high benchmark).

5.4.1 Speedup relative to sequential execution

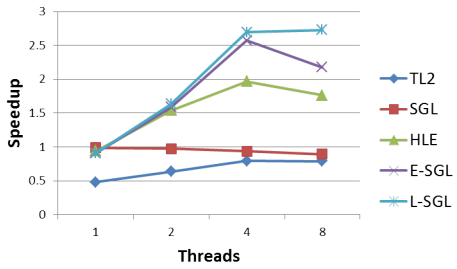
L-SGL performs best on benchmarks with medium sized transactions, such as Intruder 5.6c, Vacation Low 5.6h and Vacation High 5.6i, where it outperforms all prior methods. On the benchmarks with smaller transactions, such as Ssca2 5.6g, Kmeans Low 5.6d and Kmeans High 5.6e, L-SGL has good speedup compared to sequential execution, and outperforms TL2, which has too much overhead for these small transactions. However, L-SGL does not present a significant advantage compared to HLE on these benchmarks, because most transactions will quickly succeed in hardware, therefore making the differences between L-SGL and HLE less noticeable. For Kmeans Low 5.6d, where there is little contention, SGL performs similar to L-SGL and HLE as well. However, when there is more contention, as is the case with Kmeans High 5.6e, or when most transactions can succeed in hardware, in parallel,



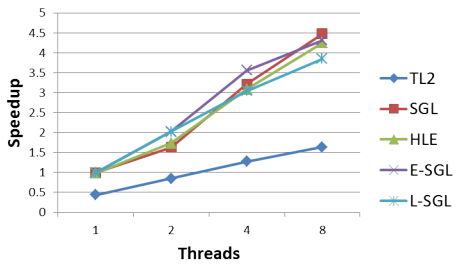
(a) Bayes.



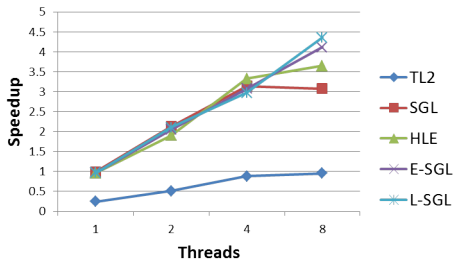
(b) Genome.



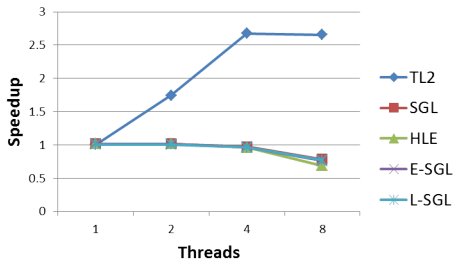
(c) Intruder.



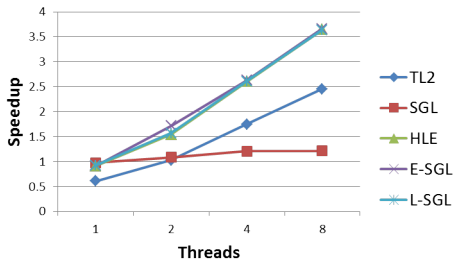
(d) Kmeans Low.



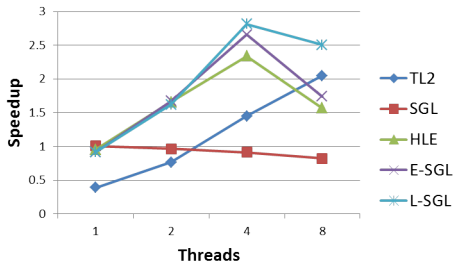
(e) Kmeans High.



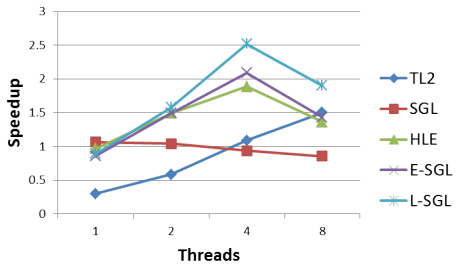
(f) Labyrinth.



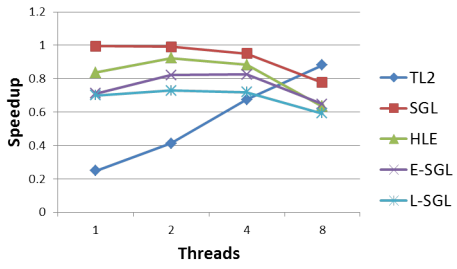
(g) Ssca2.



(h) Vacation Low.

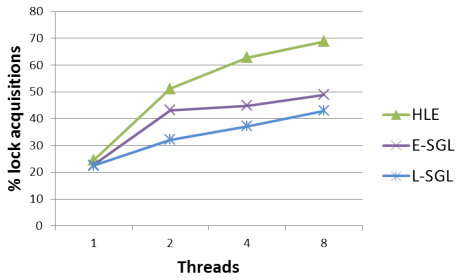


(i) Vacation High.

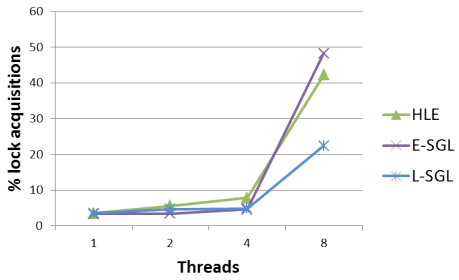


(j) Yada.

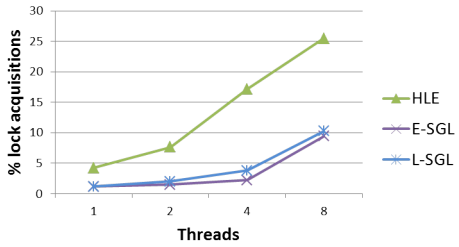
Figure 5.6: STAMP Throughput.



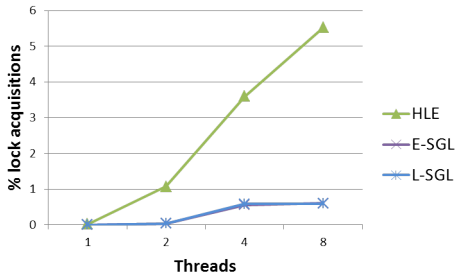
(a) Bayes.



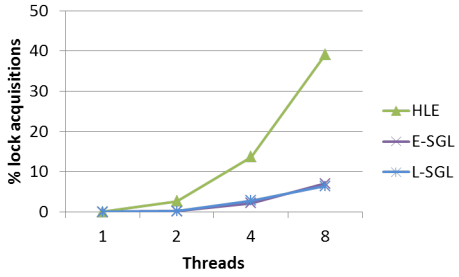
(b) Genome.



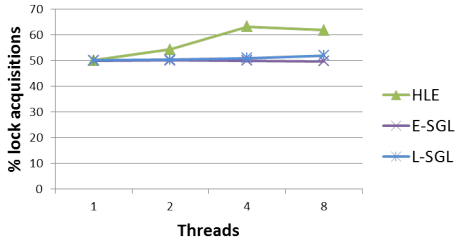
(c) Intruder.



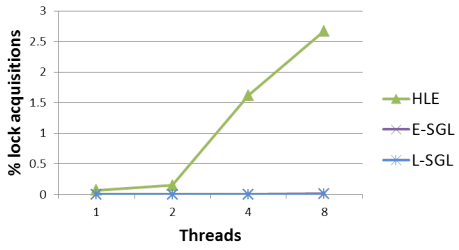
(d) Kmeans Low.



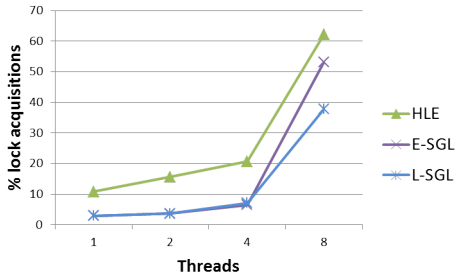
(e) Kmeans High.



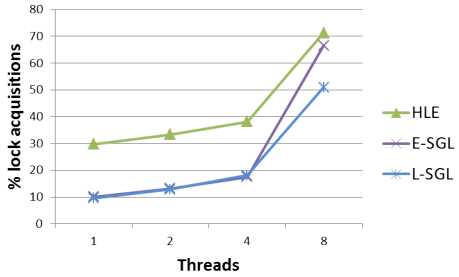
(f) Labyrinth.



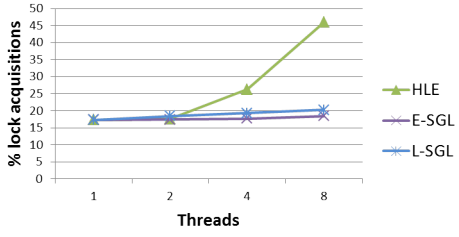
(g) Ssca2.



(h) Vacation Low.



(i) Vacation High.



(j) Yada.

Figure 5.7: STAMP Percentage of Lock Acquisitions.

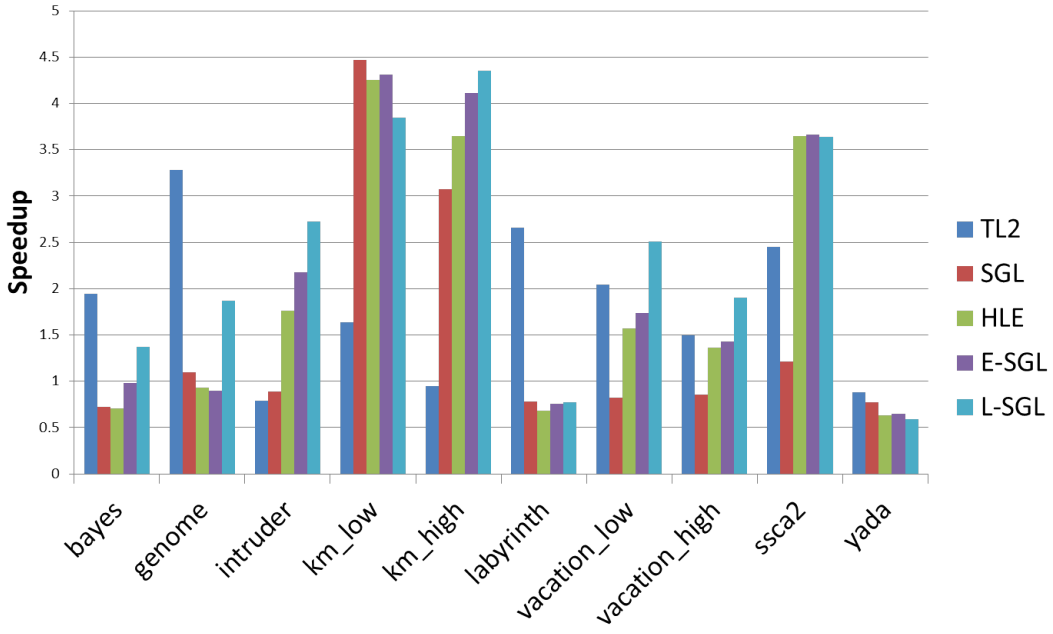


Figure 5.8: Speedup for 8 threads

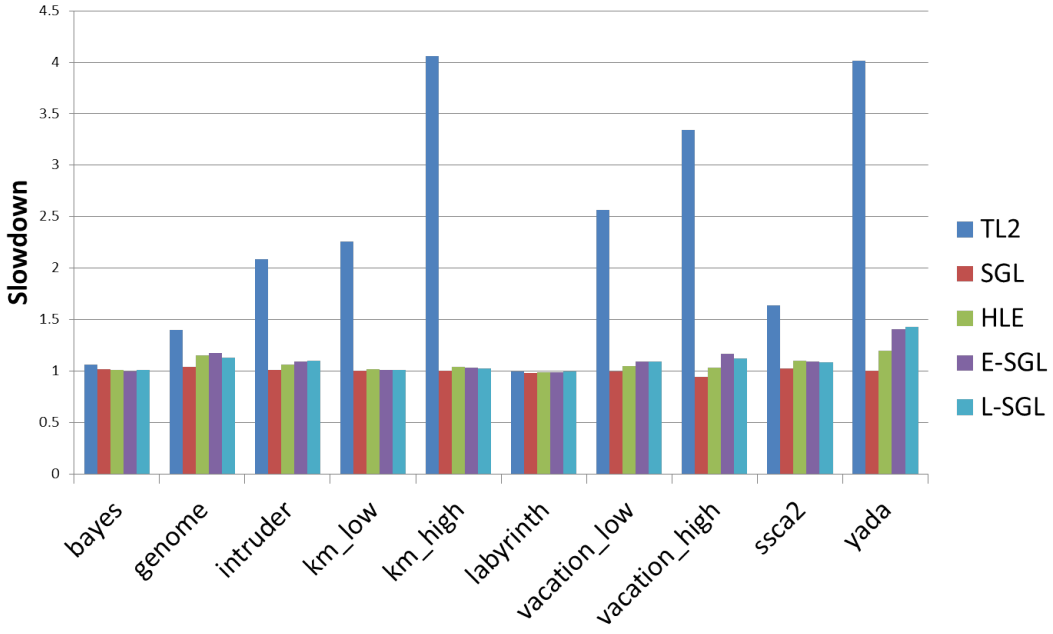


Figure 5.9: Slowdown for 1 thread.

as in Ssca2 5.6g, the performance of SGL quickly degrades.

