Fast and Scalable Software Execution on Multicores

Thesis submitted for the degree of Doctor of Philosophy
by
Adam Morrison

This work was carried out under the supervision of
Prof. Yehuda Afek

Submitted to the Senate of Tel Aviv University
December 2013
Abstract

This dissertation addresses three major challenges in designing high-performance concurrent software for modern multicore processors: realizing the potential parallelism of the problem, efficiently dealing with synchronization tasks, and overcoming system-level bottlenecks.

Realizing parallelism  Obtaining a scalable solution – in which performance improves as close as possible to linearly with the number of threads – can be difficult even for an inherently parallel problem. We demonstrate this using the rendezvous problem, in which threads of two types, consumers and producers, show up and need to be matched each with a unique thread of the other type. The parallel nature of rendezvousing allows each thread to look for a match independently of the other threads. Yet existing algorithms often waste work by attempting to match threads of the same type and by allowing a thread to idly wait until another thread attempts to match with it. We describe an efficient rendezvousing system which avoids these problems, so that every execution cycle goes towards making a successful match, leading to a significant performance improvement. Based on our rendezvousing system, we also construct a highly scalable and competitive concurrent stack implementation (Chapter 4).

Efficient synchronization  Two notorious synchronization-related performance penalties in concurrent programs are due to accessing a contended memory location (thereby serializing memory operations of threads) and issuing a memory fence (thus stalling the execution of a thread). We show that detailed understanding of the hardware mechanisms causing these penalties reveals opportunities to reduce or eliminate the penalty.

First, we find that on modern multicores performing one access to a contended location is relatively fast; contention-related performance meltdowns only occur when an algorithm repeatedly performs contended accesses. Based on this observation, we construct a high performance nonblocking concurrent FIFO queue in which one (contended) fetch-and-add operation is used to spread threads among items in the queue, allowing them to enqueue and de-
queue quickly and in parallel. Our queue therefore outperforms the best existing FIFO queues designs (Chapter 5).

Second, we show that work stealing can be done without requiring a worker to issue a memory fence on task removals, on multicores with a bounded total store order (TSO) memory model – which we show captures today’s x86 and SPARC processors. Bounded TSO is a novel restriction of the TSO memory model that places a finite bound on the capacity of the processor’s store buffer, the feature responsible for reordering in TSO. Our insight is that knowing the store buffer’s capacity allows a thief to detect when it can safely steal a task by reasoning about the number of stores pending in the worker’s store buffer, without requiring the worker to issue memory fences. We use these insights to design better performing fence-free variants of the state of the art work stealing algorithms (Chapter 6).

Overcoming system-level bottlenecks   A program can often have a performance bottleneck caused by a hardware or software system issue. Such problems are difficult to diagnose because programmers usually focus on their program and not at the lower levels. This dissertation describes ways to overcome two such system-level bottlenecks.

The first bottleneck occurs when naively using hardware lock elision (HLE) in multicores, as even a few transactional aborts severely limit the amount of concurrency and speedup obtained using HLE. We provide two software-based techniques to solve this problem and restore the lost concurrency potential in lock elision executions (Chapter 7).

Second, we observe that the manner in which modern memory allocators place allocated blocks in memory can induce cache contention and lead to excessive cache misses in both single- and multi-thread executions. We propose a methodology for building a memory allocator with cache sensitive block placement to alleviate this problem (Chapter 8).
Acknowledgements

First of all, I thank my advisor, Yehuda Afek. I take with me a lot of great lessons and excellent advice from Yehuda, but most of all I am inspired by his attitude of being both optimistic and skeptical, which made me always push for more. But above all, I thank Yehuda for doing everything with my best interests at heart, and for all the wonderful opportunities he has provided me with. It was a privilege to have you as my advisor.

I was fortunate to have the opportunity of working with Dave Dice, Eli Gafni, Haim Kaplan and Robert Tarjan. I have learned a lot from each of them, and I thank them for that. I also thank Hagit Attiya, Idit Keidar, Mooly Sagiv and Nir Shavit for their help and advice along the way, and for many interesting discussions.

I would like thank my student collaborators: Yoav Cohen, Michael Hakimi, Boris Korenfeld, Amir Levy and Guy Wertheim. Working with you was a pleasure and a great experience. I would also like to pay tribute to Moran Tzafrir, with whom I only had the chance to work briefly before she tragically passed away.

I am grateful for the financial support of the Israel Science Foundation and IBM.

Last but most, I would like to thank Meyrav, my wife, for taking this journey with me. For experiencing the highs and lows of research work together. For the practice talks and the crunch times. For the calm confidence that I could solve whatever technical problem I was struggling with. For sharing the joy of success, and sweetening the bitterness of failure. For always believing in me. And just for being you.

Meyrav, I could not have done this without you. Words cannot express how much I love you.
Previously published material


I have also worked on several projects whose focus is not on the unique aspects of modern multicore hardware, and therefore have chosen not to include them in this dissertation:


# Contents

1 Introduction ................................................. 1
   1.1 Realizing parallelism .................................... 1
      1.1.1 Fast and scalable rendezvousing .................. 2
      1.1.2 Scalable LIFO stack through rendezvousing ....... 4
   1.2 Efficient synchronization .............................. 5
      1.2.1 Contended memory locations and FIFO queues ...... 5
      1.2.2 Memory fences in work stealing .................... 7
   1.3 Overcoming system bottlenecks ......................... 10
      1.3.1 Serialization in hardware lock elision .......... 11
      1.3.2 Cache under-utilization induced by memory allocator . 13
   1.4 Organization ........................................ 14

2 Computational model and definitions ...................... 16
   2.1 System model ......................................... 16
   2.2 Concurrent object implementations .................... 17
      2.2.1 Executions and events ............................ 17
      2.2.2 Implementation correctness ....................... 17
   2.3 Example sequential specifications ...................... 18
   2.4 Concurrency levels .................................. 18
   2.5 Progress ........................................... 18
   2.6 Model vs. reality .................................... 18

3 Experimental setup ......................................... 20
   3.1 Architectures ....................................... 20
   3.2 Machine configurations ................................ 21
   3.3 General methodology .................................. 21
6.1.2 Bounded TSO model ........................................... 83
6.2 Background: work stealing synchronization .................. 85
  6.2.1 Linearizability of work stealing algorithms ............... 87
6.3 Getting around the laws of order ............................... 88
6.4 Fence-freedom by reasoning about bounded reordering ...... 90
  6.4.1 Fence-free Chase-Lev algorithm version ................... 92
6.5 Fence-free work stealing without relaxed semantics ......... 93
6.6 Bounded TSO in mainstream processors ....................... 95
  6.6.1 Cause of bounded store/load reordering .................. 95
  6.6.2 Measuring store buffer capacity .......................... 96
  6.6.3 From store buffer capacity to a reordering bound ....... 98
6.7 Evaluation ...................................................... 101
  6.7.1 The THEP algorithm ...................................... 101
  6.7.2 Chase-Lev vs. idempotent work stealing queues .......... 103
6.8 Related work .................................................. 106

7 Overcoming serialization in hardware lock elision ............... 108
  7.1 Transactional synchronization extensions ..................... 108
    7.1.1 Haswell’s TSX implementation ........................... 110
  7.2 Avalanche effect in Haswell HLE ............................. 111
  7.3 Software-assisted lock removal ............................... 117
  7.4 Software-assisted conflict management ....................... 120
  7.5 Evaluation ...................................................... 123
    7.5.1 Red-black tree micro-benchmark .......................... 124
    7.5.2 STAMP ..................................................... 126
  7.6 Related work .................................................. 127

8 Cache index-aware memory allocation ........................... 135
  8.1 Background: modern memory allocator design ................ 135
  8.2 The index-oblvious allocation problem ....................... 137
    8.2.1 Impact of index-obliviousness allocation ............... 139
  8.3 Index-aware allocation ....................................... 143
    8.3.1 Cache index-aware size-classes ......................... 143
    8.3.2 Punctuated arrays ....................................... 144
List of Figures

1.1 Time to increment a contended counter on a system with four Intel Xeon E7-4870 (Westmere EX) processors, each of which has 10 2.40 GHz cores that multiplex 2 hardware threads. The right vertical axis shows the number of CAS it takes to complete an increment. ........................................... 6

1.2 Single threaded execution time of several widely used CilkPlus [11] benchmarks [16, 55, 57, 77, 84] when not issuing a memory fence on task removal, normalized to the standard CilkPlus runtime on an Intel Haswell (Core i7-4770) 3.4 GHz processor. ................................. 8

1.3 Applying hardware lock elision to a TTAS (Test&Test&Set) lock. ............ 11

4.1 The ring data structure with the array of pointers. Node $i$ points to node $i - 1$. The ring maximum size is $T$ and is resized by changing the head’s prev pointer. 27

4.2 Asymmetric rendezvous algorithm. Lines beginning with $\triangle$ handle the adaptivity process. ................................................................. 29

4.3 Algorithm TripleAdp: definitions and helper procedures. .................. 34

4.4 Pseudo code of get() and put(). .............................................. 34

4.5 TripleAdp pseudo code for seeking and waiting threads. Lines marked by a □ handle ring size adaptivity. ........................................... 36

4.6 Pseudo code of findFreeNode(). Lines marked by □ handle ring size adaptivity. 37

4.7 Rendezvousing between producer/consumer pairs: Niagara results. L2 misses are not shown; all algorithms but JDK had less than one L2 miss/operation on average. .................................................. 43

4.8 Rendezvousing between producer/consumer pairs: Westmere results. Box plot on the right zooms in on 10 : 10 results. The bottom and top of a box are the 25th and 75th percentiles of the data, with a line at the median and a marker at the average. Whiskers extend to the minimum and maximum of the data. ......... 45
4.9 Effect of initial slot selection on $N : N$ workload throughput. Performance counter plots are logarithmic scale. There were no L3 (shared) cache misses on the Westmere processor.

4.10 $N : N$ rendezvousing with decreasing arrival rate due to increasing amount of work time between operations.

4.11 Percent of operations performed by each thread in an $N : N$ test with maximum concurrency. The horizontal line marks ideal behavior.

4.12 $N : N$ rendezvousing on both processors of the Niagara machine.

4.13 Throughput of synchronous queue bursty workload. AdaptiveAR’s ring size is sampled continuously using a thread that does not participate in the rendezvousing.

4.14 Rendezvousing between producer/consumer pairs: contribution of each technique.

4.15 Single producer and $N$ consumers rendezvousing.

4.16 Rendezvousing between $N$ producers and a single consumer.

4.17 Asymmetric rendezvousing (single producer and $N$ consumers, single consumer and $N$ producers) with OS constrained to place the single producer (or consumer) alone on a core, so that threads performing the reverse operation do not interfere with its execution.

4.18 Oversubscribed (more threads than available hardware threads) symmetric workload. Total number of threads grows, starting from the number of available threads in the hardware.

4.19 Oversubscribed (more threads than available hardware threads) asymmetric workloads. Total number of threads grows, starting from the number of available threads in the hardware.

4.20 Throughput of synchronous queue bursty $N : N$ workload in an oversubscribed setting. Workload alternates every second between 512 threads (256 pairs) and 16 threads (8 pairs).

4.21 Elimination stack variants. We omit the linked list manipulation details of the Treiber lock-free stack [117].

4.22 Stack throughput. Each thread performs both push and pop operations with probability $1/2$ for each operation type.
4.23 Analysis of stack elimination on Westmere. **Left:** Percent of stack operations that complete through elimination. **Right:** A stack operation may complete through elimination only after several rounds of failing to eliminate and going back to the main stack. Thus, we show the percent of successful elimination attempts (resulting in completion of the stack operation) out of all elimination attempts, which outnumber the stack operations.

5.1 Infinite array queue.

5.2 Pseudocode of CRQ algorithm.

5.3 CRQ linearization procedure $P$. Operations linearized at the same event are ordered based on the order of $P$'s steps.

5.4 Pseudocode of the LCRQ algorithm, using a linearizable CRQ black box.

5.5 Enqueue/dequeue throughput on a single processor. The right plot shows throughput with more threads than available hardware threads; the first point, showing the throughput at maximal hardware concurrency, is included for reference.

5.6 Enqueue/dequeue throughput on four processors (threads run on all processors from the start).

5.7 Cumulative distribution of queue operation latency at maximum concurrency.

5.8 Impact of ring size on LCRQ throughput (CC-Queue and H-Queue results are shown for reference).

6.1 Design of modern work stealing task queues. Both the Cilk THE and Chase-Lev algorithms use the flag principle to detect when the worker and thief might contend for a task. They differ in the synchronization used to manage such a conflict.

6.2 Fence-free THE algorithm. The code of `put()` and `take()` remains the same, but for the removal of the memory fence in `take()`. The parameter $\delta$ is the maximum number of stores to $T$ by `take()` operations that can exist in the store buffer.

6.3 Maximum number of stores to $T$ by `take()` operations that can exist in the store buffer. There can always be at least one such store; the number of earlier `take()` stores is dictated by the number of stores performed between `take()` operations.
6.4 Fence-free Chase-Lev algorithm. The code of \texttt{put()} and \texttt{take()} remains the same, but for the removal of the memory fence in \texttt{take()}. The parameter \( \delta \) is the maximum number of stores to \( T \) by \texttt{take()} operations that can exist in the store buffer.

93

6.5 The fence-free THEP variant of Cilk’s THE algorithm. As before, \( \delta \geq 1 \) is the maximum number of stores to \( T \) by \texttt{take()} operations that can exist in the store buffer.

94

6.6 Determining store buffer’s capacity.

97

6.7 Measuring store buffer capacity: results of running the measurement code shown in Figure 6.6 on the Westmere processor which has a documented 32-entry store buffer.

97

6.8 Program for finding executions not valid under a bounded TSO model. We vary the \( \delta \) parameter of the work stealing algorithm and \( L \), the number of stores performed by the worker.

98

6.9 Testing if the processor implements TSO\([S]\). If \( S \) is the correct bound, executions on or above the line \( y = x \) should be correct.

99

6.10 CilkPlus programs run time, normalized to the default CilkPlus runtime, on Westmere without HT.

103

6.11 CilkPlus programs run time, normalized to the default CilkPlus runtime, on Haswell without HT.

103

6.12 CilkPlus programs run time, normalized to the default CilkPlus runtime, on Westmere with HT.

104

6.13 CilkPlus programs run time, normalized to the default CilkPlus runtime, on Haswell with HT.

104

6.14 Transitive closure run time, normalized to Chase-Lev algorithm, on Westmere.

104

6.15 Transitive closure run time, normalized to Chase-Lev algorithm, on Haswell.

105

6.16 Percent of stolen work in transitive closure, on Haswell.

105

6.17 Percent of stolen work in transitive closure, on Westmere.

105

7.1 Transactional behavior in practice.

111
7.2 Impact of aborts on executions under different lock implementations. For each
tree size we show the average number of times a thread attempts to execute
the critical section until successfully completing a tree operation, and the frac-
tion of operations that complete non-speculatively. CLH and ticket results are
omitted, as they are similar to the MCS lock results. . . . . . . . . . . . . . . . . 113

7.3 MCS lock . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

7.4 Normalized throughput and serialization dynamics over time. We divide the
execution into 1 millisecond time slots. Top: Throughput obtained in each time
slot, normalized to the average throughput over the entire execution. Bottom:
Fraction of operations that complete non-speculatively in each time slot. . . . . 116

7.5 The speed-up of hardware lock elision usage with different types of locks. The
base-line of each speed-up line is the standard version of that specific lock.
By mixing different access operations we vary the amount of contention: (i)
lookups only - no contention, (ii) moderate contention - a tenth of the tree
accesses are node insertions and another tenth are node deletions and (iii) ex-
tensive contention all the accesses are either node insertion or deletion. . . . . 116

7.6 Software-Assisted Lock Removal . . . . . . . . . . . . . . . . . . . . . . . . 118

7.7 How SALR enforces correct executions in different scenarios. . . . . . . . . . 119

7.8 A block diagram of a run using our software scheme. The entry point of a
speculative section is the ‘speculative run’ rectangle. All threads acquire the
original main lock using the lock-elision mechanism. If a conflict occurs (de-
scribed by ‘x’), the conflicting threads are sent to the serializing path. Once
a thread acquires the auxiliary standard lock in a non-speculative manner, it
rejoins the speculative run. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

7.9 Software-Assisted Conflict Management . . . . . . . . . . . . . . . . . . . 122

7.10 The execution results on a small tree size (128 nodes) under moderate con-
tention. The two graphs are normalized to the throughput of a single thread
with no locking. The software assisted schemes scale well and the performance
gap between MCS and TTAS is closed. . . . . . . . . . . . . . . . . . . . . . . 125

7.11 The speed-up of software lock elision schemes compared to Haswell HLE. The
base-line of each speed-up line is the HLE version of the TTAS lock. . . . . . 129
7.12 The speed-up of our generic software lock elision schemes compared to Haswell HLE. The base-line of each speed-up line is the HLE version of the MCS lock. ................................................................. 130

7.13 Impact of software assisted conflict management for MCS lock executions. For each tree size we show the average number of times a thread attempts to execute the critical section until successfully completing a tree operation, and the fraction of operations that complete non-speculatively. ............... 131

7.14 Impact of software assisted conflict management for TTAS lock executions. For each tree size we show the average number of times a thread attempts to execute the critical section until successfully completing a tree operation, and the fraction of operations that complete non-speculatively. ............... 132

7.15 Normalized run time of STAMP applications using a TTAS lock, HLE, and software-assisted methods. ................................................................. 133

7.16 Normalized run time of STAMP applications using an MCS lock, HLE, and software-assisted methods. ................................................................. 134

8.1 Memory allocator index distribution (over 256 allocated blocks). ........... 138

8.2 Pointer chasing performance using existing memory allocators: traversal rate over a list of 256 blocks ................................................................. 140

8.3 Impact of superblock index-obliviousness on memory access scalability in multi-threaded execution. ................................................................. 141

8.4 CIF mmcache benchmark results. ................................................................. 147

8.5 Impact of superblock coloring on memory access scalability .................. 148

8.6 Malloc scalability varying thread count and allocators. Y axis is log scale. .. 149

8.7 Run-time of cache index-sensitive applications, normalized to libc ........... 152
List of Tables

1.1 Synchronization primitives supported as machine instructions on dominant multicore architectures. ................................. 6

3.1 Multicore architectures used for evaluation. ............................... 21

5.1 Single processor (queue initially empty) average per-operation statistics. Latency numbers are relative to LCRQ. There are no L3 misses because the workload fits into the shared L3 cache. Atomic operations count does not include setting of hazard pointers. ........................................... 75

5.2 Four processor (80 threads) average per-operation statistics. Atomic operations count does not include setting of hazard pointers. ............................... 77

6.1 CilkPlus benchmark applications. ............................................. 102
Chapter 1

Introduction

The driving force behind the parallelism renaissance currently experienced by the computer industry is the quest for performance. Unable to continue making software run faster simply by speeding up the processor’s execution core, vendors are turning to multicore architectures [98] which have multiple execution cores instead of one increasingly powerful core, with the goal of improving performance through parallelism [58].

However, designing high-performance software for multicore architectures poses several major challenges. This dissertation addresses three such challenges: realizing the parallelism potential of the problem, efficiently dealing with synchronization-heavy tasks, and overcoming system-level bottlenecks. The remainder of the introduction is dedicated to describing these challenges and our contributions.

1.1 Realizing parallelism

The “gold standard” for multicore performance is to obtain a scalable program, one in which performance improves as close as possible to linearly with the number of threads. Intuitively, this is possible for an inherently parallel problem, which can be broken into pieces that are then processed in parallel. Yet even for an inherently parallel problem, realizing the abstract parallelism potential in an actual program can be difficult.

In this dissertation we demonstrate this difficulty with the example of the rendezvous problem. We show that despite the parallel nature of the problem, prior rendezvous algorithms use inefficient mechanisms to realize the problem’s parallelism, resulting in sub-optimal scalability. We propose rendezvous algorithms with significantly improved performance and almost linear scalability, as described in the following.
1.1.1 Fast and scalable rendezvousing

The asymmetric rendezvous mechanism is a common abstraction in concurrent programming. In this mechanism there are two types of threads, e.g., producers and consumers, that show up. The goal is to match pairs of threads, one of each type, and send them away. Usually the purpose of the pairing is for a producer to hand over a data item (such as a task to perform) to a consumer. Asymmetric rendezvous is exactly the task performed by unfair synchronous queues (or synchronous channels) [107], in which producers and consumers handshake to exchange data. Synchronous queues are a key building block in Java’s thread pool implementation and other message-passing and hand-off designs [26, 107]. The asymmetric rendezvous abstraction also encompasses the elimination technique [112], which is used to scale concurrent stacks and queues [65, 95].

Today, the best performing technique for solving asymmetric rendezvous is elimination [22, 65, 112]. Elimination is performed by having an arriving thread pick a random slot in a collision array, hoping to meet the right kind of partner; however, if the slot picked is empty the thread waits for a partner to arrive. This standard elimination approach is vulnerable to false matches, when two threads of the same type meet, and to timeouts, when a producer and a consumer each pick a different slot and futilely wait for a partner to arrive. This sort of wasted work prevents elimination algorithms from achieving ideal scaling. In addition, elimination arrays provide no progress guarantee. Perhaps as a result of this, existing synchronous queue algorithms are not based on elimination arrays.

In this dissertation we present two highly scalable asymmetric rendezvous algorithms that improve the state of the art in both unfair synchronous queue and elimination algorithms. We start with the elimination technique, and improve its mechanisms to get rid of any wasteful work.

Our contribution Our first algorithm, named AdaptiveAR (Adaptive Asymmetric Rendezvous), is based on a simple idea that turns out to be remarkably effective in practice: the algorithm itself is asymmetric. While a consumer captures a node in a shared ring structure and waits there for a producer, a producer actively seeks out waiting consumers on the ring. As a result, AdaptiveAR does not suffer from false matches and timeouts and we show that it is nonblocking in the following sense: if both producers and consumers keep taking steps, some rendezvous operation is guaranteed to complete. (This progress property, which we refer to as pairwise nonblocking, is similar to lock-freedom [66] while taking into account the fact
1.1. REALIZING PARALLELISM

that “it takes two to tango”, i.e., both types of threads must take steps to guarantee a successful rendezvous.)

We present two new techniques, that combined with this asymmetric methodology on the ring lead to extremely high performance and scalability in practice. The first is a new ring adaptivity scheme that dynamically adjusts the ring’s size, leaving enough room to fit in all the consumers while minimizing as much as possible empty nodes that producers will needlessly need to search. Having an adaptive ring size, we can expect the nodes to be mostly occupied, which leads us to the next idea: if a producer starts to scan the ring and finds the first node to be empty, there is a good chance that a consumer will arrive there shortly. However, simply waiting at this node, hoping that a consumer arrives, would make the algorithm prone to timeouts and impede progress. Rather, we employ a peeking technique that lets the producer have the best of both worlds: as the producer traverses the ring, it continues to peek at its initial node; should a consumer arrive there, the producer immediately tries to partner with it, thereby minimizing the amount of wasted work.

Our second algorithm extends AdaptiveAR with two new forms of adaptivity. Together with AdaptiveAR’s original ring adaptivity scheme, this algorithm is triple adaptive and hence named TripleAdp. We present TripleAdp separately from AdaptiveAR because the adaptivity techniques it employs add some overhead over AdaptiveAR.

The major new dimension of adaptivity in TripleAdp is that it adapts the roles performed by the threads. AdaptiveAR fixes the roles played by consumers and producers in the algorithm: consumers wait for producers which seek them on the ring. Unfortunately, in asymmetric workloads, with more producers than consumers, the producers collide with each other when competing for the few consumers in the ring. As a result, in such workloads AdaptiveAR’s throughput degrades as the number of producers increases. However, exactly this property makes AdaptiveAR shine in workloads where consumers outnumber producers: the few producers have their pick among the many consumers occupying the ring, and AdaptiveAR maintains the peak throughput achievable by the producers. The main observation behind TripleAdp is that the mirror image of AdaptiveAR, where consumers seek producers that wait on the ring, will do well in the asymmetric workloads where AdaptiveAR fails to maintain peak throughput. TripleAdp therefore adapts to the access pattern and switches between (essentially) AdaptiveAR and its mirror image, enjoying the best of both worlds.

The second dimension of adaptivity added in TripleAdp is that of overall memory consumption. For simplicity, we designed AdaptiveAR to use a predetermined static ring size and only
adapt its effective size, i.e., the subset of the ring actually used by the currently active threads. However, this requires advance knowledge of the maximum number of threads that can simultaneously run, which may not be known a priori. TripleAdp addresses this limitation with a technique to adapt the total ring size by allocating and deallocating ring nodes as necessary.

Both AdaptiveAR and TripleAdp have several features that no prior synchronous queue algorithm possesses together. They are based on a distributed scalable ring structure, unlike Java’s synchronous queue which relies on a non-scalable centralized structure. They are non-blocking and uniform, in that no thread has to perform work on behalf of other threads. This is in contrast to the flat combining (FC) based synchronous queues of Hendler et al. [64], which are blocking and non-uniform.

Most importantly, our algorithms perform extremely well in practice on a number of hardware architectures with different characteristics. On an UltraSPARC T2 Plus multicore machine with 64 hardware threads and a write-through cache architecture AdaptiveAR outperforms Java’s synchronous queue by up to 120×, the FC synchronous queue by up to 5.5×, and the Java Exchanger algorithm [72, 82] (an elimination-array implementation) by up to 2.5×. On an Intel Xeon E7 multicore with 20 hardware threads and a write-back cache architecture our algorithms outperform or match the performance of prior synchronous queues in most benchmarks.

1.1.2 Scalable LIFO stack through rendezvousing

We adapt our rendezvousing algorithms for use as an elimination layer on top of Treiber’s lock-free stack [117], yielding a highly scalable stack that outperforms existing concurrent stack implementations. Here push operations act as producers and pop operations as consumers. A pair of operations that successfully rendezvous can be linearized together without having to access the main stack [65].

In addition to using elimination as a backoff scheme that threads enter after sensing contention on the main stack, as was done in prior work [65], we present an optimistic elimination stack. Here a thread enters the elimination layer first, accessing the main stack only if it fails to find a match. If it then encounters contention on the main stack it goes back to try the rendezvous, and so on.

Our AdaptiveAR optimistic elimination stack yields more than a factor of three improvement over the prior elimination-array stack [65] and FC stack [63] algorithms on the Ultra-SPARC T2 Plus architecture.
1.2 Efficient synchronization

Synchronization operations often impose a significant performance penalty on important concurrent algorithms. This part of the dissertation shows how insightful understanding of the hardware mechanisms underlying these penalties reveals opportunities to reduce or eliminate the penalty.

1.2.1 Contended memory locations and FIFO queues

Writing a cache line that was last written by another thread is not scalable, because the processor’s coherence protocol serializes such write operations. Therefore, avoiding contended memory operations – in which many threads write to the same location – is a fundamental principle in the design of concurrent algorithms [69].

But what happens when the problem being solved seems to require such shared writing? The concurrent FIFO queue, a fundamental and commonly used data structure, is a prime example: every enqueue needs to modify the queue’s tail, and every dequeue needs to modify the queue’s head. Indeed, both of Michael and Scott’s classic algorithms [92], one lock-based and one nonblocking, do not scale past a low level of concurrency because threads contend on the queue’s tail and head [63, 69].

To get around this seemingly inherent bottleneck, researchers have recently applied flat combining approaches in which one thread gathers pending operations of other threads and executes them on their behalf [53, 54, 63].

Most non-combining concurrent algorithms synchronize using compare-and-swap (CAS) loops: a thread observes the shared state, performs a computation, and uses CAS to update the shared state. If the CAS succeeds, this read-compute-update sequence appears to be atomic; otherwise the thread must retry. Essentially, the idea behind flat combining is that the synchronization cost of a contended CAS hot spot (due to cache coherency traffic on the contended location) is so large that performing all the work serially, to save synchronization, performs better [63].

In this dissertation we show that the truth is more nuanced: by itself, a contended memory operation is relatively fast; only the compounded cost of repeated CAS failures causes the poor performance of algorithms with a CAS hot spot.

Observing this distinction, let alone exploiting it, is not possible on most commercial multicore architectures which only support the universal primitives CAS or
load-linked/store-conditional (LL/SC). While in theory these can implement weaker primitives in a wait-free manner [66], such implementations are heavyweight and in practice vendors direct programmers to use CAS loops [5]. However, there is an interesting exception: the (64 bit) x86 architecture, which dominates the server and desktop markets, directly supports various theoretically weaker primitives whose crucial property for our purpose is that they always succeed (see Table 1.1).

Consider, for example, the fetch-and-add (F&A) primitive. Figure 1.1 shows the difference in the time it takes a thread to increment a contended counter on a modern x86 system when using F&A vs. a CAS loop. Avoiding the retries and paying only the synchronization price leads to a $4 \times - 6 \times$ performance improvement. Our contribution is in transferring this insight to the domain of FIFO queues, henceforth simply queues.

**Our contribution** We present LCRQ, a linearizable nonblocking FIFO queue that uses contended F&A operations to spread threads among items in the queue, allowing them to enqueue

<table>
<thead>
<tr>
<th></th>
<th>compare-</th>
<th>swap</th>
<th>test-and-</th>
<th>fetch-</th>
<th>and-set</th>
<th>and-add</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>LL/SC</td>
<td>deprecated</td>
<td>no</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POWER</td>
<td>LL/SC</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPARC</td>
<td>yes</td>
<td>deprecated</td>
<td>yes</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x86</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Synchronization primitives supported as machine instructions on dominant multicore architectures.

Figure 1.1: Time to increment a contended counter on a system with four Intel Xeon E7-4870 (Westmere EX) processors, each of which has 10 2.40 GHz cores that multiplex 2 hardware threads. The right vertical axis shows the number of CAS it takes to complete an increment.
1.2. EFFICIENT SYNCHRONIZATION

and dequeue quickly and in parallel, in contrast the inherently serial behavior of combining-based approaches. As a result, LCRQ outperforms prior queue implementations by $1.5 \times$ to $2.5 \times$ on a system with four Intel Xeon E7-4870 multicore processors, both on single-processor and on multi-processor executions. Because LCRQ is nonblocking, it maintains its performance in oversubscribed scenarios in which there are more threads than available hardware threads. In such workloads it outperforms by more than $20 \times$ lock-based combining queues, which cannot make progress if a combiner gets scheduled out.

Our LCRQ algorithm is essentially a Michael and Scott linked list queue [92] in which a node is a concurrent ring (cyclic array) queue, CRQ for short. A CRQ that fills up becomes closed to further enqueues, who instead append a new CRQ to the list and begin working in it. Most of the activity in the LCRQ occurs in the individual CRQs, making contention on the list’s head and tail a non-issue. Within a CRQ, the head and tail are updated using a F&A operation which is used to spread threads around the slots of the ring, where they synchronize using (uncontended in the common case) CAS.

One of the CRQ’s distinctive properties compared to prior concurrent circular array queues [31, 32, 43, 56, 61, 111, 118] is that in the common case an operation accesses only the CRQ’s head or tail but not both. This reduces the CRQ’s synchronization cost by a factor of two, since the contended head and tail are the algorithm’s bottleneck.

1.2.2 Memory fences in work stealing

Modern multicores support relaxed memory models [17], in which some of a thread’s memory operations can be reordered. To enforce order between two memory operations a thread needs to issue a special memory fence instruction. Memory fences are relatively expensive in today’s multicores, since they require stalling the thread’s execution.

In this dissertation we consider the use of memory fences in work stealing algorithms, which are used for dynamic load balancing in the runtimes of task-based parallel programming systems [11, 28, 39, 57, 81, 103, 104]. The task-based model exposes parallelism by expressing a computation as a set of tasks that can be scheduled in parallel, and is used in many programming languages and frameworks, as well as in (multicore) MapReduce.

In a runtime using work stealing, each worker thread has a queue of tasks from which it continuously removes the next task to execute. While executing a task the worker may place new tasks in its queue. If a worker’s queue empties, the worker becomes a thief and tries to steal a task from another worker.
Figure 1.2: Single threaded execution time of several widely used CilkPlus [11] benchmarks [16, 55, 57, 77, 84] when not issuing a memory fence on task removal, normalized to the standard CilkPlus runtime on an Intel Haswell (Core i7-4770) 3.4 GHz processor.

Today’s state of the art work stealing algorithms [13, 40, 57] require a worker to issue a costly (store-load) memory fence instruction when removing a task from its own queue, to ensure that a thief does not take the same task as the worker. As Figure 1.2 shows, on a modern processor this fence can account for up to ≈ 25% of execution time.

The memory fence is needed to prevent reordering of memory operations in the work stealing synchronization protocol, which is typically based on the flag principle [69]. The worker publishes the task it is about to take, then checks whether a thief intends to steal the same task. If not, the worker can safely take the task because any future thief will observe its publication and avoid stealing this task – but this reasoning holds only if the checking load is not reordered before the publishing store!

Thus, it would seem that the worker’s memory fence is unavoidable. In fact, Attiya et al.’s “laws of order” [27] are sometimes interpreted as saying just this [25]. In truth, however, Attiya et al.’s proof holds only under certain assumptions [27], giving hope of circumventing the impossibility result by violating its assumptions.
Our contribution In this dissertation we demonstrate that fence-free work stealing is possible on mainstream processors implementing the total store ordering (TSO) [2, 109] memory model, such as the popular x86 and SPARC. We show that these processors in fact implement a bounded TSO memory model – a novel restriction of TSO which bounds the amount of reordering possible – and that work stealing can be implemented without fences on bounded TSO machines.

The only type of reordering possible in TSO is store/load reordering, which can be modeled as happening due to store buffering [109]: each stored value gets buffered in the store buffer before reaching memory, allowing a later load to be satisfied from memory before the earlier store is written to memory. The bounded TSO model places a fixed bound on the size of the store buffer.

Our insight is that knowing the store buffer’s capacity allows a thief to detect when it can safely steal a task by reasoning about the number of stores pending in the worker’s store buffer. Because this allows the thief to safely steal without needing to see the worker’s latest store in memory, we no longer need the worker to issue a memory fence when taking a task.

To see this, consider a typical work stealing task queue, in which the worker works its way from the tail of the queue to its head and the thief works from the head towards the tail. Consider what happens if we remove the memory fence between the worker writing to memory that it is about to take a task and its subsequent check that no thief intends to steal this task. (We are intentionally glossing over some details in how the worker and thief synchronize, but Section 6.2 gives a full description.) Now, consider the system’s state after a worker working its way from task #10 to task #1 has taken tasks #10, #9 and #8, on a processor with a 4-entry store buffer:

- **stores by worker**

<table>
<thead>
<tr>
<th>stores:</th>
<th>about to take task #8</th>
<th>another store</th>
<th>(latest store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffered</td>
<td>some store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>about to take task #9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>about to take task #10</td>
<td>(earliest store)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A thief reading from memory in this state only sees that the worker is about to take task #10. But the thief knows it is missing at most 4 worker announcements due to store buffering,
which implies that the furthest store the worker could have issued at this time is “about to take task #6.” Therefore, if the thief intends to steal task #1, it is assured that the worker has not already taken this task.

However, bounding the worker’s position in this way may not rule out the possibility that the worker and thief are about to access the same task (say, if the thief intended to steal task #6 in the above scenario). One solution we describe for this case is to have the thief refuse to steal the task, returning a designated ABORT value. In doing so we are effectively relaxing the semantics of the work stealing task queue in a way that makes the “laws of order” not applicable to the new nondeterministic specification.

To deal with the cases in which reasoning about the store buffer leaves the thief uncertain about whether it can safely steal without changing the work stealing semantics we propose a heartbeat mechanism which allows the thief to learn the worker’s updated position when these scenarios arise. This yields a task queue with the standard deterministic work stealing semantics, but one in which a thief’s progress might depend on the worker taking steps – a behavior that invalidates another assumption of the “laws of order” proof.

Introducing our techniques into Intel’s Cilk Plus [11] runtime, which is used by the C/C++ parallel extensions in Intel’s compilers, improves the running time of a suite of parallel benchmarks by up to 25% on Intel Westmere and Haswell processors. Additionally, a fence-free version of the Chase-Lev algorithm [40] performs comparably to Michael et al.’s idempotent work stealing queues [93], which are also fence-free but allow a task to be dequeued by both a worker and a thief, a behavior that many applications cannot tolerate.

1.3 Overcoming system bottlenecks

The third challenge addressed in this dissertation concerns situations in which a concurrent program’s performance is negatively impacted by the system – the elements outside of the program’s control, such as the memory allocator, operating system, or shared hardware resource management policies. We describe two bottlenecks caused by the system’s policies, one due to a hardware mechanism and one due to memory allocator policies, and propose ways to overcome them.
1.3. OVERCOMING SYSTEM BOTTLENECKS

1.3.1 Serialization in hardware lock elision

Transactional memory (TM) [67] is being introduced into mainstream mass-market multicores, e.g., Intel’s latest Haswell microarchitecture, and practitioners try to use the offered hardware-based lock elision [101] to improve the performance of their lock-based programs [14].

However, naively using hardware lock elision can lead to disappointing performance results, as hardware limitations cause excessive serialization and prevent the materialization of the full concurrency in the application from being exposed. We therefore provide software techniques for assisting the hardware’s lock elision, getting around these limitations and significantly improving the performance obtainable when using lock elision.

