2015

Crafting Concurrent Data Structures

Yujie Liu
Lehigh University

Crafting Concurrent Data Structures

by

Yujie Liu

A Dissertation
Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy
in
Computer Science

Lehigh University
August 2015

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Yujie Liu
Crafting Concurrent Data Structures

Date

Michael Spear, Dissertation Director, Chair
(Must Sign with Blue Ink)

Accepted Date

Committee Members

Hank Korth

Victor Luchangco

Gang Tan

Dedication

To an anonymous friend.

Acknowledgments

This dissertation would not be possible without the support from many people.

First of all, I want to thank Professor Kunlong Zhang who introduced me to computer science research, and my advisor Mike Spear who taught me how to do it well. They provided continuous inspiration and aspiration along these years.

I thank many people from the Scalable Synchronization Research Group at Oracle Labs, including Dave Dice, Tim Harris, Alex Kogan, Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, and many others for deepening my knowledge of shared memory synchronization. I am especially thankful to Victor Luchangco for coaching me in writing sensible correctness proofs of concurrent programs.

I am grateful to Wei Peng, Jian Wang, and Haobo Zhang, who made me addicted to (playing games on) computers, and to Lonnie Heinke who showed me programming can be a fun profession.

I enjoy the friendship with Yazhou Li, Andi Ni, Mengtao Sun, Yunkai Zhang, and many others, who cheered me up when times got tough. I am especially thankful to Mengtao Sun for accompanying me through many pleasant road trips. I thank Chef Chizmar, Mr. and Mrs. Shindo, and the folks from Full of Crepe, who nurtured this dissertation with delightful food.

Finally, I would like to thank Mike again for being generous on geographic and time freedom, and my lovely parents for providing sweet “life improvement” funds, which together granted me the privilege to travel around the world comfortably during my PhD study.

Contents

List of Tables

List of Figures

Abstract

Concurrent data structures lie at the heart of modern parallel programs. The design and implementation of concurrent data structures can be challenging due to the demand for good performance (low latency and high scalability) and strong progress guarantees. In this dissertation, we enrich the knowledge of concurrent data structure design by proposing new implementations, as well as general techniques to improve the performance of existing ones.

The first part of the dissertation present an unordered linked list implementation that supports nonblocking insert, remove, and lookup operations. The algorithm is based on a novel "enlist" technique that greatly simplifies the task of achieving wait-freedom. The value of our technique is also demonstrated in the creation of other wait-free data structures such as stacks and hash tables.

The second data structure presented is a nonblocking hash table implementation which solves a long-standing design challenge by permitting the hash table to dynamically adjust its size in a nonblocking manner. Additionally, our hash table offers strong theoretical properties such as supporting unbounded memory. In our algorithm, we introduce a new "freezable set" abstraction which allows us to achieve atomic migration of keys during a resize. The freezable set abstraction also enables highly efficient implementations which maximally exploit the processor cache locality. In experiments, we found our lock-free hash table performs consistently better than state-of-the-art implementations, such as the split-ordered list.

The third data structure we present is a concurrent priority queue called the "mound". Our implementations include nonblocking and lock-based variants. The mound employs randomization to reduce contention on concurrent insert operations,

and decomposes a remove operation into smaller atomic operations so that multiple remove operations can execute in parallel within a pipeline. In experiments, we show that the mound can provide excellent latency at low thread counts.

Lastly, we discuss how hardware transactional memory (HTM) can be used to accelerate existing nonblocking concurrent data structure implementations. We propose optimization techniques that can significantly improve the performance (1.5x to 3x speedups) of a variety of important concurrent data structures, such as binary search trees and hash tables. The optimizations also preserve the strong progress guarantees of the original implementations.

Chapter 1

Introduction

1.1 Background

Concurrent data structures are a fundamental building block for scalable multi-threaded programs. They are widely used in software systems that handle concurrent tasks, e.g., operating systems and web servers. In these systems, a concurrent data structure is often shared by the whole system, and can significantly impact the overall performance. As an example, a web cache service [2] may store the cached key-value pairs in a shared hash table, which is accessed by concurrent threads that perform lookup or update operations on the data structure. To maximize the scalability of the cache service, the hash table often requires careful design to avoid performance bottlenecks.

Arguably, the simplest way to obtain a concurrent data structure implementation is to protect a sequential implementation with a *mutual exclusion lock* [13]. In this approach, programmers can mark each data structure operation as a *critical section* protected by a single mutual exclusion lock, which allows at most one thread to execute a critical section at a time. However, this approach is not scalable since it permits at most one thread to access the data structure at a time. Applications sometimes admit relaxations of the mutual exclusion property. For example, *readers-writer exclusion* [53] (a.k.a, readers-writer locks) allows a thread to declare itself as a *reader* or a *writer* for the critical section, where multiple readers can share the critical section at the same time but a writer precludes any other request for the critical section. In reader-dominated workloads, using readers-writer exclusion may improve performance by exploiting parallelism among reader requests. However, the approach fails to support concurrent writers even if the writers operate on different parts of the data structure.

Variants of mutual exclusion also include a popular technique known as *fine-grained locking*, in which an operation only locks parts of a data structure as needed. This approach may introduce significant overhead to a streamlined implementation, as locks are aquired and released at each part of the data structure. In addition, the approach inherits the scalability problems incurred from locks. An in-progress operation that locks parts of a data structure can prevent other concurrent operations from proceeding. The overall performance can be impacted by the scheduling decisions made by the underlying operating system.

The design and implementation of concurrent data structures can be challenging. An ideal implementation would provide intuitive semantics and good performance (low latency and high scalability). However, it is difficult [22] and sometimes impossible [5] to achieve these properties at the same time.

Specifically, the challenges in concurrent data structure design stem from a number of aspects. First, the implementations must achieve good performance across a variety of workloads. A concurrent data structure ought to have low latency when accessed by a single thread. This property is valuable for applications whose threads rarely access the data structure at the same time: if latency is too high, then the programmer may instead opt to protect a sequential data structure with a single mutual exclusion lock. However, an implementation should also exhibit high scalability. That is, in highly concurrent workloads, threads should not impede each others' progress when they access disjoint parts of the data structure. There is typically a tension between these goals: to ensure good scalability, a greater amount synchronization is required to coordinate potential concurrent accesses to the data structure; however, the injection of synchronization introduces overhead to the streamlined sequential implementation, because it often requires the use of expensive atomic synchronization primitives, such as compare-and-swap (CAS) instructions. Introducing fine-grained synchronization can result in more synchronization primitives per operation, and thus more latency.

Secondly, many programs expect progress guarantees from concurrent data structures. Nonblocking progress, where an implementation can tolerate an unbounded number of thread failures, has been achieved for many practical data structures. Herlihy [32] demonstarted the theoretical feasibility (known as universal constructions) of converting any sequential data structure to a nonblocking concurrent implementation. Nonblocking data structures derived from universal constructions tend to be expensive, since the universal constructions often incur significant instrumentation overhead with increased time and space complexity [9, 19, 33]. Therefore, there has been an increasing interest in finding practical and efficient solutions to design specific nonblocking concurrent data structures using synchronization primitives [32] that are supported on existing hardware, such as compare-and-swap (CAS) instructions. Over the past two decades, dozens of concurrent data structures have been proposed, providing highly scalable stacks [31], queues [41,57], lists [26], trees [17,61], hash tables [65], skiplists [67], and many other data structures.

However, constructing nonblocking concurrent data structures directly upon these low-level synchronization primitives can be very difficult [22]. In many data structures, an operation needs to update multiple locations, such as resizing a hash table and rotating a balanced search tree, but there is often significant overhead to make concurrent updates to multiple locations appear atomic [28, 51]. Even for simple data structures, such as linked lists and queues, their nonblocking concurrent implementations can be substantially more complicated than their sequential counterparts, since instructions of multiple threads tend to interleave in a highly concurrent manner, creating a much larger state space for programmers to reason about.