Finally, for large transactions and those with unsupported instructions, as in Bayes 5.6a, Labyrinth 5.6f and Yada 5.6j, TL2 is more advantageous, because it can execute transactions in parallel, in software, without overflowing the cache. The effects of hyperthreading when running with 8 threads are even more pronounced on these benchmarks, because most transactions are large. We note that Labyrinth in particular is very suitable for STM systems because it uses very large transactions, whose initial memory accesses are all local. Therefore, these memory accesses do not contribute towards generating conflicts in an STM. Unfortunately, Haswell RTM does not have escape actions, therefore counting local accesses as transactional and overflowing the cache unnecessarily.

5.4.2 Percentage of lock acquisitions

We measured the percentage of lock acquisitions in L-SGL by inserting statistics in our code to measure the total number of transactions and the percentage executed non-speculatively. We measure the percentage of lock acquisitions in HLE using perf with support for TSX, a performance analysis tool for Linux.

We can notice in fig. 5.7 that L-SGL achieves a better rate of transactional execution than HLE on all STAMP benchmarks (its rate of lock acquisitions is lower than HLE's rate on all benchmarks). L-SGL uses lazy subscription, so the lock is read transactionally at the end of the critical section. In contrast, HLE subscribes to the lock address in the beginning of the critical section, suffering more aborts due to changes to the lock.

5.4.3 Single-threaded penalty

One of the biggest advantages of L-SGL is that it manages to improve performance for 4 and 8 threads without paying a big penalty for single threaded execution, as is the case with most STMs. For example, fig. 5.8 shows L-SGL's speedup relative to sequential for 8 threads and fig. 5.9 shows the slowdown for 1 thread. We can see that TL2 pays a huge penalty for single-threaded execution, while L-SGL execution is almost as good as sequential execution.

5.5 Fine-grained SGL

L-SGL allows multiple hardware transactions to execute concurrently with a software transaction as long as the software transaction commits first (Fig. 5.4c and Fig. 5.4d). Unfortunately, hardware transactions that attempt to commit while a software transaction is in progress will abort (Fig. 5.4a and Fig. 5.4b). This is the correct and expected behavior if there are conflicts between the hardware transactions and the software transaction, but otherwise these hardware transactions could successfully commit. Despite being an improvement over the simple single global lock algorithm, L-SGL does not enable the maximum amount of concurrency possible between multiple hardware transactions and a software transaction.

In this section, we describe another SGL fallback mechanism that performs finer grained conflict detection than E-SGL and L-SGL, based on Bloom filters (BF-SGL). BF-SGL increases the amount of concurrency offered by the hybrid transactional memory system in Fig. 5.4a and Fig. 5.4b. In order to detect conflicts between the software transaction and hardware transactions, we add a Bloom filter for each thread. Each read and write is annotated to add the memory location to the Bloom filter. Hardware transactions consult the global lock before committing and, if they

find it free, they can commit successfully. However, if the lock is taken, they can compare their Bloom filter with the software transaction's Bloom filter to determine if there are conflicts. The Bloom filter allows false positives, but not false negatives. Therefore, it could detect a conflict despite the transactions not having any conflicts, but it will never report zero conflicts if the transactions accessed the same memory. So the hardware transactions can commit successfully even if the lock is taken as long as the Bloom filters do not report conflicts. L-SGL represents a particular case of BF-SGL. Specifically, L-SGL can be obtained from BF-SGL if the Bloom filter set intersection operation between the hardware transaction trying to commit and the currently executing software transaction always returns that there exists at least one conflict.

5.5.1 Use Cases

Using BF-SGL, many small hardware transactions that access disjoint memory locations and concurrently executing large software transactions can commit. The same is not true for any other system that we are aware of. This is because we provide precise conflict detection using the Bloom filters for the HW and SW transactions to track memory accesses. Consider, for example, an array representing an open addressing hash-table. Threads can perform lookup(x) operations and insert(x) operations in this hash-table. Once a threshold of occupancy is achieved, a thread decides to double the size of the hash-table by allocating a new array and rehashing elements from the old array to the new array. Lookup and insert are short transactions and can succeed in hardware most of the time. Rehashing is always executed as a software transaction, so the thread needs to acquire the single global lock.

Using L-SGL, no lookup and insert operations can succeed during rehashing. However, using BF-SGL with precise conflict detection between the software transaction

and the concurrent hardware transactions, lookup operations executed as hardware transactions can commit using data from the old array while the rehashing to the new array is taking place. Moreover, insert operations executed as hardware transactions at the end of the old array, in the part that has not been rehashed yet, can also commit during rehashing. Therefore, BF-SGL improves throughput by allowing small hardware transactions to commit concurrently with long executing software transactions.

5.5.2 Performance and Practicality

Adding the software Bloom filter to hardware transactions incurs some overhead compared to simple hardware transactions. However, the Bloom filter adds the benefit of being able to commit hardware transactions even when a software transaction is executing as long as there are no real conflicts or false conflicts caused by the Bloom filter. An efficient Bloom filter implementation allows insertion and set intersection in O(1) time, minimizing the overhead.

In addition, reading these two locations in the hardware transaction only adds two additional cache lines to the read set of the transaction. This can be optimized so that a bit of the Bloom filter is used to indicate whether the lock is taken or not and the rest is used as a Bloom filter. Therefore, the lock location can serve both purposes, reducing the read set size of the hardware transaction to just one additional location. The transaction's own Bloom filters add additional cache lines to the write set, but this could be as low as only one cache line, depending on the Bloom filter size.

Hardware transactions read the software Bloom filter only right before committing, narrowing the window when hardware transactions could be aborted by software transactions. Unfortunately, the software transaction needs to modify its Bloom

filters for every read and write, causing many spurious aborts for the hardware transactions. We found that this behavior significantly affects the performance of BF-SGL, so we did not include results for this system. However, we note that this is a strong motivation why escape actions should be included with any HTM. If we had escape actions, the Bloom filters could be read non-transactionally at the end of the hardware transaction, avoiding the spurious aborts caused by the software transaction updating its Bloom filters. Correctness would still be maintained because any conflicting read or write performed by the software will still abort the hardware transaction. We believe this support will be available in the future, making the bloom filter based conflict detection a viable option. For example, IBM's Power ISA suspended mode [8] provides the necessary functionality.

5.6 Hardware Optimizations

Lazy Hardware Lock Elision (LHLE). Haswell's HLE works by eliding locks prefixed with HLE-Acquire and executing the critical sections as hardware transactions. If the speculation fails for any reason, the lock is acquired and the critical section is re-executed non-speculatively in software. HLE is similar to E-SGL: hardware transactions need to subscribe to the lock in the beginning of their execution to ensure correctness. However, we have shown that L-SGL, implemented in software, outperforms the hardware only HLE. Therefore, we speculate that Lazy Hardware Lock Elision (LHLE), where the lock is added to the read set at the end of the critical section, would perform better than HLE. Similar to HLE, LHLE enables multiple speculative critical sections to execute in parallel if there are no conflicts detected at run-time and it simplifies programming by enabling more parallelism for coarse-grained critical sections. In contrast to HLE, LHLE supports parallelism between one non-speculative critical section and multiple speculative critical sections. More-

over, LHLE is designed to be implemented entirely in hardware, so the sandboxing issues described in Section 5.3.2 do not arise, as the hardware can ensure that the subscription to the lock occurs whenever the xend instruction is invoked.

Bloom Filter Hardware Lock Elision. As described in Section 5.5, BF-SGL can improve the granularity of conflict detection with an SGL, but causes spurious aborts because the SGL transaction's Bloom filters become part of the read set of the hardware transactions. This could be avoided if the HTM allowed escape actions. In that case, the Bloom filters would be read non-transactionally to detect conflicts. Alternatively, if the Bloom filters were handled by the hardware instead of the software, they could avoid the tracking mechanism of HTM and avoid the unnecessary aborts. Haswell HLE could be extended with Bloom filters for the hardware transaction, as well as for the SGL transaction. With this design, conflict detection would be realized at a finer-grained level than it is currently done.

5.7 Summary

The naïve SGL fallback's simplicity makes it an appealing alternative to more complicated, even if better-performing, HyTM schemes. In this chapter, we introduced novel SGL methods that improve the performance of the simple SGL fallback, while maintaining its simplicity. First, we described L-SGL, a simple SGL-based fallback for HTM that uses lazy subscription to allow hardware-software transaction concurrency. L-SGL improves performance on current machines by up to 4X compared to state-of-the-art software and hardware solutions.

In addition, L-SGL has some appealing properties. For example, it does not require read and write annotations, making it suitable for implementation in a real system, either in the compiler or even in hardware. Our L-SGL software implementation

improves performance over native Haswell lock elision by almost a factor of 2, and reduces the rate of lock acquisitions by up to 35%. We conjecture this difference would be even higher if L-SGL were implemented in hardware.

We also described BF-SGL, an alternative SGL fallback mechanism with more accurate conflict detection. Our BF-SGL results, perhaps counter-intuitively, show that adding a mechanism to support better conflict detection, such as Bloom filters, hinders performance by increasing the abort rate. If the HTM were to support escape actions, allowing precise conflict detection to be performed outside of transactional tracking, we speculate that this comparison would change in favor of BF-SGL. Finally, we showed how to use these ideas to improve future HTMs with minimal microarchitectural changes.

Chapter 6

Hybrid Transactional Memory

The Intel Haswell processor includes restricted transactional memory (RTM), which is the first commodity-based hardware transactional memory (HTM) to become publicly available. However, like other real HTMs, such as IBM's Blue Gene/Q, Haswell's RTM is best-effort, meaning it provides no transactional forward progress guarantees. Because of this, a software fallback system must be used in conjunction with Haswell's RTM to ensure transactional programs execute to completion. To complicate matters, Haswell does not provide escape actions. Without escape actions, non-transactional instructions cannot be executed within the context of a hardware transaction, thereby restricting the ways in which a software fallback can interact with the HTM. As such, the challenge of creating a scalable hybrid TM (HyTM) that uses Haswell's RTM and a software TM (STM) fallback is exacerbated.

In this chapter, we present Invyswell, a novel HyTM that exploits the benefits and manages the limitations of Haswell's RTM. After describing Invyswell's design, we show that it outperforms NOrec, a state-of-the-art STM, by 35%, Hybrid NOrec, NOrec's hybrid implementation, by 18%, and Haswell's hardware-only lock elision by 25% across all STAMP benchmarks.

6.1 Background

Traditionally, locks have been the predominant mechanism used to synchronize shared memory in multithreaded programs [44]. Yet, developing software that correctly and efficiently uses locks is notoriously challenging, even for the most seasoned programmers. Transactional memory (TM) has been proposed as an alternative to locks, where much of the mechanical complexity of synchronization is managed by the underlying system, not the programmer [43, 84].