We use Haswell’s hardware lock elision (HLE) as an example. This HLE mechanism is vulnerable to an avalanche behavior pathology—also known as the lemming effect [48]—which causes unnecessary serialization and limits the concurrency exposed. We propose several simple software-assisted lock elision algorithms to address this problem.

The idea behind lock elision [101] is to execute a speculative transaction with the same scope as a critical section. This allows multiple critical sections to run and complete concurrently, provided that they do not access the same data (i.e., conflict). Implementation-wise, with standard HLE (Figure 1.3) one prefixes the store instruction acquiring the lock with a new XACQUIRE prefix and the store releasing the lock with an XRELEASE prefix. Executing an XACQUIRE store initiates a speculative transactional execution in which the hardware elides the lock acquisition, treating it as a load instead. This allows multiple transactions to run concurrently without conflicting on the lock, while still believing that each of them has acquired the lock and is running alone in the critical section. In a successful conflict-free execution, the XRELEASE store which ends the critical section causes the transaction to commit, making its memory updates globally visible.

However, when a transaction aborts (e.g., due to a conflict) the execution rolls back to the

```
1 shared variable:
2 lock : 1 bit (boolean), initially FALSE

4 lock() {
5   while (TRUE) {
6     while (lock = TRUE) { // busy wait
7       }
8   }
9   ret := XACQUIRE test&set(lock)
10  if ( ret = FALSE)
11     return
12 } 
13 }

15 unlock() {
16   XRELEASE lock := FALSE
17 }
```

Figure 1.3: Applying hardware lock elision to a TTAS (Test&Test&Set) lock.
acquiring store which is now re-issued non-transactionally. In other words, an abort causes the thread to try to reenter the critical section in a standard manner. The globally visible lock acquisition by the aborted thread conflicts with the speculative loads of the lock performed by speculative HLE transactions, and causes all the threads that are in a transaction to abort. In addition, new threads arriving at the critical section see that the lock is taken. This in turn causes such threads to delay their entrance into a transactional execution, or even (in the case of fair locks) serializes the run until a quiescent period in which no thread tries to access the lock.

To address this problem, we propose several lightweight software-assisted lock elision algorithms:

**Software-assisted lock removal:** Rajwar and Goodman [102] observed that one can simply execute transactions with the same scope of the critical section (i.e., start a transaction instead of acquiring the lock and commit instead of releasing the lock) without accessing the lock at all, provided the TM offers some progress guarantee for conflicting transactions. (Otherwise, livelock situations, in which transactions repeatedly abort due to a conflict and restart only to abort again, can occur.) However, Haswell’s TM has a simple “requestor wins” conflict resolution policy [14] which is prone to livelock [34]. We therefore propose software-assisted lock removal (SALR), in which a transactionally executing critical section does not access the lock until it is ready to commit. It then reads the lock and commits if the lock is not held; otherwise, it aborts and retries, giving up and acquiring the lock after a few attempts. In SALR a thread acquiring the lock does not automatically conflict with running transactions nor does it prevent an arriving thread from speculatively starting its transaction. However, SALR is not applicable to every application, as transactions may observe inconsistent state while running.

**Software-assisted conflict management:** To prevent the avalanche problem in HLE transactions without resorting to lock removal, we propose a simple conflict management technique that allows the non-conflicting threads to continue their speculative HLE-based run without any interference from conflicting threads.

To do this, we use an external lock to serialize all threads that aborted due to a conflict, but allows other non-conflicting threads to run concurrently. We implement this by adding a serializing path to the lock implementation, in which an aborted thread acquires a distinct auxiliary lock (without using lock elision) and then rejoins the speculative execution with the other threads. Using this approach conflicting threads are serialized among themselves and do
not interfere with other threads, greatly reducing the probability that a thread aborts so many
times that it must give up and acquire the original lock.

We integrate our conflict management scheme into the SALR, to further reduce the periods
of time in which the lock is acquired non-transactionally, as holding the lock can affect the
progress of SALR transactions.

1.3.2 Cache under-utilization induced by memory allocator

Modern state-of-the-art multi-threaded memory allocators \([30, 35, 36, 91]\) share a common
design: the allocator associates each thread with a private memory region from which allocation
requests are satisfied. These regions consist of superblocks (or free-lists), one for each possible
size class. Requested allocations are rounded up to the nearest size class and satisfied from the
appropriate superblock in the thread’s region. Superblocks typically have at least page — and
often much higher — alignment. (For example, Hoard \([30]\) uses 64K-aligned superblocks.)

This allocator design is cache index-oblivious, and may inadvertently place blocks on a
restricted fraction of the available cache indices. Repetitive access to those blocks by the ap-
plication might then incur excessive conflict misses and degrade performance, in both single-
and multi-threaded executions.

To illustrate the problem, consider a `malloc` allocator that maintains arrays of 128-byte
blocks to satisfy requests of size suitably close to 128 bytes. As blocks are packed densely in
the array we find blocks starting every 128 bytes. If the base of the array is \(B\) then the block
addresses will be \(B, B + 128, B + 256\), and so on. Consider a data cache with 16-byte lines and
128 sets, as is the case for the shared L1 data cache in the UltraSPARC T2+ processor \([3]\). The
set of L1 data cache indices associated with the base addresses of the blocks within the arrays
is restricted to just 16 instead of the full complement of 128. This cache index underutilization
can cause the following performance problems:

**Single-threaded conflict misses** If an application accesses just a few fields in the allocated
blocks (as is often the case \([35]\)), these few fields will fall on a subset of the available cache
lines, and the application may suffer excessive conflict misses as some cache indices are “hot”
and others underutilized and “cold”.

**Multi-threaded conflict misses** Next, consider what happens if multiple threads co-
scheduled in the same core have a similar allocation pattern. Because the superblocks in the
private thread heaps have the same alignment (in other words, $B$ is page-aligned), the cache sets mapped to the allocated blocks will be identical. If the size classes allocated from are such that the returned blocks underutilize the cache, contention on these cache sets will result. Cache set associativity provides some relief here. But it does not eliminate the problem entirely, especially if (but not only if) the associativity level is less than the number of hardware threads that share the cache.

**Our contribution** We demonstrate that many existing state-of-the-art allocators are index-oblivious, admitting performance pathologies for certain block sizes. We show that a simple adjustment within the allocator to control the spacing of blocks can provide better index coverage, which in turn reduces the superfluous conflict miss rate in various applications, improving performance with no observed negative consequences. The result is an *index-aware* allocator. Our technique is general and can easily be applied to most memory allocators and to various processor architectures.

In addition, we show how shared cache contention can be reduced by coloring the base addresses of the arrays of blocks so that allocation arrays for different threads and processes start on different cache indices.

While we describe our techniques in terms of the implementation of a specific `malloc` allocator, it is general and can easily be applied in other environments such as pool allocators [80] or to the object allocators found in managed runtime environments with automatic garbage collection.

### 1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 defines the formal computational model used throughout this dissertation. In Chapter 3 we describe the multicore machines used for performance evaluation throughout the dissertation.

Chapter 4 describes our rendezvous algorithms and their utility in designing a concurrent stack. We prove that our algorithms are correct and nonblocking, using notions of correctness and nonblocking adapted to the rendezvous setting. Further, we provide an extensive performance evaluation of the algorithms.

In Chapter 5 we develop the LCRQ FIFO queue, prove that it is linearizable and nonblocking, and evaluate its performance. Chapter 6 discusses our techniques for fence-free work stealing and evaluates their performance. To reason about these techniques, this chapter extends
1.4. ORGANIZATION

the computational model to capture the store buffering in TSO processors.

Chapter 7 describes the avalanche pathology in hardware lock elision systems, and presents our software-assisted solutions to mitigate it. In Chapter 8 we survey modern memory allocators, showing that they are index-oblivious and admit performance pathologies for certain block sizes. We then discuss our methodology for designing an index-aware allocator, and evaluate its performance. Chapter 9 concludes the dissertation and discusses directions for future work.
Chapter 2

Computational model and definitions

This chapter presents the basic formal definitions required to reason about the algorithms presented in this dissertation. Where needed, individual chapters adapt these definitions for the specific setting studied.

2.1 System model

We use a standard shared memory system model with a set of sequential asynchronous threads that communicate via shared memory. The memory is an array of locations, each holding a 64-bit value. We use the notation $m[a]$ to denote the value stored in address $a$ of the memory.

The memory supports the following primitives:

1. read($a$) (also referred to as load), which returns $m[a]$,

2. write($a, x$) (also referred to as store), which changes $m[a]$’s value to $x$,

3. fetch-and-add, denoted F&A($a, x$), which returns $v = m[a]$ and changes $m[a]$’s value to $v + x$,

4. swap, denoted SWAP($a, x$), which returns $v = m[a]$ and changes $m[a]$’s value to $x$,

5. test-and-set, denoted T&S($a$), which returns $v = m[a]$ and changes $m[a]$’s value to 1,

6. compare-and-swap, denoted CAS($a, o, n$), which changes $m[a]$’s value to $n$ if $m[a] = o$ and returns TRUE, or returns FALSE otherwise,

7. compare-and-swap2, denoted CAS2($a, \langle o_0, o_1 \rangle, \langle n_0, n_1 \rangle$), which changes $m[a]$’s value to $n_0$ and $m[a + 1]$’s value to $n_1$ if $m[a] = o_0$ and $m[a + 1] = o_1$ before returning TRUE, or else returns FALSE. (This primitive is available on x86 processors.)
2.2. Concurrent object implementations

Using the memory operations the threads implement a high-level object defined by a sequential specification, a state machine consisting of a set of states, a set of operations used to transition between the states, and transitions between the states. A transition \( T(s, op) = (s', r) \) specifies that if \( op \) is applied at state \( s \), the object state changes to \( s' \) and \( op \) returns \( r \). An operation \( op \) is total if a transition \( T(s, op) \) is defined for every state \( s \). Otherwise \( op \) is partial [70].

An implementation is an algorithm specifying the shared memory operations each thread has to perform in order to complete an operation on the high-level object.

2.2.1 Executions and events

The computation is modeled as a sequence of events. A high-level event consists of a thread invoking or returning from a high-level operation. A primitive event is a computation step in which a thread invokes a memory operation, receives a response and changes its state according to the algorithm, all in one step. An execution is a (possibly infinite) sequence of events. We consider only well formed executions, in which the subsequence of thread \( T \)'s events consists of (1) zero or more instances of a high-level operation invocation followed by primitive events and finally a high-level response, (2) a possible final pending operation that consists of a high-level invocation without a matching high-level response.

Thread \( T \) is active in an execution \( E \) if \( E \) contains an event by \( T \). Two high-level operations are concurrent in an execution if one of them starts before the other one returns.

2.2.2 Implementation correctness

We judge an implementation of an object to be correct if it satisfies the linearizability [70] condition. Informally, linearizability requires that a high-level operation appears to take place atomically at one point in time during its execution interval.

Formally, we say an execution \( E \) is linearizable if the following holds: (1) one can obtain an execution \( E' \) by adding high-level responses to zero or more of \( E \)'s pending operations, (2) reorder all completed operations in \( E' \) to form a sequential execution \( S \) (i.e., in which there are no concurrent or pending operations) in such a way that the order of non-concurrent operations remains the same, and (3) \( S \) respects the high-level object’s sequential specification.

An implementation is linearizable if every possible execution is linearizable.
2.3 Example sequential specifications

**LIFO stack** A *stack* is an object supporting the operations `push()` and `pop()`. Its state is a sequence of \( m \geq 0 \) items \( \alpha = x_m, \ldots, x_1 \) where \( x_m \) is called the *top* of the stack. A stack whose state is the empty sequence, \( \epsilon \), is said to be *empty*. A `push(y)` operation on state \( \alpha \) makes \( y \) the top of the stack by changing the state to \( y \alpha \). A `pop()` operation on a non-empty state \( y \alpha \) returns \( y \), the top of the stack, and changes the state to \( \alpha \). A `pop()` operation on an empty stack returns a reserved value \( \bot \).

**FIFO queue** A *queue* is an object whose state, \( Q \), is a (possibly empty) sequence of items. It supports an `enqueue(x)` operation that appends \( x \) to the end of \( Q \) and returns OK, and a `dequeue()` operation which removes the first item \( x \) from \( Q \) and returns \( x \), or returns EMPTY if \( Q \) is the empty sequence.

2.4 Concurrency levels

Unless stated otherwise, we consider a model in which the number of threads participating in each execution is finite, but can differ between executions. The algorithm does not know the finite bound on the number of threads in the execution. (This is essentially the *bounded concurrency* model [89].)

In Chapter 4 we additionally consider a stronger model in which the number of threads in the execution is known by the algorithm in advance.

2.5 Progress

An implementation is *wait-free* [66] if it guarantees that a thread completes its operation in a finite number of its own steps, i.e., regardless of the actions of other threads.

An implementation is *nonblocking* [66] if it guarantees that some thread completes an operation in a finite number of steps. In other words, an individual operation may starve, but some operation always makes progress.

2.6 Model vs. reality

Our modeling of the shared memory as *sequentially consistent* [79] is widely used in the concurrent algorithms literature, as it simplifies reasoning about algorithms by representing the
execution as a sequence of interleaved memory operations. However, in practice multicore architecture have relaxed memory models [17] in which some of the threads’ memory operations can be reordered.

How, then, did we implement and evaluate our algorithms? Fortunately, it turns out that sequential consistency suffices to reason about most of our algorithms under the x86 and SPARC multicore architectures used throughout this dissertation. These architectures have a relatively strong total store order (TSO) [109] memory model. Under TSO the only reordering possible is when a read by a thread is ordered before an earlier write by the same thread. Most of the algorithms we present do not rely on load-store ordering, and synchronize using only atomic operations – which are globally ordered in TSO. The only exception are the work stealing algorithms in Chapter 6, which therefore expands the system model.

This dissertation does not consider the ordering requirements of implementing our algorithms on memory models weaker than TSO.
Chapter 3

Experimental setup

This chapter details the multicore architectures and machine configurations that were used for performance evaluation in this dissertation.

For brevity, in the remaining chapters we refer to these architectures by their code names, which are given in bold below.

3.1 Architectures

This section describes the multicore architectures used in our performance evaluation. Table 3.1 lists the full specification of each architecture.

Niagara The UltraSPARC T2 Plus (Niagara II) processor is a multithreaded (CMT) processor. It has 8 1.165 GHZ in-order cores with 8 hardware threads per core, for a total of 64 hardware threads. Each core has a private L1 write-through cache. The L2 cache is shared by all cores.

Westmere The Intel Xeon E7-4870 (Westmere EX) processor has 10 2.4 GHz cores, each multiplexing 2 hardware threads. Each core has private write-back L1 and L2 caches and the L3 cache is shared by all cores.

Haswell The Intel Core i7-4770 (Haswell) processor has 4 3.4 GHz cores, each with 2 hyperthreads. Each core has private write-back L1 and L2 caches and the L3 cache is shared by all cores.
3.2 Machine configurations

For each architecture, we run experiments on the following machine:

**Niagara** We use a Sun SPARC T5240 server with two Niagara processors and 64 GB of RAM. In total this system supports 128 hardware threads.

**Westmere** We use a Fujitsu PRIMERGY RX600 S6 server with four Westmere processors, and 128 GB of RAM. In total this system supports 80 hardware threads.

**Haswell** We use a desktop machine with a Haswell processor and 8 GB of RAM.

3.3 General methodology

Our main interest in this dissertation is exploring the performance opportunities (and pitfalls) created by the shift from classic SMP-style parallelism to multicore chips with low synchronization and communication overheads. Therefore, our evaluation focuses on the single (multicore) processor case – in most experiments we restrict execution to a single processor on the machine. Of course, we do perform multi-processor evaluation when doing so is appropriate. But we perform the more in-depth analysis and experiments on single processor configurations.

<table>
<thead>
<tr>
<th># cores</th>
<th># threads per core</th>
<th>L1 cache</th>
<th>L2 cache</th>
<th>L3 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niagara</td>
<td>8 × 1.165 GHz</td>
<td>8</td>
<td>8 KB, shared</td>
<td>4 MB, shared</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(by core’s threads)</td>
<td>(by all cores)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>16-byte lines</td>
<td>64-byte lines</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4-way assoc.</td>
<td>16-way assoc.</td>
<td></td>
</tr>
<tr>
<td>Westmere</td>
<td>10 × 2.4 GHz</td>
<td>2</td>
<td>32 KB, private</td>
<td>256 KB, private</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64-byte lines</td>
<td>64-byte lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8-way assoc.</td>
<td>8-way assoc.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64-byte lines</td>
<td>64-byte lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8-way assoc.</td>
<td>24-way assoc.</td>
</tr>
<tr>
<td>Haswell</td>
<td>4 × 3.4 GHz</td>
<td>2</td>
<td>32 KB, private</td>
<td>256 KB, private</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64-byte lines</td>
<td>64-byte lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8-way assoc.</td>
<td>16-way assoc.</td>
</tr>
</tbody>
</table>

Table 3.1: Multicore architectures used for evaluation.
Chapter 4

Realizing the parallelism of rendezvousing

A common abstraction in concurrent programming is that of an asymmetric rendezvous mechanism. In this mechanism there are two types of threads, e.g., producers and consumers, that show up. The goal is to match pairs of threads, one of each type, and send them away.

This chapter presents our highly scalable asymmetric rendezvous algorithms, AdaptiveAR and TripleAdp. Section 4.1 first defines correctness and nonblocking notions appropriate for the rendezvousing problem. We then review background and related work in Section 4.2. Next, Section 4.3 describes AdaptiveAR and Section 4.4 describes TripleAdp. We evaluated both algorithms together in Section 4.5. Finally, we describe and evaluate our rendezvous-based elimination stacks in Section 4.6.

4.1 Preliminaries

4.1.1 Asymmetric rendezvous

Informally, in the asymmetric rendezvous problem there are threads of two types, producers and consumers. Producers perform put(\(x\)) operations which return OK. Consumers perform get() operations that return some item \(x\) passed by a producer. Producers and consumers show up and must be matched with a unique thread of the other type, such that a put(\(x\)) and the get() that returns \(x\) must be concurrent.

A rendezvous operation cannot always complete: for example, a thread running alone cannot rendezvous. We formally capture this issue by modeling the rendezvous problem using a
sequential specification of an object that supports partial operations [70] which are undefined for some states. An implementation of a partial operation requires having the operation wait until the object reaches a state in which the operation is defined. We now define an object with partial operations whose specification captures the handoff semantics of rendezvous, in which producers pass data to consumers.

**Definition 1.** A handoff object consists of a single variable, v, that initially contains a reserved value ⊥. It supports two partial operations, put(x) and get(). A put(x) operation is defined only when v = ⊥; it sets v to x and returns OK. A get() operation is defined only when v = x for any x ≠ ⊥; it stores ⊥ back in v and returns x. For simplicity, we assume each put(x) hands off a unique value.

Even with partial operations a sequential specification alone cannot capture the synchronous nature of rendezvous, i.e., that threads need to wait for each other. To see this, notice that there must be an operation that is defined in the initial state and such an operation can therefore complete alone. The following definition addresses this limitation by reasoning directly about the concurrent rendezvous implementation.

**Definition 2.** A concurrent algorithm A implements asymmetric rendezvous if the following hold:

- **Linearizability:** A is a linearizable implementation of the handoff object.
- **Synchrony:** For any execution E of A, and any put(x) operation that completes in E, E can be extended (adding zero or more events) into an execution E' which contains a get() operation that returns x and is concurrent to put(x).

The idea behind these definitions is that when many producers and consumers show up, each matched producer/consumer pair can be linearized [70] as a put(x) followed by a get() returning x.

### 4.1.2 Pairwise nonblocking progress

To reason about progress for rendezvous we must take into account that rendezvous inherently requires waiting – a thread cannot complete until a partner shows up. We therefore consider the joint behavior of producers and consumers, using the following notion:

**Definition 3.** Let E be an execution of asymmetric rendezvous, e ∈ E an execution step, and e' ∈ E the step preceding e. We say that e is a pairwise step if one of the following holds: (1) e
CHAPTER 4. REALIZING THE PARALLELISM OF RENDEZVOUSING

is a step by a put() operation and e' is a step by a get() operation, (2) e is a step by a get() operation and e' is a step by a put() operation.

For example, if we denote an execution step of producer \( p_i \) with \( p_i \)'s id, and analogously use \( c_i \) for consumers, then in the following execution fragment the pairwise steps are exactly the steps marked in bold:

\[ \ldots, p_1, p_5, p_1, c_3, p_5, c_4, c_4, p_1, \ldots \]

We now define our notion of nonblocking progress, which requires that if both producers and consumers take enough steps, some rendezvous operation completes:

**Definition 4.** An algorithm \( A \) implementing asymmetric rendezvous is pairwise nonblocking if, at any point in time, after a finite number of pairwise steps some operation completes.

Note that, as with the definition of the lock-freedom property [66], there is no fixed a priori bound on the number of steps after which some operation must complete. Rather, we rule out implementations that make no progress at all, i.e., that admit executions in which both types of operations take steps infinitely often and yet no operation successfully completes.

### 4.2 Background and related work

#### 4.2.1 Synchronous queues

A synchronous queue using three semaphores was described by Hanson [62]. The Java 5 library improved on this using a coarse-grained locking synchronous queue, which was superseded in Java 6 by Scherer, Lea and Scott’s algorithm [107]. Their algorithm is based on a Treiber-style nonblocking stack [117] that at all times contains rendezvous requests by either producers or consumers. A producer finding the stack empty or containing producers pushes itself on the stack and waits, but if it finds the stack holding consumers, it attempts to partner with the consumer at the top of the stack (consumers behave symmetrically). This creates a sequential bottleneck. Motivated by this, Afek, Korland, Natanzon, and Shavit described elimination-diffracting (ED) trees [22], a randomized distributed data structure where arriving threads follow a path through a binary tree whose internal nodes are balancer objects [113] and the leaves are Java synchronous queues. In each internal node a thread accesses an elimination array in an attempt to avoid descending down the tree. Recently, Hendler et al. applied the flat
4.2. BACKGROUND AND RELATED WORK

combining paradigm of [63] to the synchronous queue problem [64], describing single combiner and parallel versions. In a Single FC channel a thread attempts to become a combiner by acquiring a global lock. Threads that fail to grab the lock instead post their request and wait for it to be fulfilled by the combiner. The combiner continuously scans the pending requests. As it does so it keeps a private stack which contains pending operations of one kind. If it reads a request that can be matched to the top operation in the stack, it pops that operation and releases the two matched threads. Otherwise, it pushes the request on the stack and continues. In the Parallel FC version there are multiple combiners, each handling a subset of participating threads. Each combiner posts leftover operations that remain in its subset to an exchange Single FC synchronous channel, in an attempt to match them with operations from other combiners’ subsets.

4.2.2 Concurrent stacks

Several works exploit the elimination idea to obtain a scalable stack implementation. Originally, Touitou and Shavit [112] observed that concurrent push and pop stack operations can be eliminated by having the push pass its item directly to the pop. Their stack algorithm is not linearizable [70] as it is based on diffracting trees [113]. Shavit and Zemach described a linearizable stack using the elimination concept, but their algorithm is blocking [114].

Hendler, Shavit and Yerushalmi [65] applied elimination as a backoff scheme on top of Treiber’s classic lock-free stack algorithm [117] to obtain a scalable, lock-free, linearizable stack. They used a collision array in which each thread picks a slot trying to collide with a partner. Thus, the collision array is (implicitly) a form of rendezvous.

In Hendler, Shavit and Yerushalmi’s adaptive scheme threads adapt locally: each thread picks a slot to collide in from a sub-range of the collision array centered around the middle of the array. If no partner arrives, the thread eventually shrinks the range. Alternatively, if the thread sees a waiting partner but fails to collide due to contention, it increases the range. Unlike this scheme, in which each thread independently maintains its own range, in our adaptivity technique (Section 4.3), threads make local decisions that have global impact: the ring is resized.

Several concurrent stacks have been shown in works on combining-based constructions. The idea in combining is to have one thread perform the combined work of other threads, saving them from accessing the main structure. Hendler et al. implemented a stack using the flat combining approach [63]. Fatourou and Kallimanis recently presented a combining tech-
nique that provides bounds on the number of remote memory references performed by waiting threads, and demonstrated a stack implementation using that technique [54]. Both FC and Fatourou and Kallimanis’ techniques are blocking, whereas we are interested in nonblocking algorithms.

Fatourou and Kallimanis have also described an efficient *wait-free* universal construction [53], and showed that a concurrent stack implemented using this construction performs as well as the (blocking) FC based stack.

### 4.2.3 Other applications of elimination

Moir et al. used elimination to scale a FIFO queue [95]. In their algorithm an enqueuer picks a random slot in an elimination array and waits there for a dequeuer; a dequeuer picks a random slot, giving up immediately if that slot is empty. It does not seek out waiting enqueuers. Scherer, Lea and Scott applied elimination in their *symmetric exchange channel* [72], where there is only one type of a thread and so the pairing is between any two threads that show up. Scherer, Lea and Scott also do not discuss adaptivity, though an improved version of their algorithm that is part of the Java concurrency library [82] includes a scheme that resizes the elimination array. However, here we are interested in the more difficult *asymmetric* rendezvous problem, where not all pairings are allowed.

### 4.2.4 Rendezvous semantics

Scherer and Scott proposed to model concurrent operations such as rendezvousing, that must wait for another thread to establish a precondition, by splitting them into two: a *reservation* (essentially, announcing that the thread is waiting) and a subsequent *followup* at which the operation takes effect after its request is fulfilled [108]. In contrast, our model does not require splitting an operation into two. In fact, in our algorithms some operations do not have a clear reservation point. Thus, we demonstrate that asymmetric rendezvous can be both reasoned about and implemented efficiently without using the reservation/followup semantics.

### 4.3 AdaptiveAR

#### 4.3.1 Algorithm description

The pseudo code of the AdaptiveAR algorithm is provided in Figure 4.2. The main data structure (Figure 4.2a) is a ring of nodes. The ring is accessed through a central array `ring`, where
ring\[i\] points to the \(i\)-th node in the ring (Figure 4.1) \(^1\). A consumer attempts to capture a node in the ring. It traverses the ring searching for an empty node, and once a node is captured, waits there for a producer. A producer scans the ring, seeking a waiting consumer. Once a consumer is found, it tries to match with it. For simplicity, in this section we assume the number of threads in the system, \(T\), is known in advance and pre-allocate a ring of size \(T\). However, the effective size of the ring is adaptive and it is dynamically adjusted as the concurrency level changes. (We expand on this in Section 4.3.3.) In Section 4.4 we handle the case in which the maximum number of threads that can show up is not known in advance; in this case the nodes are allocated and deallocated dynamically, i.e., the total ring size adapts to the number of threads.

Conceptually, each ring node contains a pointer that encodes the node’s state:

1. **Free**: pointer points to a global reserved object, FREE, that is distinct from any object a producer may enqueue. Initially all nodes are free.

2. **Captured by consumer**: pointer is NULL.

3. **Holding data (of a producer)**: pointer points to the data.

In practice, ring traversal is more efficient by following a pointer from one node to the next rather than the alternative of traversing the array. Array traversal in Java suffers from two inefficiencies. First, in Java reading from the next array cell may result in a cache miss (it is an array of pointers), whereas reading from the (just accessed) current node does not. Second, maintaining a running array index requires an expensive test+branch to handle index boundary conditions or counting modulo the ring size, while reading a pointer is cheap. The

![Figure 4.1: The ring data structure with the array of pointers. Node \(i\) points to node \(i - 1\). The ring maximum size is \(T\) and is resized by changing the head’s prev pointer.](image)

\(^1\)This reflects Java semantics, where arrays are of references to objects and not of objects themselves.
pointer field is named \texttt{prev}, reflecting that node $i$ points to node $i-1$. This allows the ring to be \textit{resized} with a single atomic \texttt{CAS} that changes \texttt{ring[1]}'s (the head's) \texttt{prev} pointer. To support mapping from a node to its index in the array, each node holds its index in a read-only \texttt{index} field.

For clarity we start in Section 4.3.2 by discussing the non-adaptive ring size algorithm. The adaptivity code, however, is included in the pseudo code, marked by a $\triangle$ symbol. It is explained in Section 4.3.3. Finally, Section 4.3.4 discusses correctness and progress.

### 4.3.2 Nonadaptive algorithm

**Producers (Figure 4.2b):** A producer searches the ring for a waiting consumer, and attempts to pass its data to it. The search begins at a node, $s$, obtained by hashing the thread’s id (Line 25). The producer passes the ring size to the hash function as a parameter, to ensure the returned node falls within the ring. It then traverses the ring looking for a node captured by a consumer. Here the producer periodically \textit{peeks} at the initial node $s$ to see if it has a waiting consumer (Lines 30-32); if not, it checks the current node in the traversal (Lines 33-36). Once a captured node is found, the producer tries to deposit its data using a \texttt{CAS} (Lines 31 and 34). If successful, it returns.

**Consumers (Figure 4.2c):** A consumer searches the ring for a free node and attempts to capture it by atomically changing its \texttt{item} pointer from \texttt{FREE} to \texttt{NULL} using a \texttt{CAS}. Once a node is captured, the consumer spins, waiting for a producer to arrive and deposit its item. Similarly to the producer, a consumer hashes its id to obtain a starting point, $s$, for its search (Line 45). The consumer calls the \texttt{findFreeNode} procedure to traverse the ring from $s$ until it captures and returns node $u$ (Lines 65-80). (Recall that the code responsible for handling adaptivity, which is marked by a $\triangle$, is ignored for the moment.) The consumer then waits until a producer deposits an item in $u$ (Lines 47-55), reads the deposited value and frees $u$ (Lines 57-58), and returns (Line 62).

### 4.3.3 Adding adaptivity

If the number of active consumers is smaller than the ring size, producers may need to traverse through a large number of empty nodes before finding a match. It is therefore important to \textit{decrease} the ring size if the concurrency level is low. On the other hand, if there are more concurrent threads than the ring size (high contention), it is important to dynamically \textit{increase}
4.3. ADAPTIVE AR

struct node {
    item : pointer to object
    index : node’s index in the ring
    prev : pointer to previous ring node
}

7 constants (Section 4.3.3):
8 \( T_i \) : increase threshold
9 \( T_d \) : decrease threshold
10 \( T_w \) : wait threshold

12 shared vars:
13 ring : array \([1,...,T]\) of pointers to nodes,
14 \( \text{ring}[i].item = \text{FREE}, \text{ring}[i].index = i \)
15 \( \text{ring}[1].prev = \text{ring}[T], \text{ring}[i].prev = \text{ring}[i-1] \) (\( i > 1 \))

18 utils:
19 getRingSize () {
20 node tail := ring[1].prev
21 return tail.index
22 }

(a) Global variables

23 put(threadId , object item) {
24 node s := ring[hash(threadId ,
25 getRingSize ())]
26 node v := s.prev
27 while (true) {
28     if (s.item == NULL) {
29         if (CAS(s.item, NULL, item))
30             return OK
31     } else if (v.item == NULL) {
32         if (CAS(v.item, NULL, item))
33             return OK
34     } else
35         v := v.prev
36     }
37 }

(b) Producer code

40 get(threadId) {
41 int ringsize , busy_cotr , ctr := 0
42 node s , u
43 ringsize := getRingSize ()
44 s := ring[hash(threadId , ringsize )]
45 (u, busy_cotr) := findFreeNode(s , ringsize )
46 while (u.item == NULL) {
47     if (cas(s.item, NULL, item))
48         return OK
49     if (cas(u.item, FREE))
50         return OK
51     if (u.index > ringsize and
52         CAS(u.item, FREE, NULL))
53         (u, busy_cotr) := findFreeNode(s , ringsize )
54         ctr := ctr + 1
55 }
56 item := u.item
57 u.item := FREE
58 if (busy_cotr < \( T_d \) and \( T_w \) and ringsize>1)
59 // Try to decrease ring
60 CAS(ring[1].prev, ring[ringsize ], ring[ringsize +1])
61 return item
62 }
63 }
65 findFreeNode(node s , int ringsize ) {
66 int busy_cotr := 0
67 while (true) {
68     if (s.item == FREE and
69         CAS(s.item, FREE, NULL))
70         return (s, busy_cotr)
71     if (cas(s.item, BUSY, NULL))
72         s := s.prev
73         busy_cotr := busy_cotr + 1
74     if (busy_cotr > \( T_w \) - ringsize)
75         if (ringsize < T) // Try to increase ring
76         CAS(ring[1].prev, ring[ringsize ], ring[ringsize +1])
77         ringsize := getRingSize ()
78         busy_cotr := 0
79 }
80 }

(c) Consumer code

Figure 4.2: Asymmetric rendezvous algorithm. Lines beginning with \( \triangleright \) handle the adaptivity process.
the ring. The goal of the adaptivity scheme is to keep the ring size “just right” and allow threads to complete their operations with a small number of steps.

The logic driving the resizing process is in the consumer’s code, which detects when the ring is overcrowded or sparsely populated and changes the size accordingly. If a consumer fails to capture a node after passing through many nodes in `findFreeNode()` (due to not finding a free node or having its CASes fail), then the ring is too crowded and should be increased. The exact threshold is determined by an *increase threshold* parameter, $T_i$ ($0 < T_i \leq 1$). If in `findFreeNode()`, a consumer fails to capture a node after passing through more than $T_i \cdot \text{ring.size}$ nodes, it attempts to increase the ring size (Lines 73-78). To detect when to decrease the size of the ring, we observe that when the ring is sparsely populated, a consumer usually finds a free node quickly, but then has to wait longer until a producer finds it. Thus, we add a *wait threshold*, $T_w$, and a *decrease threshold*, $T_d$. If it takes a consumer more than $T_w$ iterations of the loop in Lines 47-55 to successfully complete the rendezvous, but it successfully captured its ring node in up to $T_d$ steps, then it attempts to decrease the ring size (Lines 59-61).

Resizing the ring is made by CASing the `prev` pointer of the ring head (ring[1]) from the current tail of the ring to the tail’s successor (to increase the ring size by one) or its predecessor (to decrease the ring size by one \(^2\). If the CAS fails, then another thread has resized the ring and the consumer continues. The head’s `prev` pointer is not a sequential bottleneck because resizing is a rare event in stable workloads. Even if resizing is frequent, the thresholds ensure that the cost of the CAS is negligible compared to the other work performed by the algorithm, and resizing pays off in terms of better ring size which leads to improved performance.

Notice that if some consumer is waiting at the tail of the ring before the ring size decreases, this consumer will be left outside the new ring and so new arriving producers will not find it, which may impact progress. We next describe how to solve this problem.

**Handling consumers left out of the ring:** Each consumer periodically checks if its node’s index is larger than the current ring size (Line 49). If so, it tries to free its node using a CAS (Line 50) and find itself a new node in the ring (Lines 51-52). However, if the CAS fails, then a producer has already deposited its data in this node and so the consumer can take the data and return (this will be detected in the next execution of Line 47).

\(^2\)The tail node, which is now left out of the ring, remains pointed to by the `ring[]` array and may be reused later if the ring size increases. Therefore, we do not reclaim its memory.
4.3.4 Correctness proofs

Theorem 5. AdaptiveAR implements asymmetric rendezvous.

Proof. We need to show: (1) linearizability (that AdaptiveAR is a linearizable implementation of the handoff object) and (2) synchrony (that a get() returning $x$ executes concurrently with put($x$)).

Synchrony: Let $p$ be some put($x$) operation which completes. Then $p$ reads NULL from $s$.item, for some node $s$ in the ring (Line 30 or 33), and deposits $x$ in $s$.item using CAS (Line 31 or 34). Thus, there exists a consumer $c$ who changed $s$.item from FREE to NULL (Line 69) and is therefore executing the loop in Lines 47-55 when $p$ CASes $x$ to $s$.item.

We prove that if $c$ continues to take steps after $p$’s CAS, it returns $x$. When $c$ executes Line 47, it notices that $x$ has written to $s$.item and returns $x$ (Lines 57-62). Notice that even if $c$ attempts to free its node due to a ring resize (Line 50), its CAS fails since $s$.item contains $x$ and thus $c$ proceeds to execute Line 47.

Linearizability: We assign linearization points as follows. Let $p$ be a put($x$) that completes its operation. Then $p$ is linearized at its successful CAS of $x$ into $s$.item for some node $s$ (Line 31 or 34). As shown above, there exists a unique concurrent get() operation, $c$, that captured $s$ before $p$’s linearization point and which returns $x$ if it takes enough steps. We linearize $c$ returning $x$ immediately after $p$, at the same CAS event as $p$. Conversely, let $c$ be a get() operation which completes its operation. Then $c$ captures a node $s$ and then spins until observing some item $x$ stored in $s$.item (Line 47). This occurs as a result of some producer CASing $x$ into $s$.item, and so it follows that in the process described above of assigning linearization points to put($x$)/get() pairs, we assign a linearization point to $c$. Thus, any operation that completes has been assigned a linearization point.

Because we always linearize a put() followed immediately by its matching get() at the put()'s successful CAS operation, our linearization points assignment yields a sequence of matching put()/get() pairs. It remains to show that this sequence is a valid execution of the handoff object. To do this, we show using induction on the number of put()/get() pairs linearized that our linearization point assignment constructs a valid execution of the handoff object that leaves the object in state $\bot$. In the base case no operation is linearized, and this is the (valid) empty execution.