Hardware Transactional Memory (HTM) [35] was originally designed to simplify the task of creating concurrent data structures. The idea behind HTM is simple: programmers mark regions of code that ought to execute as a single, indivisible operation, and then the hardware runs these "transactions" concurrently, while tracking their memory accesses. By tracking accesses, the hardware can identify conflicting memory accesses among transactions. By also providing a buffering mechanism, the hardware can abort, roll-back, and retry some of the transactions involved in a conflict, so that each transaction appears to execute in isolation.

Unlike research HTM proposals, the first-generation HTM systems from IBM [39, 70] and Intel [38] impose significant restrictions, which limit their suitability for lock-free programming. These "best effort" HTMs [12, 45] do not guarantee progress for arbitrary transactions: a transaction attempt will fail if it (a) attempts to access too many distinct locations; (b) executes for longer than a scheduler quantum of the operating system; or (c) attempts to perform an unsupported operation, such as a system call. Transaction attempts can also fail due to memory accesses that conflict with concurrent operations from transactions, or accesses that conflict with concurrent nontransactional code. This property, called "strong atomicity" [6], is a natural outcome of implementing HTM through the cache coherence protocol. It also allows for clever composition of transactional and nontransactional code [12, 16, 72].

Even if these limitations did not exist, it is unlikely that HTM could ever fully replace the best concurrent data structure implementations. As recently reported by Gramoli [24], concurrent data structures implemented directly from synchronization primitives (i.e. CAS) tend to provide the best performance in comparison to those implemented by using locks or transactions.

1.2 Terminology

Model Our system model consists of a set threads communicating via shared memory objects. An *object interface* defines the set of operations that a thread can invoke on the object. An *operation* consists of an *invocation*, followed by zero or more internal steps and a *response*.

We model the *history* of an object as a sequence of invocation and response events. A history is *sequential* if every invocation is immediately followed by its corresponding response. A *sequential specification* of an object is a set of sequential histories of the object. We say a sequential history is *legal* if it is a member of the object's sequential specification.

An operation is *pending* if it has an invocation but no response in some history. A history is *complete* if it has no pending operations. We say a *completion* of history H is a complete history that concatenates H with H' where H' consists of only responses. For two operations o_1 and o_2 in history H, o_1 happens before o_2 (denoted as $o_1 \rightarrow o_2$) if the response of o_1 precedes the invocation of o_2 in H.

Correctness Conditions Linearizability [36] is the canonical correctness condition for concurrent data structure implementations over the past decades. In a *linearizable* implementation, every operation must happen atomically at some instantaneous point (known as the *linearization point*) between the invocation and response of the operation. The intuition behind linearizability is that a concurrent history should "look like" a sequential one, that is, non-overlapping operations should happen in the same order in the sequential history.

More precisely, an object implementation is linearizable if for every history H, there exists a legal sequential history H_{seq} such that:

- H_{seq} is a permutation of some completion of H;
- For any operations $o_1 \rightarrow o_2$ in H, $o_1 \rightarrow o_2$ in H_{seq}.

Progress Guarantees In nonblocking implementations, an operation can make progress regardless of the states of concurrent operations. Nonblocking implementations can be classified according to the strength of their progress properties. *Wait-freedom* [32] ensures that every thread completes its operation in a finite number of steps. In contrast to wait-freedom, a *lock-free* implementation ensures that in a finite number of steps, some thread completes its operation but individual threads may fail to make progress.

More precisely, an object implementation is wait-free if for every infinite execution, every operation has a response. An object implementation is lock-free if every infinite execution with a pending operation has infinite responses.

Synchronization Primitives A synchronization primitive is an abstract atomic operation that serves to coordinate concurrent accesses to the shared memory, which can either be supported by the hardware or implemented in software. We now define several synchronization primitives used in the dissertation as follows:

- **Compare-and-swap** (CAS) takes three arguments, the *address* of a given memory word, the *expected* value, and the *new* value, and returns a boolean result. It atomically compares the content of the memory word with the expected value, and if they are the same, changes the content to the new value and returns true. Otherwise the operation returns false. We say a compare-and-swap succeeds if it returns true.
- **Fetch-and-increment** (FAI) takes the address of a given memory word, atomically reads its content as an integer, and increments the content value of the memory word by one. The value before the increment is returned.
- **Double-compare-and-swap** (DCAS) is similar to compare-and-swap, but operates on two locations. The operation atomically compares both locations with their expected values, changes them to new values if the contents of both locations match their expected values, and returns true. Otherwise the operation returns false.
- **Double-compare-single-swap** (DCSS) is similar to double-compare-and-swap. The operation atomically compares both locations with their expected values, but only attempts to change one of the locations.

The compare-and-swap (CAS) and fetch-and-increment (FAI) primitives are usually supported by modern processor architectures, such as Intel x86. The double-compare-and-swap (DCAS) and double-compare-single-swap (DCSS) are rarely supported by the architecture, but they can be implemented by using multiple CAS instructions [28, 51].

ABA Problem Concurrent data structures implemented using CAS and DCAS primitives sometimes need to deal with an issue known as the *ABA problem* [26, 57]. The ABA problem is a pattern of interleaved execution, described as follows:

1. Initially, the value of location X is A;

2. Thread T is about to execute a CAS on X that attempts to change its value from A to C, and stalls;
3. The value of X is subsequently changed to B, and then changed back to A by concurrent updates;
4. Thread T resumes and performs the CAS which successfully changes the value of X from A to C.

The above pattern becomes a problem only if the CAS instruction in step 4 succeeds *unexpectedly*. A common solution to the ABA problem is to augment the location X with a version number, which forms a *wide word* $\langle X, V \rangle$. Every time a CAS is performed on X, it is performed on the augmented wide word $\langle X, V \rangle$ and increments the version number by one. In the above scenario, the CAS in the last step will fail since the version number changed in step 3. On modern architectures such as Intel x86 and Oracle SPARC, CAS are usually supported on both single memory words, as well as wide words that are twice the size of a single memory word.

1.3 Contributions

We expand the knowledge base of concurrent data structure by designing and implementing the following new concurrent data structures.

We first present a nonblocking unordered linked list-based set implementation, which admits lock-free and wait-free variants. The list algorithm is based on a novel "enlist" technique which is the key insight to achieving practical wait-freedom. The list algorithm also serves as the building block of other data structures such as stacks and hash tables.

We then discuss a nonblocking dynamic-sized hash table algorithm, which is inspired by the unordered list algorithm. The hash table can adjust its size dynamically (both growing and shrinking), and operations can proceed in a nonblocking manner, even during some thread performing a size adjustment. Our hash table implementation outperforms the state-of-the-art implementations by improving the processor cache utilization and memory efficiency, and at the same time, our implementation delivers stronger theoretical properties.

The final data structure we present is an array-based, concurrent priority queue which we call the "mound". Our implementations include nonblocking and lock-based variants. The mound employs randomization to reduce contention on concurrent insert operations, and decomposes a remove operation into smaller atomic operations so that multiple remove operations can execute in parallel within a pipeline. The mound exhibits comparable performance to state-of-the-art skip list based priority queue implementations, and in particular, provides lower latency at low thread counts.

In addition, we present techniques that employ hardware transactional memory to accelerate existing concurrent data structure implementations. Our optimizations can bring a significant performance boost to a variety of concurrent data structures. The optimizations also preserve the strong progress guarantees of the original implementations, such as lock-freedom and wait-freedom.

1.4 Organization

The remainder of the dissertation is organized as follows. In Chapter 2, we discuss various existing concurrent data structure implementations and their design techniques. Our linked list, hash table, and priority queue algorithms are respectively presented in Chapters 3, 4 and 5. We discuss how hardware transactional memory (HTM) can be used to accelerate existing nonblocking data structure implementations in Chapter 6. We conclude in Chapter 7 and discuss future research directions.