Experience with software transactional memory (STM), where transactions are implemented entirely in software, has demonstrated the simplicity of transactional programming, but has raised challenging performance issues. Modern STMs tend to be scalable at high thread counts [26], meaning that beyond a certain point (and up to a limit), adding more threads typically increases throughput for many benchmarks, yielding performance that is often competitive with fine-grained locking. Unfortunately, these STMs tend to perform poorly at low or medium thread counts, because of non-amortized transactional overhead, resulting in performance that is not competitive with fine-grained locking.

To improve the performance of transactions, hardware vendors such as Intel and IBM have included support for hardware transactional memory (HTM). One such example is Intel's Haswell processor [49], which includes restricted transactional memory (RTM), a cache-based HTM design that uses the microarchitecture's existing cache coherence protocol to manage transactional conflicts. Yet, it is unclear how RTM can be most effectively used by software. One cannot simply substitute hardware transactions for software transactions, because RTM, like other HTMs, such as IBM's Blue Gene/Q [89] and System z [51], is *best-effort*, providing no progress

guarantees. [1] Whether a transaction succeeds depends on whether its data set fits in the processor's cache, whether the transaction finishes without interruption, and a myriad of other architectural and platform-specific limitations best hidden from the programmer.

It has been recognized that effectively integrating best-effort HTM with the software that uses it requires an intermediate software fallback when hardware transactions fail. Such a system is called hybrid transactional memory (HyTM) [19, 17, 22, 58], where hardware and software transactions execute under the umbrella of a single TM system. In this chapter, we present a novel HyTM, called *Invyswell*, that uses hardware transactions from Haswell's RTM in conjunction with software transactions from a heavily modified design of InvalSTM [30], an STM designed to provide scalability and performance for large transactions with notable contention.

Invyswell enables the concurrent execution of both hardware and software transactions with the aim of being performant for all transaction sizes and degrees of contentions. Haswell's RTM performs best for small transactions with low contention, as it imposes no instrumentation overhead, but is limited to a "requester-wins" contention policy. InvalSTM performs best for large transactions with high contention, because it can make highly informed contention management decisions through its commit-time invalidation process. Yet, challenges remain in finding an efficient solution for the "transactional twilight zone" - midsize transactions that are small enough to successfully execute in hardware but have a non-trivial degree of contention. Furthermore, even after designing a TM that addresses the unique challenges of each of these categories, that system must ensure that each individual component does not negatively impact the overall performance by mismanaging transactions for which it was not intended. Invyswell addresses this by using a sophisticated design that

[1] Although System z supports constrained transactions, which are guaranteed to commit, we believe this does not present a generalized mechanism for HTM forward progress as constrained transactions are size-restricted.

employs several hardware and software modes of execution. This gives the system the flexibility to trade execution overhead for precision in conflict detection.

Haswell's RTM does not support *escape actions*, non-transactional instructions executed within transactions [70]. This limitation complicated our design, especially with respect to opacity [32], a correctness conditions that guarantees consistency of eventually-aborted transactions. Another challenge we encountered was designing Invyswell's *contention manager* (CM), a decision-making process aimed at improving throughput, due to the different isolation properties for hardware and software transactions. The lack of escape actions further complicated this issue, as well, as it restricts the way a hardware transaction can abort a software transaction before the hardware transaction itself commits.

We evaluate Invyswell's performance using the STAMP benchmark suite. Invyswell's performance compares favorably to that of pure software, pure hardware, and hybrid solutions. Invyswell is 35% faster than NOrecSTM [18], a state-of-the-art software transactional memory, and 18% faster than NOrecHy [17], a state-of-the-art hybrid transactional memory, as shown in Figure 6.1. It also outperforms Haswell's native *hardware lock elision* (HLE) [48, 75], a hardware mechanism that attempts to elide locks by executing critical sections as transactions and supports transactional re-execution with single global lock fallback implemented purely in hardware. Although on the average Invyswell is only 25% faster than HLE, the performance difference is significant for some benchmarks with large transactions, where Invyswell outperforms HLE by 2× to 5.4×.

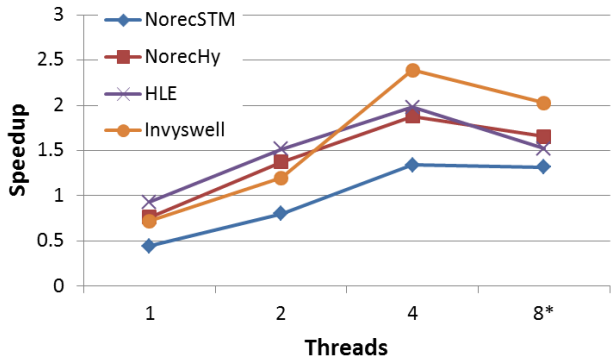


Figure 6.1: STAMP Performance Differential by Geometric Mean. *Hyperthreading is enabled for 8 threads. (Note: NOrec and Hybrid NOrec are abbreviated as NorecSTM and NorecHy, respectively, in the legend)

6.2 Overview of InvalSTM

One of the key differences between InvalSTM and other STMs is that it performs commit-time invalidation [30]. This approach requires that a transaction identify and resolve conflicts with all other in-flight (i.e., concurrently executing) transactions during its commit phase. InvalSTM achieves this by storing read and write sets in transaction-specific Bloom filters so it can perform conflict detection using constant-time set intersection. With commit-time invalidation, InvalSTM has complete knowledge of all conflicts between a committing transaction and other in-flight transactions, allowing it to make informed decisions on how to best mitigate contention. *All* InvalSTM transactions perform validation to achieve opacity in $O(N)$ total computational complexity, where N is the number of read elements, which is notably faster than the $O(N^2)$ overhead incurred by incremental validation and can drastically reduce the opacity cost for large transactions. Additionally, read-only transactions commit without incurring any commit-time serialization overhead.

For these reasons, InvalSTM naturally complements Haswell's RTM. Haswell's RTM can be used for small transactions and low thread counts, while InvalSTM can be used for large transactions and high thread counts. Moreover, Haswell's RTM can lever-

SpecSW

```
tx_begin:
fetch_and_add(&sw_cnt, 1);
do { local_cs = commit_sequence;
} while (local_cs & 1);
tx_read:
if addr in write hash table
   return val;
add addr to read bf;
val = *addr;
validate();
return val;
tx_write:
if (status == INVALID) restart;
add addr to write bf;
add addr,val to local hash table;
```

```
tx_end:
if (readonly)
  fetch_and_sub(&sw_cnt, 1);
  return;
acquire commit_lock;
validate();
if !CM_can_commit() restart();
fetch_and_sub(&sw_cnt, 1);
commit();
tx_post_commit:
invalidate();
release commit_lock;
```

LiteHW

```
tx_end:
if (!commit_lock && !sw_cnt) _xend();
else _xabort();
```

BFHW

```
tx_read: add addr to read bf
tx_write: add addr to write bf
tx_end:
if (!commit_lock)
    ++hw_post_commit;
    _xend();
else
  if (!conflict with SW txn)
      ++hw_post_commit;
      _xend();
else _xabort();
tx_post_commit:
if (!readonly) invalidate();
fetch_and_sub(&hw_post_commit, 1);
```

IrrevocSW

```
tx_begin: acquire commit_lock
tx_read: add addr to read bf
tx_write: add addr to write bf
tx_end: Do nothing
tx_post_commit:
if (!readonly) invalidate();
release commit_lock
```

SglSW

```
tx_begin:
acquire commit_lock
++commit_sequence
tx_end:
++commit_sequence
Release commit_lock
```

Figure 6.2: Transactional Events for Invyswell's Different Transaction Types.

age InvalSTM's use of Bloom filters for conflict detection by augmenting Haswell's hardware transactions with Bloom filters to enable many hardware transactions to execute concurrently with many software transactions. These Bloom filters are a good fit for Haswell's cache-based HTM design because they can be structured for constant-sized cache line alignment, thereby minimizing the negative impact of introducing hardware-to-software conflict detection into an already restricted HTM space. Finally, because InvalSTM's read-only transactions do not introduce any serialization in their execution, the performance overhead for transactions is transparent to Haswell RTM's faster executing hardware transactions. This enables Haswell's RTM to perform without interference when read-only software transactions are executing within InvalSTM, regardless of their size.

6.3 Invyswell's Design

In this section, we describe Invyswell, a HyTM that supports the concurrent execution of multiple hardware and multiple software transactions while guaranteeing forward progress. Invyswell uses Haswell's RTM [49] and a modified version of InvalSTM [30]. In InvalSTM, when a transaction is ready to commit, it marks conflicting in-flight transactions as invalid. InvalSTM uses Bloom filters for fast conflict detection between software transactions. Invyswell also uses Bloom filters at times, but not always, for conflict detection between hardware and software transactions.

Because Haswell's RTM does not support escape actions, the communication between in-flight hardware and software transactions is essentially impossible without introducing conflicts between them. For example, if a software transaction writes to memory shared by a hardware transaction, the latter will abort. Yet, communication between hardware and software transactions might be useful to improve the

precision of conflict detection between them, thereby increasing throughput in cases when conflicts do not occur.

To manage this space, Invyswell generally performs conflict detection between a hardware and a software transaction *after* the hardware transaction has committed. This enables increased throughput in cases where no conflicts exist while minimizing the chance of aborting a hardware transaction because of communication with in-flight software transactions.

Furthermore, Invyswell exploits the observation that hardware transactions do not need to check for conflicts with software transactions until just before committing, a mechanism called *lazy subscription*, which was introduced by Dalessandro et al. in their NOrec HyTM system [17]. By using lazy subscription, Invyswell reduces the "window of vulnerability" in which a write to a software transaction's conflict detection metadata (e.g., its read set, its execution lock, etc.) will abort a non-conflicting, in-flight hardware transaction.

Invyswell supports five transactions types, motivated by the need for progress guarantees and adaptability to different types of workloads. Two types are in hardware, lightweight (LiteHW) and bloom filter-based (BFHW), and three types are in software, speculative (SpecSW), irrevocable (IrrevocSW), and single global lock (SglSW). The pseudocode for these transaction is shown in Figure 6.2. Invyswell's state transitions between them are shown in Figure 6.3.

6.3.1 SpecSW: An HTM-Friendly InvalSTM

Invyswell's first type of transaction is the speculative software transaction (SpecSW), which is similar to an InvalSTM transaction, and is shown in Figure 6.4. It tracks its read and write locations in transaction-specific Bloom filters and stores its write

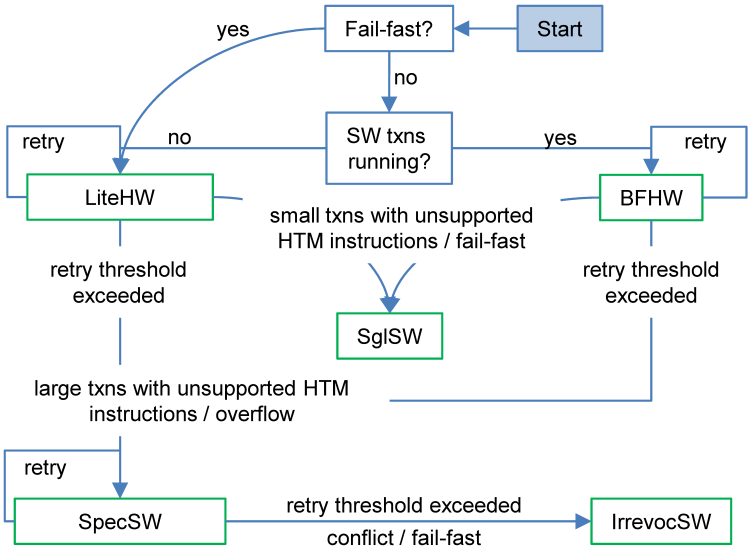


Figure 6.3: Invyswell's State Machine Describing the Transitions Between the Different Transaction Types.

set's values in a hash table for deferred update during its commit phase. Note that a memory barrier is necessary after inserting a memory location in a read bloom filter and before reading the value from memory. At commit-time, a SpecSW performs invalidation, where it compares its write Bloom filter against all other in-flight SpecSWs' read Bloom filters. If a conflict is found, it consults the contention manager (CM) on how to proceed. The CM then either aborts the committing transaction or permits it to commit. If permitted to commit, the SpecSW transaction updates all write locations and then marks all conflicting in-flight transactions as invalid. During a SpecSW's execution, it checks to see if it has been marked as invalid prior to each read and write and prior to committing. If it has, it aborts and it retries again as a SpecSW or another type as illustrated in Figure 6.3.