For the induction step, we have a valid execution of the handoff object that ends with the
object in state ⊥. In each put() / get() linearized, we linearize the put() first. Thus the next operation linearized is a put(x), which is valid in state ⊥, and changes the object’s state to x. We next linearize a get() returning x, which is valid and changes the object’s state to ⊥.

**Theorem 6.** AdaptiveAR is pairwise nonblocking.

**Proof.** We say that the rendezvous of a put() and get() pair occurs at the linearization point of the pair, as defined in the proof of Theorem 5. Assume towards a contradiction that there is an execution E with infinitely many pairwise steps and yet no rendezvous occurs. Consider E₁, the (infinite) suffix of E in which any operation whose rendezvous occurs in E no longer takes steps. (E₁ is well defined because any thread whose rendezvous occurs performs a finite number of steps before completing, and so after some point in E any such operation either completes or never takes another step.)

The following two lemmas show that eventually the ring size does not change, allowing us to consider an execution where the ring size remains fixed.

**Lemma 7.** If the ring size changes in E₁, it only increases.

**Proof.** The ring size is decreased only after a rendezvous occurs (Lines 59-61). By definition of E₁, no thread whose rendezvous completes takes steps in E₁, so the ring size cannot decrease in E'.

**Lemma 8.** From some point onwards in E₁, the ring size does not change.

**Proof.** Consider an event e ∈ E₁ by some operation op that changes the ring size. There are two possible cases: (1) e is a CAS decreasing the ring size, which is impossible by Lemma 7, or (2) e is a CAS increasing the ring size (Line 75). Before executing e, op executes Line 74, and so e increases the ring size to some r ≤ T. Therefore there can be only finitely many such ring increasing events. Since E₁ is infinite, from some point onwards the ring size does not change.

Let E' be the (infinite) suffix of E₁ in which the ring size does not change. Since there is a finite number of threads, T, there must be a producer/consumer pair, p and c, each of which runs forever without completing in E'. We now prove that c cannot run forever without capturing some ring node.

**Lemma 9.** At some point in E', there is a node that has been captured by c.
4.4. **TRIPLEADP**

**Proof.** If \( c \) captures a node before \( E' \) and holds it captured in \( E' \), we are done. Otherwise, suppose \( c \) never captures a node in \( E' \). Then the ring size must be \( T \). Otherwise, when \( c \) tries to increase the ring size (Lines 73-75) it either succeeds or fails because another thread changes the ring size, both of which are impossible by definition of \( E' \).

A ring of size \( T \) has room for \( c \), as \( T \) is the maximum number of threads. Thus \( c \) fails to capture a node only by encountering another consumer, \( c' \), twice at different nodes. This \( c' \) left one node and captured another only as a result of the following: (1) noticing the ring decreasing or (2) completing its operation and returning again, both of which are impossible by definition of \( E' \). It therefore cannot be that \( c \) never captures a node.

Consider now the execution fragment in \( E' \) in which \( c \) holds some node \( s \) captured. Then \( s.item = \text{NULL} \). There are two possible cases: (1) \( p \) visits \( s \) or (2) \( p \) never visits \( s \).

Suppose that \( p \) visits \( s \) while executing its loop. Then \( p \) attempts to rendezvous with \( c \) by executing Line 31 or 34. If \( p \)’s CAS succeeds, a rendezvous has occurred. Otherwise, either another producer rendezvouses with \( c \) or \( c \) leaves \( s \). As shown above, \( c \) leaving \( s \) implies that a rendezvous has occurred. In any case we reach a contradiction.

Therefore, suppose \( p \) never visits \( s \). This implies that \( s \) has been left out of the ring by some decreasing resize. Since in \( E' \) the ring does not decrease, \( c \) eventually executes Lines 49-52 and moves itself into \( p \)’s range. Because the ring size does not change, this means that \( p \) eventually visits \( s \), a contradiction.

4.4 **TripleAdp**

Here we present TripleAdp, an algorithm that adds two new forms of adaptivity to address some of AdaptiveAR’s limitations. First, TripleAdp adapts the roles performed by the threads to solve the problem wherein AdaptiveAR’s throughput degrades as the number of producers grows in workloads with more producers than consumers. Second, TripleAdp adapts its total ring size to match the changing number of threads. Thus TripleAdp can run without knowing the maximum number of threads in the system. In addition, adapting the total ring size makes TripleAdp’s memory consumption adaptive as well.

4.4.1 **Algorithm description**

The main idea behind TripleAdp is to switch between running AdaptiveAR and its mirror image, where consumers seek producers that wait on the ring. TripleAdp’s main data struc-
CHAPTER 4. REALIZING THE PARALLELISM OF RENDEZVOUSING

Figure 4.3: Algorithm TripleAdp: definitions and helper procedures.

ture is a ring of nodes with similar structure to AdaptiveAR’s node (Figure 4.3, Lines 1-5); the difference from AdaptiveAR is explained below. To support reallocating the ring we use a level of indirection: a global shared ring variable points to the ring (Lines 8-9). This allows a thread to install a new ring/array pair. The thread first allocates a new ring (Lines 14-23) and then installs it with a single atomic CAS to point ring at the new array and thereby the new ring. Naturally, threads waiting in the old ring need to move to the new ring, a procedure we describe later.

A thread in TripleAdp is either a waiter which captures a node in the ring and waits for a partner, or a seeker that scans the ring looking for waiters. To determine what role an arriving thread plays the thread consults a global mode variable (Figure 4.4). If mode=1 then consumers wait and producers seek, as

3This is standard array semantics in Java, but not in C++.
in AdaptiveAR. If mode=2 then roles are reversed, i.e., AdaptiveAR’s mirror image is run. The mode can change during the execution in response to the workload and so the wait and seek procedures periodically check for a mode switch, upon which the thread’s role is reversed. This means that role reversal does not occur atomically with the mode change, so threads in different modes can be active concurrently. To prevent such threads from stepping on each other in a ring node, each node has a two member array field named item, where item[m] is accessed only by threads in mode m and encodes the node’s state in mode mode:

1. **Free**: item[mode] points to a global reserved object, FREE, distinct from any object a producer may enqueue.

2. **Captured (by a waiter)**: If mode = 1 (consumers waiting), item[1]=NULL. Otherwise (producers waiting), item[2] holds producer’s item.

3. **Rendezvous occurred**: If mode = 1 item[1] holds producer’s item. Otherwise, item[2]=NULL.

Figure 4.5 shows the pseudo code of the wait and seek procedures. For clarity, we initially describe the code without going into the details of adapting the ring size and memory consumption, to which we return later. The relevant lines of code are shown in the pseudo code marked by a but are ignored for the moment.

**Waiters (Figure 4.5a)**: First (Line 70), a waiter determines whether it is a consumer or producer by invoking the helper procedure waitItemIndex() whose code is given in Figure 4.3 (Lines 30-35). If the waiter does not have a value to hand off then it is a consumer, implying the algorithm is in the consumers waiting mode (m = 1). Otherwise, it is a producer and m = 2. Recall that determining the value of m indicates which of two item slots the waiter next waits on. The waiter then attempts to capture a node in the ring and waits there for a seeker to rendezvous with it. This process generally follows that of AdaptiveAR. The waiter starts by searching the ring for a free node to capture. It hashes its id to obtain a starting point for this search (Lines 73-74) and then invokes the findFreeNode procedure to perform the search (Line 75). The pseudo code of findFreeNode is shown in Figure 4.6. Here the waiter traverses the ring and upon encountering a FREE node, attempts to CAS the node’s item to its value (Lines 153-156). (Recall that value holds the thread’s item if it is a producer, and NULL otherwise.) The traversal is aborted if the global mode changes from the current mode (Lines 157-158) and the waiter then reverses its role (Lines 76-77). Once findFreeNode captures a
wait(int threadId, object value) {
    node s, u
    node[] r

    ctr := 0
    i := waitItemIndex(value)
    while (true) {
        r := ring // private copy of ring pointer
        s := r[hash(threadId, ringsize)]
        (u, busy, ret) := findFreeNode(value, r, ringsize, s)
        if (ret = MODE-CHG)
            return seek(threadId, value)
        if (ret = RING-CHG)
            continue; // Restart loop
        while (true) {
            // Wait for partner
            if (u.item[i] != value)
                object item := u.item[i]
                // try to decrease ring
                if (ringsize - 1 == size(r)/4)
                    node[] nr = allocateRing(ringsize - 1, size(r)/2)
                else
                    CAS(r[1].prev, r[ringsize - 1], r[ringsize - 1])
                return item
            if (mode != i)
                return wait(threadId, value)
            if (ring != r)
                break; // Restart outer loop
        }
    }
}

seek(int threadId, object value) {
    object item
    int i, ctr
    node s, v
    node[] r

    i := seekItemIndex(value)
    while (true) {
        r := ring // private copy of ring pointer
        s := r[hash(threadId, getRingSize(r))]
        v := s.prev
        ctr := 0
        while (true) {
            // Wait for partner
            if (mode != i)
                return wait(threadId, value)
            if (ring != r)
                break; // Restart outer loop
        }
    }
}

CHAPTER 4. REALIZING THE PARALLELISM OF RENDEZVOUSING

Figure 4.5: TripleAdp pseudo code for seeking and waiting threads. Lines marked by a □ handle ring size adaptivity.
node $u$ (Line 75), the waiter spins on $u$ and waits for a seeker. If the waiter detects a global mode change during this period it releases $u$ and reverses its role (Lines 98-99). The waiter returns (Line 96) once it notices that $u$’s content has changed (Lines 81-83). If the waiter is a consumer, it returns the seeking producer’s deposited value, or NULL otherwise.

**Seekers (Figure 4.5b):** A seeker traverses the ring, looking for a waiter to rendezvous with. Symmetrically to the waiter code, the seeker starts by determining its operation mode (Line 115). The seeker then begins traversing the ring from an initial node $s$ obtained by hashing the thread’s id (Lines 117-118). As in the waiter case, the traversal process follows that of AdaptiveAR: the seeker proceeds while peeking at $s$ to see if a waiter has arrived there in the meantime (Lines 126-131). If $s$ has no waiter, the next node in the traversal is checked. To determine if a node is captured by a waiter, the seeker uses the helper procedure hasWaiter() (Figure 4.3, Lines 46-Lines 51). Once a waiter is found, the seeker tries to match with it by atomically swapping the contents of the node’s item with its value using CAS. If successful, it returns the value previously stored in the node. During each iteration of the loop, the seeker validates that it is in the correct mode and switches roles if not (Lines 122-123).

**Adapting effective and total ring size (□-marked lines):** The waiter adjusts both the effective and total ring size. Deciding when to resize the ring and adjusting its effective ring size is performed as in AdaptiveAR (Lines 161-175, 163-165, 84 and Line 92). Where AdaptiveAR would fail to increase the effective ring size due to using a statically sized ring, in TripleAdp the waiter allocates a new ring of double...
the size and tries to install it using a CAS on ring (Lines 168-171). In the other direction, when
the effective ring size is a quarter of the ring size the waiter swings ring to point to a new
allocated ring of half the old size (Lines 85-91). Seekers and waiters check whether a new ring
has been allocated (Lines 124-125 and 101,159-160). If so they move themselves to the new
ring by restarting the operation.

Adapting to the access pattern: Because ring size adaptivity sizes the ring according to the
number of waiters, in an asymmetric access pattern we want to assign the role of waiter to
the majority thread type (producer or consumer). Otherwise, there will be less ring slots than
seekers (which outnumber the waiters) and this will cause seekers to collide with each other on
the ring. For example, with \( N \) producers and a single consumer, we want a ring of size \( N \) and not
\( N \) producers contending on one slot. Following this intuition, we make seekers responsible for
adapting the operation mode to the access pattern, by having a seeker deduce that the operation
mode should be changed if it starts to experience collisions with other seekers in the form of
CAS failures. A seeker maintains a counter of failed CAS attempts (Lines 133, 143). A seeker
that completes a rendezvous checks if its counter is above a contention threshold \( T_f \), and if so
it toggles the global operation mode using a CAS on mode (Lines 129-130, 139-140).

Memory reclamation: We assume the runtime environment supports garbage collection
(GC).\(^4\) Thus, once the pointer to an old ring is overwritten and all threads move to the new
ring, the garbage collector reclaims the old ring’s memory.

We assume GC for simplicity. A different memory management scheme that allows safe
memory reclamation for concurrent nonblocking code will work as well, but may require
changes to TripleAdp to interface with the memory manager. For example, consider the use
of hazard pointers [90]. An operation must point a hazard pointer to the ring it will access after
reading the global ring pointer (Lines 72,117). After pointing the hazard pointer at the ring, the
operation must check that the global ring pointer has not changed and retry the process if it has.
If the ring has not changed since the operation wrote its hazard pointer, it is guaranteed
that the ring will not be reclaimed until the operation’s hazard pointer changes [90]. The op-
eration must therefore nullify its hazard pointer once it completes. The proof of Theorem 11
(Section 4.4.2) can be extended to show that TripleAdp remains pairwise-nonblocking when
using hazard pointers.

\(^4\)GC is part of modern environments such as C# and Java, in which most prior synchronous queue algorithms
were implemented [22, 64, 107].
4.4. TRIPLEADP

4.4.2 Correctness proofs

**Theorem 10.** TripleAdp implements asymmetric rendezvous.

**Proof.** We prove linearizability and synchrony of TripleAdp:

**Synchrony:** Let \( p \) be some \( \text{put}(x) \) operation that successfully \( \text{CAS} \)es \( x \) into \( \text{item}[i] \) pointer while in \( \text{seek}(p,x) \) (Figure 4.5b, Lines 128,138). It follows that \( i = 1 \), since that is the index \( \text{seekItemIndex()} \) returns when passed \( x \neq \text{NULL} \) (Figure 4.3, Lines 39-40). This, in turn, implies that \( p \) observes \( \text{NULL} \) in \( \text{item}[1] \) before proceeding with its \( \text{CAS} \), as that is when \( \text{hasWaiter()} \) returns TRUE when \( i = 1 \) (Lines 47-48). Thus the waiter found by \( p \) is some \( \text{get()} \) operation, \( c \), that invokes \( \text{wait}(c,\text{NULL}) \) before \( p \) \( \text{CAS} \)es \( x \) into the node.

We prove that if \( c \) continues to take steps after \( p \)'s \( \text{CAS} \), it returns \( x \). When \( c \) executes Line 81, it notices \( x \) written to \( \text{item}[1] \) and returns \( x \) (Lines 81-96). Notice that even if \( c \) attempts to free its node due to a mode change (Lines 98-99) or ring resize (Lines 101-104), its \( \text{CAS} \) will fail and thus \( c \) will proceed to execute Line 81.

A symmetric argument shows that a \( \text{get()} \) operation which successfully \( \text{CAS} \)es \( \text{NULL} \) into some node's \( \text{item}[i] \) pointer must have \( i = 2 \) and must therefore observe a producer waiting in some \( \text{wait}(p,x) \) call.

**Linearizability:** We define linearization points for the operations as follows, depending on the roles they play in TripleAdp. Let \( p \) be a \( \text{put}(x) \) operation that completes a rendezvous while executing \( \text{seek}(p,x) \). Then \( p \) is linearized at its successful \( \text{CAS} \) of \( x \) into \( \text{item}[1] \), for some node \( s \). As shown above, there is then a unique concurrent \( \text{get()} \) operation, \( c \), that \( \text{CAS} \)es \( \text{NULL} \) to \( \text{item}[1] \) from \( \text{wait}(c,\text{NULL}) \) before \( p \)'s linearization point, and which returns \( x \) if it takes enough steps. We linearize \( c \) returning \( x \) immediately after \( p \), at the same \( \text{CAS} \) event. Conversely, let \( c \) be a \( \text{get()} \) operation which completes its operation in \( \text{wait}(c,\text{NULL}) \). Then \( c \) captures a node \( s \) by \( \text{CAS} \)ing \( \text{NULL} \) into \( \text{item}[1] \), and then spins until observing some item \( x \) stored in \( \text{item}[1] \) (Line 81). This occurs as a result of some producer \( \text{CAS} \)ing \( x \) into \( \text{item}[1] \). It follows that in the process described above of assigning linearization points to \( \text{put}(x)/\text{get()} \) pairs, we assign a linearization point to \( c \).

Symmetrically, let \( c \) be a \( \text{get()} \) operation that completes a rendezvous from \( \text{seek-}(c,\text{NULL}) \). Then \( c \) successfully \( \text{CAS} \)es \( \text{NULL} \) into \( \text{item}[2] \), for some node \( s \), and there exists a \( \text{put}(x) \), \( p \), executing \( \text{wait}(p,x) \) which previously \( \text{CAS} \)es \( x \) into \( \text{item}[2] \). We linearize \( p \) at \( c \)'s successful \( \text{CAS} \), and linearize \( c \) returning \( x \) immediately after \( p \), at the same \( \text{CAS} \) event. Con-
versely, let \( p \) be a \( \text{put}(x) \) operation that completes a rendezvous while in \( \text{wait}(p,x) \). Then \( p \) captures a node \( s \), \( \text{CAS}es \) \( x \) into \( s.\text{item}[2] \), and then spins until observing that \( x \) is not stored in \( s.\text{item}[2] \) (Line 81). This occurs as a result of some \( \text{get()} \) running \( \text{seek()} \) \( \text{CAS}ing \) \( \text{NULL} \) into \( s.\text{item}[2] \). It follows that in the process described above of assigning linearization points to \( \text{get()}/\text{put}(x) \) pairs, we assign a linearization point to \( p \).

Thus, any operation that completes has a linearization point assigned to it. Because we always linearize a \( \text{put()} \) followed immediately by its matching \( \text{get()} \) at a successful \( \text{CAS} \) operation by one member of the pair, our linearization points assignment yields a sequence of matching \( \text{put()}/\text{get()} \) pairs. It remains to show that this sequence is a valid execution of the handoff object. To do this, we show using induction on the number of \( \text{put()}/\text{get()} \) pairs linearized that our linearization point assignment constructs a valid execution of the handoff object that leaves the object in state \( \bot \). In the base case no operation is linearized, and this is the (valid) empty execution.

For the induction step, we have a valid execution of the handoff object that ends with the object in state \( \bot \). The next operation linearized is a \( \text{put}(x) \), which is valid and changes the object’s state to \( x \). We then linearize a \( \text{get()} \) returning \( x \), which is valid and changes the object’s state to \( \bot \).

**Theorem 11.** \( \text{TripleAdp} \) is pairwise nonblocking.

**Proof.** Similarly to Theorem 6’s proof, we say that the rendezvous of a \( \text{put()} \) and \( \text{get()} \) pair occurs at the linearization point of the pair, as defined in the proof of Theorem 10. The proof is by contradiction. Assume that there is an execution \( E \) in which there are infinitely many pairwise steps and yet no rendezvous occurs. We consider \( E' \), the infinite suffix of \( E \) in which any operation whose rendezvous occurs in \( E \) no longer takes steps. Notice that \( E' \) cannot contain mode switches, since a mode switch happens after a seeker’s rendezvous occurs (Lines 128,138). Thus, in \( E' \) \( \text{TripleAdp} \) either stays in producers waiting mode (AdaptiveAR mirror image) or consumers waiting mode (essentially AdaptiveAR). We first prove that if no ring reallocations occur in \( E' \), then \( \text{TripleAdp} \) running with a fixed mode is pairwise nonblocking, leading to a contradiction (Lemmas 12 and 13). Thus \( \text{TripleAdp} \) must reallocate the ring in \( E' \). In fact, we show that \( \text{TripleAdp} \) must reallocate the ring infinitely often (Lemma 14). We conclude by showing that this too implies a contradiction.

**Lemma 12.** If from some point onwards in \( E' \), no ring reallocations occur and \( \text{TripleAdp} \) runs with mode \( = 2 \) then a rendezvous occurs.
4.4. **TRIPLEADP**

*Proof.* Assume towards a contradiction that the lemma is false. Then, as in Theorem 6’s proof, there is a producer/consumer pair, \( p \) and \( c \), each of which runs forever without completing. Let \( r \) be the ring after the last ring reallocation in \( E' \). Then \( p \) and \( c \) must eventually execute on \( r \): eventually each of them executes Lines 124-125 or 101,159-160) and, if its current ring differs from \( r \), moves itself to \( r \). Once \( p \) and \( c \) are on the same ring, the same arguments as Theorem 6’s proof show that \( p \) must eventually capture a node. This implies that \( c \) doesn’t find \( p \) on the ring. This can only occur if \( p \) was left out of the ring because some producer decreased the ring. But, since from that point on, the ring size cannot decrease (otherwise a rendezvous completes), \( p \) eventually moves itself into \( c \)'s range, where either \( c \) or another consumer rendezvous with it, a contradiction. 

**Lemma 13.** If from some point onwards in \( E' \), no ring reallocations occur and TripleAdp runs with mode = 1 then a rendezvous occurs.

*Sketch of proof* Following Lemma 12’s proof.

**Lemma 14.** If \( E' \) contains a finite number of ring reallocations, a rendezvous occurs.

*Proof.* After the last ring reallocation, either Lemma 12 or Lemma 13 applies.

It follows that \( E' \) must contain infinitely many ring reallocations. A ring reallocation that reduces the total ring size (Lines 85-91) is performed after a waiter’s rendezvous occurs, which is impossible. Thus, the only ring reallocations possible are ones that increase the total ring size. To complete the proof, we show that there cannot be infinitely many such ring reallocations. Here we use the assumption that the concurrency is finite. Let \( T \) be the maximum number of threads that can be active in \( E \); we show that the ring size cannot keep growing indefinitely. Assume w.l.o.g. that the increase threshold \( T_i \) is 1. Let \( R \) be the initial ring to be installed with size \( \text{size}(R) \geq T \). Then \( R \)'s effective size always remains \( \geq T \), because a ring size decrease implies that a rendezvous has occurred. Consider the first waiter \( w \) that tries to reallocate \( R \). Then \( w \) observes \( \text{size}(R) \) occupied nodes, i.e., it observes another waiter \( w' \) in two different nodes. \( w' \) leaves one node and returns to capture another node due to one of the following reasons: a mode switch (impossible), a ring reallocation (impossible, since \( w \) is the first to reallocate the ring), or because \( w' \) successfully rendezvous, a contradiction.
4.5 Evaluation

We study the performance of our algorithms on a number of rendezvous workloads. We compare them to Java’s unfair synchronous queue (or channel) (JDK), ED tree, and FC synchronous channels. We also compare our algorithms to a rendezvous implementation based on Java’s Exchanger algorithm [72, 82]. The Exchanger is an adaptively sized collision array whose design is inspired by the collision arrays used for elimination in stacks and queues [65, 95], but with an optimized implementation as appropriate for a package in Java’s concurrency library. As in the original collision array, the Exchanger is symmetric and any two arriving threads can be matched. Therefore, our Exchanger-based rendezvous implements the put() and get() operations by having the thread repeatedly invoke the Exchanger until obtaining a match with an opposite operation. Since threads may continuously obtain false matches, this algorithm is not a practical approach for implementing rendezvous. However, its similar characteristics to our algorithm make it an interesting yardstick to compare to.

**Experimental setup:** Evaluation is carried out on both the Niagara and Westmere machines. Both Java and C++ implementations are tested. For the prior algorithms, we use the original authors’ implementation. Following Hendler et al. [64], we test both JDK and JDK-no-park, a version that always uses busy waiting instead of yielding the CPU (parking), and report the results of the variant that performs best in each workload.

Unless stated otherwise our benchmarks use dedicated threads, each performing only put() or get() operations in a loop over a period of 10 seconds, and results are averages of ten such runs of the Java implementation on an idle machine. Except when stated otherwise, our results had very little variance. Throughout the evaluation we use a modulo hash function ($hash(t) = t \mod \text{ringsize}$) in AdaptiveAR and the Exchanger to map thread ids to ring nodes. Thread ids are assigned uniformly at random at the start of each run, to model the fact that in practice little is known about which threads try to rendezvous at any point in time. The adaptivity thresholds we use for both AdaptiveAR and TripleAdp are $T_i = 1$, $T_d = 2$, $T_w = 64$. In addition, we set $T_f = 10$ for TripleAdp.

---

5Java benchmarks were ran with HotSpot Server JVM, build 1.7.0_05-b05. C++ benchmarks were compiled with Sun C++ 5.9 on the Niagara machine and with gcc 4.3.3 (-O3 optimization setting) on the Westmere machine. In the C++ experiments we used the Hoard 3.8 [30] memory allocator.

6We remove all statistics counting from the code and use the latest JVM. Thus, the results we report are usually slightly better than those reported in the original papers. On the other hand, we fixed a bug in the benchmark of [64] that miscounted timed-out operations of the Java channel as successful operations; thus the results we report for it are sometimes lower.
Figure 4.7: Rendezvousing between producer/consumer pairs: Niagara results. L2 misses are not shown; all algorithms but JDK had less than one L2 miss/operation on average.

**Thread placement:** Unless stated otherwise, we ran the experiments without pinning threads to specific cores.

### 4.5.1 Symmetric producer/consumer workload

We measure the throughput at which data is transferred from $N$ producers to $N$ consumers. Figure 4.7a shows the results from running on a single multi-core chip of the Niagara machine. Only AdaptiveAR, TripleAdp, Exchanger and Parallel FC show meaningful scalability. In low concurrency settings AdaptiveAR outperforms the Parallel FC channel by $2 \times - 3 \times$ and the Exchanger by $2 \times$. At high thread counts AdaptiveAR obtains 5.5 times the throughput of Parallel FC and 2.7 times that of the Exchanger. TripleAdp falls below AdaptiveAR by 20% for all thread counts (this difference becomes noticeable in the graphs as the thread count increases). Despite this, TripleAdp outperforms Parallel FC by $4.45 \times$ and the Exchanger by $2.3 \times$.

The remaining algorithms scale poorly or not at all. Hardware performance counter data
(Figures 4.7b-4.7d) shows that these implementations exhibit some kind of serialization: the number of instructions per rendezvous for JDK and Single FC increases as concurrency grows; for the ED tree, the cache miss plot shows a growing trend (Figure 4.7d). Taking JDK for example, as concurrency grows a thread requires more attempts until its CAS to the top of the stack succeeds (Figure 4.7c) which causes the overall instruction count to grow too. Consequently, as concurrency increases JDK’s throughput deteriorates while AdaptiveAR’s increases, resulting in around $120 \times$ throughput difference at 64 threads (32 producer/consumer pairs).

Turning back to the more scalable algorithms, we find their throughput differences correlated with the implementations’ path length, i.e., the average number of instructions per successful rendezvous (Figure 4.7b). The path length is crucial on the Niagara, which is an in-order processor. AdaptiveAR completes a rendezvous in less than 170 instructions of which one is a CAS, and this number remains stable as concurrency grows. Compared to AdaptiveAR, TripleAdp performs extra work in each rendezvous when checking for mode switches and ring reallocations. This extra work is wasted in this workload, in which mode switches and ring reallocations are rare, leading to the 20% throughput drop. In comparison, while Parallel FC hardly performs CASes (Figure 4.7c) the time spent waiting for the combiner adds up and Parallel FC requires $2.6 \times$ to $5.8 \times$ more instructions than AdaptiveAR to complete a rendezvous. Finally, an Exchanger rendezvous takes $1.5 \times$ to $2.74 \times$ more instructions. The main reason is false matches which occur with probability $1/2$, so on average two exchanges are required to complete a rendezvous. The remaining throughput gap is due to other algorithmic differences between AdaptiveAR and the Exchanger, such as peeking, which we expand upon in Section 4.5.1.4.

The throughput results on the Westmere processor (Figure 4.8a) are similar to the low concurrency Niagara results for all algorithms but AdaptiveAR and TripleAdp, which now obtain comparable throughput to the Exchanger. Here the average results, shown in Figure 4.8a, do not tell the whole story. Figure 4.8b, which includes more details about the results, shows that AdaptiveAR’s throughput is much more stable than the Exchanger. Still, why does the Exchanger surpass AdaptiveAR in some executions?

It turns out that on this processor and workload the Exchanger’s symmetric behavior can be advantageous due to the write-back caching architecture. In the Exchanger algorithm a thread captures a slot by pointing that slot’s pointer field (using CAS) to a private memory location on which the thread spins. A thread finding an occupied slot tries to nullify the pointer using CAS and, if successful, to rendezvous at the private location. Now, consider a *stable* producer/con-
4.5. EVALUATION

Figure 4.8: Rendezvousing between producer/consumer pairs: Westmere results. Box plot on the right zooms in on 10 : 10 results. The bottom and top of a box are the 25th and 75th percentiles of the data, with a line at the median and a marker at the average. Whiskers extend to the minimum and maximum of the data.

Consumer pair, $p$ and $c$, whose ids hash to the same array slot $s$. Because thread ids remain constant throughout the run, this pair is likely to repeatedly rendezvous at $s$. As we now explain, this results in $s$ moving between these threads’ caches, allowing them to complete a rendezvous more efficiently. When an Exchanger match occurs at $s$, the thread that arrives later performs a CAS on $s$ and leaves shortly after, whereas the thread that occupies $s$ can only leave after noticing that a match has occurred in its private location. Thus the thread that arrives last at $s$, say $p$, will return first and when it does its cache will have exclusive access to $s$’s cacheline. This enables it to immediately capture $s$; when its partner $c$ shows up shortly after it will see $s$ occupied and complete the match. Thus the Exchanger’s symmetry allows a stable thread pair $p$ and $c$ to repeatedly change their roles in the collision array. In different executions there are different number of stable pairs, hence the Exchanger’s erratic performance.

In AdaptiveAR, however, thread roles are fixed. So while in a stable pair the consumer $c$ always writes to the slot $s$ last, by then its partner $p$ has already returned for another rendezvous. When $p$ reads $s$, $c$’s cache loses exclusive access to $s$, which means that when $c$ returns and tries to CAS $s$, a cache miss will occur. Furthermore, $p$ which sees that $s$ is not occupied may proceed to other nodes in the ring, experiencing cache misses as a result and causing cache misses for threads whose ids map to these nodes. Because this workload is symmetric, Triple-Adp essentially fixes the thread roles and thus behaves similarly to AdaptiveAR.

Figures 4.9a-4.9b demonstrate this effect. In Figure 4.9a we additionally plot the results of running the benchmark with an artificial sequential ids distribution that creates $N$ stable pairs. Thus the Exchanger no longer wastes work on false matches, resulting in a $2.4 \times$ reduction.
CHAPTER 4. REALIZING THE PARALLELISM OF RENDEZVOUSING

Figure 4.9: Effect of initial slot selection on $N:N$ workload throughput. Performance counter plots are logarithmic scale. There were no L3 (shared) cache misses on the Westmere processor.

of instructions spent per operation. Every match is also more efficient due to the previously described effect, yielding an $2.6 \times$ reduction in cache misses and almost $3 \times$ better throughput. With AdaptiveAR and TripleAdp, however, there is no improvement in throughput. The Niagara (Figure 4.9c) has a write-through cache architecture in which all CAS operations are applied in the shared L2 cache and invalidate the L1 cacheline. There is therefore no advantage for being the last thread to CAS a slot. The Exchanger’s only benefit from the elimination of false matches is in the amount of work done, yielding a $1.6 \times$ throughput improvement. Similarly, AdaptiveAR’s and TripleAdp’s throughput increases by 12% – 14% because consumers capture slots immediately without contention and producers find them faster.

**Varying arrival rate:** How important are the effects described above in practice, where threads do some work between rendezvous? To study this question we measure how the throughput of the producer/consumer pairs workload is affected when the thread arrival rate decreases due to increasingly larger amounts of time spent doing “work” before each rendezvous.
4.5. EVALUATION

Figures 4.10a and 4.10b show that as the work period grows the throughput of all algorithms that exhibit scaling deteriorates, due to the increase in the sequential part of the workload at the expense of the parallel time. Going from no work to 1.5μs of work, Parallel FC on the Niagara degrades by 2×, the Exchanger by 2.5×, and AdaptiveAR and TripleAdp degrades by about 3× (because they start much higher). Still, the Niagara has sufficient parallelism to allow AdaptiveAR and TripleAdp to outperform the other implementations by at least 2×. On the Westmere EX even a minimal delay between rendezvousing diminishes the impact that stable pairs have on the Exchanger’s throughput, and as the amount of work increases AdaptiveAR’s and TripleAdp’s throughput increases up to 1.30× that of the Exchanger.

4.5.1.1 Work uniformity

One clear advantage of AdaptiveAR over FC is work uniformity, since the combiner in FC spends time doing work for other threads at the expense of its own work. We show this by comparing the percent of total operations performed by each thread in a multiple producer/multiple consumer workload. In addition to measuring uniformity, this test is a yardstick for progress in practice: if a thread starves we will see it as performing very little work compared to other threads. In Figure 4.11 we pick the best result from five executions with maximum concurrency and plot the percent of total operations performed by each thread.

On the Westmere the ideal uniform behavior is for each thread to perform 5% (1/20) of the work. AdaptiveAR (Figure 4.11a) and TripleAdp (Figure 4.11b) are relatively close to uniform, with the highest performing thread completing about 1.28× (1.31× for TripleAdp) the operations as the lowest performer. However, the other algorithms show non-uniform behavior.
Figure 4.11: Percent of operations performed by each thread in an $N : N$ test with maximum concurrency. The horizontal line marks ideal behavior.

Parallel FC (Figure 4.11c) has a best/worst ratio of $30 \times$, showing the impact that performing combining has on the combiner. On the Niagara the results are mostly similar and we therefore omit them.

Figure 4.12: $N : N$ rendezvousing on both processors of the Niagara machine.
4.5. EVALUATION

4.5.1.2 NUMA (multi-processor) architecture

When utilizing both processors of the Niagara machine the operating system’s default scheduling policy is to place threads round-robin on both chips. Thus, the cross-chip overhead is noticeable even at low thread counts, as Figure 4.12 shows. Since the threads no longer share a single L2 cache, they experience an increased number of L1 and L2 cache misses; each such miss is expensive, requiring coherency protocol traffic to the remote chip. The effect is catastrophic for serializing algorithms; for example, the Java channel’s throughput at 128 threads is $5 \times$ worse than its throughput with 64 threads running on a single chip.

Despite the cross-chip communication overhead, the algorithms with a scalable distributed design, namely AdaptiveAR, TripleAdp, Parallel FC and the Exchanger, show scaling trends similar to the single chip ones but with overall lower throughput. AdaptiveAR’s NUMA throughput with 32 producer/consumer pairs is about half of its single-chip 32 pair throughput, TripleAdp’s drops by 67%, the Exchanger by 30% and Parallel FC by 10%.

We believe that the throughput of all the scalable algorithms can be improved using NUMA-aware scheduling to minimize cross-chip communications; we leave this for future work.

4.5.1.3 Bursts

To evaluate the effectiveness of our concurrency level adaptivity technique, we measure the rendezvous rate on the Niagara machine in a workload that experiences bursts of concurrency changes. For 10 seconds the workload alternates every second between 31 thread pairs and 8 pairs. The 63rd hardware strand is used to take ring size measurements. This sampling thread

![Diagram](image-url)

Figure 4.13: Throughput of synchronous queue bursty workload. AdaptiveAR’s ring size is sampled continuously using a thread that does not participate in the rendezvousing.
continuously reads the ring size and records the time whenever the current read differs from the previous read. Figure 4.13a shows the throughput results; here AdaptiveAR obtains $2.62 \times$ the throughput of the Exchanger, $5 \times$ that of Parallel FC and $40 \times$ that of JDK. TripleAdp lags behind AdaptiveAR by 20%. Figure 4.13b depicts how AdaptiveAR successfully adapts its ring size as the concurrency level changes over time. The results for TripleAdp are essentially identical, so we omit them to keep the graph readable.

### 4.5.1.4 AdaptiveAR performance breakdown

In this section (and Figure 4.14) we quantify how much each of the techniques used in AdaptiveAR improves performance. For the purpose of this test (and only in this section), we start with the Exchanger-based collision array implementation, referred to as Baseline, and introduce our techniques one by one (each on top of all the previous ones), eventually obtaining a new implementation of AdaptiveAR from a different code base (as opposed to our from-scratch implementation of AdaptiveAR). This methodology also provides further assurance that AdaptiveAR...
4.5. EVALUATION

tiveAR’s performance advantages are due to our techniques and not implementation artifacts. To focus solely on rendezvousing performance we disable the algorithms’ adaptivity schemes and use an array/ring whose size equals the number of producer/consumer pairs taking part in the test. Our modifications to Baseline yield the following versions:

- **Asymmetrical**: Distinguishes between producers and consumers. A consumer thread attempts to occupy a slot and then waits for a partner. A producer thread searches the array for a waiting consumer.

- **Exchange in array**: In the Exchanger algorithm a thread captures a slot using CAS to change a pointer field in the slot to point to a private memory location on which the thread spins. A thread finding an occupied slot attempts to nullify this pointer using a CAS and, if successful, to rendezvous at the private location. Here we change the protocol so that rendezvous occurs in the array slot.

- **Ring**: In the Exchanger a thread maintains an index to traverse the array. Here we turn the array into a ring and traverse it using pointers.

- **Peeking**: Implements peeking (Section 4.3.2). This version is algorithmically identical to AR, our from-scratch AdaptiveAR implementation with adaptivity disabled as in Section 4.3.2.