Chapter 2

Related Work

In this chapter, we survey various concurrent data structure implementations by briefly summarizing their main ideas, and discussing some of the general design techniques.

2.1 Linked Lists

Valois [69] presented the first lock-free list implementation based on a technique that encodes in-progress operations with auxiliary nodes. Michael and Scott [56] corrected a memory management related bug later found in the Valois algorithm.

Harris [26] proposed a practical lock-free ordered list implementation, which requires language support for garbage collection. The Harris algorithm employs a technique that marks the lower bits of the successor pointers of nodes, in order to mark nodes as "logically deleted". The algorithm uses a separate phase to physically remove from the list the nodes that are logically deleted. The obligation of physical deletion is assigned to each insert, remove and lookup operation. Michael [54] proposed a variant of the Harris algorithm with manual memory management, which also improves the performance of the original algorithm.

Heller et al. [29] designed a lock-based linked list with lazy synchronization. Similar to the Harris-Michael algorithm, the Heller algorithm also separates the logical and physical deletion of nodes, however, it uses a boolean field to indicate that a node is logically deleted. This avoids the requirement for pointer marking, which is not generally available in every programming language. More importantly, the Heller algorithm incorporates a more efficient wait-free design of the lookup operation, in which logically deleted nodes are skipped instead of being removed (as in Harris-Michael). The resulting lookup operation is wait-free and contains no side effect on the shared memory, which significantly improves the performance of the original Harris-Michael algorithm. The Heller algorithm also demonstrates that concurrent object implementations can achieve competitive performance by leveraging hybrid progress guarantees for different operations, i.e. blocking updates and non-blocking searches.

Timnat et al. [68] constructed the first practical wait-free ordered list implementation according to the fast-path-slow-path methodology [42]. The algorithm composes a less efficient (and more complicated) wait-free algorithm with the Harris-Michael algorithm.

2.2 Hash Tables

The first practical nonblocking hash table was designed by Michael [54] by creating a fixed-size bucket array of lock-free linked lists. The lists are a streamlined version of the lock-free ordered list by Harris [26]. Independently, Greenwald [25] implemented a lock-free closed addressing hash table. Greenwald's hash table is resizable, but relies on a DCAS (double-compare-and-swap) operation. Unfortunately, simulating DCAS in a lock-free manner is expensive [51], requiring multiple CAS operations, and implementing it via hardware transactional memory can only achieve obstruction-freedom.

Shalev and Shavit [65] presented a lock-free extendible hash table using the recursive split-ordering technique. Their hash table consists of two substructures: an ordered linked list based on the work of Michael [54], and a directory structure based on an array of arrays. The ordered list contains both data and marker nodes, where marker nodes roughly partition the list into constant-size contiguous sublists. To find an element (or its predecessor), threads first perform a constant-time traversal of the directory to locate the closest preceding "marker" node, and then inspect the sub-list that follows. A clever bit-reversal technique used on the hash value of an element ensures that as buckets are split, and new marker nodes added, the order of elements within the list need not change. Thus while resizing may require a large number of updates to the directory, the relative position of elements in the list does not change. Zhang and Larson [73] announced that they had implemented a lock-free linear hash table also using recursive split-ordering technique.

Gao et al. [23] proposed a resizable, lock-free, open addressing hash table. They maintain a second table during resizing; to migrate a key, they first mark the key as being moved, then copy it to the second table, and finally update the original key's mark to indicate that it has moved. Whenever an operation finds a marked key, it must help finish resizing the entire table, and then resume its execution on the second table. Purcell and Harris [62] proposed another lock-free open addressing hash table that is not resizable, but is space-efficient. In particular, their hash table can reuse the space occupied by deleted keys.

Lastly, Feldman, LaBorde, and Dechev [20] demonstrated that with perfect hashing, it is possible to implement a wait-free hash table. Their implementation makes use of a tree-like array-of-arrays structure, with data stored in single-element leaf arrays.

2.3 Balanced Search Structures

Fraser [22], Fomitchev and Ruppert [21], and Sundell and Tsigas [67] separately discovered the first lock-free skip list set algorithms, which support probabilistically balanced search operations. The Fraser [22] algorithm is adopted by the Java concurrency library as the canonical implementation of concurrent containers (Set and Map classes). In the Fraser algorithm, the skip list is built from a hierarchy of non-blocking linked lists (i.e. the Harris-Michael algorithm). To insert an element, the process first searches for a *window* that consists of a predecessor node with smaller key value and a successor node with larger key value. The process then tries to link the node at the bottom level linked list by a CAS, and the whole operation linearizes if the CAS succeeds. Finally, the process links the node to the linked list at each level from the bottom to the top. Similarly, a remove operation first locates a window and if the successor's key matches the specified key value, the operation tries to mark the least significant bit of the pointer at the bottom level list. The remove operation linearizes if the marking (which uses a CAS) succeeds. Logically deleted nodes are removed from the skip list by subsequent search operations.

It remained an open challenge to implement a nonblocking tree-based search data structure [14], until Ellen, Fatourou, Ruppert and Breugel [17] proposed the first nonblocking binary search tree implementation to support linearizable insert, remove and lookup operations. The nonblocking binary search tree implementation employs a technique based on intermediate nodes: an update operation that changes the child pointer of a tree node must first set the pointer to point to an intermediate "Info" record, which contains the information of the in-progress operation, such that any concurrent operation that observes the intermediate record can first help the in-progress operation to finish, before proceeding to its own operation.

Prokopec et al. [61] designed a nonblocking Trie (called C-Trie) implementation based on an intermediate node technique similar to [17]. It is worth noting that the C-Trie implementation also provides a novel feature that allows nonblocking iteration over the set elements. The iteration is implemented by copying and timestamping, and hence, imposes extra time and space overhead (and complexity) on other operations.

2.4 Priority Queues

Lotan and Shavit [50] described a method to construct quiescently consistent [34, Chapter 3] priority queues using skip list sets. In the underlying skip list, each node is augmented with a boolean field "deleted". To insert an element to the priority queue, the process simply invokes the insert method on the skip list set. To remove the minimal element, the process traverses from the head of the bottom list and attempts to use a CAS to mark the first un-deleted node as deleted, and returns the element if the CAS succeeds. The skip list based priority queue is shown to scale well in practice. However the object is not linearizable: a removeMin operation may sometimes remove an element that is not the minimum value in the priority queue. The paper also introduces an algorithm variant that provides linearizability by employing a timestamp mechanism.

The Hunt heap [37] used fine-grained locking, and avoided deadlock by repeatedly un-locking and re-locking in insert operations to guarantee a global locking order. Dragicevic and Bauer presented a linearizable heap-based priority queue that uses lock-free software transactional memory (STM) [15]. Their algorithm improved performance by splitting critical sections into small atomic regions, but the overhead of STM resulted in unacceptable performance. Another skiplist-based priority queue was proposed by Sundell and Tsigas [67]. While this implementation was lock-free and linearizable, it required reference counting.

2.5 HTM-Accelerated Implementations

A variety of combining techniques have gained prominence for their ability to accelerate concurrent data structures [30]. Unfortunately, these techniques do not perform well on search data structures and they sacrifice nonblocking progress.

Neelakantam et al. used HTM to optimize existing software [60]. Their focus was not on concurrency, but rather on speculative optimization of a program trace. The system replaced unlikely code paths with explicit transactional aborts.

Dice et al. analyzed the impact of a real HTM system on concurrent data structures [12]. They showed that many concurrent applications could be simplified by attempting to execute operations in HTM. Early work on hardware lock elision [63] suggested that a locking fallback would suffice. Calciu et al. proposed lazy subscription as optimizations to the lock-based fallback path which could have significant impact on throughput [8]. Similarly, hybrid TM researchers have embraced the need for an intermediate point between HTM execution and serialized fallback. Recently, Dice et al. pointed out several subtle pitfalls [11] in the lazy subscription technique.