A key difference between Invyswell and InvalSTM is that SpecSWs perform invalidation *after* committing changes to memory, unlike InvalSTM, which performs invalidation *before*. The reason for doing this is the following. In InvalSTM, new transactions acquire an in-flight lock to insert their transaction ID into an in-flight linked list. If Invyswell did the same, hardware transactions would have to read this

lock before committing, to ensure correctness in their conflict detection. However, reading such a lock could subsequently cause many unnecessary hardware transaction aborts because whenever a new SpecSW was added to the list the in-flight lock would be acquired, automatically aborting all hardware transactions that previously read it.

To avoid this behavior, Invyswell performs invalidation after committing SpecSW's changes to memory and uses a slotted array for the in-flight SpecSWs, rather than a linked list. The combination of these changes results in Invyswell's elimination of the InvalSTM in-flight lock, thereby reducing the likelihood of unnecessary hardware transaction aborts. Instead, if a new transaction starts while the committing transaction is updating memory, it will be detected by the invalidation phase of the committing transaction, which will follow the memory update phase. Alternatively, if the new transaction starts after the memory was already updated, it could be missed by the invalidation phase. However, this new transaction is guaranteed to only read consistent states because the committing transaction has finished updating the memory, making the bloom filter check unnecessary for this transaction.

Initially, this modification results in the loss of opacity for SpecSWs, however, we restore opacity for SpecSWs by adding inexpensive validation to each read as described in Section 6.3.7. This change makes SpecSWs compatible with hardware transactions that can invalidate in-flight SpecSWs and it permits Invyswell to eliminate the need for an in-flight lock and the per-transaction locks that are required by InvalSTM.

6.3.2 BFHW: Hardware-Software Conflict Detection

Invyswell's second type of transaction is the Bloom filter hardware transaction (BFHW). BFHWs execute in hardware and, like SpecSWs, record the memory loca-

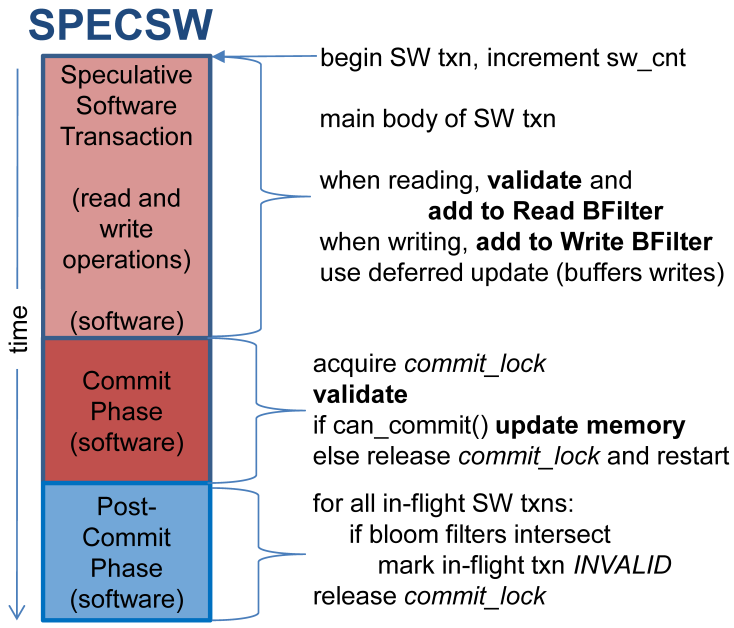


Figure 6.4: Speculative Software Transaction (SpecSW).

tions they read and write in transaction-specific software Bloom filters.

At commit time, if a BFHW sees the software *commit_lock* is free, it increments the *hw_post_commit* counter, which subsequently prevents SpecSWs from committing or reading new values while its value is non-zero, and then commits its speculative writes to memory and performs post-commit invalidation on all in-flight SpecSWs, where all conflicting transactions are marked as invalid. The BFHW then decrements the *hw_post_commit* counter to indicate its post-commit phase has completed, allowing software transactions to again commit, as shown in Figure 6.5.

The *hw_post_commit* counter is necessary because there is a window of vulnerability after a BFHW has committed, but before it has finished executing the invalidation phase, when SpecSWs can read inconsistent values written by the BFHW. Without the *hw_post_commit* counter these SpecSWs will be marked as *invalid* by the BFHW during its invalidation phase, but they could still execute momentarily returning inconsistent reads, causing SpecSWs to lose their opacity.

Alternatively, if the *commit_lock* is taken when a BFHW enters its commit phase, this means a SpecSW is committing. In this scenario, the simplest option is for the BFHW to abort, because there may be a conflict with the committing SpecSW. However, because BFHWs track their read and write accesses, Invyswell can instead perform conflict detection between the committing BFHW and the committing SpecSW via Bloom filter set intersection. If an overlap is found, the BFHW is aborted. Otherwise, no conflict exists between the BFHW and the SpecSW, and, because their respective read and write sets are immutable during their commit phases, the BFHW is permitted to commit.

When a SpecSW commits, it releases the *commit_lock* before clearing its read and write sets. This ensures that if the SpecSW commits before a committing BFHW performs conflict detection against the committing SpecSW, that the BFHW is automatically aborted because the write performed by the SpecSW to the *commit_lock* would trigger a hardware conflict with the BFHW from its prior read.

Note that if a BFHW transaction aborts, its *hw_post_commit* counter increment never becomes visible, because it is part of its speculative write set. Moreover, the new counter value becomes visible only when the hardware transaction commits. If a SpecSW reads this counter after it has been written to by a BFHW, but before the BFHW has committed, the BFHW will be automatically aborted by Haswell RTM's strong isolation property, thereby avoiding a race.

6.3.3 LiteHW: Optimizing for Small Transactions

Although BFHWs enable the concurrent execution of hardware and software transactions, they come with added overhead because each load and store requires an associated Bloom filter insert operation. Invyswell addresses this limitation with its third type of transaction, the LiteHW.

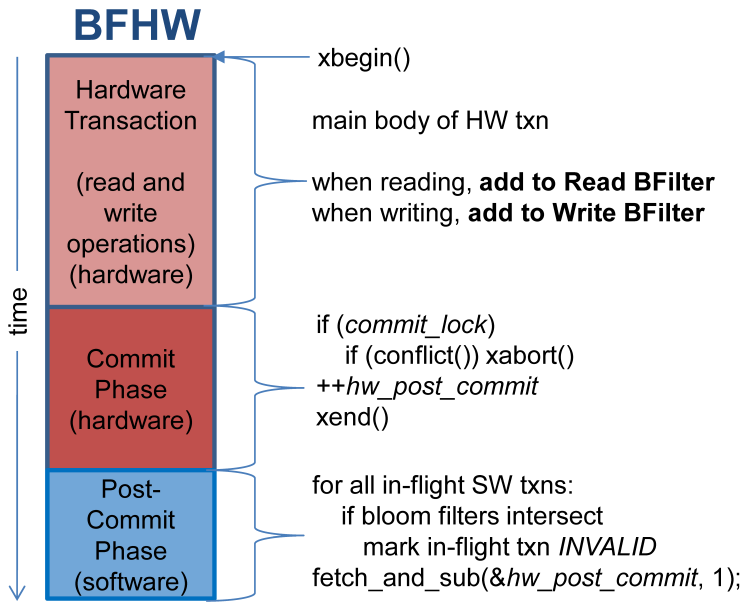


Figure 6.5: Bloom Filter Hardware Transaction (BFHW).

LiteHWs are lightweight hardware transactions, which execute without read or write annotations. They can only commit if there are no in-flight software transactions when they begin their commit phase. Unfortunately, because LiteHWs do not maintain read or write set metadata, if a software transaction is in-flight when a LiteHW enters its commit phase, Invyswell must assume a conflict exists between the LiteHW and the software transaction and, therefore, must abort the LiteHW. LiteHWs determine if there is an in-flight software transaction by reading the *commit_lock* and the software transaction counter, *sw_cnt*, prior to committing. Because LiteHWs do not perform conflict detection against software transactions, they require no post-commit phase.

6.3.4 IrrevocSW: Progress Guarantees

InvalSTM guarantees forward progress by using transaction-specific priorities that are incremented each time a transaction is aborted. Using this mechanism, a contin-

uously aborted transaction will eventually yield the highest priority and is guaranteed to commit. Invyswell's BFHWs, however, deviate from this model and instead commit memory changes first and perform invalidation second, at which point all conflicting software transactions are aborted. Because of this change, there is a danger that BFHWs could repeatedly abort high-priority SpecSWs, resulting in their starvation.

To address this problem, Invyswell introduces a fourth transaction type, the IrrevocSW, a direct update irrevocable transaction type that cannot be aborted. To ensure irrevocability, IrrevocSWs acquire the *commit_lock* as soon as they begin their execution and hold it until they have committed. To enable conflict detection with other transactions, an IrrevocSW transactions records its read and write locations in Bloom filters. An IrrevocSW needs no commit phase, because its writes are in-place. Its post-commit phase invalidates conflicting in-flight SpecSWs. While an IrrevocSW is executing, SpecSWs are required to perform validation and are disallowed from committing. Furthermore, LiteHW transactions must abort if their commit phase overlaps with any part of an IrrevocSW's execution. However, BFHWs can execute concurrently with an IrrevocSW. Yet, to ensure correctness, a BFHW needs to check for conflicts with the IrrevocSW transaction prior to committing its changes to memory and it must abort itself if a conflict is found.

6.3.5 SglSW: Progress Guarantees with Reduced Overhead

Small transactions that execute instructions not supported by Haswell's RTM need to be executed in software. However, both SpecSWs and IrrevocSWs add transactional metadata that may be too expensive for transactions that only access a few memory elements. To address this need, Invyswell employs a final transaction type that uses a single global lock without any associated transactional metadata.

This transaction type, SglSW, uses direct update and is irrevocable. SglSW is fast, but it does not allow the concurrent execution of other software transactions. Because SglSW does not track its reads or writes, it cannot perform conflict detection. Instead, it uses a sequence lock to force all in-flight SpecSWs to abort and acquires the *commit_lock* when it begins its execution to prevent IrrevocSWs from starting. BFHW and LiteHW transactions abort if an SglSW is executing when they try to commit. However, SglSWs allows for some overlap in execution with BFHWs and LiteHWs, as long as the executing SglSW commits before the hardware transactions do, thereby ensuring that the hardware's strong isolation property aborts any BFHWs and LiteHWs that conflict with the SglSW.

6.3.6 Transitioning Between Transaction Types

Transactions are scheduled opportunistically, first as fast, high-risk hardware transactions, then as slower, low-risk software transactions as shown in Figure 6.3. Each transaction is first tried in hardware, as LiteHW or BFHW, depending on whether other software transactions are present. If the hardware abort status suggests that a transaction is unlikely to succeed in hardware, then it is retried as a SpecSW. If it fails again, it is either retried as a SpecSW or it is escalated to irrevocable status, preventing it from aborting and ensuring progress. The transitions between the different types are decided automatically at runtime based on a heuristic that is application-independent. [2]

[2]Due to limitations in Intel's first generation HTM (e.g., imprecision on a transaction's abort status and limitations of only four concurrent hardware threads, eight with hyperthreading) Invyswell's state transitions deviate slightly from that shown in Figure 6.3. In particular, we use a modified design that transitions to SglSW when SpecSWs fail.