Figure 4.14a shows the throughput results on the Niagara. Asymmetrical obliterates false matches as evidenced by the reduced instruction count shown in Figure 4.14b. As a result, throughput increases by $2 \times$. Exchange in the array adds up to 30% by simplifying the code and reducing the number of CASes. Traversing pointers (Ring) saves a conditional on each iteration and yields a 5% gain. Peeking gives a final 20% boost. On the Westmere (Figure 4.14c) processor Asymmetrical serves to stabilize the algorithm’s performance (Figure 4.14d). Because the processor is an out-of-order processor with $2 \times$ the clock rate of the Niagara the cost of cache misses dominates and the remaining techniques have less impact.

4.5.2 Asymmetric workloads

The throughput of one producer rendezvousing with a varying number of consumers is presented in Figures 4.15a (Niagara) and 4.15b (Westmere). We focus on the Niagara results as the results from the Westmere processor are similar. Since the throughput is bounded by the rate of a single producer, little scaling is expected and observed. However, for AdaptiveAR
and TripleAdp it takes several consumers to keep up with a single producer. This is because the producer hands off an item and completes, whereas the consumer needs to notice the rendezvous has occurred. And while the single consumer is thus busy the producer cannot make progress on a new rendezvous. However, when more consumers are active, the producer can immediately return and find another (different) consumer ready to rendezvous with. Figure 4.15a shows that most algorithms do not sustain their peak throughput. Exchanger’s throughput degrades by 50% from its peak to 32 consumers; on the Westmere the degradation is a much more severe $2.8 \times$. The FC channels have the worst degradation on the Niagara: $4 \times$ for Single FC and $1.4 \times$ for Parallel FC. JDK’s degradation is minimal (20% from peak to 32 consumers), and it along with the ED tree achieves close to peak throughput even at high thread counts. Yet this throughput is low: AdaptiveAR outperforms JDK by up to $3.6 \times$ and the ED tree by $5 \times$ to $8 \times$ despite degrading by 33% from peak throughput. As in the $N:N$ workload, TripleAdp pays a 20% throughput penalty compared to AdaptiveAR on the Niagara and achieves compa-
Figure 4.17: Asymmetric rendezvousing (single producer and \( N \) consumers, single consumer and \( N \) producers) with OS constrained to place the single producer (or consumer) alone on a core, so that threads performing the reverse operation do not interfere with its execution.

rable performance on the Westmere processor. Both TripleAdp and AdaptiveAR maintain peak throughput up to 16 threads before experiencing some degradation.

Why do AdaptiveAR and TripleAdp degrade at all? After all, the producer has its pick of consumers in the ring and should be able to complete a hand-off immediately. The reason for this degradation is not algorithmic but due to contention on chip resources. A Niagara II core has two pipelines, each shared by four hardware strands. Thus, beyond 16 threads some threads must share a pipeline – and our algorithm indeed starts to degrade at 16 threads. To prove that this is the problem, we present in Figure 4.17a results when the producer runs on the first core and consumers are scheduled only on the remaining seven cores. While the trends of the other algorithms are unaffected, our algorithms now maintain peak throughput through all consumer counts.

Results from the opposite workload, where multiple producers try to serve a single consumer, are given in Figures 4.16a (Niagara) and 4.16b (Westmere). Here the producers contend over the single consumer node and as a result AdaptiveAR’s throughput degrades as the number of producers increases, as do the Exchanger and FC channels. Despite this degradation, on the Niagara AdaptiveAR outperforms the FC channels by \( 2 \times -2.5 \times \) and outperforms the Exchanger up to 8 producers and the JDK up to 48 producers (falling behind by 35% and 15%, respectively, at 32 producers).

The advantage of TripleAdp becomes apparent in this workload. TripleAdp maintains its peak throughput (modulo chip resource contention, as Figure 4.17b shows), outperforming all other algorithms by at least \( 2 \times \) on both the Niagara and Westmere architectures. The 20%
penalty in the $1:N$ workload was for this benefit.

### 4.5.3 Oversubscribed workloads

Here we explore the algorithms’ performance in oversubscribed scenarios, in which the number of threads exceeds the number of hardware execution strands and forces the operating system to context switch between the threads.

Figure 4.18 shows the results when the number of producers in the system equals the number of consumers. For reference, the first point in the plots shows the performance when the number of threads equals the number of hardware threads. The JDK and ED tree show little degradation from this point. Both FC variants experience a sharp drop in throughput as concurrency grows because the odds for the combiner threads being descheduled by the operating system and causing the algorithm to block increase. On the Niagara, AdaptiveAR’s and TripleAdp’s throughput degrades from the starting point with no oversubscription ($2.3 \times$ for AdaptiveAR and $1.67 \times$ for TripleAdp).

This degradation occurs because when the operating system switches out a thread waiting at some ring node, that node remains captured until the thread runs again. Even if a match occurs in the mean time, that thread must wake up to mark the node as FREE. The Exchanger does not show a similar degradation because its exchanges are done outside of the nodes. Recall that in the Exchanger a captured node points to a private memory location where the waiting thread spins. A thread encountering a captured node nullifies the node pointer using CAS before proceeding to rendezvous at the private location. Thus, a captured Exchanger node can be freed.

![Oversubscription throughput graphs](image)

Figure 4.18: Oversubscribed (more threads than available hardware threads) symmetric workload. Total number of threads grows, starting from the number of available threads in the hardware.
Figure 4.19: Oversubscribed (more threads than available hardware threads) asymmetric workloads. Total number of threads grows, starting from the number of available threads in the hardware.

Figure 4.20: Throughput of synchronous queue bursty $N : N$ workload in an oversubscribed setting. Workload alternates every second between 512 threads (256 pairs) and 16 threads (8 pairs).
even if its waiting thread is not running.

Despite their throughput degradation, AdaptiveAR and TripleAdp obtain the best throughput results and outperform the Exchanger, which is the next best performer, by $1.55 \times$. On the Westmere the throughput of AdaptiveAR, TripleAdp and the Exchanger remain stable, with the Exchanger outperforming AdaptiveAR by 56% and TripleAdp by 26%.

Figure 4.19 depicts throughput results for asymmetric oversubscribed workloads, where the threads performing one type of operation outnumber the threads invoking the reverse operation by a factor of three. The performance trends follow those of the $N:N$ case, with the exception that TripleAdp outperforms AdaptiveAR on the Westmere by adapting its ring structure to the scenario with more consumers than producers.

Finally, Figure 4.20 shows the throughput results in an oversubscribed bursty workload on the Niagara machine. For 10 seconds the workload alternates every second between 256 thread pairs and 8 pairs. TripleAdp, AdaptiveAR and the Exchanger are the best performers, achieving orders of magnitude better throughput than the other algorithms. TripleAdp outperforms the Exchanger by $1.88 \times$ and AdaptiveAR by 17%. TripleAdp outperforms AdaptiveAR because TripleAdp maximal ring size is not bounded, allowing its ring to grow and adapt itself to the high concurrency level in this test.

### 4.6 Concurrent stack

Hendler et al. improved the performance and scalability of a Treiber-style lock-free stack [117] by adding an elimination layer as a backoff scheme for the lock-free stack [65]. In their algorithm an arriving thread that fails to complete its operation (due to contention in the main stack) enters a collision array where it hopes to collide with an operation of the opposite type. If such a match occurs, the resulting \texttt{Push()} / \texttt{Pop()} operation pair can be linearized next to each other without affecting the state of the stack. Thus, the collision array implements a form of asymmetric rendezvous.

In this section we use AdaptiveAR as an elimination layer on top of a Treiber-style stack. This requires extending AdaptiveAR with support for \textit{aborts}: the ability to give up on a pending rendezvous operation after a certain period of time. Without aborts a single thread might remain in the elimination layer forever, making the resulting stack algorithm blocking.

\textbf{Abort semantics:} We change the specification of the handoff object as follows. Both \texttt{put()} and \texttt{get()} operations can nondeterministically return a reserved value $\bot$ indicating an aborted
operation, in which case the state of the object is not changed. Note that since there is no
notion of time in our model, we do not define when it is considered legal to timeout and abort
a rendezvous operation. This allows for trivial rendezvous implementations that always return
⊥, though they would be of little use in practice.

Abort implementation: We add a parameter to the algorithm specifying the desired time-
out. Timeout expiry is then checked in each iteration of the main loop in put(), get() and
findFreeNode() (Figure 4.2c). If the timeout expires, a producer or a consumer in
findFreeNode() aborts by returning ⊥. A consumer that has captured a ring node cannot
abort before freeing the node by CASing its item from NULL back to FREE. If the CAS fails,
the consumer has found a match and its rendezvous has completed successfully. Otherwise its
abort attempt succeeded and it returns ⊥.

We consider two variants of an elimination stack, where we abstract the elimination task to
an abortable rendezvous object. The first variant (Figure 4.21a) uses elimination as a backoff
scheme: threads try to eliminate upon detecting contention on the main stack. (This is essen-

---

```plaintext
shared objects:
rendezvous : either our algorithm or a collision array

push(item x) {
    while (true) {
        // Perform Treiber stack code first
        if (successfully CAS x to head of linked list )
            return
        // Fall back to elimination
        if (rendezvous.put(x) != ⊥ )
            return
    }
}

pop() {
    while (true) {
        // Perform Treiber stack code first
        return EMPTY
        if (successfully remove x, the head of list , using CAS)
            return x
        // Fall back to elimination
        x := rendezvous.get()
        if (x != ⊥ )
            return x
        // Fall back to Treiber stack
        return x
    }
}

(a) Backoff elimination stack [65]

shared objects:
rendezvous : either our algorithm or a collision array

push(item x) {
    while (true) {
        // Try to eliminate first
        if (rendezvous.put(x) != ⊥ )
            return
        // Fall back to Treiber stack
        return
    }
}

pop() {
    while (true) {
        // Try to eliminate first
        x := rendezvous.get()
        if (x != ⊥ )
            return x
        return EMPTY
        if ( successfully remove x, the head of list , using CAS)
            return x
    }
}

(b) Optimistic elimination stack
```

Figure 4.21: Elimination stack variants. We omit the linked list manipulation details of the
Treiber lock-free stack [117].
CHAPTER 4. REALIZING THE PARALLELISM OF RENDEZVOUSING

Originally Hendler et al.’s elimination stack [65].) The second variant (Figure 4.21b) is optimistic: a thread enters the elimination layer first, accessing the main stack only if it fails to find a match. If it then encounters contention on the main stack it goes back to try the rendezvous, and so on.

We implemented these variants in C++ within the benchmark framework of Hendler et al. [63]. We tested each variant with two algorithms for performing elimination: AdaptiveAR and a collision array. Since code for the original collision array implementation [65] is not available we chose to port Java’s optimized Exchanger to C++ for use as the collision array. We set AdaptiveAR’s wait threshold to a value smaller than the abort timeout, for otherwise ring size decreases never occur as the thread always aborts first.

We compared these implementations (referred to as Backoff elim (AdaptiveAR/Exchanger) and Optimistic elim (AdaptiveAR/Exchanger) in the plots) to the implementations of Treiber’s lock-free stack (Lock-free) and the FC based stack [63] (Flat combining) in the framework.

We report results of a benchmark measuring the throughput of an even push/pop operation mix averaged over 10 runs. Each test consists of threads repeatedly invoking stack operations over a period of ten seconds. Each thread performs both push() and pop() operations; this is the same benchmark used in [63]. In the Westmere test each thread performs a small random number (≤ 64) of empty loop iterations between completing one stack operation and invoking the next one. We found this delay necessary to avoid long runs [92] where a thread holding the top of the stack in its cache quickly performs a long sequence of operations, leading to unrealistically high throughput. On the Niagara the long run pathology does not occur because

---

Figure 4.22: Stack throughput. Each thread performs both push and pop operations with probability 1/2 for each operation type.

---

7 We reduced the overhead due to memory allocation in the original implementations [63] by caching objects popped from the stack and using them in future push operations.
all CASes are performed in the shared L2 cache, so we did not add delays in the Niagara test.

Figure 4.22 shows the results of the benchmark. The backoff variants fail to scale beyond a certain level of concurrency in all workloads and machine combinations, illustrating the price paid for accessing the main stack: both the overhead of allocating a node to push on the stack as well as trying (and failing) to CAS the stack’s top to point to the node. However, AdaptiveAR’s backoff stack achieves $3.5 \times$ the throughput of the Exchanger backoff stack on the Niagara by avoiding false matches.

The lock-free stack has no elimination to fall back on and does not scale at all. Consequently, the optimistic AdaptiveAR elimination stack outperforms it by $33 \times$ on the Niagara and by $4.8 \times$ on the Westmere. However, the optimistic algorithms achieve this scalability at the expense of lower throughput under low thread counts. The backoff AdaptiveAR elimination stack achieves $3 \times$ the throughput of the optimistic AdaptiveAR variant with 2 threads on the Westmere, and $1.65 \times$ on the Niagara.

The performance trends of the optimistic elimination stacks match those observed in Section 4.5 on the Niagara. The optimistic AdaptiveAR elimination stack obtains the highest throughput. It outperforms the backoff Exchanger elimination stack by $7.5 \times$, the optimistic Exchanger elimination stack by $2.1 \times$, and the FC stack by $3.6 \times$. On the Westmere, the optimistic Exchanger and AdaptiveAR elimination stacks show different scaling trends. The optimistic Exchanger stack is the best performer at 8 to 16 threads, however its throughput flattens beyond 12 threads. In contrast, the optimistic AdaptiveAR stack scales up to 20 threads, where
it outperforms the Exchanger stack by 1.25×.

This behavior occurs because at low concurrency levels, entering the collision array and failing to eliminate provides sufficient backoff to allow threads to complete their operations quickly. We show this by plotting the number of optimistic stack operations that complete through elimination in this benchmark (Figure 4.23a). Even as concurrency increases, the Exchanger elimination stack completes about 40% – 50% of the operations on the main stack. The reason, as Figure 4.23b shows, is that half of the Exchanger matches are false. Threads that fail to eliminate go back to the stack and – because concurrency is low – often manage to quickly complete their operation there. In contrast, AdaptiveAR obtains almost 100% elimination success rate. It pays a price in the low concurrency levels, where threads wait longer until a partner comes along, leading to lower throughput.
Chapter 5

Fast FIFO queues from contended fetch-and-add

This chapter describes our LCRQ algorithm. An LCRQ algorithm is essentially a Michael and Scott linked list queue [92] in which a node is a concurrent ring (cyclic array) queue, CRQ for short. A CRQ that fills up becomes closed to further enqueues, who instead append a new CRQ to the list and begin working in it. Most of the activity in the LCRQ occurs in the individual CRQs, making contention on the list’s head and tail a non-issue.

A major part of our contribution is the design of the CRQ ring queues. Within a CRQ, the head and tail are F&A objects which are used to spread threads around the slots of the ring, where they synchronize using (uncontended in the common case) CAS. One of the CRQ’s distinctive properties compared to prior concurrent circular array queues [31, 32, 43, 56, 61, 111, 118] is that in the common case an operation accesses only the CRQ’s head or tail but not both. This reduces the CRQ’s synchronization cost by a factor of two, since the contended head and tail are the algorithm’s bottleneck.

This chapter starts by defining a variant of the nonblocking progress guarantee which is more suitable for queues (Section 5.1). Section 5.2 then describes the LCRQ and CRQ, and Section 5.3 contains the performance evaluation. We compare LCRQ’s design to prior queue implementations in Section 5.4.

5.1 Op-wise nonblocking progress

The standard definition of nonblocking progress (Section 2.5) guarantees that some thread completes an operation in a finite number of steps. In other words, an individual operation may
starve, but some operation always makes progress. This guarantee still allows some undesirable scenarios for queues, e.g., an execution in which enqueuers are starved by dequeuers returning EMPTY.

Nonblocking queues in the literature [43, 92, 111, 118] actually provide a stronger guarantee, which we call op-wise nonblocking: some enqueue() completes in a finite number of steps by enqueuing threads, and some dequeue() completes in a finite number of steps by dequeuing threads.

Achieving the op-wise nonblocking progress guarantee is our goal when designing the LCRQ algorithm.

5.2 The LCRQ algorithm

LCRQ can be viewed as a practical realization of the following simple but unrealistic queue algorithm (Figure 5.1). The algorithm represents the queue using an infinite array, $Q$, with (unbounded) head and tail indices that identify the part of $Q$ which may contain items. Initially, each cell $Q[i]$ is empty and contains a reserved value $\bot$ that may not be enqueued.

An enqueue(x) operation obtains a cell index $t$ via a F&A on tail. The enqueue then atomically swaps the value in $Q[t]$ with $x$. If the swap returns $\bot$, the enqueue operation completes; otherwise, it repeats this process.

A dequeue, $D$, obtains a cell index $h$ using F&A on head and atomically swaps the value in $Q[h]$ with another reserved value $\top$. If $Q[h]$ contained some $x \neq \bot$, $D$ returns $x$. If $D$ finds $\bot$ in $Q[h]$, the fact that $D$ stored $\top$ in the cell guarantees that an enqueue operation which later stores an item in $Q[h]$ will not complete. $D$ then returns EMPTY if $\text{tail} \leq h + 1$ ($h + 1$ is the value of head following $D$’s F&A). If $D$ cannot return EMPTY, it repeats this process.

While this algorithm is a linearizable FIFO queue it has two major flaws that prevent it from being relevant in practice: using an infinite array and susceptibility to livelock (a dequeuer

---

1We are not aware of this property being explicitly pointed out before.

2We omit the full proof, which is similar to the proof in Section 5.2.1.2.
The LCRQ algorithm builds on CRQ to prevent the livelock problem. We represent the queue as a linked list of CRQs. An enqueue(x) operation failing to make progress in the tail CRQ closes it to further enqueues. Upon noticing the tail CRQ is closed, each enqueuer tries to append a new CRQ, initialized to contain its item, to the list. One enqueuer succeeds and completes; the rest move into the new tail CRQ, leaving the old tail CRQ with only dequeuers inside it, which allows the dequeuers to complete. The LCRQ is thus op-wise nonblocking.

5.2.1 The CRQ algorithm

The pseudocode of the basic CRQ algorithm appears in Figure 6.1. The CRQ represents the queue as a ring (cyclic array) of R nodes, with 64-bit head and tail indices (Figure 5.2a). An index with value i points to node i mod R, which we denote by node(i). We reserve the most significant bit of tail to denote the CRQ’s CLOSED state. We thus make the realistic assumption that both head and tail do not exceed $2^{63}$.

The synchronization protocol in a CRQ ring node needs to handle more cases than the infinite array queue, which only needs to distinguish whether an enqueue or dequeue arrives first at the node. We proceed to describe this protocol and how it handles these cases.
\textbf{CHAPTER 5. FAST FIFO QUEUES FROM CONTENDED FETCH-AND-ADD}

\begin{verbatim}
struct Node {
  safe : 1 bit (boolean)
  idx : 63 bits (int)
  val : 64 bits (int or pointer)
  // padded to cache line size
}

struct CRQ { // fields are on distinct cache lines
  head : 64 bit int
  tail : struct { closed : 1 bit, t : 63 bits }
  next : pointer to CRQ, initially null
  ring : array of \( R \) Nodes, initially node \( u = <1, u, \perp> \)
}

(a) Globals

\begin{verbatim}
defqueue(crq : pointer to CRQ) {
  // local variables
  val, idx : 64 bit int
  h, t : 64 bit int
  node : pointer to Node
  closed : boolean
  safe : boolean

  while (true) {
    h := F&A(&crq.head, 1)
    node := &crq.array [h mod R]
    while (true) {
      val := node.val
      \(<\text{safe}, \text{idx}> := <\text{node.safe}, \text{node.idx}> // 64-bit read
      if (val \neq \perp) goto Line 85
      \} else {
        if (idx > h) goto Line 85
        if (val \neq \perp) {
          // try dequeue transition
          if (CAS2(node, <\text{safe}, \text{idx}, val>),
            <\text{safe}, \text{idx} + R, \perp>)) return val
        } else {
          // mark node unsafe to prevent future enqueue
          if (CAS2(node, <\text{safe}, \text{idx}, \perp>),
            <0, \text{idx}, val>)) goto Line 85
        }
      }
      // end of while loop, go back to Line 63
      if (t + R \geq h+1) { fixState (crq) return EMPTY }
    }
  }
}

(b) Dequeue

\begin{verbatim}
while (true) {
  h := F&A(&crq.head, 1)
  node := &crq.array [h mod R]
  while (true) {
    if (CAS(&crq.tail, \perp)) return CLOSED
    node := &crq.array [t mod R]
    val := node.val
    \(<\text{closed}, t> := F&A(&crq.tail, 1)
    // (the F&A is on all 64 bits of tail)
    if (closed) return CLOSED
    node := &crq.array [t mod R]
    \(<\text{closed}, t> := F&A(&crq.tail, 1)
    // atomically set tail\times closed to true
    if (val <= t) and (safe = 1 or \text{crq.head} \leq t)
      CAS2(node, <\text{closed}, \perp>),
      <1, t, \text{arg}>)) {
      return OK
      }
  }
  \text{fixState} (crq)
  return EMPTY
}

(d) Enqueue
\end{verbatim}
\end{verbatim}

Figure 5.2: Pseudocode of CRQ algorithm.
5.2. THE LCRQ ALGORITHM

Node structure (Figure 5.2a) Physically, a ring node contains two 64-bit words. Logically, a ring node is a 3-tuple \((s, i, v)\) consisting of (1) a safe bit \(s\) (used by a dequeuer to notify the matching enqueuer that storing an item in the node is unsafe as the dequeuer will not be around to dequeue it; we explain the details below), (2) an index \(i\), and (3) a value \(v\). Initially, node \(u\)'s state is \((1, u, \perp)\) for every \(0 \leq u < R\).

A node can be in one of two states: if its value is \(\perp\) the node is empty; otherwise the node is occupied. A CRQ operation attempts to transition a node from empty to occupied or vice versa using CAS2. We say that an operation, \(op\), accesses node \(u\) using index \(i\) if \(op\)'s F&A returns \(i\) and \(u = i \mod R\), and refer to the operation as \(enq_i\) or \(deq_i\) as appropriate. An operation accessing a node uses the value of the node’s safe bit and index, as described next, to determine whether it can attempt a transition or should obtain a new index and try accessing another node.

Dequeuing an item When a node is in an occupied state, \((s, i, x)\), it holds the item \(x\) that has been stored by \(enq_i(x)\). In this case, only \(deq_i\), the dequeue operation accessing the node using index \(i – \text{exactly } i\) and not just equal to \(i\) modulo \(R – \text{can return } x\). Such a dequeue attempts to remove \(x\) by performing the transition \((s,i,x) \rightarrow (s,i+R,\perp)\) using CAS2 (Figure 5.2b, Line 71). We refer to this as a dequeue transition.

Dequeueing arrives before enqueuer while a node is empty This case occurs when an empty node whose state is \((s,i,\perp)\) is accessed by \(deq_j\) with \(j = i + kR (k > 0)\), i.e., before the matching enqueue \(enq_j\) completes. Similarly to the infinite array queue, \(deq_j\) tries to prevent \(enq_j\) from storing its item in the node by performing an empty transition \((s,i,\perp) \rightarrow (s,j+R,\perp)\) (Line 80). In fact, the empty transition prevents any operation using some index \(j – kR\) from performing a transition on the node. This stronger property was not needed in the infinite array queue, where only one enqueuer and one dequeuer ever access a node.

Dequeueing arrives before enqueuer while a node is occupied This case has no analog in the infinite array queue. It occurs when a dequeuer \(deq_j\) accesses an occupied node \((s,i,x)\) where \(j > i\), i.e., before \(deq_i\) which is the dequeuer supposed to dequeue \(x\). While \(deq_j\) cannot remove \(x\), before it moves on \(deq_j\) must somehow signal to \(enq_j\) not to enqueue an item in the node even if it finds the node empty in the future. \(deq_j\) uses the safe bit for this purpose, making an unsafe transition \((s,i,x) \rightarrow (0,i,x)\) (Line 76). Once a node is unsafe, all dequeue transitions keep the safe bit at 0. This prevents any enqueuer from storing its item in the node unless it first verifies that the corresponding dequeuer has not yet started, as explained next.
Enqueueing an item  When a node is in an empty state \((s, i, \perp)\), any \(enq_j(x)\) operation with \(j = i + kR\) \((k \geq 0)\) may attempt an enqueue transition to store \(x\) in the node. If \(s = 1\), \(enq_j\) simply performs the transition \((1, i, \perp) \to (1, j, x)\) (Figure 5.2d, Line 128). However, if \(s = 0\), \(enq_j\) needs to make sure that \(deq_j\) is not the dequeuer that set \(s\) to 0, since then \(deq_j\) will not dequeue from the node. \(enq_j\) determines this by checking if \(head \leq j\) (Line 127), which means that \(deq_j\) has not yet started. If so, then \(enq_j\) makes the transition \((0, i, \perp) \to (1, j, x)\) (Line 128) which undoes the previous unsafe transition. If \(deq_j\) then starts after \(enq_j\)’s check that \(head \leq j\) and performs a transition on the node before \(enq_j\)’s transition, then \(deq_j\) performs an empty transition (Line 82) which changes the node’s index and causes \(enq_j\)’s CAS2 to fail.

We now turn to a walk-through of the algorithm’s pseudocode. For simplicity, we describe optimizations in Section 5.2.1.1, omitting them from the pseudocode.

Enqueue (Figure 5.2d)  An enqueue \(enq\) repeats the following. It obtains an index to a ring node with a F&A on \(tail\) (Line 117). If the CRQ is closed, \(enq\) returns CLOSED (Lines 119-120). Otherwise, \(enq\) attempts an enqueue transition (Lines 121-132). If this fails (because the node is occupied or the CAS2 fails), \(enq\) decides to give up and closes the queue in one of two cases: (1) \(enq\)’s index has passed \(head\) by \(R\) places, indicating a possibly full queue, or (2) \(enq\) is failing to make progress for a long time (checked by \(starving()\)) (Lines 133-138).

Dequeue (Figure 5.2b)  A dequeue \(deq\) repeats the following. It obtains an index, \(h\), to a ring node using F&A on \(head\) (Lines 61-62). It then enters a loop in which it attempts to read a consistent state of the node and perform a transition. If \(h < i\), where \(i\) is the node’s index, then \(deq\) has been overtaken between its F&A and reading the node, and so it exits the loop (Line 67). Otherwise, if the node is occupied, \(deq\) attempts a dequeue transition (Lines 68-79). If the node is empty, \(deq\) attempts an empty transition (Line 80) and exits the loop if successful. Throughout this process, if \(deq\)’s CAS2 fails (implying the node’s state changes) then \(deq\) restarts the loop of reading the node and performing a transition. Whenever \(deq\) exits the loop without successfully dequeuing an item, it verifies that the queue is not empty before trying to dequeue with a new index (Line 86-87). If the queue is empty, \(deq\) fixes (see below) the queue’s state so that \(head \leq tail\) before returning EMPTY (Lines 88-89).

Fixing the queue state (Figure 5.2c)  A dequeuer F&A may bring the queue into an invalid state in which \(head > tail\). In such a case, the dequeuer can just perform an empty transition and return EMPTY. However, doing so prevents the enqueuer with the same index from using
the node, forcing it to F&A tail again and increasing contention. To avoid this problem, a dequeuer always verifies that head ≤ tail before returning EMPTY (Lines 95-107).

5.2.1.1 Optimizations

**Bounded waiting for matching enqueues** When an enqueue and dequeue operation using the same index are active concurrently, the dequeue may arrive at the node before the matching enqueuer. Performing an empty transition in such a case just leads to both operations restarting and accessing the F&A again, needlessly increasing contention on head and tail.

To avoid this, before performing an empty transition (Line 80), a dequeue operation checks whether tail ≥ h + 1, where h is the dequeue’s index. If so, then the matching enqueuer is (or was) active and the dequeuer spins for a short while, waiting for the enqueue transition to take place. Only after timing out on this spin loop does the dequeue perform an empty transition.

**Hierarchy awareness** Large servers are typically built hierarchically, with clusters of cores such that inter-core communication inside a cluster is cheap, but cross-cluster communication is expensive. For example, in a multiprocessor system a cluster consists of all the cores on a (multicore) processor. In these hierarchical machines, creating batches of operations that complete on the same cluster without interference from remote clusters reduces synchronization cost.

To achieve this, we add a cluster field to the CRQ, which identifies the current cluster from which most operations should complete. Before starting a CRQ operation, a thread checks if it is running on cluster. If not, the thread waits for a while, and then CASes cluster to be its cluster and enters the algorithm (even if the CAS fails). Similarly to prior NUMA-aware lock-based algorithms [50, 54], this divides the execution into segments such that in each segment most operations in the CRQ are from the same cluster. However, our optimization does not rely on locks nor does it introduce blocking, as every operation eventually enters the CRQ.

5.2.1.2 CRQ linearizability proof

The CRQ is not a standard FIFO queue because an enqueue can return CLOSED. To deal with this we give the CRQ the semantics of a tantrum queue: a queue in which an enqueue can nondeterministically refuse to enqueue its item, returning CLOSED instead and moving the queue to a CLOSED state. When a tantrum queue is in the CLOSED state, every enqueue operation returns CLOSED without enqueuing its item.
In the following, we prove that CRQ is a linearizable tantrum queue. Let $E = e_1, e_2, \ldots$ be a possibly infinite execution of CRQ. We assume every thread whose next local step is to complete does indeed complete in $E$. We denote an operation $op \in \{\text{deq, enq}\}$ that returns $ret$ in $E$ by $\langle op : ret \rangle$. We now describe a procedure $P$ (Figure 5.3) to assign linearization points to the operations in $E$.

Essentially, the linearization order of $\langle \text{enq}(x) : \text{OK} \rangle$ operations is by the index the enqueuer uses when successfully enqueuing its item, and similarly $\langle \text{deq}() : x \rangle$ operations ($x \neq \text{EMPTY}$) are linearized in the order of the index used to dequeue. The trick is to order enqueues and dequeues consistently, since for example a dequeuer’s F&A returning index $i$ can occur before the corresponding enqueuer’s F&A.

To do this, we track the CRQ’s state at each point in $E$ using an auxiliary sequential queue. The auxiliary queue consists of an infinite array, $Q$, coupled with indices $\text{head}(Q)$ and $\text{tail}(Q)$ representing $Q$’s head and tail. (Note that $Q$ is not cyclic.) Initially, $\text{tail}(Q) = \text{head}(Q) = 0$ and $Q[i] = \perp$ for all $i$.

We process the execution one event at a time, in order of execution, but using information about future events to decide when to linearize an operation. When we linearize an operation

```
Input: CRQ execution, $E = e_1, e_2, \ldots$

for $j = 1, \ldots$
if $e_j$ is a T&S by $\langle \text{enq}(x) : \text{CLOSED} \rangle$ that sets tail’s closed bit
   (Figure 5.2d, Line 136) then
   Linearize $\langle \text{enq}(x) : \text{CLOSED} \rangle$ at $e_j$
   else if $e_j = \langle t := \text{F&A}(\text{tail}, 1) \rangle$ by $\langle \text{enq}(x) : \text{CLOSED} \rangle$ which
      returns a closed value (Figure 5.2d, Line 117) then
      Linearize $\langle \text{enq}(x) : \text{CLOSED} \rangle$ at $e_j$
   else if $e_j = \langle t := \text{F&A}(\text{tail}, 1) \rangle$ is the last F&A in $E$ by $\langle \text{enq}(x) : \text{OK} \rangle$
      (Figure 5.2d, Line 117) then
      $Q[t] := x$
      $\text{tail}(Q) := t + 1$
      Linearize $\langle \text{enq}(x) : \text{OK} \rangle$ at $e_j$
   else if $e_j = \langle t := \text{tail} \rangle$, is a read returning $t \leq \text{head}$ by $\langle \text{deq} : \text{EMPTY} \rangle$
      (Figure 5.2b, Line 86) then
      Linearize $\langle \text{deq} : \text{EMPTY} \rangle$ at $e_j$
   endif
while $\text{head}(Q) < \text{tail}(Q)$

$h := \min \{ i \mid Q[i] \neq \perp \}$
if dequeue whose F&A (Line 61) returns $h$ not active in $e_1, \ldots, e_j$ then
   goto Line 167
endif

$x := Q[h]$
$Q[h] := \perp$
$\text{head}(Q) := h + 1$
Linearize $\langle \text{deq} : x \rangle$ at $e_j$
endwhile
```

Figure 5.3: CRQ linearization procedure $P$. Operations linearized at the same event are ordered based on the order of $P$’s steps.
we also apply it to the auxiliary queue. We linearize an \( \langle \text{enq}(x) : \text{OK} \rangle \) on its final F&A, the one returning index \( t \) such that the operation enqueues \( x \) in \( \text{node}(t) \). At this point we also set \( \text{tail}(Q) \) to \( t + 1 \). We linearize the dequeue of item \( x = Q[h] \) as soon as the dequeue becomes active and \( h \) is the lowest indexed non-\( \bot \) cell in \( Q \), and set \( \text{head}(Q) \) to \( h + 1 \) at this point.

We linearize a \( \langle \text{deq} : \text{EMPTY} \rangle \) on its read of \( \text{tail} \) that returns a value \( \leq \text{head} \) (we later show that \( \text{head}(Q) = \text{tail}(Q) \) at this point). The full pseudocode of \( P \) in Figure 5.3 also includes the straightforward cases of linearizing \( \langle \text{enq}(x) : \text{CLOSED} \rangle \) operations.

By construction, the linearization point of an operation is within its execution interval, and all completed enqueues and all dequeues that return \( \text{EMPTY} \) are linearized. We now show that completed dequeues which do not return \( \text{EMPTY} \) are also linearized. Here we denote by \( \text{enq}(x) \) the \( \langle \text{enq}(x) : \text{OK} \rangle \) operation whose last F&A on \( \text{head} \) in \( E \) returns \( i \), causing \( P \) to set \( Q[i] := x \) and linearize it. Similarly, we denote a dequeue operation whose last F&A on \( \text{head} \) in \( E \) returns \( i \) by \( \text{deq}_i \).

**Lemma 15.** Suppose \( P \) linearizes \( \text{enq}_i(x) \). If there exists a dequeue operation \( \text{deq} \) that performs a F&A on \( \text{head} \) in \( E \) which returns \( i \), then: (1) \( \text{deq} = \text{deq}_i \) (i.e., \( \text{deq} \) performs no further F&As in \( E \)), (2) \( \text{deq}_i \) returns \( x \) if it completes, and (3) \( P \) linearizes \( \langle \text{deq}_i : x \rangle \).

**Proof.** Let \((s, j, \bot) \mapsto (1, i, x)\) be \( \text{enq}_i(x) \)'s enqueue transition storing \( x \) into \( u = \text{node}(i) \) (Figure 5.2d, Line 128). Notice that \( j \leq i \). If \( \text{deq} \) takes sufficiently many steps after obtaining \( i \) from its F&A on \( \text{head} \), it performs a transition on \( u \) using index \( i \). To see this, notice that \( \text{deq} \) moves on from \( u \) without performing a transition only if it reads an index \( > i \) from \( u \) (Figure 5.2b, Line 67). Because \( \text{enq}_i \)'s transition succeeds, \( \text{deq} \) is the only operation that can move \( u \)'s index beyond \( i \), so this is impossible.

Now, consider \( \text{deq}_i \)'s transition. It cannot be \((\cdot, k, \bot) \mapsto (\cdot, i + R, \bot)\) (Line 80) since that implies \( \text{enq}_i \)'s transition fails. \( \text{deq}_i \)'s transition also cannot be of the form \((\cdot, k, v) \mapsto (0, k, v)\) (Line 76) because then, \( \text{enq}_i \)'s transition succeeding implies that some enqueue (possibly \( \text{enq}_i \)) subsequently obtains index \( t \leq i \) and then observes \( \text{head} \leq t \), which is impossible since \( \text{head} > i \). Thus, \( \text{deq}_i \)'s transition can only be a dequeue of \( x \). Hence (1) and (2) hold.

We prove (3) using induction on \( k \), the number of linearized enqueue operations. For \( k = 0 \) the claim is vacuously true. Suppose now that the \( k \)-th enqueue operation linearized is \( \text{enq}_i(x) \).

If \( \text{deq}_i \) exists in \( E \), then it does not complete before \( \text{enq}_i(x) \)'s F&A which returns \( i \), since otherwise \( \text{deq}_i \) does not return \( x \), contradicting (2). Therefore, there exists a first event \( e \) in which \( Q[i] = x \) and \( \text{deq}_i \) is active. Thus at some event \( e' \), at or after \( e \), \( Q[i] = x \) and \( \text{deq}_i \) has performed the F&A on \( \text{head} \) which returns \( i \). Let \( \text{idx} = \{ j : j < i, Q[j] \neq \bot \text{ at } e' \} \). For all \( j \in \text{idx} \)
idx, deq_j starts by e’ (because deq_i’s F&A has returned i) and does not complete before e’ (as that implies it is not linearized before completing, contradicting the induction hypothesis). Therefore, at e’ P linearizes deq_j for all j ∈ idx and subsequently linearizes deq_i. □

To complete the linearizability proof, we must show that our linearization order meets the tantrum queue specification. Because we enqueue to Q’s tail, dequeue from Q’s head, and following the first enqueue to return CLOSED all enqueues do so, this amounts to showing that the auxiliary queue is empty when we linearize a ⟨deq : EMPTY⟩ operation. Lemma 16 below implies this, because we linearize a ⟨deq : EMPTY⟩ when it reads a value t from tail (Figure 5.2b, Line 86) such that t ≤ h + 1, where h < head is the value that the deq’s prior F&A returns (Line 61).