Yoo et al. studied the Intel HTM implementation, applying it to high-performance-computing applications [72]. Like Dice et al., they employed HTM in ad-hoc fashion to a number of applications. They identified several techniques that can improve the performance of applications. They also presented valuable guidelines for users of Intel's HTM, such as the importance of tuning retry parameters, and the possibility of different behavior for read-only and writing hardware transactions.

Chapter 3

Unordered Linked List Based Nonblocking Sets

Our first contribution is a practical implementation of unordered linked list based set that supports nonblocking insert, remove, and lookup operations. The algorithms were published in Proceedings of the 27th International Symposium on Distributed Computing (DISC 2013) [74].

The implementation is linearizable and uses only a single-word compare-and-swap (CAS) primitive. Our wait-free implementation is built from a novel lock-free unordered list algorithm, where each insert and remove operation first linearizes by appending an intermediate "request" node at the head of the list, followed by a lazy search phase that computes the return value of the operation (which depends on whether the key value is already in the set); lookup operations have no side-effects on the shared memory. The implementation achieves scalable wait-freedom by adapting a technique originally designed for wait-free queues [41], and to further improve performance, we applied the fast-path-slow-path methodology [42] to construct adaptive variants of our algorithm.

We first present the lock-free unordered list algorithm, which serves as the basis for our wait-free implementation. The algorithm implements a set object, where the elements can be compared using an equality operator (=), even if they cannot be totally ordered.

3.1 Overview

The list supports three operations: Insert(k) attempts to insert value k into the set. It returns true (success) if k was not present in the set, and returns false otherwise. Remove(k) returns true if it successfully removes value k from the set and returns false if k did not exist in the set. Contains(k) indicates whether k is contained by the set.

Figure 3.1 presents the basic algorithm. The list is comprised of Node objects, where each Node stores a *key* value, a *next* pointer to the successor node, and a *state* field for coordinating concurrent operations. The *prev* and *tid* fields are reserved for the wait-free algorithm (Chapter 3.4). We maintain a global pointer *head* that points to the first element of the list. Elements are always inserted at the head position. The key insight of the algorithm is to maintain a refinement mapping function that maps a linked list object (starting from node h) to an abstract set object AbsSet(h):

$$\text{AbsSet}(h) \equiv \begin{cases} \emptyset & \text{if } h = \textbf{nil} \\ \text{AbsSet}(h.next) & \text{if } h.state = INV \\ \text{AbsSet}(h.next) \cup \{h.key\} & \text{if } h.state = INS \vee h.state = DAT \\ \text{AbsSet}(h.next) \setminus \{h.key\} & \text{if } h.state = REM \end{cases}$$

To maintain this property, an Insert or Remove operation first places a node with an intermediate state (*INS* or *REM*) at the head of the list. Then it searches the list for the value being inserted or removed, removing logically deleted nodes along the way. Finally, it sets the intermediate node to a final state (*DAT* or *INV*).

In more detail, an Insert operation allocates an *INS* node (h) and links it to the head of the list by invoking Enlist (lines 2 - 3). It then invokes HelpInsert (line 4) to determine whether the insertion is effective, that is, to check whether the key is already present in the set. The return value of HelpInsert dictates the return value of the Insert operation, as well as the final state of h (line 5): if

```
record Node
  key    : ℕ        // integer data field
  state  : ℕ        // INS, REM, DAT, or INV
  next   : Node     // pointer to the successor
  prev   : Node     // pointer to the predecessor
  tid    : ℕ        // thread id of the creator

shared variables
  head   : Node     // initially nil

function Insert($k : \mathbb{N}$) : $\mathbb{B}$
    $h \leftarrow$ new Node$\langle k, INS,$ nil, nil, threadid$\rangle$
    Enlist($n$)
    $b \leftarrow$ HelpInsert($h, k$)
    if $\neg CAS(\&h.state, INS, (b?\ DAT : INV))$ then
        HelpRemove($h, k$)
        $h.state \leftarrow INV$
    return $b$

function Remove($k : \mathbb{N}$) : $\mathbb{B}$
    $h \leftarrow$ new Node$\langle k, REM,$ nil, nil, threadid$\rangle$
    Enlist($h$)
    $b \leftarrow$ HelpRemove($h, k$)
    $h.state \leftarrow INV$
    return $b$

function Contains($k : \mathbb{N}$) : $\mathbb{B}$
    $curr \leftarrow head$
    while $curr \neq$ nil do
        if $curr.key = k$ then
            $s \leftarrow curr.state$
            if $s \neq INV$ then
                return $(s = INS) \vee (s = DAT)$
        $curr \leftarrow curr.next$
    return false

procedure Enlist($h$ : Node)
    while true do
        $old \leftarrow head$
        $h.next \leftarrow old$
        if $CAS(\&head, old, h)$ then
            return

function HelpInsert($h$ : Node) : $\mathbb{B}, k : \mathbb{N}$
    $pred \leftarrow h$
    $curr \leftarrow pred.next$
    while $curr \neq$ nil do
        $s \leftarrow curr.state$
        if $s = INV$ then
            $succ \leftarrow curr.next$
            $pred.next \leftarrow succ$
            $curr \leftarrow succ$
        else if $curr.key \neq k$ then
            $pred \leftarrow curr$
            $curr \leftarrow curr.next$
        else if $s = REM$ then
            return true
        else if $(s = INS) \vee (s = DAT)$ then
            return false
    return true

function HelpRemove($h$ : Node, $k : \mathbb{N}$) : $\mathbb{B}$
    $pred \leftarrow h$
    $curr \leftarrow pred.next$
    while $curr \neq$ nil do
        $s \leftarrow curr.state$
        if $s = INV$ then
            $succ \leftarrow curr.next$
            $pred.next \leftarrow succ$
            $curr \leftarrow succ$
        else if $curr.key \neq k$ then
            $pred \leftarrow curr$
            $curr \leftarrow curr.next$
        else if $s = REM$ then
            return false
        else if $s = INS$ then
            if $CAS(\&curr.state, INS, REM)$ then
                return true
        else if $s = DAT$ then
            $curr.state \leftarrow INV$
            return true
    return false
```

Figure 3.1: A Lock-free List Based Set

the key was absent from the set, *h.state* is set to *DAT*, and the insertion becomes effective; otherwise, *h.state* is set to *INV*, indicating that the insertion failed due to the key already being present in the set, and *h* becomes a garbage node that will be physically removed by some subsequent operation. The update of *h.state* must use a CAS instruction (line 5), since a concurrent Remove that deletes the same key may attempt to change *h.state* concurrently. If the CAS fails, it means the key was deleted concurrently and the thread will invoke HelpRemove (lines 6 - 7) to help the deleting thread to clean up the list.

Similarly, a Remove operation starts by inserting a *REM* node at the head position (lines 10 - 11). The real work of removal is delegated to the HelpRemove operation (line 12), which traverses the list to delete the specified key and returns a boolean value indicating whether the key was found (and deleted). Then node *h* is set to the *INV* state (line 13), allowing some subsequent operation to remove it from the list.

The Contains operation has no side effect on shared memory (it is read-only). The operation traverses the list to find the specified key and skips any *INV* nodes (lines 18 - 20). If a non-*INV* node with the specified key is encountered, the operation returns true (found) if the node is in state *DAT* or *INS* (line 21). Otherwise, the node is in *REM* state, which represents a Remove operation that can be thought of as having already deleted the key from the suffix of the list, and hence, the Contains operation immediately returns false.