6.3.7 SpecSW Validation

InvalSTM performs invalidation before committing a transaction's writes to memory. It uses a per-transaction *invalid* flag which is set to true when a committing transaction invalidates a conflicting in-flight transaction. For reasons described in Section 6.3.1, Invyswell departs from this design and performs invalidation after committing a SpecSW's writes to memory. Unfortunately, this change makes InvalSTM's approach to ensure opacity – using the transaction's *invalid* flag – insufficient for SpecSWs. Instead, on every new read that is not present in a SpecSW's write set, Invyswell inserts the new read location into the SpecSW's Bloom filter and only then is the SpecSW permitted to read the value. This ensures that a potential conflict will not be missed by another transaction's invalidation phase. Next, the SpecSW performs the validation process shown in Figure 6.6. This validation process is necessary because of the interactions SpecSWs can have with different transactions and the inconsistent reads they might cause, as we explain next.

SglSW First, a SpecSW read could be inconsistent due to a concurrently executing SglSW. Because SglSWs do not store reads and writes using Bloom filters, conflict detection cannot be performed between them and a SpecSW. Thus, the SpecSW must abort if the *commit_sequence* has changed (Line 1 in Figure 6.6) after it was read at *tx_begin* (Figure 6.2).

IrrevocSw Second, a concurrently executing IrrevocSW or a committing SpecSW could cause an inconsistent read. Thus, the SpecSW read must check if the read location is in the Bloom filter of the transaction holding the *commit_lock* (Line 2 in Figure 6.6). If so, it must abort. If *commit_lock* changes during the read validation, the conflict may go unnoticed by the validation code. However, if the lock has changed, it means the transaction that released it must have finished the invalidation

phase. Therefore, it is sufficient to check if the SpecSW has been invalidated in the meantime (Line 4 in Figure 6.6).

BFHW Finally, a SpecSW must wait for all committed BFHWs to finish invalidation (*hw_post_commit* to reach zero) before using a new read value (Line 3 in Figure 6.6). If the SpecSW is not marked as invalid, the read is safe (Line 4 in Figure 6.6).

```
1. if commit_sequence changed: restart();
2. if conflict with committing SW txn: restart();
3. while (hw_post_commit!=0);
4. if (status==INVALID): restart();
```

Figure 6.6: Overview of Invyswell's SpecSW Validation Process.

6.3.8 Contention Manager (CM)

SpecSWs consult the CM during the commit phase to acquire permission to commit. As in InvalSTM, the CM considers all in-flight transactions that would be aborted if the committing transaction was allowed to commit. Any CM policy can be used. Invyswell uses iBalanced [29], which makes decisions based on priority, read and write set sizes, and other factors.

Invyswell has trade-offs that the original InvalSTM design does not have. For example, InvalSTM's ability to make decisions based on complete knowledge of in-flight transactions is lost. Essentially, there is no CM for Invyswell's hardware transactions because Haswell's RTM does not support escape actions, and thus a hardware transaction has to abort all conflicting software transactions after the hardware transaction has committed. The side-effect of this approach is that, conceptually, hardware transactions are likely to scale to high thread counts only when there is little to no contention, even if mitigation of that contention could be possible with an intelligent

CM. On the other hand, software transactions retain a complete knowledge of the CM decision-making process, enabling them to scale for high thread counts amidst high contention when the contention can be managed to provide wide transactional throughput.

6.4 Correctness

Figures 6.2 and 6.3 show the five types of Invyswell transactions and the transitions between them, respectively. In this section, we give an informal explanation why these five transaction types can run concurrently with one another without violating atomicity, as shown in Figure 6.7. However, atomicity by itself does not guarantee that aborted transactions are opaque; that is, that they only observe consistent states, a topic we discuss in Section 6.4.1.

Types	BFHW	LiteHW	SpecSW	IrrevocSW	SglSW
BFHW	yes	yes	yes	yes	yes
LiteHW	yes	yes	yes	yes	yes
SpecSW	yes	yes	yes	yes	no
IrrevocSW	yes	yes	yes	no	no
SglSW	yes	yes	no	no	no

Figure 6.7: Invyswell's Concurrent Execution Matrix.

LiteHW and BFHW vs. LiteHW and BFHW Haswell's hardware transactions are *strongly isolated*, meaning that their changes to memory become visible to other threads only on commit, whether those threads are executing a transaction or not. The hardware automatically detects conflicts between these types of transactions, and any conflict will abort at least one transaction. There is no need for additional mechanisms to synchronize concurrently executing LiteHWs and BFHWs with respect to each other.

LiteHW vs. Software Transactions LiteHWs can execute concurrently with Invyswell's software transactions, but they cannot commit while such software transactions are executing. A LiteHW that overlaps execution with a software transaction (SpecSW, IrrevocSW, or SglSW) can commit only after the software transaction has committed, otherwise the resulting execution may be not serializable. A LiteHW that tries to commit while a software transaction is executing will abort. Such behavior is detected by the *sw_cnt* counter and the commit lock (see Figure 6.2).

BFHW vs. SpecSW or IrrevocSW Unlike LiteHWs, BFHWs use software Bloom filters to keep track of the memory locations they access. By performing explicit conflict detection with these Bloom filters, BFHWs can commit in the presence of software transactions. If a committing SpecSW conflicts with an in-flight BFHW, then the BFHW will automatically be aborted by the hardware when the SpecSW writes its speculative data to memory. If a committing BFHW conflicts with an in-flight SpecSW, the SpecSW will be aborted during the BFHW's post-commit invalidation phase. Moreover, BFHWs' use of lazy subscription means it is sufficient to compare the Bloom filters of BFHWs and SpecSWs at the end of the hardware transaction.

Postponing conflict detection to the end of the BFHW's execution narrows the window in which it will be aborted by false conflicts. Moreover, SpecSWs' Bloom filters do not change while it is committing, so a BFHW can read them without being aborted due to metadata interference (i.e., non-transactional interference). Note that SpecSWs that are doomed to abort after a BFHW invalidates them could read inconsistent memory before they notice they were aborted, generating faulty behavior. For this reason, atomicity by itself is not the only TM correctness property that Invyswell guarantees, an issue we discuss in Section 6.4.1.

SpecSW vs. SpecSW Conflict detection between multiple SpecSWs uses invalidation. A committing SpecSW checks for conflicts with other in-flight SpecSWs and, if conflicts are found, the committing SpecSW either aborts itself or invalidates the SpecSWs it conflicts with. No SpecSW can commit during another SpecSW's invalidation process because the committing SpecSW holds the commit lock.

IrrevocSW vs. Software Transactions An IrrevocSW acquires the commit lock as soon as it becomes active, ensuring that no other software transaction can become irrevocable (i.e., other IrrevocSWs and SglSWs cannot start) or commit. When an IrrevocSW commits, it invalidates in-flight conflicting SpecSWs.

SglSW vs. Everything When an SglSW begins, it acquires the commit lock and aborts all other concurrently executing transactions. While it holds that lock, SglSWs and IrrevocSWs are prevented from starting, and LiteHWs and BFHWs cannot commit. The SglSW also updates the *commit_sequence* lock at the transaction's start and end, aborting all concurrently executing SpecSWs and BFHWs.

6.4.1 Opacity and Sandboxing

Opacity is a correctness property that ensures that aborted transactions do not observe inconsistent states [32]. The principal challenge to achieving opacity for Invyswell occurs when a hardware transaction and a software transaction conflict. Haswell's hardware transactions are strongly isolated, but InvalSTM's software transactions are not, so care must be taken when managing their interaction.

Invyswell's initial modification to InvalSTM's design permits doomed SpecSWs, i.e. SpecSWs that are guaranteed to abort, to observe inconsistent states because committing SpecSWs perform invalidation *after* writing their changes to memory. To

prevent these transactions from observing inconsistent states, Invyswell performs validation at commit-time and before each new read as described in Section 6.3.7.

Unlike SpecSWs, Invyswell's IrrevocSWs and SglSWs cannot observe inconsistent states because these transactions are never aborted and are, therefore, never doomed. Finally, Haswell's shared memory writes executed by a hardware transaction become visible only when the transaction commits, and writes by aborted transactions never become visible. Moreover, Haswell's transactions are (mostly) sandboxed, meaning that faulty behavior caused by inconsistent reads will cause the transaction to abort. Unfortunately, however, there is one leak in the Haswell sandbox, described in detail in the next section.

6.4.2 Hardware Sandboxing Limitations

For the most part, hardware sandboxing ensures that no consistency violation within a hardware transaction can affect other transactions. There is, however, one vexing "loophole", an unlikely sequence of events in which (1) mutually inconsistent reads cause a spurious memory write, (2) which overwrite an address later used as the target of an indirect jump in that same transaction, (3) thereby causing a jump to a location that happens to contain either an *_xend* (commit transaction) instruction, or immediate data that looks like one. Executing this instruction without the final commit lock check could prematurely commit an inconsistent set of changes.

This hazard, however unlikely, presents a challenge for any HyTM system implemented in an unmanaged language. Broadly speaking, without escape actions, hardware transactions cannot guarantee transactional consistency if they execute concurrently with either in-place update software transactions or with the commit phase of a deferred update software transaction.

To address this hazard, Invyswell's hardware transactions check the *commit_lock* before doing an indirect jump using function pointers. Simple optimizations can reduce the cost of such a policy. For example, there is no need to check the lock if the transaction has an empty write set, because it could not have corrupted the jump address. If a transaction makes multiple indirect jumps, it suffices to check the lock before the first jump, because once read, the *commit_lock* remains in the transaction's read set, and the transaction will be aborted if the lock is changed externally.

In the results presented in Section 6.6, we performed these optimizations by hand. For some benchmarks, we found that early checking slightly improved performance, probably because transactions with indirect jumps are often longer, hence less likely to succeed in hardware, and more likely to benefit from a quicker fallback to software.

In the long term, there is a trend toward compiler support to help with this issue. The danger posed by indirect jumps in transactions is similar to the danger posed by common security threats such as buffer overflow in general-purpose programs. The security literature has many examples of compiler techniques to protect jump addresses, such as moving vtables and return addresses in a separate memory space [7] marked as read-only. The latest GCC supports security functionality to check vtable integrity.

Static validity checking for function pointers is difficult, in general, but feasible for common special cases, such as initializers. GCC uses devirtualization and inlining for the most likely target for indirect pointers for optimization levels -O2 or higher. When inlining is possible, GCC can make indirect jumps direct. A transactional compiler could be more aggressive about eliminating or protecting indirect jumps.

6.5 Optimizations

In this section, we describe the modifications that we made to Invyswell's original design to improve its performance. We found these optimizations to be effective for the first-generation Intel Haswell RTM processor, however, some optimizations are designed specifically for performance of low thread counts (as indicated by the * below) and may degrade performance as thread counts increase. As a result, when Intel's RTM scales to higher thread counts, these "low thread count" changes should be eliminated.

Hardware Transactions Hardware transactions are retried with exponential back-off. Before starting a hardware transaction, the *commit_lock* and the software transaction counter, *sw_cnt*, are read non-transactionally to increase the likelihood of finding these data cached, and to optimize for the case when only hardware transactions are active.

Validation Consider two SpecSWs, T_A and T_B. Assume that T_A has entered its commit phase and T_B is about to validate a read. Furthermore, assume that T_B has higher priority than T_A and that they conflict with one another. When T_B performs its validation, it could notice that T_A has acquired the commit lock and abort because of the conflict it identifies. At the same time, T_A could consult the CM and abort because T_B has a higher priority, resulting in both transactions aborting because of each other. A similar situation could also occur between a committing SpecSW and a committing BFHW.

To avoid such scenarios, we introduce two global flags, *hw_check* and *sw_check*, in addition to the *commit_lock*, to indicate the different phases of a SpecSW's commit phase. At the highest level, these flags are used to ensure that SpecSWs and BFHWs

are only aborted by a SpecSW that is guaranteed to commit. These flags change the SpecSW and BFHW commit process in the following way.

At commit, a SpecSW, called T_C, acquires the *commit_lock* and then consults the CM to receive permission to commit. If permitted to commit, T_C sets the *hw_check* = *true* to signal to BFHWs that it is committing its writes to memory. With this approach, BFHWs only read the *hw_check* flag at commit-time, instead of the *commit_lock*, which ensures that a BFHW can only be aborted by a SpecSW that will eventually commit, rather than reading the *commit_lock*, where a BFHW could be aborted by a SpecSW that has only started its commit phase but may eventually be aborted by the CM.