Lemma 16. If at event e, head ≥ tail, then head(Q) = tail(Q).

Proof. Suppose towards a contradiction that head(Q) < tail(Q) at e. Then there exists a minimal i such that Q[i] ≠ ⊥ at e. Because we update tail(Q) following the order of F&As on tail, i < tail(Q) ≤ tail ≤ head at e. Thus, deq_i is active before e and should have been linearized by P, a contradiction. □

In conclusion, we have shown the following.

Theorem 17. CRQ is a linearizable implementation of a tantrum queue.

5.2.2 The LCRQ algorithm

We now present LCRQ using the CRQ as a black box. The LCRQ is simply a linked list of CRQs in which dequeuing threads access the head CRQ and enqueuing threads access the tail CRQ (Figure 5.4a). An enqueue(x) operation that receives a CLOSED response from the tail CRQ creates a new CRQ, initialized to contain x, and links it after the current tail, thereby making it the new tail (Figure 5.4c). Dequeues dequeue from the head of the LCRQ, moving to the next next node (if one exists) after the head CRQ becomes EMPTY. If the head CRQ becomes EMPTY and there is a node linked after it, dequeues move to the next node, after installing it as the new head (Figure 5.4b). Before moving the LCRQ’s head, a dequeue tries to dequeue from the head CRQ again (Lines 183-184) to verify that no items have been enqueued between the first time it observed the head CRQ EMPTY (Line 180) and the time it read the head’s next pointer (Line 180).
5.2. THE LCRQ ALGORITHM

Memory reclamation A dequeue that successfully changes the head pointer cannot reclaim the memory used by the old CRQ because there may be concurrent operations about to access it (i.e., stalled just before Line 180 or Line 197). We address this problem by using hazard pointers [90] to protect an operation’s reference to the CRQ it is about to access. We omit the details, which are standard.

Linearizability Assuming that the CRQ is a linearizable tantrum queue, proving that LCRQ is a linearizable queue implementation is straightforward:

Theorem 18. If CRQ is a linearizable tantrum queue implementation, then LCRQ is a linearizable queue implementation.

Proof. We linearize an enqueue that completes after appending a new CRQ to the list at the CAS which links the new CRQ (Figure 5.4c, Line 200). We linearize any other completed operation at the point in which its final CRQ operation takes place. The next pointer of a CRQ changes from null only after it becomes CLOSED, and conversely, after a CRQ becomes CLOSED no new enqueue completes until a new CRQ is linked after it. Thus, if \( q_0 \) precedes \( q_1 \) in the list, any \( q_1 \) enqueue is linearized after any \( q_0 \) enqueue. Similarly, no dequeue accesses \( q_1 \) before \( q_0 \) becomes EMPTY after being CLOSED. Thus, any \( q_0 \) dequeue is linearized before any \( q_1 \) dequeue. Linearizability follows. \( \square \)
CHAPTER 5. FAST FIFO QUEUES FROM CONTENDED FETCH-AND-ADD

5.2.2.1 LCRQ nonblocking proof

In this section, we prove the following theorem:

**Theorem 19.** LCRQ is op-wise nonblocking.

**Proof.** An enqueuer that does not complete within a finite number of steps in the tail CRQ closes it. Once the CRQ is closed, every enqueuer taking enough steps tries to append a new CRQ to the LCRQ. The first one to CAS the CRQ’s next pointer (Figure 5.4c, Line 200) succeeds and completes. Thus, an enqueue operation completes within a finite number of steps by enqueuing threads.

Now, consider a dequeuer \( \text{deq} \) taking an infinite number of steps without completing. Suppose first that \( \text{deq} \) remains in one LCRQ node, \( q \). If enqueuers take infinitely many steps in \( q \), then \( q \) does not close and so, because \( q \)’s size is finite, dequeuers remove items from \( q \). If enqueuers take only finitely many steps in \( q \), then from some point only dequeuers take steps in \( q \) and so eventually \( q \)’s head exceeds its tail. Then \( \text{deq} \) finds that \( q \) is empty (Lines 86-87), enters \( \text{fixState()} \) but never returns from this method. Thus, new dequeuers continue to enter \( q \) and increment head. Since the number of dequeuers is finite, this implies some dequeuer completes.

The other possibility is that \( \text{deq} \) returns EMPTY in each CRQ node \( q_i \) it enters but never reaches the LCRQ’s tail. Each node \( q_i \) contains at least one item, and so there is a dequeuer \( d_i \) that holds the index to this item. After traversing through \( T \) nodes, where \( T \) is the number of threads in the system, it must be that \( d_i = d_j \) for some \( j > i \). This means \( d_i \) completes and returns. Overall, we have shown that a dequeue must complete within a finite number of steps by dequeuing threads. \( \square \)

5.3 Evaluation

**Evaluated algorithms** We compare LCRQ to the best performing queues reported in the recent literature, all of which are based on the combining principle: Hendler et al.’s FC queue [63] and Fatourou and Kallimanis’ CC-Queue and H-Queue [54]. We also test Michael and Scott’s classic nonblocking MS queue [92].

The FC queue is based on flat combining, in which a thread becomes a combiner by acquiring a global lock, and then applies the operations of the non-combining threads. The queue we test is a linked list of cyclic arrays, with a new tail array allocated when the old tail fills.
The CC-Queue replaces each of the two locks in Michael and Scott’s two-lock queue [92], which serialize accesses to the queue’s head and tail, with an instance of the CC-Synch universal construction [54]. The CC-Synch universal construction maintains a linked list to which threads add themselves using SWAP. The thread at the head of the list traverses the list and performs the requests of waiting threads. Since the enqueue and dequeue CC-Synch instances work in parallel, the CC-Queue outperforms the FC queue [54].

The H-Queue is a hierarchical version of the CC-Queue. It uses an instance of the H-Synch universal construction [54] to replace the two-lock queue’s locks. The H-Synch construction consists of one instance of CC-Synch per cluster and a lock that synchronizes the CC-Synch instances. Each CC-Synch combiner acquires the lock and performs the operations of the threads on its cluster.

To obtain the most meaningful results, we use the queue implementations from Fatourou and Kallimanis’ benchmark framework [53, 54], all of which are in C. We incorporate the FC queue implementation into this framework.

**LCRQ implementation** We use CRQs whose ring size, $R$, is $2^{17}$. (We include a sensitivity study of LCRQ to the ring size below.) In addition to baseline LCRQ, we also evaluate LCRQ+H, in which we enable our hierarchical optimization (with a timeout of 100 µs). To explore the impact of CAS failures, we test LCRQ-CAS, a version of LCRQ in which we implement the F&As using a CAS loop. All LCRQ variants include the overhead of pointing a hazard pointer at the CRQ before accessing it.

**Methodology** We evaluate the algorithms on the Westmere machine. We follow the testing methodology of prior work [54, 92]. We measure the time it takes for every thread to execute $10^7$ pairs of an enqueue followed by a dequeue operations, averaged over 10 runs. Note that in prior work each thread does $10^7 / T$ operations, where $T$ is the number of threads. However, with high levels of concurrency this leads to extremely short executions, hence we keep the number of operations fixed.

As in prior work, in every test we avoid artificial *long run* scenarios [92], in which a thread zooms through many consecutive operations, by having each thread wait for a random number of nanoseconds (up to 100) between operations. Each thread is pinned to a specific

---

3. We fixed a memory leak bug in the CC and H-Queue implementations, thereby improving their performance.
4. This consists of a writing the CRQ’s address to a thread-private location, issuing a memory fence, and rereading the LCRQ’s head/tail.
Figure 5.5: Enqueue/dequeue throughput on a single processor. The right plot shows throughput with more threads than available hardware threads; the first point, showing the throughput at maximal hardware concurrency, is included for reference.

hardware thread, to avoid interference from the operating system scheduler. Our tests use the jemalloc [52] memory allocator to prevent memory allocation from being a bottleneck. Results’ variance is negligible (we use a dedicated test machine).

**Single processor executions (Figure 5.5a)** Here we restrict threads to run on one of the server’s processors. This evaluates the queues in a modern multicore environment in which all synchronization is handled on-chip and thus has low cost. We omit results of LCRQ+H and H-Queue, since they are relevant only for multi-processor executions.

LCRQ outperforms all other queues beyond 2 threads. From 10 threads onwards, LCRQ outperforms CC-Queue by $1.5 \times$, the FC queue by $>2.5 \times$, and the MS queue by $>3 \times$. LCRQ-CAS matches LCRQ’s performance up to 4 threads, but at that point its performance levels off. Subsequently, LCRQ-CAS exhibits the throughput “meltdown” associated with highly contended hot spots. Its throughput at maximum concurrency is 33% lower than its peak performance at 8 threads. Similarly, MS queue’s performance peaks at 2 threads and degrades as concurrency increases.

Table 5.1 explains the above results. LCRQ, LCRQ-CAS and the MS queue all complete in a few instructions, but some of these are expensive atomic operations on contended locations. LCRQ-CAS and the MS queue suffer from CAS failures, which also lead to more cache misses as the algorithm wastes work. In the combining algorithms, communication between combiners and waiting threads causes more cache misses compared to LCRQ.
### 5.3. EVALUATION

<table>
<thead>
<tr>
<th></th>
<th>LCRQ</th>
<th>LCRQ-CCAS</th>
<th>CC-Queue</th>
<th>FC queue</th>
<th>MS queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency</strong></td>
<td>0.13 $\mu$s</td>
<td>0.96 $\times$</td>
<td>0.95 $\times$</td>
<td>0.91 $\times$</td>
<td>0.88 $\times$</td>
</tr>
<tr>
<td><strong>Instructions</strong></td>
<td>278.46</td>
<td>284.96</td>
<td>294.96</td>
<td>284.96</td>
<td>228.77</td>
</tr>
<tr>
<td><strong>Atomic operations</strong></td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td><strong>L1 misses</strong></td>
<td>0.51</td>
<td>0.51</td>
<td>0</td>
<td>0.03</td>
<td>0</td>
</tr>
<tr>
<td><strong>L2 misses</strong></td>
<td>0.05</td>
<td>0.06</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LCRQ</th>
<th>LCRQ-CCAS</th>
<th>CC-Queue</th>
<th>FC queue</th>
<th>MS queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Latency</strong></td>
<td>0.44 $\mu$s</td>
<td>2.70 $\times$</td>
<td>1.45 $\times$</td>
<td>3.51 $\times$</td>
<td>5.95 $\times$</td>
</tr>
<tr>
<td><strong>Instructions</strong></td>
<td>280.27</td>
<td>302.08</td>
<td>867.27</td>
<td>3846.29</td>
<td>321.05</td>
</tr>
<tr>
<td><strong>Atomic operations</strong></td>
<td>2</td>
<td>3.04</td>
<td>1</td>
<td>0.21</td>
<td>4.3</td>
</tr>
<tr>
<td><strong>L1 misses</strong></td>
<td>3.85</td>
<td>7.64</td>
<td>9.62</td>
<td>5.46</td>
<td>13.19</td>
</tr>
<tr>
<td><strong>L2 misses</strong></td>
<td>3.86</td>
<td>7.23</td>
<td>6.91</td>
<td>3.81</td>
<td>13.15</td>
</tr>
</tbody>
</table>

Table 5.1: Single processor (queue initially empty) average per-operation statistics. Latency numbers are relative to LCRQ. There are no L3 misses because the workload fits into the shared L3 cache. Atomic operations count does not include setting of hazard pointers.
Oversubscribed workloads (Figure 5.5b) Problems related to blocking usually occur in oversubscribed scenarios, in which the number of software threads exceeds the hardware supported level and forces the operating system to context switch between threads. If a thread holding a lock is scheduled out, the algorithm cannot make progress until it runs again. We show this by increasing the number of threads in a single processor execution beyond 20. The throughput of the lock-based combining algorithms plummets, with FC queue dropping by $40 \times$ and CC-Queue by $15 \times$, whereas both LCRQ and the MS queue maintain their peak throughput. As a result, LCRQ outperforms the CC-Queue by $20 \times$.

Four processor executions (Figure 5.6) To measure the effect of the increased synchronization cost between processors, we pin the threads across the processors in a round-robin manner, so that the cross-processor cache coherency cost always exists. One can see how, in contrast to the single processor experiment, when going from 1 to 2 threads the throughput of all algorithms drops due to cross-processor synchronization, except for LCRQ and LCRQ+H. Figure 5.6a shows the results when the queue is initially filled with $2^{16}$ elements, thus keeping the queue’s head and tail apart. This causes the throughput of CC-Queue to degrade by $\approx 10\%$ compared to the initially empty case (Figure 5.6b), due to reduced locality: in an initially empty queue, the queue’s state keeps hovering around empty and so there is a 1 in 4 chance that dequeued items will have just been enqueued on the same processor by the enqueuing combiner. In contrast, switching to an initially filled queue improves LCRQ’s throughput by $\approx 5\%$. The reason is that when the queue is not empty an LCRQ dequeuer does not wait for an enqueuer to arrive at its ring node. (Table 5.2 shows that LCRQ operations take less instructions to complete.) The reduced locality does not hurt LCRQ because dequeued items are fetched in parallel by all operations, and not sequentially by a single combiner. Overall, using an initially filled queue increases LCRQ’s advantage over CC-Queue from $\approx 1.5 \times$ to $\approx 1.8 \times$.

Heavy synchronization cost due to lack of locality also explains why only the hierarchical LCRQ+H and H-Queue scale past 16 threads. These algorithms amortize the synchronization cost by running batches of operations on a single processor while operations on other processors wait. However, H-Queue suffers much more from the reduced locality caused when switching to an initially filled queue: it triples the number of L3 misses (Table 5.2), which must be satisfied from off-chip resources and so its throughput drops by $\approx 40\%$. In contrast,

---

5On a single processor this test yields similar results to an initially empty queue and so we did not discuss it earlier.
Figure 5.6: Enqueue/dequeue throughput on four processors (threads run on all processors from the start).

<table>
<thead>
<tr>
<th>Queue initially empty</th>
<th>LCRQ+H</th>
<th>LCRQ</th>
<th>LCRQ-CAS</th>
<th>H-Queue</th>
<th>CC-Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>2.19 µs</td>
<td>6.20 µs</td>
<td>13.50 µs</td>
<td>3.28 µs</td>
<td>9.70 µs</td>
</tr>
<tr>
<td>Instructions</td>
<td>1456.65</td>
<td>307.15</td>
<td>338.98</td>
<td>5670.17</td>
<td>16249.94</td>
</tr>
<tr>
<td>Atomic operations</td>
<td>2</td>
<td>2</td>
<td>2.88</td>
<td>1.05</td>
<td>1</td>
</tr>
<tr>
<td>L1 misses</td>
<td>4.12</td>
<td>2.91</td>
<td>4.15</td>
<td>9.99</td>
<td>10.70</td>
</tr>
<tr>
<td>L2 misses</td>
<td>4.15</td>
<td>2.83</td>
<td>4.01</td>
<td>7.10</td>
<td>8.65</td>
</tr>
<tr>
<td>L3 misses</td>
<td>0.51</td>
<td>1.47</td>
<td>2.23</td>
<td>0.34</td>
<td>5.90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue initially full</th>
<th>LCRQ+H</th>
<th>LCRQ</th>
<th>LCRQ-CAS</th>
<th>H-Queue</th>
<th>CC-Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>2.05 µs</td>
<td>5.81 µs</td>
<td>13.45 µs</td>
<td>5.19 µs</td>
<td>10.55 µs</td>
</tr>
<tr>
<td>Instructions</td>
<td>1515.60</td>
<td>278.62</td>
<td>293.86</td>
<td>9173.94</td>
<td>18224.62</td>
</tr>
<tr>
<td>Atomic operations</td>
<td>2</td>
<td>2</td>
<td>2.95</td>
<td>1.05</td>
<td>1</td>
</tr>
<tr>
<td>L1 misses</td>
<td>3.43</td>
<td>3.01</td>
<td>4.31</td>
<td>10.60</td>
<td>11.33</td>
</tr>
<tr>
<td>L2 misses</td>
<td>3.54</td>
<td>2.90</td>
<td>4.17</td>
<td>7.74</td>
<td>9.07</td>
</tr>
<tr>
<td>L3 misses</td>
<td>0.81</td>
<td>1.43</td>
<td>2.22</td>
<td>0.95</td>
<td>6.19</td>
</tr>
</tbody>
</table>

Table 5.2: Four processor (80 threads) average per-operation statistics. Atomic operations count does not include setting of hazard pointers.
CHAPTER 5. FAST FIFO QUEUES FROM CONTENDED FETCH-AND-ADD

Figure 5.7: Cumulative distribution of queue operation latency at maximum concurrency.

LCRQ+H maintains its performance, increasing its advantage over H-Queue from $1.5 \times$ to $2.5 \times$.

Latency of operations (Figure 5.7) Examining the latency distribution of queue operations at maximum concurrency provides more refined insight on the performance of the algorithms. For instance, while the average latency of an LCRQ+H operation is $2.19 \mu s$ (Table 5.2), 80% of the operations complete in $\leq 0.5 \mu s$ and 97% in $\leq 2 \mu s$. The remaining operations are those that complete only after the timeout expires. The spinning these operations do while waiting accounts for the increased average instruction count of LCRQ+H compared to LCRQ shown in Table 5.2. In general, LCRQ operations have better latency than combining-based operations, which spend time either servicing other threads or waiting for the combiner. On a single processor, 42% of LCRQ operations finish in $\leq 0.24 \mu s$ while none of the combining operations do. On four processors, 80% of LCRQ operations finish in $\leq 9.6 \mu s$ compared to 50% of CC-Queue operations. Similarly, 80% of LCRQ+H operation finish in $\leq 0.5 \mu s$ compared to 30% of H-Queue operations.

Ring size sensitivity study (Figure 5.8) The ring size plays an important role in the performance of LCRQ. Intuitively, as the ring size decreases an LCRQ operation needs more tries before it succeeds in performing an enqueue/dequeue transition.

To quantify this effect, we test LCRQ on an initially empty queue at maximum concurrency with various ring sizes. On a single processor, taking $R \geq 32$ is enough for LCRQ to outperform the CC-Queue by $1.33 \times$. As $R$ increases LCRQ’s throughput increases up to $\approx 1.5 \times$ that of the CC-Queue. In other words, as long as an individual CRQ has room for all running threads,
5.4. COMPARISON TO EXISTING WORK

LCRQ obtains excellent performance.

On the four processor benchmark the results are similar, but due to the higher concurrency level, LCRQ outperforms CC-Queue starting with $R = 128$ and the advantage becomes $\approx 1.5 \times$ starting with $R = 1024$. LCRQ+H requires $R = 512$ to match H-Queue and $R = 4096$ to outperform H-Queue by $1.5 \times$.

5.4 Comparison to existing work

We refer the reader to Michael and Scott’s extensive survey [92] for discussion of additional work that predates theirs.

List based queues Michael and Scott present two linked list queues, one nonblocking (henceforth MS queue) and one lock-based [92]. However, due to contention on the queue’s head and tail, their algorithms do not scale past a low level of concurrency [54, 63]. Kogan and Petrank introduce a wait-free variant of the MS queue with similar performance characteristics [76]. Several works attempt to improve the MS queue’s scalability, however all these still suffer from the CAS retry problem [71, 78, 95].

Ladan-Mozes and Shavit [78] describe an optimistic queue that reduces the number of CAS operations required to enqueue in the MS queue from two to one. Still, each operation needs to modify one end of the queue list using a CAS. Moir et al. [95] apply an elimination backoff scheme on top of the MS queue. The idea is that enqueues and dequeues can pair up and eliminate each other. To maintain FIFO semantics, an enqueue cannot be eliminated unless all previous items are dequeued, limiting applicability of this approach. Hoffman et al. [71] apply
a baskets approach to the MS queue, replacing each node with an abstract basket, implemented using linked lists, that contains items which may be dequeued in any order. If an enqueue fails to link its node to the main queue, it attempts to link its node to the tail basket. However, insertions and removals from the main queue and from individual baskets are still implemented using CAS.

**Cyclic array queues** Prior concurrent cyclic array queues are bounded and can contain a fixed number of items. One of the challenges in these algorithms is correctly determining when the queue is full and empty.

The queues of Gottlieb et al. [61] and of Freudenthal and Gottlieb [56] maintain a size counter that is updated using F&A. Such a F&A might bring the queue into an inconsistent state (e.g., size < 0) and the algorithm then tries to recover using a compensating F&A. Still, the inconsistent states make these queues non-linearizable. Blelloch et al. [31] use room synchronization, which prevents enqueues from running concurrently to dequeues, to construct a queue that is linearizable despite temporarily entering inconsistent states when its head/tail are updated using F&A. Another queue by Blelloch et al. [32] avoids inconsistent states of the head and tail by updating these indices using hardware memory block transactions which are not supported by commercial hardware. Tsigas and Zhang [118], Colvin and Groves [43] and Shafiei [111] present cyclic array queues that avoid inconsistent head and tail states by performing the updates using CAS, but are therefore prone to the CAS failure effect.

In contrast to these prior designs, LCRQ is an unbounded queue formed by linking CRQs (array queues) in a list, with a new CRQ added when an enqueue operation fails to make progress. The ability to close a CRQ, forcing enqueues to move to the next CRQ in the list, makes LCRQ nonblocking whereas prior F&A-based designs [31, 32, 56, 61] are blocking. In addition, since we do not need to determine when the queue is full in a linearizable way, we can recover from inconsistent states that result from using F&A for head/tail updates without compromising linearizability. Performance-wise, a CRQ operation accesses only one end of the queue in the common case, whereas the operations in the previous designs access both the head and tail indices. Moreover, in the CAS-based designs [43, 111, 118] a thread first tries to enqueue/dequeue at the array node pointed to by the tail/head of the queue, and only after succeeding updates the tail/head index using CAS. Therefore, all (say) dequeuers that observe the same head value compete on the same node, where all but one lose and must try again.

---

6Blelloch et al. [31] show a non-linearizable execution for Gottlieb et al.’s queue. A similar scenario applies to Freudenthal and Gottlieb’s queue.
leading to the CAS failure effect. In CRQ we first reserve a node using F&A and only then access it.

**Combining** Researchers have recently shown that *combining-based* queues scale better than CAS-based list queues [53, 54, 63]. A combining algorithm is essentially a *universal construction* [66] that can implement any shared object. The idea is that a single thread scans a list of pending operations and applies them to the object. Such algorithms greatly reduce the synchronization cost of accessing the object, at the cost of executing work serially.

Hendler et al. describe a linked list queue based on *flat combining*, a lock-based combining construction [63]. Fatourou and Kallimanis present SimQueue [53], a queue based on a wait-free combining construction, and CC-Queue, a queue based on a blocking combining algorithm [54]. Section 5.3 details these algorithms.

Both of Fatourou and Kallimanis’ algorithms use weak synchronization primitives (F&A and SWAP). However, they do so to reduce the synchronization cost of the combining algorithm, which still needs to perform serial work that is linear in the number of threads. In contrast, we use F&A to enable parallelism in the seemingly inherently sequential FIFO queue.
Chapter 6

Fence-free work stealing on bounded TSO processors

This chapter describes how to implement fence-free work stealing on multicores with a bounded total store ordering (TSO) memory model which allows only a bounded amount of reordering, as in current popular x86 and SPARC processors.

Setting the stage We start by formally defining the work stealing task and extending our computational model to capture bounded TSO machines (Section 6.1). Then we describe how existing work stealing algorithms work, and why they require issuing an expensive memory fence in the process of removing a task (Section 6.2). Because issuing such a memory fence was proved necessary by Attiya et al. [27], Section 6.3 next explains how we get around Attiya et al.’s impossibility results by invalidating some of their assumptions.

Our results In Section 6.4 we describe how a thief can approximate the worker’s position in the work stealing queue by reasoning about the worker’s store buffer, and how this can be used to obtain safe work stealing without requiring the worker to issue memory fences. To deal with the cases in which approximating the worker’s position leaves the thief uncertain about whether it can safely steal, Section 6.5 proposes a heartbeat mechanism which allows the thief to learn the worker’s updated position. Section 6.6 shows that the bounded TSO model captures the behavior of commercial x86 and SPARC processors. We evaluate the performance benefit of our techniques in Section 6.7. Finally, we discuss related work (Section 6.8).
6.1 Preliminaries

6.1.1 Work stealing specification

A work stealing queue supports three methods: \texttt{put()}, \texttt{take()} and \texttt{steal()}. Its state is a double-ended queue of items. A \texttt{put(y)} enqueues \emph{y} to the tail of the queue. A \texttt{take()} applied to a non-empty queue dequeues from its tail. A \texttt{steal()} applied to a non-empty queue dequeues from its head. A \texttt{take()} or \texttt{steal()} applied to an empty queue return \texttt{EMPTY}.

6.1.2 Bounded TSO model

This section extends our computational model (Section 2.1) to bounded TSO systems. Our model is inspired by Sewell et al.’s x86-TSO model \cite{109}. However, unlike x86-TSO, we make the store buffer’s capacity a part of the model.

6.1.2.1 TSO[$S$] computational model

In the TSO[$S$] model, a load can reordered with at most $S$ prior stores. We model TSO[$S$] as a shared memory system with a set of sequential threads that communicate via shared memory, similarly to Section 2.1. Here, however, each thread $T$ is equipped with a FIFO store buffer that holds $T$’s stores before they get written to memory. The store buffer holds at most $S$ stores, and $S$ is called the buffer’s capacity. We refer to thread $T$’s store buffer by $sb(T)$, and denote the number of stores held in $sb(T)$ by $|sb(T)|$. A store buffer is full if $|sb(T)| = S$ and empty if $|sb(T)| = 0$. In the following, we describe the changes to Section 2.1’s model needed to support store buffering. One technical difference is that now we have some restrictions on when operations can be invoked (e.g., a thread cannot write if its store buffer is full). For simplicity, we describe these restrictions directly. Their formal interpretation is that we only consider executions in which the restrictions are not violated.

**Writes go into the store buffer** A store by thread $T$ is enqueued in $sb(T)$. Formally, thread $T$ invokes write($a, x$) only if $sb(T)$ is not full. The write($a,x$) operation then enqueues the pair ($a,x$) into $sb(T)$. (Notice that if a thread performs a store to an address for which there is already a store in the store buffer, this store is enqueued into a distinct location in the store buffer.)
Threads read their writes If stores to address $a$ resides in $sb(T)$, a read of address $a$ by $T$ returns the value written by the newest matching store. Formally, a read($a$) returns $m[a]$ if there is no entry $(a,x)$ in $sb(T)$, otherwise the read returns the newest matching value in $sb(T)$. (Recall that $m[a]$ denotes the value stored in address $a$ of the memory.)

Writes get flushed nondeterministically A store enqueued in $sb(T)$ is flushed to memory at some nondeterministically chosen time. Formally, we define a new primitive event named flush($a, x$). If the oldest entry in $sb(T)$ is $(a, x)$, a flush($a, x$) event by thread $T$ dequeues $(a, x)$ from $sb(T)$ and changes $m[a]$’s value to $x$. If thread $T$’s store buffer is not empty, then instead of making the next primitive event specified by its algorithm, $T$ can nondeterministically invoke the next valid flush operation. We do not allow threads to explicitly control when an individual flush happens, except by issuing memory fences, which are described next. (Section 6.1.2.2 expands on this.)

Memory fences To explicitly flush all entries in its store buffer to memory, $T$ invokes a fence operation. Formally, we define a fence() operation that can only be invoked if $sb(T)$ is empty.

Atomic primitives are fences Thread $T$ can invoke the atomic primitives F&A, SWAP, T&S, CAS or CAS2 only if $sb(T)$ is empty.

6.1.2.2 Implementations

An algorithm specifying an implementation cannot explicitly issue an individual flush; it may only use fences and atomic operations. From the algorithm’s perspective, these operations flush the thread’s entire store buffer. To see this, consider that in any valid execution, a fence (or atomic operation) by $T$ appears only when $sb(T)$ is empty. Thus, any valid execution contains sufficiently many flush events to empty $sb(T)$ before the fence (or atomic operation) appears. $T$ is therefore guaranteed that all its stores are written to memory once its flush executes.

6.1.2.3 Discussion

The model is not meant to describe the hardware implementation of a TSO[S] machine, but the observable behavior of the machine. For example, a TSO[S] machine may execute read operations out-of-order under the hood, whereas our model specifies in-order execution. However, a TSO[S] processor guarantees that any read reordering is not observable by the program: reordered reads are executed speculatively and retired only when their results are consistent with
in-order execution. We can therefore think about executions as if reads are executed in-order as dictated by the TSO[S] model.

6.2 Background: work stealing synchronization

Modern work stealing algorithms [13, 40, 57] strive to reduce the overhead experienced by workers performing the computation [57], even at the cost of making steal operations more expensive. As a result, these algorithms have converged on a similar design. In this design the task queue is represented using a double-ended queue. A worker puts and takes tasks from the queue’s tail and a thief steals from the queue’s head, so that a worker and thief usually do not contend for the same task. When trying to take a task, the worker uses a protocol based on the flag principle [69] to detect if a conflict with a thief might exist. If so, the worker switches to a more heavyweight synchronization protocol to decide whether the worker or thief gets the task.

Figure 6.1a shows the general design. Each worker has a queue consists of a (cyclic) array of \( W \) tasks with strictly increasing head and tail indices. (For simplicity, we omit details of resizing the array if it becomes full.) An index with value \( i \) points to element \( i \mod W \) of the tasks array. The head, \( H \), points to the first task in the queue. The tail, \( T \), points to the first unused array element. If \( T = H \) the queue is empty.

A worker performs a \( \text{put()} \) by storing the task at the tail of the queue, and then incrementing the tail index \( T \). The TSO model guarantees that storing the task and incrementing \( T \) are not reordered.

To \( \text{take()} \) a task, the worker decrements \( T \) from \( t + 1 \) to \( t \), thereby “raising the flag” and publishing its intent to take task \( t \). It then reads the head index \( H \) after issuing a memory fence to ensure that reading \( H \) does not get reordered before decrementing \( T \).

If the worker observes that \( t > H \), it can safely remove task \( t \) from the queue, as it has verified there can be no conflict for task \( t \): when the tail update became globally visible, thieves have announced intent to steal only tasks up to \( H < t \), which means that a new steal operation will observe a queue that does not contain task \( t \).

However, if \( t \leq H \) there may be a conflict with a thief. The algorithms differ in the synchronization protocol used to handle such a conflict. In the following we continue with the description of each algorithm’s protocol:
CHAPTER 6. FENCE-FREE WORK STEALING ON BOUNDED TSO PROCESSORS

// shared variables
H : 64-bit int
T : 64-bit int
tasks : array of W work items

put(task) {
    t := T
    tasks[t mod W] := task
    T := t + 1
}

take() {
    t := T
    T := t
    fence()
    h := H
    if (t > h) {
        // Thief observes t and will not try to steal task t.
        return tasks[t mod W]
    }
}

steal() {
    H := H + 1
    fence()
    if (H < T) {
        ret := tasks[h mod W]
    } else {
        H := h
        ret := EMPTY
    }
    unlock()
    return ret
}

(a) Algorithm outline.

(b) Cilk THE [57].

(c) Chase-Lev [40].

Figure 6.1: Design of modern work stealing task queues. Both the Cilk THE and Chase-Lev algorithms use the flag principle to detect when the worker and thief might contend for a task. They differ in the synchronization used to manage such a conflict.

Cilk’s THE algorithm (Figure 6.1b) Cilk’s THE algorithm uses a per-queue lock to synchronize between a worker and a thief, and also to enforce mutual exclusion among thieves. In case of a conflict on a task, the protocol picks the worker as the winner.

A thief acquires the queue lock and then increments the head index \( H \) from \( h \) to \( h + 1 \), tentatively publishing its intent to steal task \( h \). It then issues a memory fence and afterwards checks if \( H \leq T \). If so, the thief knows its increment of \( H \) will be observed by any future worker and thus the thief can safely steal task \( h \). Otherwise \((H > T)\), there are two possible cases: either the queue was empty \((T = H)\) when the thief arrived, or a worker has just published its intent to take the same task (e.g., initially \( T = 1 \) and \( H = 0 \), then the worker’s decrement and the thief’s increment cross, leading to a state in which \( T = 0 \) and \( H = 1 \)). Either way, the thief restores \( H \) to its original value and aborts the steal attempt.

This behavior makes it safe for a worker which (following its decrement of \( T \)) finds that \( T = H \) to take the task. The remaining case, in which a worker observes \( T < H \) after its decrement,
is again caused either by an initially empty queue or because of a concurrent steal attempt. The worker therefore acquires the queue lock – meaning that if a conflicting thief exists, it has backed off and released the lock – and returns the task or restores the queue to a consistent state if it was empty.

**Chase-Lev algorithm (Figure 6.1c)** The Chase-Lev nonblocking [66] algorithm uses an atomic compare-and-swap (CAS) operation to pick the winner in a conflict on a task. A thief reads the queue’s head and tail, and if the queue is not empty (i.e., $T > H$) the thief tries to atomically increment $H$ from $h$ to $h + 1$ using a CAS. If the CAS succeeds, the thief has stolen the task.

To support this simple stealing protocol (and in contrast to the THE algorithm) a worker must always increment $H$ to remove the last task. After decrementing $T$, if the worker finds that $T = H$, it restores $T$ to its original value and attempts to take this last task by incrementing $H$ with a CAS. Otherwise ($T < H$) then the queue was initially empty or a thief has concurrently incremented $H$. In either case, the worker returns EMPTY after fixing the queue’s state by setting $T$ to $H$.

### 6.2.1 Linearizability of work stealing algorithms

The Cilk THE and Chase-Lev work stealing algorithms are not linearizable under TSO [86]. For example, a put() may be delayed in the worker’s store buffer and missed by a thief. This can result in a linearizability violation, as shown in this Chase-Lev execution:

<p>| initially, $T = 0$ and $H = 0$ |</p>
<table>
<thead>
<tr>
<th>worker</th>
<th>thief</th>
</tr>
</thead>
<tbody>
<tr>
<td>put() invoked</td>
<td>steal() invoked</td>
</tr>
<tr>
<td>buffered stores:</td>
<td>read $H = 0$</td>
</tr>
<tr>
<td>tasks[0] := item</td>
<td>read $T = 0$</td>
</tr>
<tr>
<td>$T := 1$</td>
<td>return EMPTY</td>
</tr>
<tr>
<td>put() completes</td>
<td>steal() returns</td>
</tr>
</tbody>
</table>
Though these algorithms can be made linearizable by issuing a fence before the put() returns [86], this linearizability violation does not impact correctness of the task parallel runtime using the work stealing algorithm: The worker will process the task before terminating, if it was not already stolen by a thief that arrives after the put()’s stores flush to memory. Thus, deployed work stealing implementations do not issue a fence on a put().

We take the same approach, and focus not on obtaining linearizable algorithms, but rather algorithms in which linearizability violations (if any) do not impact the more general correctness of the task parallel runtime.

### 6.3 Getting around the laws of order

This section explains how we get around Attiya et al.’s “laws of order” impossibility result [27] to obtain fence-free work stealing algorithms. The “laws of order” result is sometimes interpreted as proving that certain concurrent data structures, including work stealing queues, cannot be implemented without atomic operations or memory fences – without any reservations [25]. But as with any theorem, the result may not hold if any of its underlying assumptions are invalidated. Indeed, Attiya et al. suggest the research question of “taking advantage of our result by weakening its basic assumptions in order to build useful algorithms” (i.e., without expensive operations).

**Impossibility result** The “laws of order” theorem states that any linearizable implementation of a strongly non-commutative (SNC) method must use an atomic operation or memory fence in some execution. The sequential specification of the implemented data structure determines whether methods are SNC. Method $M$ is SNC if there is another method $M'$ (possibly the same method as $M$) such that applying $M$ followed by $M'$ from some initial state $\rho$ yields different outputs for $M$ and $M'$ than applying $M'$ followed by $M$.

**Example (take() and steal() are SNC)** Consider the state $\rho$ in which the work stealing task queue contains one task, $x$. When applying take() first it returns $x$ and a subsequent steal() returns EMPTY. Similarly, if steal() is applied first it returns $x$ and then take().

---

1The actual theorem statement does not mention memory fences, as it uses a sequentially consistent system model. Instead, the theorem states that either an atomic operation or a read-after-write (RAW) pattern must be used, where a RAW means a write to shared variable $X$ is followed by a read to another shared variable $Y$ without a write to $Y$ in between. However, TSO requires issuing a memory fence after the write to $X$ to prevent the read of $Y$ from being reordered before it.
returns EMPTY when applied. (It is easy to see that $\rho$ is the only state from which $\text{take()}$ and $\text{steal()}$ can influence each other in this way.)

**Tight implementations** The “laws of order” result assumes that the concurrent implementation is *tight* – that any sequential execution which complies with the specification can occur in a sequential execution of the implementation. (This is referred to as “Assumption 1” in the paper [27].) The proof needs this assumption to argue that an execution exhibiting the strong non-commutativity of a method actually occurs in the implementation. Our results break the tightness assumption.