Both Insert and Remove use the Enlist operation to insert a node at the head position. In the lock-free algorithm, Enlist repeatedly performs a CAS operation (line 28), attempting to change *head* to point to *h*, until the CAS succeeds. However, this approach fails to provide *wait-freedom*, because the CAS operation at line 28 of a specific thread may fail an unbounded number of times (due to contention), the thread may starve in the Enlist operation and make no progress. In Chapter 3.4, we introduce a wait-free Enlist implementation, and show the algorithm can be made wait-free without any change to the other parts.

3.2 Coordination Protocol

The core protocol of coordinating concurrency is encapsulated by the HelpInsert and HelpRemove operations. The two operations share a similar code structure:

each takes a pointer parameter h, which points to the node inserted by the prior Enlist operation. In both operations, the thread traverses the list starting from h, and reacts to the different types of nodes it encounters.

As a common obligation of both operations, logically deleted nodes are purged during the traversal (lines 35 - 38 and lines 52 - 55). That is, once an *INV* node is encountered (pointed to by *curr*), the node is physically removed from the list by setting the predecessor's *next* pointer to the successor of *curr*. Note that since new nodes cannot be added to the list at any point other than the head, the problems that plague node removal in sorted lists do not apply. In particular, it is not possible that removing one node can inadvertently lead to a new arrival disappearing from the list. While it is possible for a removed node to re-appear in the list on account of conflicting writes to the next pointer, such a node will necessarily already be marked *INV*, and thus there will be no impact on the correctness of the list.

During the traversal, the *curr* node is skipped if $curr.key \neq h.key$ (lines 39 - 41 and 56 - 58). Otherwise, we say the *curr* node is a "related node" with respect to the current operation. There are three possibilities if *curr* is a related node: *curr* is a *DAT* node, an *INS* node, or a *REM* node. In the latter two cases, the related node was created by some concurrent Insert or Remove operation. We call such operations "related operations".

In HelpInsert, if a related *REM* node is encountered, there is a concurrent Remove operation finalizing a removal of the same key. Hence, the HelpInsert returns true (success) immediately (lines 42 - 43), since the concurrent Remove operation ensures that the key is absent in the set. Otherwise (lines 44 - 45), if the related node is an *INS* node, then the related Insert operation inserted the same key earlier (or is determining that the key already exists in the list) and the HelpInsert operation must return false. Finally, if the related node is a *DAT* node, HelpInsert returns false since the key already exists in the set.

In HelpRemove, if a related *REM* node is found (lines 59 - 60), the operation returns false immediately since the key was already deleted by a concurrent Remove operation. If the related node is an *INS* node (lines 61 - 63), then the key was inserted by a concurrent Insert operation. In this case, the thread attempts to change the node from *INS* to *REM* (line 62); a CAS instruction is needed to prevent data races on the *state* field (i.e., line 5). In the last case, the related node is a *DAT* node, meaning that the key is in the set, and the node is deleted by setting its *state* to *INV* (line 65).

3.3 Correctness

To show that the algorithm is lock-free, we show that *some* operation completes when any thread executes a bounded number of local steps. We first notice that the Enlist operation is lock-free: a thread's CAS at line 28 may fail only due to another thread performing a CAS and completing its Enlist operation. Since Enlist is invoked exactly once in each Insert and Remove, for n threads, at least one list operation will complete if some thread fails the CAS for n times in its Enlist operation.

To show that every HelpInsert and HelpRemove operation terminates, it is sufficient to show the list is acyclic. There are three places where the *next* pointer of a node is changed: executing line 27 cannot form a cycle, since the node h is newly allocated and is not reachable from any other node; when a thread executes line 37 or line 54, *pred* is clearly always a predecessor of *succ* in some total order R, which can be defined as the order in which nodes are inserted to the list (by the CAS at line 28).

Since the size of the list is bounded by the total number of completed Enlist operations (denoted as E), every HelpInsert and HelpRemove operation finishes in $O(E)$ steps. Note that in HelpRemove, a thread never executes the CAS at line 62 twice on the same node: if the CAS fails, the *curr* node is changed to a final state (*DAT* or *INV*) and the loop will exit or skip the node in the next iteration. Thus, for n threads, either a thread completes its own list operation in $O(n + E)$ local steps, or some other thread completes a list operation during this period of time.

We define the linearization point for each operation: An Insert(k) or Remove(k)

operation linearizes at the successful CAS at line 28 in Enlist. A Contains(k) linearizes at line 16 if $k \notin$ AbsSet($head$) when p executes this line. In cases where $k \in$ AbsSet($head$) when p executes line 16, if the operation returns true, the Contains(k) linearizes at this line. If the operation returns false, there exists a concurrent Remove(k) that linearizes after p executes line 16 and before p's Contains(k) returns. We let p's Contains(k) linearize *immediately after* the linearization point of this Remove(k). Note that multiple Contains(k) operations may be required to linearize after the same Remove(k) operation, and any two of these Contains(k) operations can be ordered arbitrarily.

3.4 Achieving Wait-freedom

The major challenge of the wait-free list algorithm lies in the implementation of a wait-free Enlist operation. In this chapter, we present a wait-free Enlist implementation adapted from the wait-free enqueue technique introduced by Kogan and Petrank [41]. We also introduce an adaptive wait-free algorithm which allows applications to balance average latency and worst-case latency.

The enqueue technique introduced by Kogan and Petrank [41] provides a wait-free approach to append nodes at the tail of a list, but it is not immediately available as a solution to the Enlist problem where nodes are appended at the head position. We employ *prev* fields to solve this problem. The additional code for implementing a wait-free Enlist is presented in Figure 3.2.

The basic idea of the wait-free Enlist algorithm is to let different Enlist operations help each other complete. The helping mechanism must ensure that every Enlist operation reaches the response point in a finite number of steps (wait-freedom). This is achieved by requiring every thread to announce its intention by creating a descriptor entry in a *status* array before starting an operation. During its operation, the thread must visit each entry in the status array, helping other threads make progress. To prevent starvation, each operation is assigned a *phase* number from a strictly increasing counter, and an operation helps only those with

```
    record Desc
      phase     : ℕ        // integer phase number
      pending   : 𝔹        // whether op is pending
      node      : Node     // ptr to enqueueing node

    shared variables
      head      : Node
      dummy     : Node
      counter   : ℕ
      status    : Desc[THREADS]

    initially
      head ← new Node⟨−1, REM, nil, nil, −1⟩
      dummy ← new Node⟨−, −, −, −, −⟩
      counter ← 0
      foreach d in status do
        d ← new Desc⟨−1, false, nil⟩

68  procedure Enlist(h : Node)
69    phase ← FAI(&counter)
70    status[threadid] ← new Desc⟨phase, true, h⟩
71    for tid ← 0 ... (THREADS − 1) do
72      HelpEnlist(tid, phase)
73    HelpFinish()

74  function IsPending(tid : ℕ, phase : ℕ) : 𝔹
75    d ← status[tid]
76    return d.pending ∧(d.phase ≤ phase)

77  procedure HelpEnlist(tid : ℕ, phase : ℕ)
78    while IsPending(tid, phase) do
79      curr ← head
80      pred ← curr.prev
81      if curr = head then
82        if pred = nil then
83          if IsPending(tid, phase) then
84            n ← status[tid].node
85            if CAS(&curr.prev, nil, n) then
86              HelpFinish()
87              return
88        else
89          HelpFinish()

90  procedure HelpFinish()
91    curr ← head
92    pred ← curr.prev
93    if (pred ≠ nil) ∧(pred ≠ dummy) then
94      tid ← pred.tid
95      d ← status[tid]
96      if (curr = head) ∧(pred = d.node) then
97        d′ ← new Desc⟨d.phase, false, d.node⟩
98        CAS(&status[tid], d, d′)
99        pred.next ← curr
100       CAS(&head, curr, pred)
101       curr.prev ← dummy
```

Figure 3.2: Wait-free List: Enlist Operation

smaller phase numbers. The wait-free Enlist operation goes through six steps:

(a) The thread first announces its operation by creating a descriptor entry in its slot (indexed by its thread id) in the *status* array (line 70). The descriptor contains the *phase* number of the operation, a boolean *pending* field that indicates whether the operation is incomplete, and a pointer to the enlisting node. Once the descriptor is announced, the subsequent steps can be performed by the thread itself or by some helper thread.