Next, T_C waits for the *hw_post_commit* counter to reach zero and, once it has, it checks if it was invalidated by a concurrently committing BFHW. If still valid, T_C sets the *sw_check* = *true*, which informs other SpecSWs about to read new memory to perform conflict detection against T_C's Bloom filters. At this point, T_C and many concurrently reading SpecSWs may perform simultaneous conflict detection on each other. If conflicts are found, the reading SpecSWs are aborted. If no conflicts are found between reading SpecSWs and T_C, the reading SpecSWs subsequently check their *valid* flag to ensure they were not invalidated by T_C, which may have performed conflict detection before the reading SpecSWs had, and subsequently cleared its Bloom filters before the reading SpecSWs could identify conflicts with them. Any reading SpecSWs that are still valid are permitted to continue their execution. Without the *sw_check* flag, the scenario of conflicting transactions T_A and T_B might occur. With it, a reading SpecSW's validation can only fail if it conflicts with a concurrently executing SpecSW that is guaranteed to commit.

***Bloom Filters** In principle BFHWs and IrrevocSWs enable more concurrency than LiteHWs or SglSWs, yet, in practice the overhead associated with BFHWs' and

IrrevocSWs' Bloom filters can negate their concurrency benefits. This is especially true at low thread counts where there is not enough concurrency to justify such overhead. Because of this, we use SglSWs, rather than IrrevocSWs, as the fallback from SpecSWs for our experiments (see Figure 6.3), as SglSWs do not employ Bloom filters. However, once RTM becomes available with higher core counts, we plan to reinstate IrrevocSWs as the fallback for SpecSWs because they enable SpecSWs to execute alongside them, while SglSWs do not.

To reduce the overhead of BFHWs, we optimize away their read set Bloom filters. This optimization is possible because BFHWs only invalidate SpecSWs – SpecSWs never invalidate BFHWs – thereby only requiring write-write and write-read conflict detection for BFHWs invalidation phase. [3] However, this change prohibits BFHWs and SpecSWs from committing concurrently, which the original Invyswell design permitted. For low thread counts, however, we have found this change to only positively impact performance. Yet, for higher thread counts, this change will likely degrade performance and, therefore, it would be advisable to revert back to Invyswell's original Bloom filter design.

***Fail-Fast** When there is contention, many SpecSWs will repeatedly abort before reaching their retry threshold and falling back to SglSWs. The amount of wasted work that this process can incur could be substantial if contention is consistent, or even bursty, throughout the entire benchmark.

To address this, we add a counter to count the number of high-priority software transactions aborted during the invalidation phase. Whenever a thread notices that this number is over a threshold, it increments a racy shared counter. Once this counter reaches a pre-defined threshold, our optimized system switches to Fail-Fast

[3] BFHWs can be aborted by other hardware transactions, but that is handled automatically by the hardware.

mode, which only uses LiteHWs and SglSWs. We have found this optimization to be efficient because it identifies the cases when STMs are wasting work with too many retries, which eventually fail to irrevocable mode. In these cases, we have found it is better to use irrevocable software transactions immediately.

Read-Only We employed optimizations for both read-only SpecSWs and BFHWs. Read-only SpecSWs can commit when they reach commit phase without acquiring the commit lock, even if they were invalidated. First, the validation process in the read annotations ensures that the transaction's read set was consistent at the time of the last read. Second, read-only SpecSWs, as well as read-only BFHWs do not need to perform invalidation, as they can be serialized before conflicting in-flight software transactions.

6.6 Evaluation

Our experimental results were gathered on an Intel Haswell four-core processor (Core i7-4770) with RTM and HLE support, running at 3.40GHz. Each core has a 32KB L1 cache, and a total of 8GB RAM shared across all cores. We enabled hyper-threading to collect data for up to eight threads. Because of L1 cache sharing due to hyperthreading, we noticed that at eight threads some hardware transactions that previously executed without failure began to abort due to overflow, thereby degrading performance. We used the GCC 4.8 compiler with -O3 optimizations for all benchmarks.

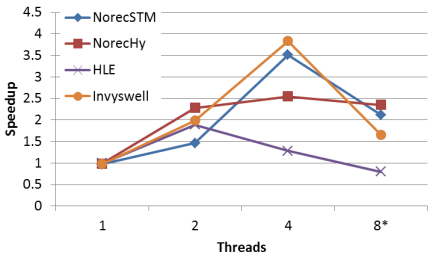
We used the STAMP benchmark suite [15] to measure the speedup that Invyswell provides relative to sequential execution. We compare this speedup against NOrec, which we call NorecSTM, Hybrid NOrec, which we abbreviate as NorecHy, and Haswell's HLE. For each of these systems, we executed each STAMP benchmark five

times and present the median result as shown in Figure 6.8. Variance was generally low, except for Bayes.

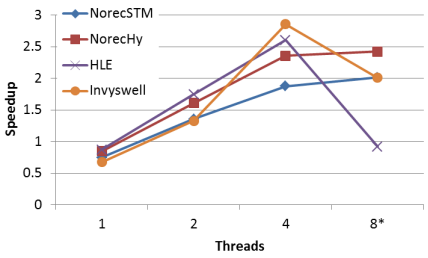
Invyswell Details We instrumented the STAMP code using its macros to use a thread-local transaction type indicator for choosing which code path to execute. This instrumentation incurs a run-time performance penalty. A compiler could generate different code paths for these transaction types, but it would not need to generate a code path for each type. In particular, LiteHW and SglSW have similar read/write annotations, as do BFHW and IrrevocSW. Moreover, the overhead incurred for manual instrumentation is higher than the overhead incurred by compiler instrumentation.

Hardware transactions are retried N times, where $N = 10$ for our experiments, unless the abort status indicates that the transaction is unlikely to succeed in hardware, in which case the transaction is immediately retried in software. SpecSWs are retried M times, where $M = 4$, and used SglSW as a fallback if the number of retries is exceeded. Invyswell was configured to use 1024 bits and the spooky-hash function [52] for its Bloom filters. Outside of normal Bloom filter trade-offs of precision versus size, there is an additional trade-off with Bloom filters for Invyswell's hardware transactions between their precision and the aborts they cause by overflow. [4] We found 1024 bits to be a good balance across all benchmarks. For example, the Yada benchmark emits many Bloom filter false positives and makes this tradeoff apparent. Increasing the Bloom filters' size improves SpecSW performance but degrades BFHW, as it causes more aborts.

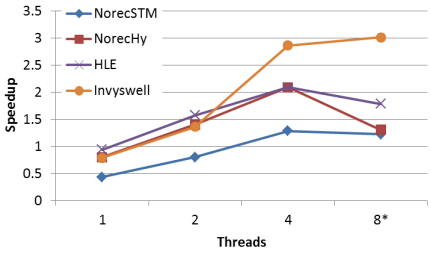
[4]The larger the Bloom filter, the better its precision, but the more likely a hardware transaction using such a Bloom filter will abort due to cache overflow, because the Bloom filter must be part of the hardware transaction's speculative state stored, in this case, in Haswell's L1D cache.



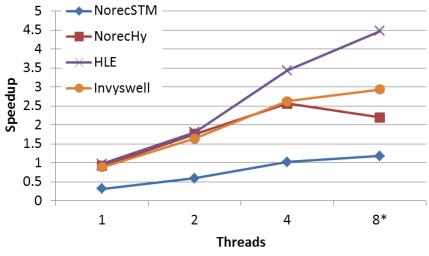
(a) Bayes.



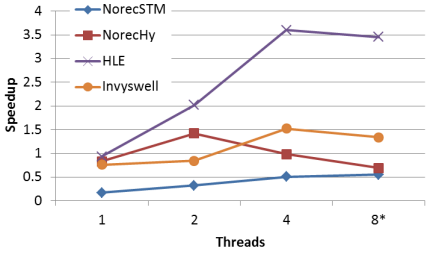
(b) Genome.



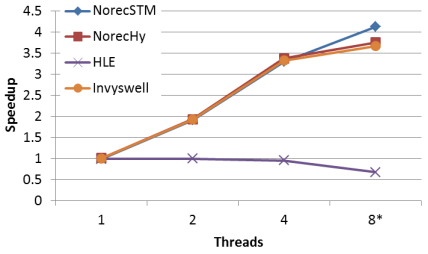
(c) Intruder.



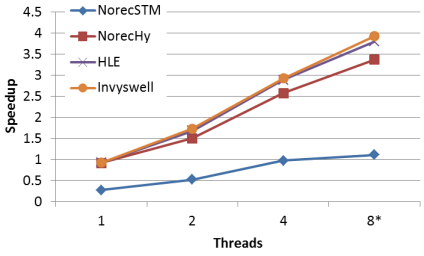
(d) Kmeans Low.



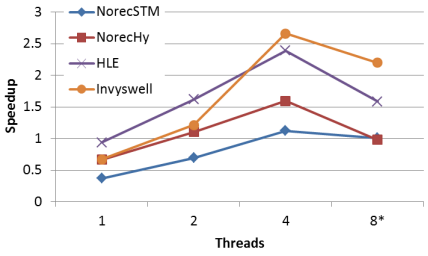
(e) Kmeans High.



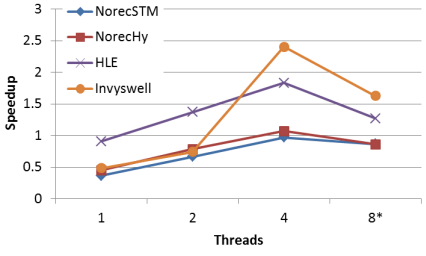
(f) Labyrinth.



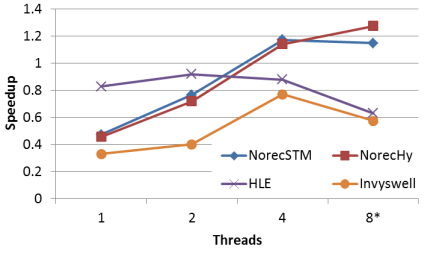
(g) Ssca2.



(h) Vacation Low.



(i) Vacation High.



(j) Yada.

Figure 6.8: Speedup on STAMP Benchmarks (Note: 8 threads using hyperthreading).

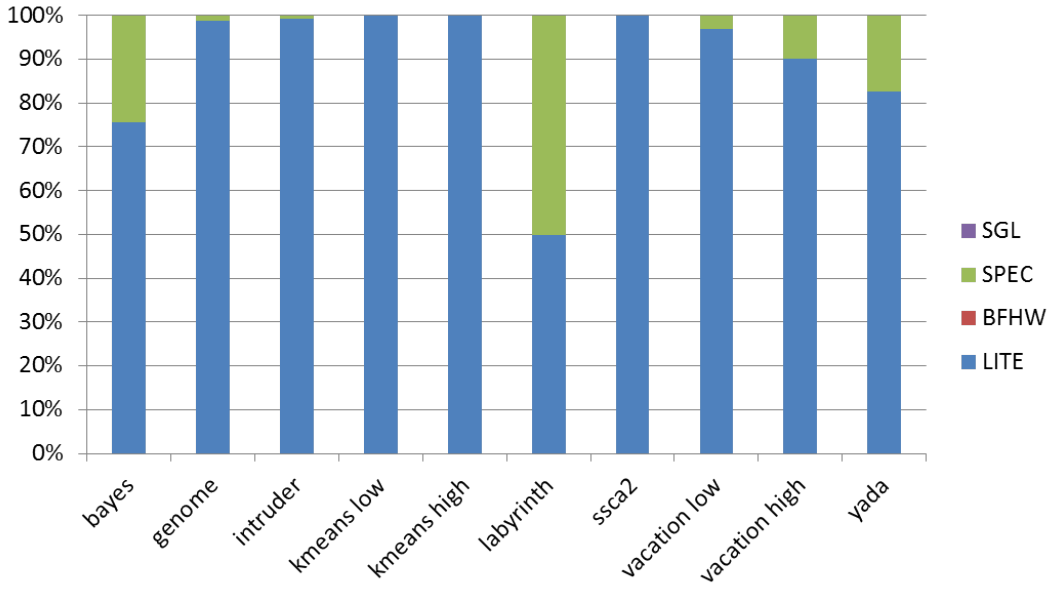


Figure 6.9: Invyswell Transaction Types: 1-threaded execution.