**Violating tightness through non-determinism** Our first technique (Section 6.4) produces an implementation in which a $\text{steal()}$ operation may return ABORT, depending on the state of the worker’s store buffer. We add the ABORT value to the work stealing specification by allowing the $\text{steal()}$ method to non-deterministically return ABORT without changing the state of the queue. In our implementation a $\text{steal()}$ invoked when the queue is empty or contains one task always returns ABORT. This violates the tightness assumption: the SNC executions of $\text{take()}$ and $\text{steal()}$ do not occur in our implementation. Instead, $\text{steal()}$ returns ABORT without changing the queue’s state.

**Violating tightness by blocking** In Section 6.5 we obtain fence-free work stealing without relaxing the work stealing specification. To do this, we turn our non-deterministic algorithm into a deterministic one by allowing a $\text{steal()}$ invoked on certain (non-empty) queue states to *block* until the next time $\text{take()}$ is invoked. In particular, when $\text{steal()}$ is invoked on a queue containing one task in this implementation, it blocks and does not return until $\text{take()}$ is invoked. This violates the tightness assumption: the SNC execution of a $\text{steal()}$ returning $x$ followed by a $\text{take()}$ returning EMPTY cannot occur, as the $\text{steal()}$ would not terminate when running alone.

Indefinite blocking is undesirable in general, but in clients (programs) using work stealing the worker keeps taking tasks until the queue empties. Thus, thieves do not block forever. In a sense, our implementation sidesteps the impossibility result by leveraging the properties of work stealing clients to avoid the SNC executions.
6.4 Fence-freedom by reasoning about bounded reordering

Correctly implementing the (deterministic) work stealing specification requires that a thief fails to steal a task only when the queue is empty. But in practice, it often makes sense for a thief to abort a steal attempt even if the queue is not empty (e.g., when the queue’s lock is taken) and move on to another victim, and such optimizations are present in existing systems [11, 40].

Such behavior does not impact the correctness of the work stealing’s client (program), as it still maintains the task queue’s safety and does not result in a task being removed twice. However, reasoning about this behavior does require making the task queue’s specification (Section 6.1.1) non-deterministic — a steal() operation is allowed to (non-deterministically) return ABORT as a result of implementation-internal conditions.

This section derives our first technique for (worker) fence-free work stealing, obtaining a task queue that complies with the non-deterministic specification. To focus the discussion, we use Cilk’s THE algorithm as a concrete running example. (We apply the same principles to develop a fence-free version of the Chase-Lev algorithm in Section 6.4.1.)

What do we need in order to only guarantee the protocol’s safety? Once a thief makes its intent to steal task $h$ globally visible (by incrementing $H$ and issuing a fence), what it needs is to verify that the worker is not concurrently trying to take the same task, i.e., that $T > h$. Knowing this makes stealing task $h$ safe: any subsequent take() attempt will observe the updated queue head and not try to remove task $h$ without acquiring the lock.

The standard THE protocol answers the safety question by ensuring the thief observes the exact value of $T$. But the point is that any technique for answering the question “is $T > h$?” will do.

Bounding worker position  The technique we propose is to leverage knowledge of the store buffer’s capacity to deduce how far off the worker’s real position is from the position read from memory.

Suppose for now the thief knows that the store buffer’s capacity is $S$. (Section 6.6 describes how to measure the store buffer’s capacity.) Each worker queue operation changes $T$ by at most 1. Therefore, at the time in which a thief observes that $T$ contains the value $t$, the last value stored by the worker must be in the range $[t - S, t + S]$. So if $t - S > h$, the thief can steal the task. Otherwise, it must return the new ABORT value.

In general, a thief can safely steal task $h$ whenever it observes $T > h + \delta$, where $\delta \geq 1$ is the maximum number of stores to $T$ by take() operations that can exist in the store buffer. Without
any assumptions, $\delta$ is simply $S$, the size of the store buffer, as discussed above. However, we soon describe how to significantly decrease $\delta$ compared to $S$ by factoring in the stores performed by the program between consecutive `take()` operations.

Figure 6.2 shows the modifications required to implement this technique in the THE algorithm. Notice that now the thief never knows for certain whether the queue is empty, because there is always uncertainty about the final store performed by the worker (i.e., $\delta \geq 1$). Thus, the condition for returning `ABORT` subsumes the condition for returning `EMPTY` in the original THE algorithm.

**Context switches** The discussion thus far assumes the worker always uses the same store buffer. This does not hold if the operating system reschedules the worker thread, moving it from one core to another. However, it is easy to see that an operating system moving a thread from core $C_1$ to core $C_2$ must drain $C_1$’s store buffer. For example, if the thread loads from a location stored to on $C_1$ while it runs on $C_2$, it must observe the value previously stored. Indeed, vendor manuals document this requirement [15]. The discussion in this section thus rightfully considers only the last core a worker runs on.

**Tightening worker position bound** Typically, a program using work stealing actually does some work between task queue operations. The code implementing the work may store to memory and these stores take up space in the store buffer. If we know that a program always does at least $x$ stores between `take()` operations, Figure 6.3 shows how we can tighten the bound from $\delta = S$ to

$$
\delta = \begin{cases} 
\frac{S}{x+1} & \text{if } x+1 \text{ divides } S \\
\left\lfloor \frac{S}{x+1} \right\rfloor + 1 & \text{if } x+1 \text{ does not divide } S
\end{cases}
$$

```plaintext
steal() {
  lock()
  h := H
  H := h + 1
  fence()
  if (T > h + \delta) {
    ret := tasks(h mod W)
  } else {
    H := h
    ret := ABORT
  }
  unlock()
  return ret
}
```

Figure 6.2: Fence-free THE algorithm. The code of `put()` and `take()` remains the same, but for the removal of the memory fence in `take()`. The parameter $\delta$ is the maximum number of stores to $T$ by `take()` operations that can exist in the store buffer.
Store buffer capacity ($S$) is 8 stores.

$\begin{array}{|c|c|}
\hline
x = 1 & x = 2 \\
\hline
\text{(earliest store)} & \text{(latest store)} \\
\hline
\text{program store} & \text{program store} \\
T := 7 & T := 8 \\
\text{program store} & \text{program store} \\
T := 8 & \text{program store} \\
\text{program store} & \text{program store} \\
T := 9 & \text{program store} \\
\text{program store} & \text{program store} \\
T := 10 & T := 10 \\
\hline
\end{array}$

$\Rightarrow \delta = 4 \quad \Rightarrow \delta = 3$

Figure 6.3: Maximum number of stores to $T$ by \texttt{take()} operations that can exist in the store buffer. There can always be at least one such store; the number of earlier \texttt{take()} stores is dictated by the number of stores performed between \texttt{take()} operations.

We can obtain a preliminary lower bound on $x$, the number of stores between \texttt{take()}s, by inspecting the runtime’s code. For example, the CilkPlus runtime [11] updates a field in the dequeued task after removing it from the queue. Thus, we trivially have that $x \geq 1$ for any CilkPlus program.

To obtain a better bound, we can search for a weighted shortest path from \texttt{take()} to itself in the basic block control-flow graph of the program, where we assign the number of stores performed in a basic block $B$ as the weight of each edge going out of $B$.

### 6.4.1 Fence-free Chase-Lev algorithm version

The technique of bounding the worker’s position applies to the Chase-Lev algorithm, but with a somewhat different correctness argument. In the Chase-Lev algorithm, a worker about to remove the last task undoes its update of $T$ and uses CAS to advance the queue’s head. Thus, a thief about to remove task $h$ needs to verify that the worker’s store writing $T := h$ cannot be in the store buffer. If this is the case, the thief is guaranteed that if the worker tries to remove task $h$, it will synchronize with the thief using a CAS. As before, checking that $T > h + \delta$ establishes this. Figure 6.4 shows the pseudo code of the modified algorithm.
6.5. **FENCE-FREE WORK STEALING WITHOUT RELAXED SEMANTICS**

A drawback of obtaining fence-freedom by having a thief approximate the worker’s position is that this can lead to missed stealing opportunities. The thief may read a true value of the queue’s tail that happens to be too close to the head, and so the thief must conservatively abort its steal attempt. This section details a solution to this problem, which we implement on top of the non-deterministic variant of Cilk’s THE algorithm from Section 6.4.

Recall that the thief needs to verify the worker has observed its update of \( H \). To establish this, the thief maintains a counter which it increments on each \( \text{steal}() \). In turn, the worker writes the value it reads from this counter to a new variable, \( P \), allowing the thief to wait for \( P \) to reflect its counter. TSO guarantees that any value the thief subsequently reads from \( T \) was written after the worker observed the thief’s update of \( H \). The thief thus monitors the worker’s “heartbeat” until it knows the worker has observed its update of \( H \).

Using this approach yields an algorithm that meets the original deterministic specification of work stealing (Section 6.1.1) and never needs to abort a steal attempt. The price we pay is that occasionally a thief cannot make progress until the worker arrives and responds. Fortunately, in clients using work stealing the worker keeps taking tasks until the queue is empty, because it cannot rely on the work being stolen. Thus, if the queue is not empty, the worker eventually arrives and the thief can proceed.

However, we must make certain that the thief does not have to wait when the queue is empty, because in that case the worker may never arrive to respond. Therefore, if while waiting the thief notices that the queue is empty \( (T = H) \), it stops and returns \text{EMPTY}. This can result in non-linearizable executions, but the same kinds of linearizability violations already exist in

```java
69 steal() {
70   while (true) {
71     h := H
72     t := T
73     if (h ≥ t) return EMPTY
74     if (h ≥ t - δ) return ABORT
75     task := tasks[h mod W]
76     if (!CAS(&H, h, h+1)) // goto Line 70
77     continue
78     return task
79   }
80 }
```

Figure 6.4: Fence-free Chase-Lev algorithm. The code of \text{put}() and \text{take}() remains the same, but for the removal of the memory fence in \text{take}(). The parameter \( δ \) is the maximum number of stores to \( T \) by \text{take}() operations that can exist in the store buffer.
work stealing algorithms (Section 6.2.1).

The THEP algorithm (Figure 6.5) The thief maintains its counter in the top bits of \( H \). (The counter can also be maintained in a separate variable, at the cost of an extra load in the \( \text{take}() \) path.) On each \( \text{steal}() \) attempt, the thief increments the counter when it updates \( H \). Then, if it is uncertain about the worker’s position, it spins, reading \( T \) and \( P \) until one of the following occurs: (1) If the queue becomes empty (i.e., \( T < H \) which means \( T \) was equal to \( H \) before the thief incremented \( H \)), the thief returns \( \text{EMPTY} \). (2) If \( P \) echoes back the updated counter value, the thief reads \( T \) and proceeds as usual.

Chase-Lev algorithm Unlike the THE algorithm, the Chase-Lev algorithm is nonblocking [66]. In particular, a thief running alone always completes its operation. Our heartbeat technique would make the algorithm blocking, and therefore we do not consider it for this algorithm.

THEP algorithm correctness To show the safety of the THEP algorithm, we need to prove that if a thief steals task \( x \), the worker does not take \( x \) as well. There are two possible cases: (1) If the thief steals \( x \) because it observes \( T - h > \delta \) (Line 125), the claim follows from the safety

```c
// shared variables
H : struct { s:32 bits, h:32 bits }
P : initially ⊥
// T and tasks remain unchanged

take() {
    t := T - 1
    T := t
    <s, h> := H
    if (t < h) {
        lock()
        P := ⊥
        <s, h> := H
        if (h ≥ t + 1) {
            T := t + 1
            unlock()
            if (T ≤ h + δ) {
                while (P ≠ s+1) {
                    goto Line 121
                }
                t := T
                if (h + 1 ≤ t) {
                    ret := tasks[h mod W]
                } else {
                    H := <s + 1, h>
                    ret := EMPTY
                }
            } else {
                ret := tasks[h mod W]
            }
        } unlock()
        return ret
    } else {
        return tasks[t mod W]
    }
}

steal() {
    lock()
    <s, h> := H
    H := <s + 1, h + 1>
    fence()
    if (T ≤ h + δ) {
        while (P ≠ s+1) {
            goto Line 121
        }
        t := T
        if (h + 1 ≤ t) {
            ret := tasks[h mod W]
        } else {
            H := <s + 1, h>
            ret := EMPTY
        }
    } else {
        ret := tasks[h mod W]
    }
    unlock()
    return ret
}
```

Figure 6.5: The fence-free THEP variant of Cilk’s THE algorithm. As before, \( \delta ≥ 1 \) is the maximum number of stores to \( T \) by \( \text{take}() \) operations that can exist in the store buffer.
of the non-deterministic THE variant of Section 6.4. (2) Otherwise, suppose the thief steals $x$ after the worker echoes back the thief’s increment of $H$ in $P$ (Line 119). It follows from TSO that $t$, which the thief reads from $T$ in Line 117, is written by an invocation of $\texttt{take()}$ that reads the thief’s increment of $H$, i.e., $h + 1$. TSO guarantees that after the write of $t$ to $T$, any execution of $\texttt{take()}$ will also read $h + 1$ from $H$, and so to steal $x$ it will attempt to acquire the lock. Since the thief holds the lock, once such a $\texttt{take()}$ acquires the lock it will find that $H = h + 1 = t + 1$ and return $\text{EMPTY}$.

The remaining issue is whether waiting for a worker to arrive while holding the queue lock can introduce deadlock. To see why this cannot happen, notice that the worker tries to acquire the queue’s lock if $T < H$. A waiting thief eventually notices and returns $\text{EMPTY}$, releasing the lock.

### 6.6 Bounded TSO in mainstream processors

Here we show that the mainstream TSO architectures – x86 and SPARC – implement bounded TSO, except for a corner case in which there are consecutive stores to the same location, which can be prevented in software.

We use the Westmere processor as a running example, since it is representative of mainstream out-of-order TSO processors. (Similar but simpler reasoning applies to in-order processors.)

First, we explain how the processor’s implementation of TSO leads to bounded reordering except when consecutive stores to the same location are coalesced (§6.6.1). Then we derive the exact bound on the amount of reordering, and show how to adjust the work stealing runtime to avoid store coalescing (§6.6.2-6.6.3).

#### 6.6.1 Cause of bounded store/load reordering

To hide the latency of writing to the memory subsystem (which may include resolving a cache miss) the processor retires a store instruction from the reorder buffer without waiting for its value to reach the memory subsystem (henceforth simply “memory”). Instead, the processor holds the instruction’s target address and data in a store buffer entry, from which it moves the data to memory as a background task once the store retires [4, 14, 15].

Store buffering makes store/load reordering possible because a load can retire, having read from memory, before the value of an earlier store to a different location gets written to memory.
In fact, this is the only way reordering may happen. Out-of-order execution does not lead to further store/load reordering because, to maintain TSO, the processor retires a load only if the value it read remains valid at retirement time [14? ].

The processor has a fixed number, $S$, of store buffer entries. However, this does not automatically imply that it implements a TSO[$S$] memory model, because the processor’s store buffer is not equivalent to the store buffer of the abstract TSO[$S$] machine. To show that the processor implements TSO[$S$], we need to show that a load instruction cannot be reordered with more than $S$ prior stores.

The reason for such bounded reordering is the implementation of store buffering, which assigns a store buffer entry to a store when it enters the pipeline and prevents it from entering the pipeline if the store buffer is full (i.e., all $S$ entries have not been written to memory) [4, 14, 15]. In such a case, the entire execution stalls since later instructions also cannot enter the pipeline as pipeline entry occurs in program order.

It thus appears that a load cannot be reordered past more than $S$ prior stores, conforming to the TSO[$S$] model. However, if the processor coalesces multiple stores to the same location into one store buffer entry, then the $S$ store buffer entries will represent more than $S$ stores and violate this reasoning. We ignore this issue for now and address it in § 6.6.3.

6.6.2 Measuring store buffer capacity

This section shows how to empirically determine the capacity of the processor’s store buffer. The idea is to measure the time it takes to complete sequences of stores of increasing length, and find the spot at which execution starts to stall.

**Measurement algorithm (Figure 6.6)** We alternate between issuing a sequence of stores and a sequence of non-memory instructions whose execution latency is long enough to drain the store buffer. As long as the length of the store sequence does not exceed the store buffer capacity, both the execution of the stores and the flushing of their store buffer entries occur in parallel to the execution of the non-memory instructions, due to out-of-order execution. Consequently, the latency of the non-memory instruction sequence dominates the entire execution time.

However, when the length of the store sequence exceeds the store buffer capacity, the resulting stalls delay the subsequent instruction sequences from entering the pipeline and increase execution time. Importantly, stalls in the $k + 1$-th iteration do not overlap the execution of the
non-memory instructions in the $k$-th iteration, and so the stalls are not absorbed by the latency of the non-memory instructions and affect every iteration. The reason is that stores in the $k + 1$-th sequence can start draining to memory only after all non-memory instructions in the $k$-th sequence retire, since store buffer entries are flushed post-retirement and instructions retire in program order.

Figure 6.7 shows the results of running the measurement algorithm on the Westmere processor. Our measurement results match Intel’s documented store buffer capacity for this processor [14, 15]. Measurement results on a Haswell processor are similarly accurate, correctly identifying the documented capacity of 42 [14, 15].
Initially: initialize work stealing queue with 512 items.

```
worker
taken := 0
while (take() ≠ EMPTY) {
taken := taken + 1
store to location #1
store to location #2
...
store to location #L
}

thief
stolen := 0
while (steal() ≠ ABORT) {
stolen := stolen + 1
}
```

Finally:
if (taken + stolen = 512)
output CORRECT
else
output INCORRECT

Figure 6.8: Program for finding executions not valid under a bounded TSO model. We vary the δ parameter of the work stealing algorithm and L, the number of stores performed by the worker.

6.6.3 From store buffer capacity to a reordering bound

Knowing the capacity of the processor’s store buffer does not necessarily provide a bound on store/load reordering. For example, if the processor coalesces multiple stores to the same location into a single store buffer entry, the S store buffer entries can represent more than S stores and thus allow reordering beyond S prior stores. There may also be other implementation issues that affect the reordering bound.

Therefore, we develop a litmus test that, given a presumed bound S on the reordering, can be used to prove that the processor violates the TSO[S] model. We can then gain confidence that a processor implements TSO[S] if extensive testing does not show a violation of the model, although testing will never prove that the processor implements TSO[S].

The test program (Figure 6.8) runs a worker and thief who concurrently try to empty a non-deterministic Chase-Lev variant queue (Section 6.4.1) initialized with 512 elements. The queue uses a user-supplied value for δ (the maximum number of take()’s not visible in memory due to store buffering). The worker performs a sequence of L stores to distinct locations between each invocation of take(), where L is a program parameter. The worker performs no other stores except for these and the single store in take(). Therefore, if the processor implements TSO[S], an execution with \( \delta \geq \left\lceil \frac{S}{L+1} \right\rceil \) should always be correct – the total number of tasks removed by the worker and the thief equals the initial size of the queue.

We run the litmus test using various values of L and δ. For each pair \((L, \delta)\), we perform \(10^7\)
runs with each of the following assignments of threads to cores: (1) default OS placement, (2) both threads placed on hyperthreads of the same core, and (3) each thread placed on a different core.

We interpret the litmus test results in light of a supposed ordering bound \( S \). For each \( \alpha \in \{1, \ldots, S\} \) we consider all pairs \((L, \delta)\) such that \(\left\lceil \frac{S}{L+1} \right\rceil = \alpha\). If any run of such a pair is incorrect, we consider the point \((\alpha, \delta)\) incorrect; otherwise, we consider it correct. Figure 6.9a depicts the results when \( S = 32 \), the processor’s store buffer capacity. Interestingly, the processor fails to implement TSO\([32]\), as demonstrated by the incorrect executions when the maximum number
of stores between `take()` operations divides 32.

We instrumented the litmus test to understand these failures, and observed that when they occur there always appear to be 33 worker stores in flight and never more. Therefore, we analyzed the litmus test data using $S = 33$ (Figure 6.9b). This time there is an almost perfect match with the model, with one exception: when $L = 0$. In this case the only stores the worker performs are to the tail of the work stealing queue, $T$. The $T$ variable is then always at the tail of the store buffer, and in fact is the only store pending in the store buffer. In such a case, the processor apparently coalesces stores.

**Store buffer coalescing** Coalescing under TSO can take place only for consecutive stores, otherwise – as the following example shows – the TSO guarantees may be violated:

<table>
<thead>
<tr>
<th>Initially $A = B = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffered [ A := 1 ]  \text{(earliest store)}</td>
</tr>
<tr>
<td>stores: [ B := 1 ]</td>
</tr>
<tr>
<td>$A := 2$ \text{(latest store, entering buffer)}</td>
</tr>
</tbody>
</table>

If next $A := 2$ is coalesced with $A := 1$ and this entry is then written to memory, another processor can now observe $A = 2$ while $B = 0$ which is illegal under TSO.

To understand the processor’s coalescing implementation, we repeat the store buffer capacity benchmark (Figure 6.6) using sequences of stores to the same location – and still obtain the same results. This shows that coalesced stores still get assigned distinct store buffer entries, so coalescing is done at a later execution stage. We hypothesize that the oldest store gets written to memory by first moving its address and data into an additional buffer $B$, which frees its store buffer entry, and then writing $B$ to the memory subsystem. If the oldest store is to the same address that $B$ holds, $B$ is overwritten with its data, resulting in coalescing. Otherwise, $B$ is overwritten only after its data is written to memory.

**Deriving the reordering bound of 33** Because $B$ holds a retired store’s data, it observably behaves as an additional store buffer entry. Therefore, if we avoid store buffer coalescing, we can assume we are running on a TSO[33] machine (since our concern is the observable store buffer capacity, i.e., the bound on the reordering). To avoid coalescing of stores by `take()` in work stealing, we need to prevent consecutive invocations of `take()` with no store in between.
This is easy to do in practice: The CilkPlus \texttt{take()} already writes to the structure of a removed task before returning it, and thus avoids coalescing. Other runtimes can similarly perform an extra store before returning from \texttt{take()}. Performing an additional store generalizes to processors with coarser coalescing granularity: for example, if a processor coalesces stores to the same cache line, we need to write to another cache line before returning from \texttt{take()}.

\subsection*{6.7 Evaluation}

This section evaluates the performance impact of applying our techniques in the THE and Chase-Lev work stealing algorithms.

In addition, we compare our techniques to the idempotent work stealing queues of Michael et al. \cite{93}, which avoid the worker’s memory fence at the cost of relaxing the queue’s safety by allowing a task to be removed more than once. Thus, we seek to understand whether comparable performance can be obtained without compromising safety.

\textbf{Platform}  
We run our experiments on the Haswell and Westmere processors.

\subsubsection*{6.7.1 The THEP algorithm}

We implement our techniques in Intel’s CilkPlus runtime library (Build 3365, released on May 2013) which uses the THE algorithm.

To understand the individual contribution of our proposals, we evaluate both THEP and the weaker variant which refuses to steal in case of uncertainty, referred to NonDet from here on. By default, both versions use a value of $\delta$ derived from the fact that the CilkPlus runtime performs two stores during a \texttt{take()}. To measure the impact of $\delta$, we analyzed the binaries and determined that due to program stores and compiler register spills, using $\delta = 4$ is also safe. Finally, we also benchmark a variant of THEP in which a thief always waits for the worker and does not rely on the approximation at all (i.e., $\delta = \infty$).

\textbf{Methodology}  
We measure the running time of a set of 11 CilkPlus programs (listed in Table 6.1), which have by now become standard benchmarks in the literature on work stealing and task parallel runtimes \cite{16, 55, 57, 77, 84}. We use the \texttt{jemalloc} \cite{52} memory allocator to prevent program memory allocation from being a bottleneck (the runtime uses its own memory allocator). Both runtime and programs are compiled version 13.1.1 of Intel’s \texttt{icc} compiler. We run each program 10 times (except for \texttt{knapsack}, which we run 50 times) and report the
CHAPTER 6. FENCE-FREE WORK STEALING ON BOUNDED TSO PROCESSORS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>Recursive Fibonacci</td>
<td>42</td>
</tr>
<tr>
<td>Jacobi</td>
<td>Iterative mesh relaxation</td>
<td>1024 × 1024</td>
</tr>
<tr>
<td>QuickSort</td>
<td>Recursive QuickSort</td>
<td>$10^8$</td>
</tr>
<tr>
<td>Matmul</td>
<td>Matrix multiply</td>
<td>1024 × 1024</td>
</tr>
<tr>
<td>Integrate</td>
<td>Recursively calculate area</td>
<td>10000</td>
</tr>
<tr>
<td></td>
<td>under a curve</td>
<td></td>
</tr>
<tr>
<td>knapsack</td>
<td>Recursive branch-and-bound</td>
<td>32 items</td>
</tr>
<tr>
<td></td>
<td>knapsack solver</td>
<td></td>
</tr>
<tr>
<td>cholesky</td>
<td>Cholesky factorization</td>
<td>4000 × 4000,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40000 nonzeros</td>
</tr>
<tr>
<td>Heat</td>
<td>Heat diffusion simulation</td>
<td>4096 × 1024</td>
</tr>
<tr>
<td>LUD</td>
<td>LU decomposition</td>
<td>1024 × 1024</td>
</tr>
<tr>
<td>strassen</td>
<td>Strassen matrix multiply</td>
<td>4096 × 4096</td>
</tr>
<tr>
<td>fft</td>
<td>Fast Fourier transform</td>
<td>$2^{26}$</td>
</tr>
</tbody>
</table>

Table 6.1: CilkPlus benchmark applications.

median run time, normalized to the default CilkPlus run time, as well as 10-th and 90-th percentiles.

Results  Figures 6.10–6.11 show the results using the maximum level of parallelism on each platform without hyperthreading (i.e., 10 threads, one per core, on Westmere, and 4 threads on Haswell).

No hyperthreading: Our fence-free THEP variant improves the run time on 8 of the benchmarks on the Westmere by up to 23% and by 11% on (geometric) average, and degrades the remaining 3 programs by 3%. (Overall, the improvement is by 7%.) On the Haswell, the run time of 9 programs improves by up to 23% (13% on average) and is not affected on the rest. (Overall, the improvement is by 11%.)

Varying $\delta$ does not significantly impact the performance of the THEP variants on most programs, as stealing is infrequent enough that waiting for the worker does not make much of a difference. However, for Heat on Westmere, introducing more stealing opportunities resolves the run time degradation caused by THEP.

In contrast, the NonDet variant is very sensitive to $\delta$. On 6 programs the default $\delta$ prevents stealing altogether and makes the programs run at single threaded speed. Decreasing $\delta$ to a more precise value resolves the problem in all programs but LUD, showing that the ability to
resolve a thief’s uncertainty can be important.

With hyperthreading (Figures 6.12–6.13): When using hyperthreading the processor can schedule one hyperthread when its sibling stalls due to a memory fence. Consequentially, the improvements from avoiding fence are somewhat reduced. On the Westmere, the mean improvement of programs that improve drops to 4% and is at most 12%, while the degradation increases to 5% yielding an overall improvement of 3.6% on average. The effect on the Haswell is similar, with overall improvement now being 7% (at most 12%).

6.7.2 Chase-Lev vs. idempotent work stealing queues

Here we compare Michael et al.’s LIFO and double-ended FIFO idempotent work stealing queues [93] to our variant of the Chase-Lev algorithm. The LIFO queue is a stack, with both worker and thieves removing (possibly the same) tasks from the top of the stack. The double-
ended FIFO is similar to the task queues discussed so far, but the last task can be removed concurrently by both the worker and a thief.
Figure 6.15: Transitive closure run time, normalized to Chase-Lev algorithm, on Haswell.

Figure 6.16: Percent of stolen work in transitive closure, on Haswell.

Figure 6.17: Percent of stolen work in transitive closure, on Westmere.

We use Michael et al.’s benchmark programs and inputs, but we implement the idempotent task queues ourselves as their code is not available. There are two benchmarks, computing the transitive closure and the spanning tree of a graph. The parallel algorithms used [44] man-
age synchronization internally, because the same task (e.g., “visit node $u$”) can inherently be repeated by different threads (e.g., who are working on different neighbors of $u$).

We use the following input graphs: (1) a $K$-graph, which is a $k$-regular graph in which each node is connected to its $K$ neighbors, (2) a random graph of $n$ nodes and $m$ edges, and (3) a two-dimensional torus (grid).

We run the programs 10 times on each input, using the maximum level of parallelism usable by the workload, both with and without hyperthreading. For the $K$-graph and random graph, this is the maximum parallelism in the machine (10 or 20 on the Westmere and 4 or 8 on the Haswell). However, on the torus graph the programs do not scale past 2 threads, so we report results from 2 threads (running on different cores or on the same core).

Figures 6.14 and 6.15 depict the results for the transitive closure application with large inputs; results for spanning tree and smaller inputs are similar. As before, we report median, 10-th and 90-th percentile run times, normalized to the standard Chase-Lev algorithm.

All the fence-free work stealing queues obtain comparable performance. Without hyper-threading our Chase-Lev variant is the only task queue that improves the run time on the random and $K$-graph inputs on the Westmere, though by only $\approx 2\%$. The torus input enjoys the greatest improvement in running time, $\approx 33\%$ for both our Chase-Lev variant and the LIFO idempotent queue. With hyperthreading, the improvement degrades, though it is still close to 20% for the torus.

Even though the fence-free Chase-Lev does not resolve a thief’s uncertainty by waiting for the worker, here the default $\delta$ value does not prevent a thread from stealing, in contrast to the CilkPlus experiments. This can be seen by observing the non-zero percent of stolen work for our variant in Figures 6.16–6.17.

Also apparent in these figures is that the vast majority of work is performed by the worker and not by thieves, thus emphasizing the importance of removing overhead from the worker’s code paths.

### 6.8 Related work

Fence-free work stealing task queues  Michael et al.’s idempotent work stealing algorithms [93] avoid worker fences, but are only applicable to applications that can tolerate a task being executed twice. Our technique is relevant for any application on a TSO processor. Kumar et al. [77] use yieldpoint mechanisms to stop the worker at a known-safe location be-
fore stealing. Such mechanisms are not available in unmanaged environments such as C/C++, whereas our technique applies there. Gidron et al. [59] force a store buffer flush by binding the thief to the worker’s core, thereby displacing the worker from the CPU. However, their technique requires that the worker can only run on a specific core, which is not true in general.

**Eliminating fence penalty in hardware** Below we describe microarchitectural designs that eliminate the penalty of fences, thereby obviating the need for our fence-free techniques. This work is orthogonal to our approach, because in contrast to all these proposals, our algorithms offer an *immediately usable software-only solution* for mainstream multicore architectures available today.

**Speculative memory fences** Store-wait-free processing [122] and Invisifence [33] use speculation to eliminate the penalty of memory fences. Instead of a fence stalling the processor until all prior stores are written to memory, these designs initiate transactional memory [68] style speculative execution which commits when all prior stores have been drained to memory. However, this speculation may interact badly with work stealing, as each time a thief reads the queue’s tail it might abort the worker’s speculative execution, which can contain several `take()`s.

**Stalling fences only when needed** In WeeFence [51] and address-aware fences [85] a fence stalls the processor only if a post-fence access is about to violate the memory model. Thus, a steal attempt can stall the worker, whereas with our techniques – which appear to be applicable to these designs when fences are not used – steals do not affect the worker.

**Multiple store buffers** Singh et al. [115] propose to use different store buffers for shared and private memory locations. Fences then only need to drain the shared-location store buffer. However, the processor needs to distinguish between private and shared accesses, which requires compiler and instruction set changes, or extending the hardware memory management unit and page table structures [115].
Chapter 7

Overcoming serialization in hardware lock elision

This chapter addresses performance problems in hardware lock elision (HLE), using Intel’s Haswell HLE as a study vehicle.

Section 7.1 provides background on Haswell’s transactional memory mechanism. Section 7.2 analyzes the performance dynamics of HLE and quantifies the impact of the avalanche effect on it. Section 7.3 describes how to mitigate HLE’s avalanche effect using software-assisted lock removal, and Section 7.4 solves the avalanche effect using conflict management, without resorting to local removal. We evaluate our proposal in Section 7.5. We conclude with a discussion of related work in Section 7.6.

7.1 Transactional synchronization extensions

In this background section we describe the specification of Intel’s transactional synchronization extensions (TSX) [12], the commercial name for Intel’s TM. We then describe the initial implementation of this specification in the Haswell processor.

A memory transaction is an instruction sequence whose memory accesses appear as if they were executed atomically at some point in time, i.e., a transaction appears to execute instantaneously without observing any concurrent memory updates.

The set of cache lines read/written by a transaction is called its read/write set. A conflict occurs if another processor reads or writes to a cache line in the transaction’s write set, or writes to a cache line in the transaction’s read set.

During its execution the processor may abort a transaction. (For example, due to a data
conflict). Upon an abort the processor rolls back the transactional execution, restoring the processor’s state to the point before the transaction started (including discarding any memory updates it performed). Execution then continues non-transactionally. TSX makes no progress guarantees and is allowed to abort a transaction even if no conflict occurs.

TSX defines two interfaces to designate the scope of a memory transaction, i.e., its beginning and end:

**Hardware lock elision (HLE):** In HLE, the scope of a lock-protected critical section defines a transaction’s scope. HLE is implemented as a backward-compatible instruction set extension of two new prefixes, XACQUIRE and XRELEASE. Upon executing an instruction writing to memory (e.g., a store or compare-and-swap) which is prefixed by XACQUIRE (see Figure 1.3), the processor starts a transaction and elides the actual store, treating it as a transactional read instead (i.e., placing the lock cache line in the read set). Internally, however, the processor maintains an illusion that the lock was acquired: if the transaction reads the lock, it sees the value stored locally by the XACQUIRE-prefixed instruction. Upon executing an XRELEASE store, the transaction commits. HLE requires that an XRELEASE store restores the lock to its original state; otherwise, it aborts the transaction. If an HLE transaction aborts, the XACQUIRE store is re-executed non-transactionally to acquire the lock in order to ensure progress. Notice that such a non-transactional store conflicts with every concurrent HLE transaction eliding the same lock, since such a transaction has the lock’s cache line in its read set.

**Restricted transactional memory (RTM):** RTM provides a generic TM interface with three new instructions: XBEGIN, XEND, and XABORT. XBEGIN begins a transaction, XEND commits and XABORT allows a transaction to abort itself. The XBEGIN instruction takes an operand that points to fall-back code which is executed if the transaction aborts. Upon an abort the processor writes an abort status to one of the registers, which the fall-back code can then consult. For example, the abort status indicates whether the abort was due to an XABORT, a data conflict, or an “internal buffer overflow” [12].

RTM can be used to implement custom lock elision algorithms [14] by replacing the lock acquisition code with custom code that begins a transaction and reads from the lock’s cache line. However, such a lock elision scheme fails to maintain the illusion that the thread wrote to the lock, as the lock’s cacheline is indistinguishable from any other line in the read-set.
7.1.1 Haswell’s TSX implementation

Various statements in Intel’s documentation shed light on Haswell’s TM implementation [14]. Of importance to us is that Haswell appears to use a requestor wins conflict management policy. If a coherency message (read or write) arrives for a cache line in the write set, the transaction aborts. Similarly, if an eviction due to a write arrives for a cache line in the read set, the transaction aborts.

In addition, experiments we conduct (detailed below) show that transactions are prone to spurious aborts that are not explained by data conflicts or read/write set overflow. The existence of spurious aborts is a serious issue because, as the next section explains, an abort can negatively impact performance on HLE executions. Spurious aborts imply that even in a perfect conflict free workload, such degradation is possible.

The remainder of this section expands on the aforementioned implementation details.

Read and write set tracking According to Intel’s manuals, “the processor tracks both the read-set addresses and the write-set addresses in the first level data cache (L1 cache) of the processor.” Therefore, running out of cache space (due to capacity or associativity) can cause a transactional abort.

Conflict management Quoting the manual:

- “Data conflicts are detected through the cache coherence protocol.”
- “Data conflicts cause transactional aborts.”
- “The thread that detects the data conflict will transactionally abort.”

This amounts to a requestor wins policy. If a coherency message (read or write) arrives for a cache line in the write set, the transaction aborts. Similarly, if an eviction due to a write arrives for a cache line in the read set, the transaction aborts.

Transactional behavior in practice To better understand transactional behavior in practice, we conduct an experiment on the Haswell machine. We run a benchmark that initiates a transaction which reads (or writes) every cache line in an array of a given size. We repeat this $10^7$ times for various sizes and collect the number of transactions that successfully commit. Figure 7.1 shows the results: It is clear that the L1 cache, whose size is 32 KB, holds the write set. Transactions that attempt to write more than 32 KB of data always abort. However, it appears
that a more sophisticated mechanism is used for tracking the read set, as transactions succeed for sizes that exceed not only the L1 cache but also the 256 KB L2 cache.

**Spurious aborts** The fact that abort probability is non-zero even when the array the benchmark accesses fits well in the L1 cache shows that transactions are prone to aborts which are not explained by data conflicts or read/write set overflow, i.e., spurious aborts. Even if these aborts are caused because a cached line accessed in the transaction gets evicted from the cache due to the activity of another process on the system, from the perspective of the transaction the resulting abort is spurious, since it is not explained by the program’s behavior.