(b) The thread finds the node pointed to by *head*, and attempts to change its *prev* field to the enlisting node h using a CAS instruction (line 85).

(c) The thread sets the *pending* flag of the operation descriptor to false by installing a new descriptor (line 98); this prevents concurrent helpers from retrying after the node is enlisted.

(d) The thread sets *h.next* to point to the original head node (line 99). The ordering of this step is important with respect to steps (b) and (e). That is, the update of *h.next* must be ordered after *head.prev* is set to *h*, since the correct successor of *h* is "unknown" until then. On the other hand, *h.next* must be updated before *head* is changed to *h*, since otherwise a concurrent CONTAINS operation may start traversing from *h* and erroneously end by discovering *h.next* is **nil**.

(e) The thread fixes *head* by changing it to *h* using a CAS (line 100), which is the linearization point of the ENLIST operation.

(f) Finally, the thread clears the *prev* field of the original head by setting it to a *dummy* state (line 101). This is necessary to allow the garbage collector to recycle deleted nodes. Since the *prev* pointers are installed by the wait-free ENLIST implementation, and the lock-free algorithm is unaware of their existence, keeping the *prev* pointers prevents the garbage collector from reclaiming a node even if the node is considered "unreachable" by the lock-free algorithm. It is worth noting that we must invalidate the *prev* pointer by setting it to a *dummy* state instead of **nil**, since the latter would admit ABA problems for the CAS instruction (line 85). Once the *prev* field of a node is set to *dummy*, it never changes.

3.5 An Adaptive Algorithm

Although the wait-free algorithm provides an upper bound on the steps required to complete an operation in the worst case, it imposes overhead in the common cases when contention is low. We employed the fast-path-slow-path methodology [42] to construct an adaptive algorithm that performs competitively in the common case while retaining the wait-free guarantee.

In the adaptive algorithm, a thread starts by executing a fast path version of the ENLIST operation, and falls back to the wait-free slow path if the fast path fails too many times (bounded by constant F). To prevent a thread from repeatedly taking the fast path while another thread starves, every thread checks the global status array after completing D operations, and performs helping if necessary. As shown in [42], for n threads, the adaptive algorithm ensures that every Enlist operation completes in $O(F + D \cdot n^2)$ local steps. The F and D parameters can be adjusted to balance worst-case and common-case latency of operations. It is worth noting that the fast path Enlist of the adaptive algorithm is *not* equivalent to the lock-free Enlist implementation in Figure 3.1. Instead, the fast path algorithm resembles the wait-free protocol, but excluding the announcing and helping steps.

3.6 Performance Evaluation

We evaluate performance of the lock-free and wait-free list algorithms via a set of microbenchmarks. These experiments allow us to vary the ratio of Insert, Remove and Contains operations, the range of key values, and the initial size of the list. We compare the following list-based set algorithms:

- **HarrisAMR**: Implementation of the Harris-Michael algorithm [54] which also incorporates the wait-free Contains technique introduced in [29]. The implementation uses Java `AtomicMarkableReference` objects to atomically mark deleted nodes.
- **HarrisRTTI**: Optimized implementation of HarrisAMR in which Java runtime type information (RTTI) is used in place of `AtomicMarkableReference`. This is the best-known lock-free list implementation.
- **LazyList**: Lock-based optimistic list implementation proposed by Heller et al [29].
- **LFList**: The lock-free unordered list algorithm discussed in Chapter 3.1.
- **WFList**: The basic wait-free unordered list algorithm discussed in Chapter 3.4.
- **Adaptive**: The adaptive wait-free unordered list algorithm discussed in Chapter 3.5.
- **FastPath**: The fast-path portion of the Adaptive algorithm from Chapter 3.5.

	Harris	LazyList	LFList	WFList	Adaptive
INSERT Cost	1 CAS	2 CAS	2 CAS	4 CAS + 1 F&I	3 CAS
REMOVE Cost	2 CAS	2 CAS	1 CAS	3 CAS + 1 F&I	2 CAS
Traverse Distance	$\frac{1}{2}k$		$(1-\frac{\alpha}{2})k$		

Figure 3.3: Update Cost and Average Traversal Distance (in uncontended cases)

In all implementations (except "HarrisAMR"), we use Java "FieldUpdaters" to perform CAS instructions on object fields. This approach provides better performance than simply using atomic fields (i.e. `AtomicInteger` and `AtomicReference`), which require expensive heap allocation cost and extra indirection overhead.

Experiments were conducted on an HP z600 machine with 6GB RAM and a 2.66GHz Intel Xeon X5650 processor with 6 cores (12 total threads) running Linux kernel 2.6.37 and OpenJDK 1.6.0. Each data point is the median of five 5-second trials.

3.6.1 Expected Overheads

Figure 3.3 enumerates the expected overheads of each of the algorithms. The cost of a successful list operation is affected by the update cost and the traversal cost. We measure the cost of an update operation (INSERT or REMOVE) by the number of atomic instructions required in the uncontended case. Compared to the Harris algorithm, LFList uses an extra CAS instruction in INSERT and one less in the REMOVE operation. The WFList requires two more CAS instructions and an extra FAI instruction to provide wait-freedom, though this cost is reduced in the Adaptive algorithm by leveraging the lock-free fast path.

The traversal cost is the average number of nodes that must be accessed. Suppose the list contains k elements uniformly selected from range $[0...M)$ and let $k = \alpha M$ $(0 \leq \alpha \leq 1)$. The average traversal distance for searching a random key value in an ordered list is: $D_o = \frac{1}{2}k$. In unordered lists, the average traversal distance is averaged among successful and unsuccessful search operations: $D_u = \alpha \cdot \frac{1}{2}k +$

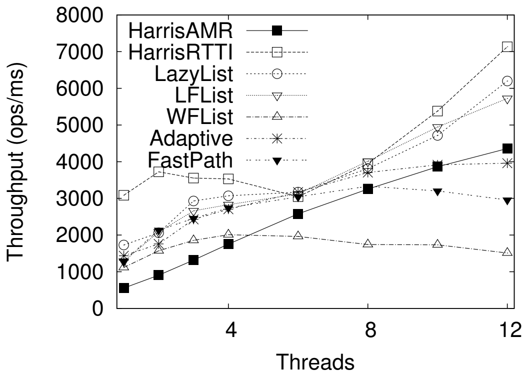
$(1-\alpha)k = (1-\frac{\alpha}{2})k$. This suggests that ordered lists have an increasing advantage over unordered lists when the set is sparse. For instance, when $\alpha = \frac{1}{2}$ (half of the key space is in the set), the average traversal distance in an unordered list is 50% longer than its ordered permutation. Note too that in the ordered lists, an unsuccessful insert/remove does not perform a CAS, whereas every insert/remove in the unordered list performs a CAS.

3.6.2 Microbenchmark Performance

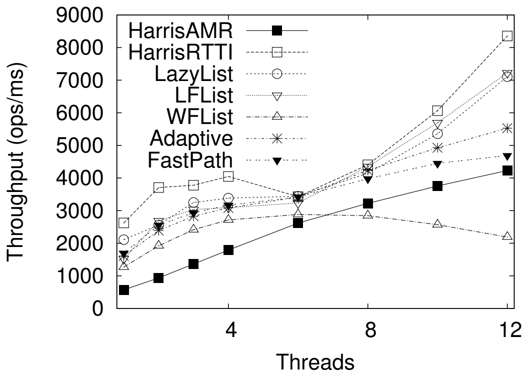
In Figures 3.4–3.6, we assess the performance of the lists for a variety of workloads. The "L" parameter indicates the percentage of operations that are lookups, with the remainder evenly split between inserts and removals. "R" indicates the key range, and "S" indicates the average size of the list. In every case, the list is pre-populated with a random selection of S unique elements in the range [0, R). These elements are chosen at random, without replacement. Thus in the unordered lists, they will not be ordered.