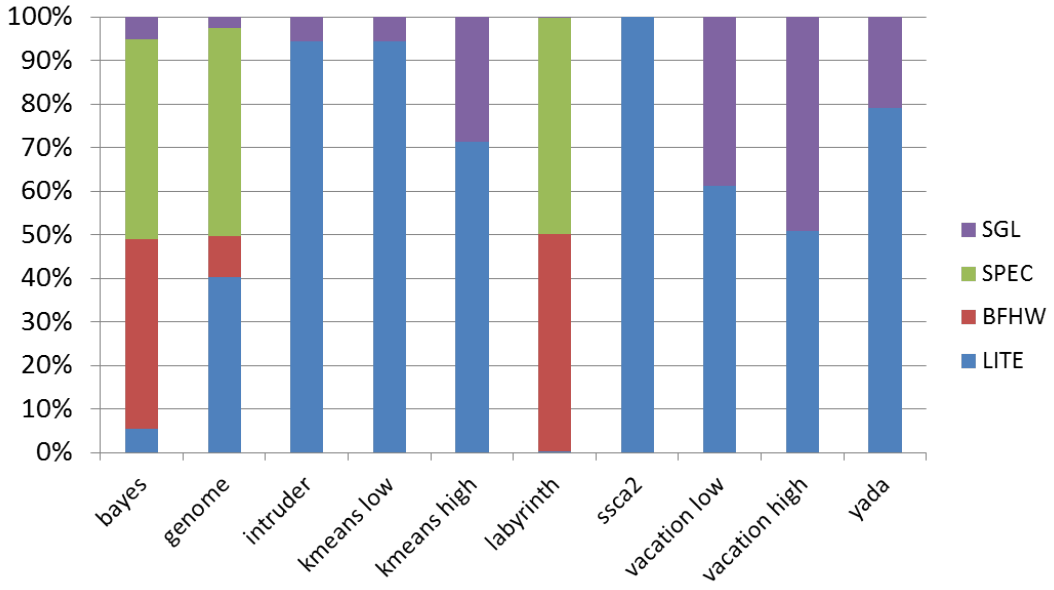


Figure 6.10: Invyswell Transaction Types: 8-threaded execution.

Hybrid NOrec and Invyswell Hybrid NOrec has many variants, many of which require nonspeculative loads. [77] requires both nonspeculative loads and nonspeculative stores. These variants cannot be implemented using TSX, and are not considered in this thesis. The version of NOrec evaluated in this chapter uses the two location variant and the sw_exists filter described in [17]. [5] Hybrid NOrec has two types of transactions, hardware and software. Both types can execute at the same time. To ensure hardware transactions do not see inconsistent memory states, they eagerly subscribe to the software transactions' commit lock as soon as they begin their execution. When a software transaction begins its commit phase, hardware transactions are automatically aborted. When a hardware transaction commits, it increments a shared counter, which notifies software transactions that they must perform value-based validation to ensure consistency. To perform validation, each software transaction maintains its own list of read memory locations. To reduce list insert computational overhead, each software transaction inserts new read element directly to the list's tail, even if the item is already in the list, resulting in $O(1)$ insert time complexity. A disadvantage of this approach is that the read list can become large if a software transaction reads many locations, thereby increasing the time it takes to perform validation, where the entire list must be walked. Each software transaction performs validation in $O(N)$ time, where N is the size of the read set, for every new read added to the transaction's read set after a software or hardware transaction has committed.

In contrast to Hybrid NOrec, Invyswell has two hardware transaction types, three software transaction types, and performs conflict detection using Bloom filters, not lists, which house the memory accessed by both hardware transactions (BFHW)

[5] Our implementation of Hybrid NOrec included all the optimizations used in [17]. In addition, we tried a variant of this algorithm that had hardware transactions lazily subscribe to the software commit lock, which also used the indirect jump annotations that we used for Invyswell. This version performed similarly to Hybrid NOrec's normal eager subscription, so we omitted the results for clarity.

and software transactions (SpecSW and IrrevocSW). With Bloom filters, Invyswell's conflict detection is performed in $O(1)$ time, yet, because Invyswell uses invalidation, it has additional overhead that Hybrid NOrec does not have, where invalidation is performed after committing a transaction's speculative writes to memory.

Invyswell's LiteHWs are similar to Hybrid NOrec's hardware transactions, but Invyswell's BFHWs have no Hybrid NOrec counterpart. Although BFHWs incur overhead not found in Hybrid NOrec's hardware transactions – the storing of read and write set data in Bloom filters – this overhead is amortized on large transactions because of the finer grained conflict detection that it enables. The improved precision of conflict detection enables wider transactional throughput between hardware and software transactions if they don't conflict (e.g., Figure 6.8f's benchmark).

If Invyswell did not include BFHWs, nearly all of Labyrinth's transactions would execute as software transactions, because Invyswell's LiteHWs often get aborted by the long-running software transactions. However, with BFHW, hardware and speculative software transactions (SpecSWs) can execute concurrently and both types of transactions can commit, as there are not many conflicts. NOrec hardware transactions do not exhibit Bloom filter overhead but, instead, incur overhead on its software transactions, which must do value based validation, re-validating the entire read set after each transactional commit. As 50% of the transactions in Labyrinth cannot succeed in hardware, the performance of both HyTMs is similar to that of NOrec STM.

Another important difference between Invyswell and Hybrid NOrec is how fast software transactions execute for different transaction sizes. Invyswell's SpecSW transactions, which are similar to InvalSTM's transactions, are fast for large transactions, while NOrec's software transactions are fast for small transactions without many reads to re-validate. Yet, because Haswell's RTM can successfully execute most

smaller size transactions (those without unsupported instructions), we believe SpecSWs are the natural choice as a fallback mechanism for hardware transactions.

Nevertheless, there is an interesting effect that occurs in the presence of hyperthreading, where hardware transactions overflow at smaller sizes than they would without hyperthreading because of cache sharing between two hyperthreads on the same core. For example, in Genome (Figure 6.8b), at eight threads about 50% of hardware transactions spill to software, for both HyTMs, because of overflow. Because of this, Hybrid NOrec performs better than Invyswell for Genome at eight threads. However, we believe this is an artifact of hyperthreading, as Invyswell is notably faster than Hybrid NOrec for Genome at four threads, where significantly fewer hardware transactions spill to software. With this in mind, we expect Invywell to perform better as HTMs scale in core count, as only large transactions will overflow the cache, resulting in the use of Invysell's SpecSWs only in the cases in which they were intended.

NOrec and HyTMs STMs typically scale at higher thread counts, but often perform poorly at low thread counts, especially for small and mid-sized transactions. NOrec, referred to as NorecSTM in our figures, like any STM, incurs instrumentation overhead that limits performance for small (Ssca2, Kmeans) and mid-sized (Intruder, Vacation, Genome) transactions. For such benchmarks, Invyswell can outperform NOrec by a factor of 3.5× (6.8g). Hybrid NOrec also outperforms NOrec on these benchmarks, indicating that a hybrid is necessary over an STM. However, Invyswell can be twice as fast as Hybrid NOrec (6.8c) because of its more lightweight SglSWs, in which Hybrid NOrec has no software equivalent.

As expected, NOrec performs best for benchmarks with longer transactions, and bigger read and write sets, such as Bayes, Labyrinth and Yada (Figures 6.8a, 6.8f, and 6.8j, respectively). Hybrid NOrec closely approaches the NOrec's speedup, as most

of the benefit in these cases comes from the software transactions. In Figure 6.8a, NOrec is 2.1× faster than sequential execution, while Invyswell is 1.6× faster. For completeness, we included results for Bayes, but its high variance suggests that these results should be interpreted with caution [78].

Labyrinth (Figure 6.8f) has long transactions, where the first portion of the transaction manipulates non-shared memory. For this benchmark, 50% of the transactions cannot complete in hardware, so HLE's performance degrades to that of a lock. In contrast, NOrec yields high throughput because it enables concurrency between its transactions. Because Haswell does not support non-transactional loads and stores, all local operations performed inside a transaction are also transactional, putting pressure on the cache. Therefore, both Hybrid NOrec and Invyswell are negatively affected, resulting in performance similar to NOrec.

Hardware Lock Elision (HLE) HLE is implemented entirely in hardware and has no instrumentation overhead, but uses a non-scalable single global lock fallback when transactions fail. For large benchmarks, such as Bayes or Labyrinth, even at small thread counts, Invyswell outperforms HLE by a notable margin. This is because many transactions overflow the cache and fall back to software, being serialized by the lock used in HLE. For medium sized benchmarks, Invyswell also outperforms HLE. However, for small transactions, HLE benefits most from the lack of overhead, so it is faster than Invyswell on benchmarks such as Kmeans Low and Kmeans High. Ssca2 is also a benchmark with small transactions, but Invyswell and HLE perform similarly.

Figure 6.11 shows the percentage of committed hardware transactions for one thread and four threads for both Invyswell and HLE. The one-threaded execution indicates, in general, the percentage of transactions that fail in hardware because of unsupported instructions or overflow. This provides a baseline of the maximum number

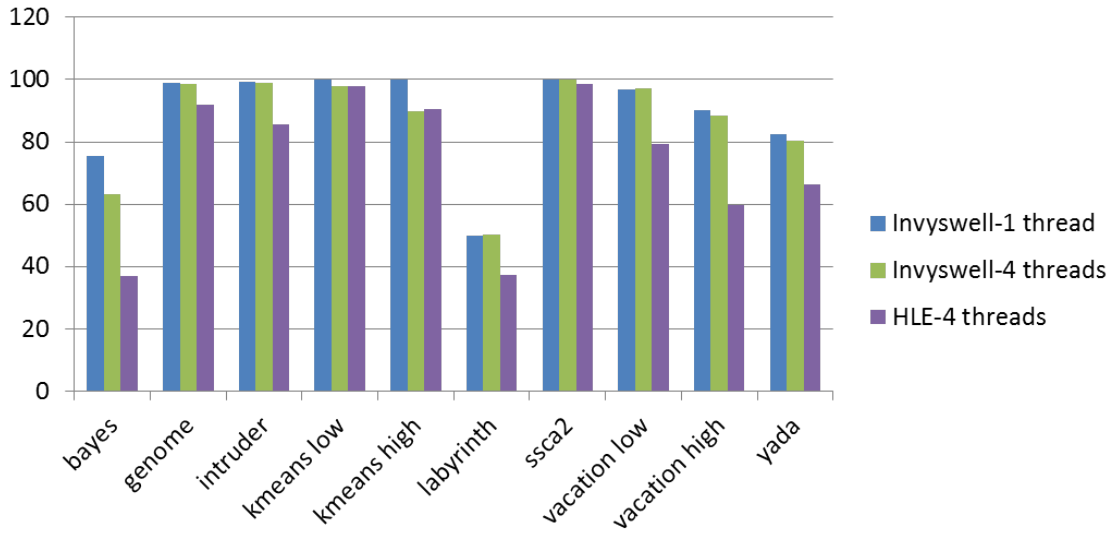


Figure 6.11: Percentage of Committed Hardware Transactions.

of hardware transaction commits that are possible for each benchmark. We also found that the number of HLE hardware transactions that begin is higher than the total number of committed transactions. This suggests that HLE also retries failed transactions before falling back to its global lock.

Invyswell's percentage of committed hardware transactions at four threads is similar to its percentage at one thread, and it is higher than HLE's percentage at four threads. This makes the argument that Invyswell generally makes more efficient use of hardware resources than the hardware (i.e., HLE) itself. Figures 6.9 and 6.10 show the breakdown of Invyswell's transaction types for one thread and eight threads executions. The eight-threaded execution suffers from the effects of hyperthreading, so the number of hardware transactions successfully committed is lower than for the one thread execution.