### 7.2 Avalanche effect in Haswell HLE

In this section we experimentally quantify the serialization penalty due to transactional aborts during an HLE execution. As before, we use the Haswell machine for our experiments. Our workload consists of threads accessing a red-black tree data structure which is protected by
a single coarse-grained lock. Initially, we pick a size, \( s \), for the tree, and fill it with random elements from a domain of size \( 2^s \). Then, the threads run for a period of 3 seconds in which each thread continuously performs random insert, delete and lookup operations, according to a specified distribution. (We always use an equal rate of inserts and deletes so that on average the tree size remains close to the initial size.)

Setting the number of threads and operation mix lets us determine the conflict level, and the tree size controls the length and amount of data accessed in the critical section. Thus, a small tree with many mutating insert/delete operations implies high conflict levels. Increasing the size of the tree reduces the chance that two operations’ data accesses conflict, as the elements accessed are more sparsely distributed.

We run the benchmark 10 times. We report averages (there is little variance since we use a dedicated test machine) of the total number of operations completed as well as the following: (1) the number of successful speculative operations, \( S \), (2) the number of aborted speculative operations, \( A \), and (3) the number of operations that complete via a normal (non-speculative) execution, \( N \). Thus, the total number of operations performed is \( S + N \). As we explain below, in some lock implementations an operation can start and abort several speculation attempts before completing, so there is no formula relating \( A \) to \( S \) and \( N \).

We note that it is not possible to count aborts when using Haswell’s HLE, since with HLE an abort results in a re-issue of the XACQUIRE write, which is completely opaque to the lock implementation. Therefore, in our tests we use an equivalent lock elision mechanism based on the RTM instructions, which allows us to count aborts before re-issuing the acquiring write. We have verified that the performance differences between this lock elision mechanism and HLE are negligible.

We focus our comparison on the test-and-test-and-set (TTAS) lock and the fair MCS [88] lock. We use the MCS lock as the representative of the class of fair locks because, unlike other fair lock algorithms such as ticket locks or CLH locks, it restores the lock to its original value when running without alone, making it compatible with HLE. We have verified that the problems reported shortly also occur with ticket and CLH locks.

Figure 7.2 shows the amount of serialization caused by aborts, as a function of the tree size, for a moderate level of tree modifications (20%). In addition to the fraction of operations that complete non-speculatively (i.e., \( \frac{N}{N+S} \)), we report the amount of work required to complete an operation, i.e., \( \frac{A+N+S}{N+S} \), the number of times a thread tries to complete the critical section before succeeding.
As the figure shows, the serialization dynamics for each lock type are quite different. With an MCS lock, the benchmark executes virtually all operations non-speculatively after an initial speculative section aborts. As a result, an HLE MCS lock offers little if any speedup over a standard MCS lock, even when there is little underlying contention.

The TTAS lock, on the other hand, manages to recover from aborts. At high conflict levels
(on small trees) it requires $2 - 3.5$ attempts to complete a single operation, but nevertheless a fraction of 30% to 70% of the operations complete speculatively. As the tree size increases and conflict levels decrease, HLE shines and nearly all operations complete speculatively.

We now turn to analyze the causes for these differences.

**TTAS spinlock (Figure 1.3, and the boxed line in Figure 7.2)** The first thread to abort successfully acquires the lock non-speculatively. As for the remaining threads, we distinguish between two behaviors. First, a thread that aborts because of this lock acquisition re-executes its acquiring TAS instruction, which returns 1 because the lock is held. The thread then spins, and once it observes the lock free re-issues its XACQUIRE TAS and re-enters a speculative execution. Second, a newly arriving thread initially observes the lock as taken and spins (the pink curve in Figure 7.2 plots the fraction of operations for which this occurs). Once the thread in the critical section releases the lock, the waiting thread issues an XACQUIRE TAS as in the first case. The bottom line is that all threads are blocked from entering a speculative execution until the initial aborted thread exits the critical section, but then all the threads resume execution speculatively. The flip side of this behavior is that a thread may thus abort several times before successfully completing its operation, either speculatively or non-speculatively.

**Fair lock (represented by the MCS lock (Figure 7.3, and the circled line in Figure 7.2))** The MCS lock represents the lock as a linked list of nodes, where each node represents a thread waiting to acquire the lock. An arriving thread uses an atomic SWAP [66] to atomically append its own node to the tail of the queue, and in the pro-
cess retrieves a pointer to its predecessor in the queue. It then spins on the \textit{locked} field of its node, waiting for its predecessor to set this field to false.

Similarly to the TTAS lock, the first thread to abort acquires the lock and causes all subsequent threads to spin. In contrast to the TTAS lock, in the MCS lock spinning threads announce their presence, which leads to an avalanche effect that makes it hard to recover and re-enter speculative execution.

Consider first a thread that aborted because of the lock acquisition. The processor re-issues its acquire SWAP operation which returns the thread’s turn in the queue. The thread then spins and once its turn arrives (its predecessor sets its \textit{locked} field to false) enters the critical section \textit{non-speculatively}. Thus, a single abort causes the serialization of all concurrent critical sections, which will now execute non-speculatively.

Now consider a newly arriving thread. It executes an XACQUIRE SWAP to obtain its turn. However, it sees a state in which a lock is held and must therefore spin (in the speculative execution) waiting for the lock to be released. As a result, its speculative execution is doomed to abort: when the thread’s predecessor releases the lock, the releasing write conflicts with the reads performed in the waiting thread’s spin loop. In fact, the speculative execution may abort earlier if the spin loops issues a \texttt{PAUSE} instruction, as is often the case. In this case, as discussed above, the thread executes the critical section non-speculatively.

Essentially, because of the fairness guarantees the MCS lock provides, it “remembers” conflict events and makes it harder to resume a speculative execution. Even when the original lock holder releases the lock, it moves it into a state that does not allow new threads to speculatively execute. The MCS lock requires a \textit{quiescence period}, in which no new threads arrive, so that all waiting threads acquire the lock, execute the critical section and leave. Only then does the MCS lock return to a state that allows speculative execution.

\textbf{Performance impact} In Figure 7.4 we divide the benchmark’s execution into 1 millisecond time slots and show the throughput obtained in each window, normalized to the throughput over the entire execution. We also show the fraction of operations that completed via a non-speculative execution in each time slot.

As can be seen, the MCS lock’s throughput hovers around the average and shows little variance because the benchmark is executing with no benefit from concurrency. In contrast, when using TTAS performance can fluctuate severely, sometimes falling by as much as $2.5 \times$. These throughput drops are correlated with periods in which more critical sections finish non-
Figure 7.4: Normalized throughput and serialization dynamics over time. We divide the execution into 1 millisecond time slots. **Top:** Throughput obtained in each time slot, normalized to the average throughput over the entire execution. **Bottom:** Fraction of operations that complete non-speculatively in each time slot.

Figure 7.5: The speed-up of hardware lock elision usage with different types of locks. The base-line of each speed-up line is the standard version of that specific lock. By mixing different access operations we vary the amount of contention: (i) lookups only - no contention, (ii) moderate contention - a tenth of the tree accesses are node insertions and another tenth are node deletions and (iii) extensive contention all the accesses are either node insertion or deletion.
speculatively, i.e., after serialization caused by an abort.

Finally, Figure 7.5 depicts the performance advantage of the lock elision usage with different types of locks. As observed, MCS lock gains no benefit with HLE usage. On the other hand the TTAS lock gains performance boost while using the HLE mechanism.

7.3 Software-assisted lock removal

The problems demonstrated in the previous section arise because an HLE transaction runs with the lock in its read set. Rajwar and Goodman observed [102] that given transactional capabilities, one can simply run transactions with the same scope as the critical section without accessing the lock. Because committed transactions appear as if executed in a serial order, this results in an execution which is equivalent to executing the critical section non-transactionally.

However, simply replacing critical sections with transactions does not automatically guarantee progress. With HLE, a thread whose transaction aborts re-issues its lock acquisition in a normal manner. Even if the thread then fails to acquire the lock, a different thread does successfully acquire it. Overall, the system makes progress and avoids a livelock scenario.

Rajwar and Goodman’s solution to this problem is a hardware conflict management scheme that guarantees starvation freedom, i.e., that every transaction eventually succeeds. Unfortunately, Haswell’s TM “requestor wins” conflict management policy guarantees neither starvation freedom nor livelock freedom [34].

Our solution to this progress problem is to go back to using the lock as a progress-guaranteeing mechanism. However, this needs to be done with care: simply having an aborted thread acquire the lock (to avoid starvation) can lead to an incorrect execution:

**Example**  Consider the case of two code segments protected by the same lock $L$:

$C_1$:  
lock(L)  
load(X)  
load(Y)  
unlock(L)

$C_2$:  
lock(L)  
store(Y,1)  
store(X,1)  
unlock(L)

Suppose now that thread $T_1$ transactionally executes $C_1$ without accessing $L$ and reads $X = 0$ from memory. Now another thread, $T_2$, executes $C_2$ non-transactionally. It therefore acquires $L$
and then stores 1 to \( Y \). Following this \( T_1 \) reads \( Y \) from memory. Since \( Y \) is not in \( T_1 \)'s read set, there is no conflict with \( T_2 \)'s previous store and \( T_1 \) observes \( Y = 1. \) \( T_1 \) then commits. Thus \( T_1 \) observes an inconsistent state, \( X = 0 \) and \( Y = 1 \).

The problem exposed in the above example is due to the *non-atomicity* of a thread running in the critical section in a non-transactional manner. Memory updates that such a thread does are made globally visible one at a time, making it possible for concurrent transactions to observe inconsistent state. (Had the thread \( T_2 \) been running transactionally in the above example, \( T_1 \) would observe either both or none of \( T_2 \)'s stores.)

**Software-assisted lock removal (SALR, Figure 7.6)** We reduce the impact of the non-atomicity problem by using the lock to verify that transactions observe consistent state. A transaction executes without accessing the lock. Once it reaches the end of the critical section, it reads the lock. If the lock is free, the transaction commits; otherwise, the transaction aborts itself using XABORT. Returning to our example, Figure 7.7 shows how SALR enforces correct executions in several scenarios, though in some scenarios the transaction can observe inconsistent state. Observing inconsistent state is usually not a problem, because the TM sandboxes the transaction and prevents it from changing the program’s state unless it commits – which the lock check is supposed to prevent. Even if the transaction enters an infinite loop as a result of observing inconsistent state, the OS will eventually deschedule the running thread, which will abort the transaction. The only exception is if observing inconsistent state causes a transaction to erroneously ex-

```c
1 shared variables:
2 lock : speculative lock

4 thread local variables:
5 retries : int
6 lock() {
7   retries := 0
8   // speculative path
9   XBEGIN (Line 12) // jump to Line 12 on abort
10  return
11
12 // fallback path
13  retries ++
14  if ( retries < MAX_RETRIES)
15     goto Line 8
16  else
17     lock.lock() // standard lock acquire
18 }
19
20 unlock() {
21  if (XTEST()) { // returns TRUE if the run is speculative
22   if (lock is locked)
23     XABORT // aborts the speculative run
24   else
25     XEND
26  } else {
27    lock.unlock() // standard lock release
28  }
29 }
```

Figure 7.6: Software-Assisted Lock Removal
### 7.3. SOFTWARE-ASSISTED LOCK REMOVAL

<table>
<thead>
<tr>
<th>Scenarios ending in abort</th>
<th>Successfully committing scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>begin SALR transaction</td>
<td>begin SALR transaction</td>
</tr>
<tr>
<td>load($X$)</td>
<td>load($X$)</td>
</tr>
<tr>
<td>lock($L$)</td>
<td>lock($L$)</td>
</tr>
<tr>
<td>store($Y$)</td>
<td>store($Y$)</td>
</tr>
<tr>
<td>$L$ is locked: ABORT</td>
<td>load($X$)</td>
</tr>
<tr>
<td>$L$ is locked: ABORT</td>
<td>lock($L$)</td>
</tr>
<tr>
<td>ABORT: conflict on $X$</td>
<td>store($Y$)</td>
</tr>
<tr>
<td></td>
<td>store($X$)</td>
</tr>
</tbody>
</table>

Figure 7.7: How SALR enforces correct executions in different scenarios.

execute an XEND without correctly checking the lock – e.g., if it branches to the middle of an instruction which happens to encode an XEND, or overwrites the lock address and makes it appear unlocked, or causes an internal corruption which makes the lock checking code access the wrong memory address. As a result, SALR is not universally applicable: one has to verify that values read from memory cannot cause transactions to misbehave in such ways.

**Performance impact**  SALR alleviates the two problems that lead to the HLE avalanche effect. First, with SALR a thread that non-transactionally acquires the lock does not automatically cause all running transactions to abort. If it manages to complete the critical section before these transactions try to commit, they may successfully commit, given they do not conflict on the data. Second, while a thread is executing non-transactionally in the critical section, arriving threads continue to enter speculative transactional execution and are not forced to wait.

**Correctness**  We assume that every committed SALR transaction reads the lock before committing, i.e., executes XEND through the SALR unlock code. We show the correctness of the
SALR by totally ordering the execution of the SALR transactions and non-transactional critical sections, which proves that their execution is indistinguishable from that of a standard (non-transactional) execution of the critical section. We define the total order \( \sim \) on \( T_1, \ldots, T_k \), the successfully committed SALR transactions, and \( C_1, \ldots, C_m \), the non-transactional executions of the critical section. First, \( T_i \sim T_j \) if \( T_i \) is ordered before \( T_j \) in the global order of committed transactions. Second, \( C_i \sim C_j \) if \( C_i \) is ordered before \( C_j \) in the global order of critical sections. Third, for any \( C_i \) and \( T_j \), \( C_i \sim T_j \) if \( T_j \) checks the lock after \( C_i \) releases the lock; otherwise, \( T_j \sim C_i \). It remains to show that the \( \sim \) order respects the actual execution order, i.e., that if a critical section execution \( A \) (whether transactional or not) reads a value stored by critical section \( B \neq A \), then \( B \sim A \). There are four possible cases: (1) If \( A \) and \( B \) are both transactional, this is immediate from the correctness of the HTM. (2) If \( A \) and \( B \) are both non-transactional, this follows from the correctness of the lock implementation. (3) If \( A = T_j \) and \( B = C_i \), then \( C_i \) acquires the lock before \( T_j \)'s read. But since \( T_j \) successfully commits, it observes an unlocked lock, and so \( C_i \) releases the lock before \( T_j \) reads it. Hence \( C_i \sim T_j \). (4) If \( A = C_i \) and \( B = T_j \), then \( C_i \) holds the lock after \( T_j \) commits. Since \( T_j \) observes an unlocked lock when it commits, \( C_i \) acquires the lock after \( T_j \) commits. Thus \( T_j \sim C_i \).

### 7.4 Software-assisted conflict management

In this section we propose a conflict management scheme which serializes conflicting threads that cannot run concurrently, but does this without acquiring the lock to avoid impact on the other threads in the system. The serialized threads then re-enter the critical section one at a time and using lock elision when they re-enter, to avoid the conflicts on the lock that standard HLE creates. This scheme is compatible with any lock implementation, and resolves the avalanche problem in HLE transactions without resorting to using lock removal, which is not always the optimal solution to use.

For example, when using lock removal in highly contended workloads with lots of aborts, new threads keep starting transactions and causing more aborts and wasted work. Here HLE’s ability to grab the lock and stop newly arriving threads from entering speculative execution turns out to be extremely helpful and greatly improves performance (as shown in Section 7.5). Our conflict management scheme thus maintains this ability of HLE, and prevents the avalanche effect in less contended scenarios.

In addition, lock removal suffers from a usability drawback in that it does not provide a
7.4. SOFTWARE-ASSISTED CONFLICT MANAGEMENT

Figure 7.8: A block diagram of a run using our software scheme. The entry point of a speculative section is the ‘speculative run’ rectangle. All threads acquire the original main lock using the lock-elision mechanism. If a conflict occurs (described by ‘x’), the conflicting threads are sent to the serializing path. Once a thread acquires the auxiliary standard lock in a non-speculative manner, it rejoins the speculative run.

transaction the illusion that the lock is acquired while it runs, unlike HLE. In contrast, our conflict management scheme can in principle maintain this illusion when applied to HLE. (However, as discussed later, a technical limitation of the current hardware requires us to implement the scheme without this support.)

Though it benefits mainly HLE, the conflict management scheme is compatible with SALR as well. It can be used to further reduce any progress problems caused when SALR threads give up and acquire the lock non-transactionally. For example, with plain SALR many transactions may end up aborting if the system contains even a few livelock-prone conflicting transactions that end up non-transactionally acquiring the lock.

**Preventing livelock** Our scheme uses two locks, the original *main* lock which is taken using the HLE/SALR mechanism and an *auxiliary* standard lock which is only acquired in a standard non-transactional manner. The auxiliary lock groups all the threads that are involved in a
shared variables:
main_lock : elided main lock
aux_lock : auxiliary standard lock
	hread local variables:
retries : int
aux_lock_owner : boolean, initially FALSE

lock () {
retries := 0
// primary path
XBEGIN (Line 16) // jump to Line 16 on abort
call HLE or SALR lock() as appropriate
return

// serializing path
if (aux_lock_owner = FALSE) {
} else {
aux_lock.lock() // standard lock acquire
aux_lock_owner := TRUE
} if (retries < MAX_RETRIES)
goto 11
else
main_lock.lock() // standard lock acquire
}

unlock() {
if (XTEST()) { // returns TRUE if the run is speculative
call HLE or SALR unlock() as appropriate
XEND
} else {
main_lock.unlock() // standard lock release
}
if (aux_lock_owner = TRUE) {
aux_lock.unlock() // standard lock release
aux_lock_owner := FALSE
}
}

Figure 7.9: Software-Assisted Conflict Management

conflict and serializes them (see Figure 7.8). When a transaction is aborted, the aborted thread
non-transactionally acquires the auxiliary lock and then rejoins the speculative execution of
the original critical section. Each standard lock is associated with a unique auxiliary lock. We
refer to the process of acquiring the auxiliary lock in order to rejoin the speculative run as the
serializing path. As with previous schemes, the thread may retry its transaction before going to
the serializing path.

To see why this scheme prevents livelock, consider two transactions, $T_1$ and $T_2$, which
repeatedly abort each other. Once $T_1$ acquires the auxiliary lock and re-joins the speculative
execution, one of the following can happen: (1) $T_1$ aborts again, but $T_2$ commits, or (2) $T_2$ aborts
and thus tries to acquire the auxiliary lock, where it must wait for $T_1$ to commit. Generalizing
this, once a thread $T$ acquires the auxiliary lock any transaction that conflicts with $T$ either
commits or gets serialized to run after $T$. Thus the system makes progress.

**Preventing starvation** In the above scheme starvation remains possible due to one of two scenarios: (1) a thread fails to acquire the auxiliary lock (as can happen with a TTAS lock), or (2) a thread holding the auxiliary lock fails to commit. To solve issue (1) we require that the auxiliary lock be a starvation-free (or “fair”) lock, such as an MCS lock. Our scheme then inherits any fairness properties of the auxiliary lock. To solve issue (2), the auxiliary lock holder non-transactionally acquires the main lock after failing to commit a given number of times. If all accesses to the main lock go through the HLE/SALR mechanism, then *only* the auxiliary lock holder can ever try to acquire the main lock and is therefore guaranteed to succeed. Otherwise (i.e., if the program sometimes explicitly acquires the lock non-transactionally), the main lock must be starvation-free as well.

**Implementation and HLE compatibility (Figure 7.9)** Our scheme maintains HLE-compatibility by *nesting* an HLE transaction within an RTM transaction. When used with HLE, we first start an RTM transaction which “acquires” the lock with an XACQUIRE store. Because TSX provides a flat nesting model [12], an abort will abort the parent RTM transaction and execute the fall-back code instead of re-issuing the XACQUIRE store and aborting all the running transactions.

Unfortunately, the initial implementation of TSX in Haswell does not support nesting of HLE within RTM. Therefore, in our experiments we use RTM also to perform lock elision (by reading the lock address), which does not provide the self-illusion that the lock is taken. More precisely, in our current implementation we omit Line 31 from Figure 7.9’s unlock() implementation, and perform the following at Line 13 of Figure 7.9 (in the lock() implementation):

```c
1 // put the main_lock in the read set and check that it is free
2 if (main_lock is locked)
3  XABORT(“non-speculative_run”)
```

---

7.5 **Evaluation**

In this section we evaluate the benefit provided by our lock elision schemes using two micro-benchmarks and applications from the STAMP suite (commonly used for evaluating hardware TM implementations [99, 120]), which consists of eight applications that cover a variety of
domains and exhibit different characteristics in terms of transaction lengths, read and write set sizes and amounts of contention.

**Methodology** As in Section 7.2, we use the Haswell machine. We run the benchmarks on an otherwise idle machine using the jemalloc memory allocator which is tuned for multi-threaded programs. We evaluate our methods as applied to both the MCS lock (as a representative of the class of fair locks) and the TTAS lock. For each lock type we test the following schemes: (1) Standard (non-speculative) version of the lock, (2) HLE version of the lock (3) HLE version of the lock with conflict management (HLE-CM), (4) Pessimistic SALR version, in which a thread acquires the lock non-speculatively after one failure (Pes SALR), (5) Optimistic SALR version, in which a thread only acquires the lock non-speculatively after retrying speculatively 10 times (Opt SALR), and (6) Optimistic SALR version with conflict management applied (SALR-CM).

**Conflict management tuning** Because SALR and HLE behave differently when the main lock is taken non-speculatively, we tune the conflict management as appropriate for each technique. Taking the lock non-speculatively in an HLE-based execution has large performance impact, and so the thread holding the auxiliary lock retries to complete its operation speculatively 10 times before giving up and acquiring the main lock. In contrast, SALR is much less sensitive to the main lock being taken and so if the bits in the abort status register indicate the transaction is unlikely to succeed, we switch to a non-speculative execution. We have verified that using other tuning options only degrades the schemes’ performance.

### 7.5.1 Red-black tree micro-benchmark

We evaluate our methods using two micro-benchmarks, the red-black tree (described in Section 7.2) and a hash table. In each test, we measure the average number of operations per second (throughput) when running the benchmark 20 times on an otherwise idle machine.

The results of the two micro-benchmarks are comparable, as hash table transactions are always short and therefore “zoom in” on the short transaction portion of the red-black workload spectrum. We therefore discuss only the red-black tree.

**Red-black tree** Figure 7.10 shows the speedups obtained by the various methods on a 128-node tree under moderate contention (20% updates). It can be seen that using our scheme, the throughput scales with the number of threads. In contrast, with HLE the MCS lock does not
Figure 7.10: The execution results on a small tree size (128 nodes) under moderate contention. The two graphs are normalized to the throughput of a single thread with no locking. The software assisted schemes scale well and the performance gap between MCS and TTAS is closed.

scale at all, and even the TTAS does not scale past 4 threads. Using our methods eliminates the performance gap between MCS and TTAS.

Figures 7.11-7.12 depict the speed-up that our methods obtain (relative to the HLE version of the specific lock) across the full spectrum of workloads. Notice that increasing the tree size also increases the size of the critical section, resulting in a lower conflict probability but also lower throughput.

**TTAS lock** On the lookup only configuration, applying our methods to the TTAS lock shows no performance improvement – the HLE-based TTAS is good enough. However, as we increase the level of contention, our methods improve over the plain HLE-based TTAS by up to $3 \times$. This is the result of letting new arriving threads immediately enter the critical section speculatively, instead of waiting for the aborted thread currently in the critical section to leave.

The HLE-CM and SALR versions of TTAS give comparable performance in general, ex-
except for short transactions. There, HLE-CM outperforms SALR and SALR-CM by up to $2 \times$, exactly because of the serialization it induces (see below).

**MCS lock** Because of spurious aborts, the MCS lock experiences severe avalanche behavior even in a read-only workload. Thus, our methods increase throughput by $2 - 10 \times$ in every MCS workload. We again see comparable results for HLE-CM and SALR, with a slight advantage to HLE-CM in short transactions. The pessimistic SALR version gives comparable performance to the plain HLE-based MCS lock, and provides a little speedup in longer transactions, in contrast to its behavior in TTAS where it fails to scale and gives overall poor results.

**Analysis** To gain deeper insight into the behavior of the benchmarks, we run them with statistics turned on (at the cost of a 5-10% degradation in throughput). Figures 7.13-7.14 show the amount of serialization caused by aborts, as a function of the tree size, for a high level of tree modifications. Figure 7.13 depicts the impact of the software assisted conflict management scheme on the HLE-based MCS lock. As the conflict level decreases (as the tree size increases), the HLE-CM requires less attempts in order to complete a single operation (converges to single attempt) and the speed-up increases. The HLE-CM manages to complete very high fraction of the operation speculatively. In Figure 7.14 one can see the impact of multiple software assisted schemes on HLE-based TTAS lock. SALR enables (at least partial) speculative execution while the lock is non-speculatively taken. Yet, serializing of conflicting threads to prevent recurrence of known conflicts helps to reduce the number of aborts and eventually to increase the performance. In the highest contention part (small tree sizes) the HLE-CM performs distinctive lower number of attempts, hence gains the better speed-up.

### 7.5.2 STAMP

To apply our methods to the STAMP suite of benchmark programs [38], we replace the transactions with critical sections that all use the same global lock. Figures 7.15-7.16 show the runtime of the STAMP programs with the various lock elision methods, normalized to the execution time using the standard lock.

As with the red-black tree micro-benchmark, MCS lock gains no benefit from HLE usage. But, MCS lock gains much benefit from HLE when combined with our conflict management scheme. The HLE-CM scheme improves the performance, typically in a substantial way (up to $2.5 \times$).
On the other hand, TTAS lock gains some benefit of HLE usage (up to $2 \times$ in intruder) but the HLE-CM scheme with TTAS gives modest improvement with the exception of genome (up to $1.5 \times$). Here too we see that the benefit of HLE usage in TTAS depends on the workload’s characteristics.

Both locks gain much benefit from lock removal usage. In most of the tests, the optimistic SALR scheme gives the highest improvement (with one exception of kmeans-high MCS HLE-CM lock), sometimes up to $2 \times$ compared to the HLE-based scheme and up to $4 \times$ compare to the plain non-speculative version of the lock.

For the most part, when the conflict management scheme is used with SALR, the performance gain is negligible with only one exception. In the vacation low test, the SALR-CM gives 15% improvement over the Opt-SALR. In general, the pessimistic SALR gives modest performance gain but for the most part substantially lower than the other software assisted schemes.

**Conclusion:** One can see the impact of our software-assisted conflict management scheme on the performance of HLE-based locks in both micro-benchmarks and STAMP no matter what the contention level is. MCS lock (or any other fair lock) gains the highest performance boost since these locks need quiescence period in order to overcome the avalanche behavior and return to speculative execution.

The impact of our conflict management method on the SALR is more modest and depends on the characteristics of the execution (such as transaction length and contention level). Yet, one can distinct the performance enhancement of conflict management in some of the STAMP application.

### 7.6 Related work

Rajwar and Goodman [101] introduced the concept of speculative lock elision (SLE). They subsequently proposed transactional lock removal [102], which uses hardware-based conflict management to serialize conflicting transactions. Our approaches achieves a similar goal, but using software to assist the hardware implementation.

Dice et al. [48] studied transactional lock elision (TLE) using Sun’s Rock processor. They point out a lemming effect which is similar to the avalanche behavior and sketch a software mechanism to speed-up recovery from the lemming effect. In contrast, our solutions prevent the lemming effect from occurring in the first place, instead of recovering from it once it happens.
Furthermore, Dice et al. sketch their solution without evaluating it, whereas we implement our solutions and evaluate them on commercial hardware.

Additionally, we address the problem of a thread arriving at the critical section while the lock is taken, which is an additional issue that prevents potential concurrency from materializing in HLE and is distinct from the lemming effect.

Bahar et al. [99] analyze both lock elision and lock removal schemes in the context of embedded systems with hardware TM. They discuss hardware approaches, whereas we are interested in what can be done using software to assist the hardware TM.
Figure 7.11: The speed-up of software lock elision schemes compared to Haswell HLE. The base-line of each speed-up line is the HLE version of the TTAS lock.
Figure 7.12: The speed-up of our generic software lock elision schemes compared to Haswell HLE. The base-line of each speed-up line is the HLE version of the MCS lock.
Figure 7.13: Impact of software assisted conflict management for MCS lock executions. For each tree size we show the average number of times a thread attempts to execute the critical section until successfully completing a tree operation, and the fraction of operations that complete non-speculatively.
Figure 7.14: Impact of software assisted conflict management for TTAS lock executions. For each tree size we show the average number of times a thread attempts to execute the critical section until successfully completing a tree operation, and the fraction of operations that complete non-speculatively.
7.6. RELATED WORK

Figure 7.15: Normalized run time of STAMP applications using a TTAS lock, HLE, and software-assisted methods.
CHAPTER 7. OVERCOMING SERIALIZATION IN HARDWARE LOCK ELISION

Figure 7.16: Normalized run time of STAMP applications using an MCS lock, HLE, and software-assisted methods.
Chapter 8

Cache index-aware memory allocation

This chapter demonstrates that many existing state-of-the-art memory allocators are index-oblivious, admitting performance pathologies for certain allocation sizes. We then describe a simple methodology for adjusting the allocator to yield an index-aware allocator with better index coverage, which in turn reduces the superfluous conflict miss rate in various applications and improves performance.

Section 8.1 provides background on modern memory allocator design. Section 8.2 describes the index-obliviousness problem and its potential for creating performance problems. Section 8.3 describes our solution methodology, and Section 8.4 details our implementation of this methodology within an existing memory allocator. Section 8.5 illustrates the index-obliviousness problem in several applications, and shows how our allocator mitigates the problem. Section 8.6 discusses related work.

8.1 Background: modern memory allocator design

Modern state-of-the-art allocators include Hoard [30], CLFMalloc [91], LFMalloc [46], libumem [35], jemalloc [52] and tcmalloc [10]. They are broadly categorized as segregated free list [73] allocators as they maintain distinct free lists based on block size. Such allocators round requested allocation sizes up to the nearest size-class where a size-class is simply an interval of block sizes and without ambiguity we can refer to a size-class by its upper bound. The set of size-classes forms a partition on the set of possible allocation sizes. The choice of size-classes is largely arbitrary and defined at the whim of the implementor, although a step size of \(1.2 \times\) between adjacent size-classes is common [30] as the worst-case internal fragmentation is constrained to 20%. 

135
We will use Hoard as a representative example of modern allocator design. Hoard uses multiple heaps to reduce contention. Specifically, Hoard attempts to diffuse contention and improve scalability by satisfying potentially concurrent malloc requests from multiple local heaps – this strategy also mitigates the allocator-induced false sharing problem.

Each heap consists of an array of references to superblocks, with one slot for each possible size-class. A superblock is simply an array of blocks of a certain size class. Superblocks are all the same size, a multiple of the system page size, and are allocated from the system via the mmap interface which allocates virtual address pages and associates physical pages to those addresses. Mmap is used instead of the more traditional sbrk operator as pages allocated through mmap may later be returned to the system, if desired, through munmap.

The superblock is the fundamental unit of allocation for Hoard. Each superblock has a local singly-linked free list threaded through the free blocks and maintained in LIFO order to promote TLB and data cache locality. A small superblock header at the base of the array contains the head of the superblock-local free list.

The Hoard implementation places superblocks on highly aligned addresses. The free operator then uses address arithmetic – simple masking – on the block address to locate the header of the enclosing superblock, which in turn allows the operator to quickly push the block onto the superblock’s free list. As such, in-use blocks do not require a header field.

If a superblock becomes depleted it can be detached from a heap and moved to a global heap. The local heap can be reprovisioned from either the global heap, assuming a superblock with sufficient free space is available, or by allocating a new superblock from the system. Superblocks can circulate between various local heaps and the global heap, but will be associated with at most one local heap at any one time.

Hoard’s malloc operator first quantizes the requested size to an appropriate size-class, identifies a heap (by hashing the identity of the current thread), locks the heap, locates a superblock of the appropriate size-class in that heap, unlinks a block from that superblock’s free list, unlocks the heap, and finally returns the address of the block’s data area.

Additional malloc implementations CLFMalloc is structurally similar to Hoard, differing mostly in the policy by which it associates malloc requests with heap instances and in that CLFMalloc is lock-free.

Libumem and tcmalloc use a central heap but diffuse contention via multiple local free lists. In the case of tcmalloc the central heap uses segregated free lists which are populated
8.2. THE INDEX-OBLIVIOUS ALLOCATION PROBLEM

by allocating runs of pages and then splitting those pages into contiguous arrays of the desired size-class.

The default Solaris libc allocator uses a single global heap protected by one mutex. Memory is allocated from the operating system by means of the sbrk system call. The global free list is organized as a splay tree ordered by size and allocation requests are serviced via a best-fit policy. The heap is augmented by a small set of segregated free lists of bounded capacity, allowing many common requests to operate in constant-time.

8.2 The index-oblivious allocation problem

Background Modern CPUs have set-associative caches, which can be envisioned as an array of several rows, each of which consists of a small set of cache lines. The cache row in which data from a memory location is placed at is determined by using a subset of the bits in the location’s address as an index that specifies which row to use. Thus, if there are \( r \) rows in the cache, \( \lceil \log_2 r \rceil \) bits of the address are used. These bits are typically the bits following the least significant \( \lceil \log_2 L \rceil \) bits of the address (where \( L \) is the size of a cache line in bytes), which are used to index a memory address inside the cache line itself. We describe the Niagara’s L1 data cache as an example. This is an 8 KB cache that is 4-way set associative with 16-byte lines. It thus has \( 128 = \frac{8192}{4} / 16 \) indexable rows. A physical address maps to a cache row index by shifting the address 4 bits to the right \( (L = 16) \) and then masking off the low 7 bits to form the index \( (r = 128) \).

Index-obliviousness An allocator in which allocation requests are satisfied from an underlying array, as in the typical implementation of a segregated free list allocator, can easily result in patterns of block addresses that map to only a few cache indices. For example, suppose such an allocator defines a size class with blocks of size \( 2L \), where \( L \) is the cache line size. Then returned blocks will map only to even cache lines, covering half the cache. This can lead to inter-block cache index conflicts and excessive conflict misses if a group of blocks of a size-class are accessed frequently by the application.

This section shows that popular allocators indeed suffer from such index-oblivious size-class selection, and that index-oblivious block placement can result in performance pathologies, both in single- and multi-threaded executions.

We illustrate the index-obliviousness of modern allocators using mcache, a simple single-threaded benchmark program. The program mallocs 256 blocks of size \( B \) bytes and then re-
ports the cache index of the base address for each of the blocks.

We run mcache on the Niagara machine, and report the results in Figure 8.1. Each point in the graph represents a distinct run of mcache. On the X-axis we vary the block size $B$ with a step of 16 bytes. The Y-axis values are the number of distinct Niagara L1D indices on which those blocks were placed, reflecting cache index distribution. A value of 128 – the number of L1D indices – is ideal. (Other more descriptive statistics might better reflect index distribution, such as a histogram, standard deviation or spread between maximum and minimum of the index population, but a simple count of the number of distinct indices serves to illustrate our assertion that many allocators have non-uniform index distribution.)

The various allocators were configured by way of the LD_PRELOAD dynamic linking fa-
8.2. THE INDEX-OBLIVIOUS ALLOCATION PROBLEM

cility. Libumem and libc are provided with Solaris. Hoard version 3.8 was obtained from [6] and CLFMalloc version 0.5.3 was obtained from [7]. We used jemalloc version 2.0.1 and tcmalloc version 1.6 in the tcmalloc-minimal configuration without call-site profiling. Where SPARC executables were not available, source code was compiled with gcc version 4.4.1 at optimization level -O3. Unless otherwise noted all data in this chapter was collected with 32-bit programs under the Solaris 10 operating system on the Niagara machine.

As can be seen in Figure 8.1, all of the allocators have one or more size values where blocks fall on only a fraction of the 128 possible indices, potentially limiting the performance of an application that repeatedly accesses a few “hot” fields (fields exhibiting strong temporal locality) in a set of such blocks.

The same experiment on the Westmere system revealed index imbalance under the default libc allocator, itself based on Lea’s dlmalloc [9], although the situation was not as dire for mid-sized blocks as the cache has higher associativity (see Section 3.1).

8.2.1 Impact of index-obliviousness allocation

Placing blocks on a restricted fraction of the available cache indices, as an index-oblivious allocator does, can artificially and needlessly increase the conflict miss rate. To see this, consider an application that allocates a set of blocks and then repeatedly accesses just a few fields in a group of blocks (e.g., graph traversal). In other words, we assume temporal locality for blocks and temporal and spatial locality within individual blocks. Such an access pattern is not atypical. Bonwick et al. [35] calls out the kernel inode construct as an example. The pattern is common in object graphs with intrusive linkage where the linkage fields reside in a “header” that precedes the body of the object. However, such an application may suffer excessive conflict misses as some cache indices are “hot” and others underutilized and “cold”.

The following sections demonstrate the performance bottlenecks caused by index-oblivious allocation, in both single- and multi-threaded execution.

8.2.1.1 Increased conflict misses in single-threaded execution

In Figure 8.2 we configure the mcache program so that the 256 blocks are configured in a ring by ascending virtual address. The first field in the block contains a pointer to the next block in the ring. The remainder of the block is not accessed during the run. Our program runs for 10 seconds, traversing the ring and then reports the number of steps per millisecond on the Y-axis. Again, we vary the block size on the X-axis. The only activity during the measurement interval
is “pointer chasing” over the ring of allocated nodes.