The x86 processor features an aggressive pipeline, a deep cache hierarchy, and low-latency CAS operations. On this platform, the cost of write-write sharing is high, and thus both the wait-free enlistment mechanism and conflicting CAS operations on the head of the list are potential scalability bottlenecks. Nonetheless, our lock-free and wait-free algorithms scale well in all but a few cases. Indeed, the difference in performance appears to be more a consequence of the increased traversal distance in the unordered algorithm than increased cache misses due to frequent updates to the head of the list.

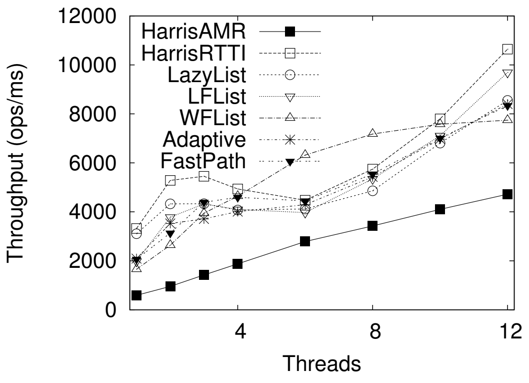
The most immediate and consistent finding is that the Harris list without RTTI optimizations has substantially higher latency and worse scalability than all other algorithms. We include this result as a reminder that concurrent data structures must be implemented using state-of-the-art techniques. Merely showing improved performance relative to the canonical Harris list presented in [34] does not give any indication of real-world performance. In particular, we caution that a direct comparison between our list and the wait-free ordered list [68] is not possible until



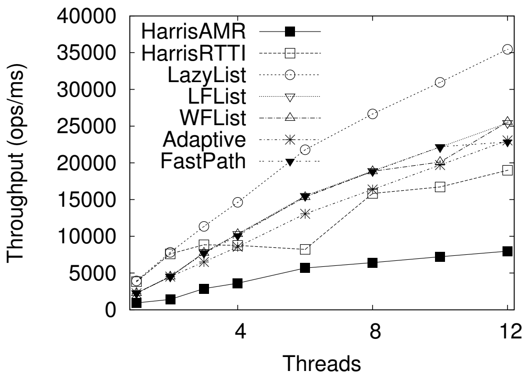
(a) L=0% R=512 S=256



(b) L=34% R=512 S=256



(c) L=80% R=512 S=256



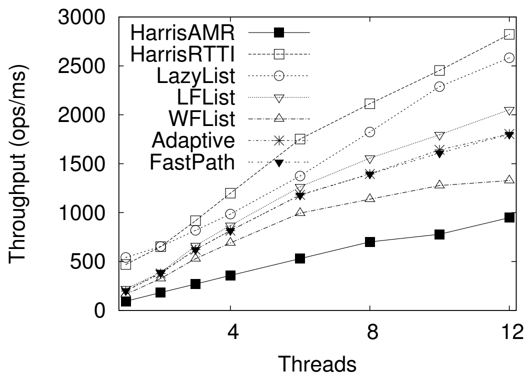
(d) L=100% R=512 S=256

Figure 3.4: Microbenchmark - Short Lists (L: Lookup Ratio, R: Key Range, S: List Size)

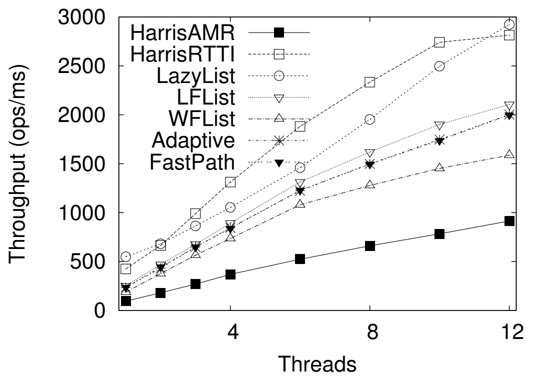
that list is redesigned to use these modern optimizations.

We also see that long-running and read-only operations significantly reduce the cost of wait-free enlistment. When lists are small and updates are frequent, the enlistment table and counter themselves become a bottleneck. Otherwise, the adaptive algorithm and its FastPath component are nearly identical.

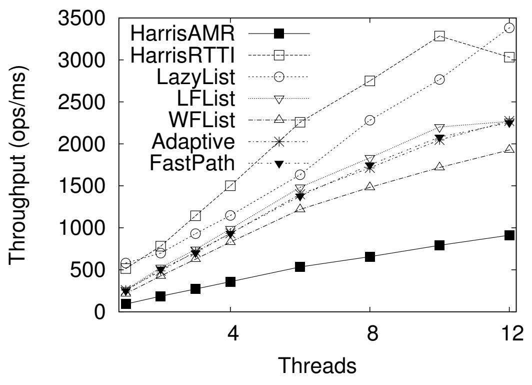
As expected, in a read-only workload, the unordered list performs about half as well as the ordered lists. This follows immediately from our analysis of the number of nodes accessed. The gap is most pronounced when the list size is 1000 elements: for smaller lists all algorithms use a small number of cache levels, and for larger



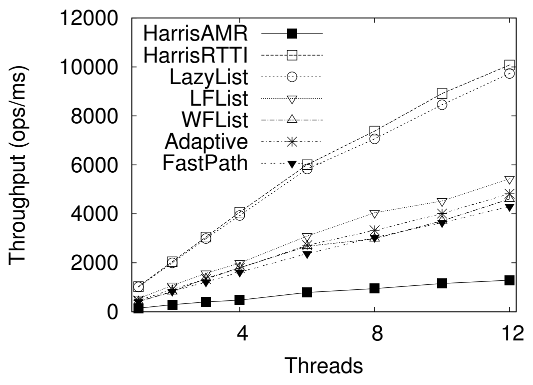
(a) L=0% R=2K S=1K



(b) L=34% R=2K S=1K



(c) L=80% R=2K S=1K

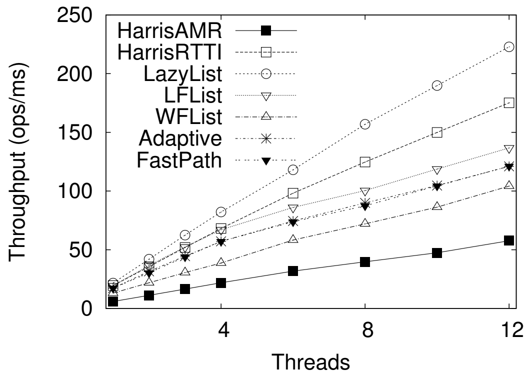


(d) L=100% R=2K S=1K

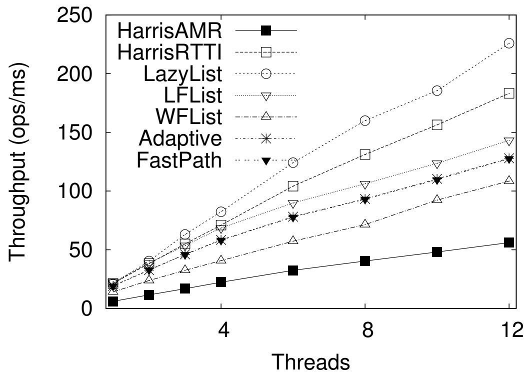
Figure 3.5: Microbenchmark - Medium Lists (L: Lookup Ratio, R: Key Range, S: List Size)

lists all algorithms use a large number of cache levels. Here, however, the ordered lists tend to keep more of their working set in the cache than the unordered lists, particularly since failed lookups traverse half as many elements.