Overall, Invyswell outperforms HLE. For Yada, however, HLE is faster than Invyswell despite using fewer hardware transactions. This benchmark has large transactions and high contention, causing a lot of conflicts between transactions. In this case, Invyswell suffers from many false positives in its Bloom filter set intersection. We

noticed an increase in performance for SpecSWs as we increase the size of the Bloom filters. However, as we previously explained, larger Bloom filters negatively impact BFHWs. Therefore, the size of the Bloom filters represents a tradeoff to balance the performance of SpecSWs and BFHWs.

Discussion In general, Invyswell outperforms prior methods across all STAMP benchmarks. Not only does Invyswell outperform HLE for all but the smallest transactions, it is inherently more flexible, because the programmer has explicit control over CM and failover policies. Although Invyswell is adapted from the earlier InvalSTM design, the existence of hardware transactions that bypass the CM means that the two systems are divergent, in terms of design and behavior.

Hardware transactions can fail for a variety of reasons, including resource exhaustion, timing anomalies, or illegal instructions. For future work, there is a need for better adaptive CM to identify when a particular approach is not working well, and when to switch to a more effective alternative.

6.7 Summary

In this chapter, we described Invyswell, a HyTM that combines Haswell's RTM transactions with software transactions from a heavily modified version of InvalSTM. We evaluated Invyswell on a 3.4 GHz 4-core Haswell processor capable of supporting up to eight hardware threads and compared it to to Haswell's native hardware lock elision (HLE), a state-of-the-art STM (NOrec), and a state-of-the-art HyTM (Hybrid NOrec).

Our main goals with Invyswell were to *(i)* improve performance for small- to medium-sized transactions, configurations where the instrumentation costs of STMs typically

cause them to perform poorly and *(ii)* to extend InvalSTM's design to support the concurrent execution of both hardware and software transactions. We found that very small transactions are handled well by a simple combination of hardware transactions with fallback to a single global lock. The most interesting challenges were *(i)* modifying InvalSTM to provide some degree of precision in its conflict detection between concurrently executing hardware and software transactions and *(ii)* improving mid-size transaction performance, transactions that are small enough to benefit from hardware transactions, but too large to work well with a single global lock.

We evaluated a variety of transactional mechanisms, both hardware and software, on a range of STAMP benchmarks. As one might expect for such heterogeneous benchmarks, no single mechanism was best for every benchmark, but overall, Invyswell outperformed prior methods by more than 18%.

Haswell supports *hardware lock elision* (HLE), which allows an annotated critical section to be first executed speculatively as a hardware transaction, and then, if that transaction fails, to be re-executed non-speculatively using the original lock. HLE already provides some of the functionality of HyTM, so it is natural to ask whether Haswell needs HyTM at all. We find that HyTM is indeed needed: on average, Invyswell is about 25% faster than HLE across all benchmarks. Moreover, for benchmarks with large transactions, such as Bayes and Labyrinth, HLE does not scale and it is 2×-5.4× slower than Invyswell. The principal reason HLE does not eliminate the need for HyTM is that HyTM allows for better contention management. HLE follows a hard-wired policy of falling back to a lock after failure, but HyTM can make more intelligent and flexible decisions about resolving conflicts, taking advantage of software-based transactions, and making more effective transitions between speculative and various non-speculative synchronization mechanisms.

We tested alternative software mechanisms that trade overhead for precision. Conflict detection can be coarse and fast (SglSW) or more precise and slower (IrrevocSW and SpecSW). In the thread-count range supported by our platform, coarse-and-fast usually slightly outperforms precise-and-slower. We conjecture that precise conflict detection will become more attractive in future hardware platforms with more cores, where Invyswell is likely to perform well.

Any HyTM faces the challenge of providing opacity, which ensures that all transactions only observe consistent states. This is more difficult than it may seem, because the composition of two opaque mechanisms (for example, Haswell's RTM and InvalSTM) is not necessarily opaque. RTM's lack of escape actions complicated our task. Escape actions could make it substantially easier to ensure opacity, and to provide more effective conflict management. For example, a hardware transaction could invalidate software transactions during its commit phase, rather than after it, allowing, in some cases, for it to abort itself to improve overall throughput, as was the case in InvalSTM's original design.

Our experience suggests that hybrid mechanisms can improve the performance of small to mid-size transactions that can execute in hardware, compared to software-only or hardware lock-elision mechanisms. We conjecture that this difference will become even more pronounced when Haswell platforms with more cores become available.

Chapter 7

Conclusion

Computer architecture design has reached a "power wall", marking the end of CPU frequency scaling. To further improve performance in this context, new hardware platforms are now increasingly focusing on leveraging more parallelism. These new architectures are continually increasing the number of cores, becoming more heterogeneous and offering new instructions in support of parallelism. However, they are also becoming harder to program. Parallel and concurrent programming have become necessities in this highly parallel environment, but our current abstractions are not up to the challenge. Locking is still the most widely used synchronization paradigm, but it either fails to deliver acceptable performance and scalability beyond a small number of cores or it comes at a very high cost in terms of development effort and expertise.

In this thesis, we proposed new techniques to simplify writing efficient parallel code that leverage the architectural features of these emerging systems. We focused on two commercially available platforms: NUMA architectures with hundreds of cores and Haswell processors with support for hardware transactional memory.

We described various abstractions that have been proposed in the concurrent comput-

ing community, such as delegation, elimination, combining and transactional memory and we showed how to use and integrate these abstractions to design scalable concurrent algorithms. We designed, implemented and evaluated a NUMA-aware concurrent stack and a scalable concurrent priority queue using these abstractions. Our designs achieve significant performance benefits compared to prior work.

Moreover, we proposed improved algorithms for transactional memory. We presented new fallback algorithms for best-effort hardware transactional memory that outperform state-of-the-art software, hardware and hybrid solutions. First, we described Lazy Single Global Lock fallback (L-SGL), which uses an optimized single global lock as the software fallback. Second, we described Invyswell, a new Hybrid Transactional Memory solution based on a modified version of InvalSTM. Our experience suggests that hybrid mechanisms can improve the performance of small to mid-size transactions, in situations where the number of threads fits in hardware, compared to software-only or hardware lock-elision mechanisms. We conjecture that this improvement will become even more pronounced when Haswell platforms with more cores become available, although the trade-offs among the various hybrid mechanisms are likely to change as platforms scale.

As hardware changes and improves to provide more parallelism potential, we need better software mechanisms to leverage these new features. The methods we discussed are a step in the direction of scalable concurrent software design, but more abstractions are needed to design highly scalable programs and to eliminate the necessity of specializing code for particular architectures. Moreover, software needs to anticipate and inform hardware developments, because only a tight collaboration between hardware and software can achieve the performance and scalability desired by developers.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Transactional programming in a multi-core environment. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *PPoPP*, page 272. ACM, 2007.

[2] Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Rassul Ayani. Lr-algorithm: concurrent operations on priority queues. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing, SPDP 1990, Dallas, Texas, USA, December 9-13, 1990.*, pages 22–25, 1990.

[4] R. Bayer and M. Schkolnick. Readings in database systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[5] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[6] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[7] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 19–19, Berkeley, CA, USA, 2003. USENIX Association.

[8] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[9] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra J. Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *OPODIS*, pages 83–97, 2013.

[10] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM.

[11] Irina Calciu, Justin Gottschlich, and Maurice Herlihy. Using delegation and elimination to implement a scalable numa-friendly stack. In *5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[12] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, pages 187–200, 2014.

[13] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 406–420, 2014.

[14] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *9th ACM Sigplan Workshop on Transactional Computing, TRANSACT '14, Salt Lake City, UT, USA, March 2, 2014*.

[15] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[16] C. Click. Azul's experiences with hardware transactional memory. HP Labs' Bay Area Workshop on Transactional Memory, August 2007.

[17] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, pages 39–52, New York, NY, USA, 2011. ACM.

[18] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[19] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[20] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

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

[22] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM.

[23] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[24] David Dice, Mark Moir, and Nir Shavit. Sun Microsystems: Transactional memory.

[25] David Dice, Ori Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

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

[27] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[28] Justin E. Gottschlich and Daniel A. Connors. Extending contention managers for user-defined priority-based transactions. In *Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, Apr 2008.

[29] Justin E. Gottschlich, Maurice P. Herlihy, Gilles A. Pokam, and Jeremy G. Siek. Visualizing transactional memory. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 159–170, New York, NY, USA, 2012. ACM.

[30] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, April 2010.

[31] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *DISC*. LNCS, 2005.

[32] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

[33] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Morgan and Claypool, 2010.

[34] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pages 8–21, Washington, DC, USA, 1978. IEEE Computer Society.

[35] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). *j-SIGPLAN*, 39(11):1–13, November 2004.

[36] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, Second Edition*. Morgan and Claypool, 2010.

[37] D. Hendler and N. Shavit. Work dealing. In *Proc. of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, pages 164–172, 2002.

[38] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[39] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, January 2010.

[40] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[41] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[42] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[43] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture*. May 1993.

[44] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.

[45] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[46] Q. Huang. An evaluation of concurrent priority queue algorithms. Technical report, Cambridge, MA, USA, 1991.

[47] Galen Hunt, M. Michael, S. Parthasarathy, and M. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151 – 157, 1996.

[48] Intel Corporation. Hardware lock elision in Haswell. Retrieved from `http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/GUID-A462FBC8-37F2-490F-A68B-2FFA8010DEBC.htm`.

[49] Intel Corporation. Transactional Synchronization in Haswell. Retrieved from `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 8 September 2012.

[50] Amos Israeli and Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, WDAG '93, pages 1–17, London, UK, UK, 1993. Springer-Verlag.

[51] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[52] Jenkins, B. SpookyHash: a 128-bit non-cryptographic hash (2010). Retrieved from `http://burtleburtle.net/bob/hash/spooky.html`, 25 June 2014.

[53] J. L. W. Kessels. On-the-fly optimization of data structures. *Commun. ACM*, 26(11):895–901, November 1983.

[54] Gokcen Kestor, Roberto Gioiosa, Tim Harris, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. Stm2: A parallel stm for high performance simultaneous multithreading systems. In Lawrence Rauchwerger and Vivek Sarkar, editors, *PACT*, pages 221–231. IEEE Computer Society, 2011.

[55] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[56] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[57] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, September 1980.

[58] Yosef Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[59] I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, 2000.

[60] Udi Manber. On maintaining dynamic information in a concurrent environment. *SIAM J. Comput.*, 15(4):1130–1142, November 1986.

[61] Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236, New York, NY, USA, 2008. ACM.

[62] Alex Matveev and Nir Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, July 2013.

[63] Alexander Matveev and Nir Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM.

[64] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. Cphash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 319–320, New York, NY, USA, 2012. ACM.

[65] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.

[66] Mark Moir. Hybrid transactional memory. Jul 2005.

[67] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.

[68] Mark Moir and Nir Shavit. Concurrent data structures. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of data structures and applications*. Chapman and Hall/CRC Press, 2007.

[69] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254–265, 2006.

[70] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 359–370, New York, NY, USA, 2006. ACM.

[71] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[72] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '87, pages 170–176, New York, NY, USA, 1987. ACM.

[73] Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '91, pages 192–198, New York, NY, USA, 1991. ACM.

[74] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently, 1999.

[75] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, pages 294–305. ACM/IEEE, 2001.

[76] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[77] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: the importance of nonspeculative operations. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

[78] Wenjia Ruan, Yujie Liu, and Michael Spear. Stamp need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*. Mar 2014.

[79] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPOPP*. ACM SIGPLAN 2006, March 2006.

[80] William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In Marcos Kawazoe Aguilera and James Aspnes, editors, *PODC*, pages 240–248. ACM, 2005.

[81] William N. Scherer, III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, May 2009.

[82] Nir Shavit. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, page 2000, 2000.

[83] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.

[84] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Principles of Distributed Computing*. Aug 1995.

[85] Michael F. Spear, Luke Dalessandro, Virendra Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP*, February 2009.

[86] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 253–264, New York, NY, USA, 2009. ACM.

[87] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *IEEE International Symposium on Parallel and Distributed Processing*, page 11 pp., april 2003.

[88] R. Kent Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, 2006.

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

[90] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.