As can be seen, block placement greatly impacts performance. Note that we selected 256 blocks intentionally, as the L1D can contain 512 distinct lines and, ideally, with uniform index distribution, could accommodate all 256 block headers in the cache without incurring any cache misses. (Each Niagara core has a 128-way fully associative data TLB and thus more than sufficient capacity to cover the blocks for a reasonable heap layout without incurring TLB misses.) As expected, when we collect CPU performance counter data when running mcache under the various allocators we see that the L1D miss rate correlates strongly with the performance reported by the application, supporting our claim that the slow-down, when present, arises from cache misses.

8.2.1.2 Contention in shared caches

The advent of shared caches in a multicore architecture means that index-oblivious allocation impacts multi-threaded and multi-process execution.
Assume for instance that threads $T_1$ and $T_2$ run concurrently on the same core and share the L1D. Both threads `malloc(100)` immediately after they start. Each thread will typically access distinct CPU-private heaps and within those heaps, superblock instances of the size-class appropriate for 100 bytes. The superblocks will be instantiated via `mmap` which returns addresses that will be at least page-aligned and in practice often have much higher alignment. Thus, if the allocator creates the superblock at the address returned from `mmap` it is very likely that the blocks returned from the `malloc` requests by $T_1$ and $T_2$ will collide at the same cache indices.

With shared caches, applications can also encounter inter-process index conflicts where different processes have threads running concurrently on the same core.

We illustrate this problem using a multi-threaded version of `mcache`. Each thread `malloc`s two blocks of 100 bytes at startup and configures them as a ring via intrusive “next” pointers. (The choice of size is largely irrelevant in this benchmark). All the threads are completely independent. During the 10 second measurement interval each thread iterates over its private ring.

![Figure 8.3: Impact of superblock index-obliviousness on memory access scalability in multi-threaded execution.](image-url)
visiting the two nodes in turn. When finished, the program reports the aggregate throughput rate of the threads.

Figure 8.3 reports the throughput rate on the Y-axis in steps per millisecond while varying the number of threads on the X-axis. In an ideal system we would see perfect linear scaling but our real system has shared resources such as the pipeline (2 per core), caches, memory channel, etc. [119]. Beyond 8 threads, assuming ideal dispersion of those threads by the scheduler [45], threads start sharing the L1D cache. At 32 threads we have 4 threads per core. Recall that the L1D is 4-way set associative, so above 32 threads index collisions start to manifest as misses and impede scaling, even to the extent of actually reducing performance in some cases.

As we can see mcache scales reasonably up to 32 threads under all the allocators. Beyond 32 threads we see that performance bifurcates: we still find reasonable scaling under libc, tcmalloc, and libumem, while under Hoard, CLFMalloc, jemalloc we find that scaling fades. libc, tcmalloc and libumem are not vulnerable as the blocks distributed to the various threads come from a centralized heap instead of per-thread mmap-ed heaps.

In fact, jemalloc originally suffered from an even more severe index-oblivious issue. We observed an interesting performance phenomenon where access performance dropped precipitously under jemalloc at thread counts above 32. We observed that only a fraction of the currently executing threads were afflicted, and those threads suffered extremely high level-2 cache (L2) miss rates. Investigation revealed that jemalloc requests memory in units of 4MB chunks via mmap – each thread that invokes malloc will have at least one such thread-private 4MB region. 4MB happens to precisely coincide with a large page size on our platform. Indeed, the pmap -s command confirmed that Solaris was placing those 4MB regions on 4MB pages.

Because of this, when a homogeneous set of threads invoke malloc they obtain addresses that are the same offset from the base of their 4MB block. 4MB pages must start on 4MB physical address boundaries. Thus the physical addresses underlying the 4MB blocks are extremely regular, differing in only a small number of bits between threads. The set of addresses returned by malloc to the threads thus tend to conflict as they select only a small set of the possible L2 banks and L2 indices, resulting in conflict misses in the L2.

The Niagara applies an XOR-based hash to physical addresses to avoid such behavior, but in our case the physical addresses were so regular that the hash did not avoid the problem. We confirmed our suspicion by using an unsupported Solaris API to translate virtual addresses to physical addresses within our benchmark program, allowing us to analyze the distribution of physical addresses underlying the blocks allocated within the 4MB regions. Once the problem
was understood we could avoid the issue by setting the `MALLOC_CONF` environment variable to “lg_chunk:20” which directs `jemalloc` to use 1MB regions instead of its default 4MB regions. All `jemalloc` data in this chapter was collected in this mode (1MB).

### 8.3 Index-aware allocation

The section describes a methodology for designing an index-aware segregated free list memory allocator, without intrusively changing other parts of the allocator design. Our methodology technique is general enough that it can easily be applied to most memory allocators.

#### 8.3.1 Cache index-aware size-classes

To avoid inter-block conflicts, we can simply choose a slightly different set of size-classes that is less prone to inter-block conflicts. That is, we simply avoid block sizes that underutilize the available indices. We can apply the adjustments, below, either statically at compile-time or at run-time, to transform a set of size-classes to be index-aware, producing an index-aware allocator.

Simplifying the problem slightly for the purposes of exposition, we will assume the effective block size $S$ inclusive of any header is always an integer multiple of 16 bytes (the cache line size). We define the cache index of a block as the index of that block’s base address. Within a given superblock the constituent blocks will have addresses of the form $(S \times n) + B$ where $B$ is the base of the block array in the superblock and $n \in \mathbb{N}$ up to the number of blocks in the superblock. Given $S$, the number of distinct indices for blocks in a sufficiently long superblock of size-class $S$ is $2048 / GCD(2048,S)$ where 2048 is the cache page size. Equivalently, we can state the number of usable indices for $S$ as $128 / GCD(128,S/16)$. Notice that we have a cyclic subgroup $\mathbb{Z}/(128)$ where 128 is the number of indices in the cache and the cycle length of the size corresponds to the number of indices on which blocks of that size can fall. As such, to provide an ideal tessellation we want to ensure $GCD(128,S/16) = 1$. That is, $S/16$ should be coprime with 128 and thus a generator of $\mathbb{Z}/(128)$, in which case blocks of size $S$ will land uniformly on all possible indices. The following simple transformation will adjust any size $S$ to be index-aware:

```plaintext
if GCD(2048,S) > 16 then S += 16
```

In the case of CLFMalloc only 4 lines of codes – an array of ints that defines the size-classes – needed change to render CLFMalloc cache index-aware.
8.3.2 Punctuated arrays

Changing the size-classes can have a profound impact on the application’s footprint. Consider an application allocating mostly 32-byte blocks working with an (index-oblivious) allocator with a free list of 32-byte blocks. Changing the size classes as proposed in the previous section would lead to the allocator satisfying 32-byte requests from a size class of size 48, increasing the memory footprint by 50%.

Therefore, this section proposes a different technique for index-aware allocation which does not suffer from such memory footprint blowup. The idea is to insert a cache line-sized and aligned spacer into the superblock array when indices start to repeat, yielding a punctuated array. This allows the allocator to retain its existing size-classes.

Say we have a superblock with 768-byte blocks and a sequence of blocks within that superblock that fall on addresses \(B, B + 768, B + 1536, B + 2304, B + 3072, B + 3840, B + 4608, B + 5376, B + 6144\), etc. The Niagara has 16-byte lines, 128 possible indices, and a 2048-byte cache page size. Our blocks would fall on indices \(I, I + 48, I + 96, I + 16, I + 64, I + 112, I + 32, I + 80\) and \(I\), respectively, where \(I\) is the cache index associated with block address \(B\). If block address \(B\) falls on index \(I\) then the \(N\)-th block beyond \(B\) falls on address \(B + (768 \times N)\) having index \(I + (((768 \times N) / 16) \mod 128)\). In our example the indices repeat after just 8 blocks or 6144 bytes as the least common multiple of 2048 (the cache page size) and 768 (the block size) is 6144 bytes. If an implementation inserts a spacer after every 8 blocks, however, then a punctuated array of such blocks will land on the full set of 128 indices.

In the worst case punctuated arrays require just one cache line of spacer per cache page within the superblock, putting a tight bound on wastage. Furthermore, the spacer lines are never accessed, so while they might increase TLB pressure and physical RAM usage, they do not influence L1D pressure.

Finally, we note that we only need to employ spacers in superblocks that have index-unfriendly size-classes, where a simple unpunctuated array of blocks would otherwise land on only a subset of the possible indices.

8.3.3 Superblock coloring

As explained in Section 8.2.1.2, index-oblivious allocation of superblocks can result in intra-core, inter-thread, inter-superblock, inter-heap index conflicts.

In this section we propose to reduce the odds of such inter-thread conflicts by inserting randomly sized variable length coloring area at the start of each superblock. This makes allocation
arrays for different threads and processes start on different cache indices, thereby reducing index conflicts. We initially placed the superblock header on the address returned by `mmap` and then inserted the coloring region after the superblock header and before the array of blocks, but noticed that the superblock header itself was vulnerable to index conflicts. We ultimately placed the coloring area before the header, providing better index distribution for the cache lines underlying the superblock headers.

To avoid inter-process index conflicts, one needs to seed the pseudo-random number generator – used to generate superblock colorings – differently for each process, perhaps based on the time-of-day, process-ID, or a system random number generator. Absent per-process seeding of the random number generator used for superblock coloring, allocations in similar but distinct processes may fall on precisely the same virtual addresses, increasing the likelihood of inter-process conflicts.

We note that Solaris randomly colors the offset of the stack for a process’s primordial thread, in part to lessen the odds of inter-process conflict between stacks. Similarly, the HotSpot Java Virtual Machine explicitly colors the stacks for threads created by the JVM.

### 8.4 The CIF memory allocator

Using the methodology of the previous section, we implemented a new index-aware segregated-free list allocator, CIF (Cache-Index Friendly). CIF is derived from LFMalloc. CIF and LFMalloc are structurally similar to Hoard. LFMalloc used hardware transactional memory [47] or restartable critical sections for concurrency control but for portability CIF uses simple mutual exclusion locks. CIF is easily portable and currently runs on Solaris SPARC and Linux x86 (32-bit and 64-bit).

#### 8.4.1 General design

In CIF each processor is associated with a processor-private heap. A superblock consists of a coloring region (described below), a header containing metadata and an array of blocks of the given size-class. As in Hoard the superblock header contains a pointer to the head of a LIFO free list of available blocks within that superblock. All blocks in a superblock are of the same length. Superblocks are 64KB in length.

CIF does not explicitly request large pages for superblocks. Large pages, if supported by the processor and operating system, can improve performance by decreasing TLB miss
rates. Solaris attempts to provision mappings with large pages as a best-effort optimization. On SPARC large pages must be physically contiguous and both physically and virtually aligned to the large page size. The Niagara supports 8KB, 64KB, 4MB and 256MB pages.

Concurrency control in CIF is implemented by heap-specific locks. Contention is rare and arises only by way of preemption. The impact of contention can be reduced by using techniques such as the Solaris schedctl mechanism [1] to advise the scheduler to defer involuntary pre-emption by time slicing for threads holding the heap lock.

Threads use the schedctl facility to efficiently identify the processor on which the thread is running, thus enabling the use of processor-specific heaps. On Linux/x86 CIF can be configured to use the CPUID or RDTSCP instructions to select a heap.

In CIF threads instantiate superblocks via mmap. On CC-NUMA systems that use a “first touch” page placement policy this means that the pages in a superblock will tend to be local to the node where the thread is running, improving performance.

All the allocators except Hoard, tcmalloc and jemalloc require at least a word-size metadata header field for in-use blocks. In CIF, for instance, an in-use block consists of a header word – a pointer to the enclosing superblock – followed by the data area. Malloc returns the address of the data area, which by convention must aligned on at least an 8-byte address boundary. The free operator consults this header to locate the free list in the superblock’s header. CIF places the header word on the last word of the cache line preceding the address returned by malloc so the address returned by malloc is always aligned on 16-byte boundaries.

8.4.2 Index-awareness

In CIF the size-classes inclusive of the header are simple powers-of-two starting at 16 bytes. (We intentionally selected powers-of-two for the purposes of comparison against other allocators, whereas a production-quality allocator would use finer-grained size-classes.) Each free list is implemented using the punctuated array technique.

CIF can also be configured by means of an environment variable to use a simple “flat” array of blocks with no spacers. We refer to this form as CIU – Cache-Index Unfriendly. This form yields extremely poor cache index distribution similar to that which would be achieved with a binary buddy allocator [75]. It serves as a useful measure of cache index sensitivity.

CIF can also be configured to use index-aware size-classes instead of the punctuated array. And in fact the form with index-aware size-classes yields the same performance as the form that uses the punctuated array. In this mode CIF uses size-classes of the form \((2^N + 1) \times 16\) for
N=0,1,2,3 etc., yielding a favorable index distribution.

The CIF and CIU allocators employ random superblock coloring – 16 possible colors in the interval [0,15] with the length of the coloring region taken as the color times the L1D line length – and process-specific seeding of the random number generator. Ideally an implementation would provide one color for each of the 128 possible indices. Recall, however, that the coloring region is never allocated from and never accessed. It exists solely to control the offset of the array of blocks. As a practical concern to bound wastage from the coloring area we restrict ourselves to just 16 colors.

### 8.4.3 Revisiting the index-awareness benchmarks

To verify the efficacy of our methodology, we use the `mcache` benchmark to compare CIF and CIU to existing allocators. For readability, we report here only the results for CIF and CIU – results of existing allocators are shown in Section 8.2.1.

Figure 8.4a shows that CIF gives an ideal uniform index distribution over all block sizes, whereas CIU distributes its blocks much less uniformly. As expected, Figure 8.4b shows that using CIF allows the benchmark to achieve the best performance across all block sizes, unlike any of the allocators evaluated.

To evaluate the efficacy of superblock coloring, we use the memory access scalability benchmark (Section 8.2.1.2) to compare CIF and CIU to CIF-NoColor, which is CIF configured to run with superblock coloring disabled. Figure 8.5 shows that superblock coloring mitigates the conflicts in the shared L1 cache, as both CIF and CIU achieve the best scalability whereas scaling fades with CIF-NoColor.

![Index distribution](image1)

![Pointer chasing performance](image2)

Figure 8.4: CIF `mcache` benchmark results.
8.4.4 CIF scalability

Here, we show that making an allocator index-aware does not affect its performance or scalability. We use the micro benchmark [46, 47] which runs concurrent threads within a single process, each of which invokes malloc and free repetitively over a 50 second measurement interval, reporting the aggregate throughput rate of the threads in malloc-free pairs per millisecond. The threads are completely independent and do not communicate or write to any shared data.

Figure 8.6 shows the results. The Niagara processor has only two pipelines per core, so scaling above 16 threads is modest and arises largely from memory-level parallelism [42]. As we can see from the graph all the allocators scale well except libc, which uses a single heap with a centralized lock. Broadly, the ratio of performance between the allocators observed at 1 thread holds as the number of threads increases, suggesting that path length through the malloc and free dominates multithreaded performance, and that the allocators have no substantial scaling impediments. As is made obvious in the graph, the application is insensitive to cache index placement as CIU effectively yields the same results as CIF.

8.5 Cache index-sensitivity in applications

This section shows that index-oblivious allocation can negatively impact performance in real applications, and demonstrates that our index-aware methodology mitigates the problem.

Cache index sensitivity is an aspect of application performance determined both by application structure and allocator design choices. We define an application as index sensitive under
Figure 8.6: Malloc scalability varying thread count and allocators. Y axis is log scale.

an allocator if it suffers excessive conflict misses because of poor index distribution. Index distribution, in turn, is largely determined by the allocator policies and design choices.

8.5.1 Applications studied

To evaluate index-sensitivity in real-world code, we examined existing benchmark suites as well as common programming patterns. We selected applications that showed benefit under an index-aware allocator relative to an index-unfriendly allocator, and that were amenable to direct analysis to establish that the benefit arose solely from index-aware allocation.

We selected the applications below as (1) they were insensitive to malloc-free performance with such operations typically confined to a brief initialization phase; (2) they were cache index-sensitive; (3) they were insensitive to cache line relative block alignment, and (4) they were sufficiently simple so as to be amenable to direct analysis, allowing us to establish that the benefit arose solely from index-aware allocation.

Regarding (3), above, during our investigation we discovered that some applications were extremely sensitive to how allocators placed blocks with respect to cache line boundaries – whether, for instance, the blocks were always aligned, never aligned, or sometimes aligned for a given size. Our set of allocators used various policies. By default CIF always returns blocks aligned on cache line boundaries, but, as a test of sensitivity can be configured otherwise. We only reported on applications that were not sensitive to cache line relative block alignment.
Evaluated applications:

- **llubenchmark** [123] allocates groups of nodes and then iterates over those nodes during the benchmark interval. We used the version of llubenchmark found in the LLVM test suite version 2.7, and augmented it to report cache index distribution and allow for variable-length node sizes. The original form from Zilles allowed the node size to be specified on the command line but used a custom allocator while the form in the LLVM test suite used malloc but with a fixed node size. Our form mallocs each node and allows the node size to be specified on the command line. We used a command line of “-i 2000000 -n 1 -l 341 -g 0.0 -s 250” which specifies one list of 341 nodes of 250 bytes and 2000000 iterations over the list. As configured by the command line, all allocation is performed at startup time, so differences in reported performance reflect the rate at which the thread iterates over the list. We collect the elapsed time by running the program under the `time` command. We note that llubenchmark behaves similarly to our own mcache.

- **egrep** is GNU grep version 2.6.3, a regular expression search utility based on deterministic finite automata. We timed the search of a 500Mb text file containing nucleotide sequences. The key block size malloc-ed by the application is 1024 bytes, which represents arrays of 256 ints which serve as transition tables for the state machine. Only a few indices are actually accessed, however. All significant allocation occurs at startup time and the run is dominated by pointer chasing operations over the DFA graph structures. The performance differences in the figure are almost entirely attributable to malloc placement policies in the different allocators. All the other benchmarks report elapsed time, but for egrep we report user-mode CPU time to factor out the IO-time required to read the file. (For reference, the run under libc took 11.2 seconds elapsed time with 2.01 seconds IO time. An IO time of 2.01 seconds is constant over the various allocators). Similar results were seen with the Google RE2 regular expression package which is also index-sensitive.

- **dnapenny**[29] is benchmark in the “phylip” Phylogenetic Inference Package component of the BioPerf bioinformatics benchmark suite. It uses a branch-and-bound algorithm to compute parsimonious trees. The source code was obtained from [8]. When starting, the application allocates 16 “tip” nodes. Each contains 4 buffers of size 6872 bytes which are allocated separately. In the main loop there is an iteration that accesses a single buffer in each tip node. The program is moderately long-running, requiring more than 8 minutes
under libc. Other phylip components such as promlk are similarly index-sensitive.

- **stdmap** creates a `std::map<int,int>` standard template library collection at startup. The benchmark then times the collection’s iterator. The key-space is [0,499] and is approximately half populated. `std::map<>` is implemented as a red-black tree. The tree nodes are 40 bytes in length which some allocators round up to 64 bytes – a size that is cache index-unfriendly. The benchmark is written in C++ and produces a 64-bit executable. By default the C++ `new` and `delete` operators map directly to `malloc` and `free`. The tree implementation and nodes are opaque to the application so we were unable to directly report node addresses and indices, but instead used the Solaris `dtrace` and `truss -fl -t !all -u ::malloc` commands to observe the allocation patterns.

- **Xml** is a 64-bit microbenchmark written in C that constructs an in-memory XML document tree via the Solaris `libxml2` package and then repeatedly iterates over the tree, reporting iteration times. The `xmlNode` instances are individually allocated by `malloc` and `free`. The `libxml2` library package parses the XML document and directly allocates the nodes which form the internal representation of the document.

- **Gauss** performs Gaussian elimination on 200x200 matrices of 64-bit floating point numbers using the partial-pivot method. Each row is individually allocated.

- **DotProduct** computes the dot-product of a 200 vectors each of 200 elements.

Both Gauss and DotProduct have array accesses of the form `a[I][J]` where an inner loop advances `I` and an outer loop advances `J` and each row is individually allocated. (That is, the outer loop varies column and the inner loop varies row). There is no temporal locality as each element is accessed just once, although spatial locality is potentially available between iterations. While processing index `I` the inner loop may access a cache line underlying a row at address `A` only to find that same line subsequently evicted later in the inner loop because of conflict displacement. On the next iteration of the outer loop the code will access index `I + 1` adjacent to `I` in that same line underlying `A`, incurring a conflict miss. In this case our code is iterating over multiple arrays simultaneously, in lock-step, and there are no hot fields. Because of avoidable index conflicts, the application may fail to leverage potential spatial locality. Cache index-aware allocation can often avoid this problem.
This is a fundamentally different mode of benefit than is seen in the other applications, where index aware allocation leverages temporal locality in a small number of “hot” fields.

### 8.5.2 Evaluation

We ran each benchmark (a single-threaded application) 5 times on the Niagara machine, and took the median result. Variation between runs was extremely low.

Figure 8.7 depicts the results normalized to the times obtained with the default `libc` allocator. Excluding CIF, no one allocator is best over all the applications as each exhibits different pathological index-unfriendly sizes as was previously seen in Figure 8.1.

`Hoard` is often comparable to CIF. `Hoard` uses size classes that are a power of \( \approx 1.2 \) apart and are therefore less regular, so many of `Hoard`’s size classes tend to cover all cache indices. Yet this is not due to index-aware design of the allocator, as evidenced by `Hoard`’s poor results on some benchmarks.

Using the CIU variants leads to poor performance in all benchmarks, sometimes worse than CLFMalloc on which it is based. This is because CIU’s size-classes are still different than CLFMalloc’s. For example, in `egrep`’s case, the DFA transition table is a 1024-byte buffer. In CIU all these allocations start on the same cache line, and in CLFMalloc on eight cache lines. This happens since CIU allocates there requests from its 2048-byte size-class, while in CLFMalloc, 1024-byte requests are allocated from a 1280-byte size-class.

![Figure 8.7: Run-time of cache index-sensitive applications, normalized to libc](image)
8.6 Related work

The literature is rich with studies that show how layout and placement can influence cache behavior and impact performance for pointer-based programs \[83\][124]. Petrank \[100\] shows that cache-conscious data placement is, in the general case, NP-hard. By exploiting common access patterns and behavior found in applications we can still, however, provide benefit in many circumstances. Broadly, the optimization techniques involve changing the access pattern; intra-object field layout changes; and inter-object placement policies. Calder et al. \[37\] introduce cache conscious data placement which reduces cache misses by profiling an application, building a temporal relationship graph of data accessed, and finally using the temporal access patterns discovered in the profiling stage to refine data placement. Kistler et al. \[74\] develop an algorithm that clusters data members to promote and enhance spatial locality. Chilimbi et al. \[41\] show the benefit of cache-conscious data layout and field placement. Their allocator interface is non-standard, however, and does not allow drop-in binary replacement under the standard malloc-free interface.

Lvin et al. \[87\] use object-per-page allocation in the Archipelago allocator to probabilistically detect errors in the heap arising from software flaws. Naively, if all objects were to start on page-aligned virtual address boundaries then applications accessing such objects could suffer from excessive conflict misses. To reduce the conflict miss rate their allocator randomly colors the offset of the object with the page. Coloring was used to salve the impact of page alignment, and not applied in a general and principled fashion that minimizes wastage. Furthermore, an object-per-page allocator may impose high TLB pressure.

Bonwick et al. \[35\] (section 4.1-4.3) also suggested superblock coloring, but only as remedy for inter-superblock intra-thread cache index conflicts and to relieve bus and bank imbalance for systems with multiple memory channels. Their paper predates commodity CMT (chip multithreading) systems with shared caches. We believe that index-aware size-classes or punctuated block arrays largely obviate and supersede the use of superblock coloring for the purpose of addressing inter-superblock intra-thread conflicts. That is, index-aware size-classes or punctuated arrays reduce both intra- and inter-superblock conflicts for a given thread accessing a set of superblocks. Superblock coloring remains useful, however, as it provides a new mode of benefit for intra-core inter-thread inter-superblock conflicts on modern shared cache CMT platforms. Bonwick also noted that binary buddy allocators are pessimal with respect to cache index distribution. We concur and generalize to sizes other than simple powers-of-two.
We also note that facilities such as `memalign` should be used judiciously as excessive unneeded alignment can induce conflict misses.

Page coloring [105] operates at the level of the operating system or virtual machine monitor by influencing the choice of physical pages to assign to virtual addresses. The color of a physical page is just the value in the intersection of the physical page number field and the cache index field of the page address. Page coloring attempts to provide a uniform distribution of page colors for the physical pages assigned to a set of virtual pages, which in turn promotes balanced utilization of the set of available cache indices. Say that in the physical address layout we find that the page number field overlaps the cache index field by 2 bits, giving 4 possible page colors. If the kernel does not provide ideal page coloring and inadvertently mapped virtual pages V0, V1, V2 and V3 to physical pages P0, P1, P2 and P3, respectively, and those physical pages happened to be of the same page color, then cache lines underlying V0, V1, V2 and V3 would be able to reside in just one quarter of the available cache indices, possibly underutilizing parts of the cache and creating a “hot spot” in other sectors. Page coloring attempts to avoid such unfavorable assignments of physical pages to virtual addresses.

Hardware-based means of reducing the rate of conflict misses were suggested Seznec [110] (skew-associative caches) and later by Gonzales [60] and Wang [121]. All entail changes to the hash function that maps addresses to cache indices and none is currently available in commodity processors. Min and Hu [94] suggest completely decoupling memory addresses from cache addresses in order to reduce conflict misses while Sanchez and Kozyrakis [106] subsequently suggest decoupling ways and associativity.

Our approach is most similar to that of page coloring except that it is implemented entirely in user-space within the virtual address `malloc` allocator and operates only on low-order bits of the cache index field of addresses that are not part of the physical page number field. Page coloring and cache index-aware allocation are complementary optimizations. Like page coloring our approach is non-intrusive in that it operates without any need to profile the application or modify the application’s source code. If the `malloc` library is implemented as a separately deliverable dynamically loadable module, as is the case on most platforms, then our approach can be used by simply substituting a new `malloc` library, eliminating the need to recompile and providing benefit to legacy binary applications. In addition, our technique is orthogonal to but benefits from complementary mechanisms that change field layout to promote spatial locality [74]. Instead of specifically increasing locality it simply leverages ambient locality already present in the application.
Chapter 9

Conclusion

This dissertation addressed three challenges in designing high-performance concurrent software for modern multicore processors: realizing the parallelism potential of the problem, efficiently dealing with synchronization-heavy tasks, and overcoming system-level bottlenecks. Specifically, we make the following contributions:

1. We have shown how to efficiently realize the parallelism in the rendezvous problem, an abstract problem of concurrent matching, in which consumers and producers show up and are matched each with a unique thread of the other type. We have presented adaptive, nonblocking, high throughput asymmetric rendezvous systems that scale well under symmetric workloads and maintain peak throughput in asymmetric workloads. This is achieved by a careful marriage of new algorithmic ideas and attention to implementation details, to squeeze all available cycles out of the processors.

2. We have presented LCRQ, a concurrent nonblocking linearizable FIFO queue that outperforms prior combining-based queue implementations in all concurrency levels on an x86 server with four multicore processors. LCRQ uses contended F&A objects to spread threads around items in the queue, allowing them to complete in parallel. Because the hardware guarantees that every F&A succeeds, we avoid the costly failures that plague CAS-based algorithms.

3. We have demonstrated that fence-free work stealing is possible on mainstream processors implementing the total store ordering (TSO) memory model, such as the popular x86 and SPARC. We have shown that these processors in fact implement a bounded TSO memory model – a novel restriction of TSO which bounds the amount of reordering possible – and that work stealing can be implemented without fences on bounded TSO machines.
4. We have described lightweight techniques by which software can help hardware with TM capabilities to significantly improve the performance of lock-based code using lock elision, by increasing the amount of concurrency exploited. Our evaluation on a Haswell processor is encouraging, improving the run time of applications from the STAMP suite by up to $3.5 \times$ and of data structure microbenchmarks by up to $10 \times$.

5. We have observed that the manner in which modern memory allocators place allocated blocks in memory can induce cache contention and lead to excessive cache misses in both single- and multi-thread executions. We proposed a methodology for building a memory allocator with cache sensitive block placement and demonstrated that it alleviates this problem.

9.1 Future directions

9.1.1 Hardware and software symbiosis

Our work shows the degree of impact that hardware architecture specific details can have on a concurrent algorithm’s performance:

- The Westmere processor’s write-back cache architecture leads to significant cost differences between CASing a location in a core’s cache compared to one in a remote core’s cache, whereas on the Niagara with its write-through cache architecture the CAS cost is more uniform.

- Using F&A – when available – can result in fundamentally better performance by avoiding “contention meltdowns” associated with CAS retries.

- Understanding the store buffering mechanisms responsible for memory reordering in the TSO model yields insights that allow omitting of an expensive memory fence in the fast-path of an algorithm.

All of these suggest that algorithms must be fully aware of their underlying hardware to maximize their performance. Yet much of the programming world is moving toward managed languages, such as Java, whose goal is to abstract away architectural details and avoid exposing them to the program. Even unmanaged environments such as C++ strive to define memory models that cover all possible hardware architectures. Reconciling these conflicting directions is a very interesting research problem.
9.1. FUTURE DIRECTIONS

9.1.2 The role of combining

Our results raise a question about the combining paradigm. We have shown that combining may lose its advantage in problems with inherent potential for parallelism, as our rendezvousing algorithms significantly outperform the combining-based rendezvous algorithms.

Thus, it would seem that the advantage of combining is in fundamentally sequential data structures, such as a FIFO or priority queue, whose concurrent implementations have central hot spots. However, our LCRQ algorithm outperforms the best known combining-based queues.

This raises some interesting questions about the role of combining in concurrent algorithms:

• Can the combining technique be improved to exceed the performance of our asymmetric rendezvous systems and LCRQ queue?

• Can one design a concurrent non-combining priority queue (PQ) that outperforms combining-based PQs?

• Which points in the design space – in terms of hardware architecture and problem being solved – is combining most suited for?

9.1.3 Tuning and adaptivity for lock elision

It would be interesting to explore more refined policies for handling conflict management in hardware TM and lock elision systems. In particular, utilizing abort information provided by the hardware (such as the location in which a conflict occurs) appears to be a promising direction.

Our software-assisted conflict management scheme groups conflicting threads into a single group. This may be too strict since a single conflicting thread does not have to conflict with the entire group. A natural extension to explore is dividing the conflicting threads to different groups, each containing only threads that conflict among themselves.

Finally, it seems plausible that the best conflict management parameters are workload-dependent. For example, in a workload in which one thread writes to a shared locations that many threads concurrently read, our schemes may end up serializing the reading threads. Avoiding this problem, and in general designing an auto-tuning library that performs well on a wide set of workloads, is an interesting challenge.
9.1.4 Applications of cache index-aware allocation

In initial testing, we have observed that the HotSpot Java virtual machine (JVM) exhibits index-oblivious placement for certain object sizes, causing reduced performance and increased L1D miss rates.

Therefore, it seems promising to investigate and implement index-aware allocation in the context of the JVM, which can enforce index-aware size-classes in the object layout manager and provide random coloring either at the start of or within thread local allocation buffers (TLABs), which are contiguous thread-local object allocation regions managed by a simple bump pointer.

In addition, we believe that cache index-aware allocation should be particularly helpful for hardware transactional memory implementations where the address-sets are tracked in the L1D and where conflict misses cause transaction aborts [49].
Bibliography


[22] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In Euro-Par 2010 - Parallel Processing, volume 6272 of LNCS, pages 151–162. Springer Berlin / Heidelberg, 2010. 1.1.1, 4.2.1, 4


[27] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms


[29] D.A. Bader, Yue Li, Tao Li, and V. Sachdeva. BioPerf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. pages 163–173, 2005. 8.5.1


[61] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. ACM Transactions on Programming Languages and Systems (TOPLAS), 5(2):164–189, April 1983. 1.2.1, 5, 5.2, 5.4


[70] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 12:463–492, July 1990. 2.2, 2.2.2, 4.1.1, 4.1.1, 4.2.2


[79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, 28(9):690–691, September 1979. 2.6


Yehuda Afek, Michael Hakimi and Adam Morrison.

Adam Morrison and Yehuda Afek.

Adam Morrison and Yehuda Afek.

Yehuda Afek, Amir Levy and Adam Morrison.

Yehuda Afek, Dave Dice and Adam Morrison.
Hardware lock elision) and an abort (HLE) to signal that the operation was aborted. This is achieved by marking the cache line as invalid (cache miss).

In multithreaded systems, HLE improves the performance of transactions that share the same cache line. This is referred to as Hardware lock elision.
בsprites ובנוסף הבחנה לתדמית ליישום. בחר זה המתבונן נישה אתחול מתון.
_fetch_and_add/_higher
デンי במשר ימויו מחנה בסיומי בנפיים בישא
דבר המאפיים את פ㎥ למגיחה מבמקיב. משוש הכל, ביצועי של חותם שוא החזית עלים
על ביצוע הזראה הבלתי מתור המוכרים לוים. (פרק 5)

שוני, ואמריא צעד ניח, במערכת מורחבת ילות על מולד זכור מהן, לבצע TSO 'יניבת בנווהל' אל ארוז ביצוע פקודה כシャר מוארת מטלת מתור העבוצה. במודל ה- TSO, פקודת ה- TSO-סידור, מתוארת ה-
memory fence מסדר את איה הח 수도ות הקומפקט בשור. לימיוסן לחות העלב באתק רור העבוצה, 'יעי י الكبير חצימה החלון (store buffer) של לחות.
חותנה שליכו איה כי אם זבל של חツי החכמים 디ו, או וח גנד ילבוס את מקומם של לחות העובד באתק החוכמה עלי מרפי החכמאות משומצא בחתוך, ואת אל תורך
хватת העבץ באתק החוכמה עלי מרפי החכמאות עלי מרפי החכמת חצימה בחתוך, ואת אל תורך
בחיצה 'יעי העבץ. תוהמה את לזרב מופסיק בין מנט 'הできて חוה לנה.continue' העבוצה.

מתמודדים על אותה ומושל, בכך להביש את חכונה של פלונית 'יניבת העבוצה'.

בונו, ואמריא צעד ניח, במעבד נ86 SPARC- יאואר מיצירת מסי עכום מתור השולש TSO-סידור. כמ כל המריא צעד ניח להמדד את גולש של
חות NHS החכמים על מועדים אז, בכך לאפשר את מעשו התכנית של

וא משלבים את כל החוכמות הנייל על מנט להנכי לכל חוסר משומרי ביצועים של י睫יב
אלגוריתמי 'יניבת העבוצה', המופעים בפתור. (פרק 6)

ה切尔בון על חיים בקוק מעקרולים: שערת קורבות, מתקור של רוחר הבקבוקים בברית
ה切尔בון תואמך על מחנה החכמים, בת יא בחרומת און בתוכנים. כש לאבחב בייהם כלר,şeש ששתוח מלב של השיחות צוקדكَ חתוך שללה אל בשכבות התוח נמכרה עלים
ה切尔בון לחנה. חיבר זה מתואר שיין זוארי בקוק באל, וריכים לתנלברעל.
תקציר

עבדו וו לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת לשפת

MIOMETPI 유효

MIOMETPI 유효

MIOMETPI 유효

MIOMETPI 유효

MIOMETPI 유효

MIOMETPI 유효

MIOMETPI 유효

MIOMETPI 유효
תמצית

עבודה זו עוסקת בשלושה אתגרי התמודדות עם בעיות בתוכנה ומעבדות מרובי ליבות מודרניי. המשימה הראשונה היא מימוש פוטנציאל התמרון של הבעיות: מעבדי התמודדות עם בעיות ביצועי על ביצועי עחייה, והתמודדות עם בעיות בתוכנה ומעבדות מרובי ליבות. אחר כך, אנו מתארים שני ציארי בקבוק של הסכימה בשכבות התוכנה וברשתות המחשביות: הבלוקה של הלקellery (Hardware lock elision או HLE) ו yardımcיה, האפשרות להשתמש בתוכנה של תוכנה מקבילה בכיוון השיכרות ומגלה המעבד. מעבדת המעבדים, ואינו פועל במעבדה של מעבדת המעבדים. ב baseman המעבדים, אנו מתארים שני ציארי בקבוק של הסכימה בשכבות התוכנה וברשתות המחשביות: הבלוקה של הלקellery (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכבות התוכנה וברשתות המחשביות: הבלוקה של הלקendency (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בوبة, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי בקבוק של הסכימה בשכמות התוכנה וברשתות המחשביות: הבלוקה של הלקenemy (Hardware lock elision או HLE) ו_metavim, או מונעים תופעות של "למטמו". על בובה, אנו מתארים שני ציארי ב 错误
הרצת תקעה עיולה ומדרגית
על מחשבים מורבי ליבלי

חיים הורן להוראות התואר "דוקטור לפילוסופיה"

על דיו

אמס מוריסון

ענדו אדם מקחוק בועשת בוגר

פרופ’ יהודה אפק

حواצ לסטטש של אוניברסיטת תל-אביב

טבת תשס”ד