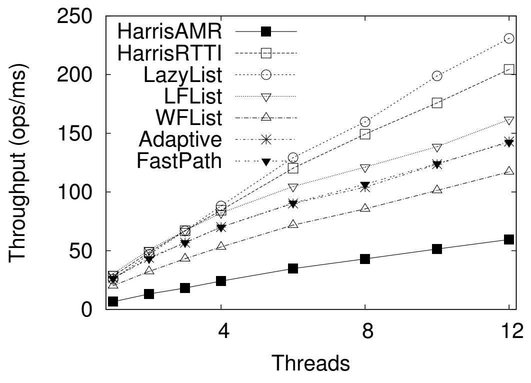
The FastPath lock-free list is always a constant factor slower than the lock-free unordered list, but the Adaptive algorithm remains close to FastPath. This finding confirms Kogan and Petrank's claim [42] that the fast-path-slow-path technique can provide worst-case wait-freedom with lock-free performance. Furthermore, since the average operation in our list accesses many locations, contention on the head node of the list, while significant, does not dominate. Thus we observed that even for small



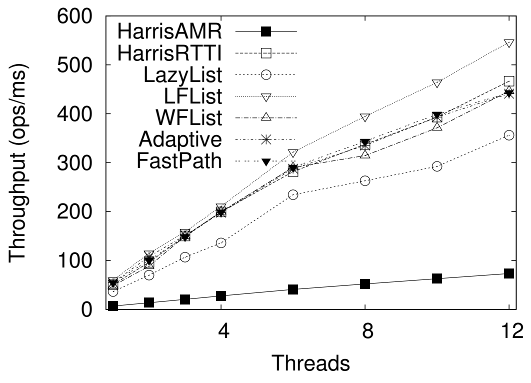
(a) L=0% R=16K S=8K



(b) L=34% R=16K S=8K



(c) L=80% R=16K S=8K



(d) L=100% R=16K S=8K

Figure 3.6: Microbenchmark - Long Lists (L: Lookup Ratio, R: Key Range, S: List Size)

thresholds, the adaptive algorithm rarely fell back to wait-free mode. However, it is important to observe that the lock-free FastPath algorithm itself is slower than our best lock-free unordered list.

3.7 Summary

In this chapter, we presented lock-free and wait-free implementations of unordered linked lists. The unordered list algorithms are suitable for use as standalone lists. In addition, we showed that the unordered nature of the list can make it substantially easier to achieve wait-freedom, and the wait-free lists can serve as the foundation for building wait-free stacks and non-resizable hash tables.

Our experience in designing the adaptive wait-free algorithm also provides insights into the fast-path-slow-path methodology [42]. We showed that in an adaptive algorithm, the lock-free fast path must be carefully designed, and sometimes modified, to be able to compose correctly with the wait-free slow path. The modification may also introduce noticeable overheads to the lock-free algorithm.

Chapter 4

Dynamic-Sized Nonblocking Hash Tables

Our second contribution is an implementation of lock-free and wait-free resizable hash tables. The algorithms were published in Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing (PODC 2014) [48].

Hash tables are often chosen as the data structure to implement concurrent set and map objects, because they offer $O(1)$ insert, remove and lookup operations. Typically, a closed-addressing hash table consists of a static *bucket array*, where each bucket is a pointer to a dynamic set object, and a *hash function* that directs operations to buckets according to the values of the operations' operands. To preserve constant time complexity when the number of elements grows, a *resize* operation (or rehash) must be performed on the hash table to extend the size of the bucket array. However, resizing a hash table in the presence of concurrent operations in a nonblocking manner is a difficult problem [65]. Shalev and Shavit proposed the split-ordered list [65], which circumvents explicit migration of key values between buckets. However, their algorithm has several limitations. A "shrinking" feature is missing in the resizing mechanism: the bucket array can only extend when the size of set grows, during which "marker" nodes are permanently inserted into the underlying linked list; it is unclear how these marker nodes can be reclaimed when the set shrinks. Furthermore, the implementation leverages the assumption that memory size is bounded and known, and relies on a tree-based indexing structure with predetermined parameters (i.e. levels and degrees).

We introduce new resizable hash table implementations that eliminate the limitations in [65], by solving the resizing problem with a direct and more efficient approach. In contrast to [65], our implementations achieve three new properties. First, they are **dynamic**: the bucket array can adjust its size both upward *and downward*, according to the size of the set. Second, the bucket array is **unbounded** and we make no assumption about the size of memory. Third, our algorithm admits **wait-free** variants, where every insert, remove, and lookup operation completes in a bounded number of steps, even in the presence of resizing. The major technical novelty of our implementations stems from the definition and use of freezable set objects. In addition to canonical set operations (i.e., insert, lookup, and remove), a freezable set provides a "freeze" operation that makes the object immutable. In our algorithms, each bucket is implemented using a freezable set. To resize a hash table, buckets in the old bucket array are frozen before their key values are copied to the new table. The migration of keys during resizing is incrementally performed in a *lazy* manner, and more importantly, the logical state of the set is never changed by migration. This ensures that every insert, remove, and lookup operation is linearizable [36].

We briefly introduce freezable set objects in Chapter 4.1, and discuss a lock-free hash set based on freezable sets (Chapters 4.2 and 4.3). A wait-free hash set algorithm is presented in Chapter 4.4. We discuss the implementations of freezable sets in Chapters 4.5 and 4.6. Finally, we present evaluation results in Chapter 4.7.

4.1 Freezable Sets

We first briefly introduce FSet, a freezable set object that serves as the common building block of our lock-free and wait-free hash table algorithms. Figure 4.1 presents the FSet specification. Discussion of nonblocking implementations of

```
abstract states of FSet
  set   : Set of $\mathbb{N}$
  ok    : $\mathbb{B}$

abstract states of FSetOp
  type  : $\{INS, REM\}$
  key   : $\mathbb{N}$
  done  : $\mathbb{B}$
  resp  : $\mathbb{B}$

GetResponse($op$ : FSetOp) : $\mathbb{B}$
  atomic
    return $op.resp$

HasMember($b$ : FSet, $k$ : $\mathbb{N}$) : $\mathbb{B}$
  atomic
    return $k \in b.set$

Invoke($b$ : FSet, $op$ : FSetOp) : $\mathbb{B}$
  atomic
    if $b.ok \land \neg op.done$ then
      if $op.type = INS$ then
        $op.resp \leftarrow op.key \notin b.set$
        $b.set \leftarrow b.set \cup \{op.key\}$
      else if $op.type = REM$ then
        $op.resp \leftarrow op.key \in b.set$
        $b.set \leftarrow b.set \setminus \{op.key\}$
      $op.done \leftarrow$ true
    return $op.done$

Freeze($b$ : FSet) : Set of $\mathbb{N}$
  atomic
    if $b.ok$ then
      $b.ok \leftarrow$ false
    return $b.set$
```

Figure 4.1: Specification of FSet and FSetOp Objects

FSet appears in Chapter 4.5 and Chapter 4.6.

An FSet object implements an integer set with insert, remove, and lookup operations, and in addition, provides a special Freeze operation. The abstract states consist of a set of integers, and an *ok* bit indicating whether the set is mutable.

Modification of an FSet object can be either insertion or removal of a key. Logically, an insert returns true if the key was not in the set, and a remove returns true if the key was in the set; otherwise, the modification operation returns false. We encode insert and remove operations as FSetOp objects. The states of a FSetOp object include the operation type (*INS* or *REM*), the key value, a boolean *done* field that indicates whether the operation is ever applied, and a boolean *resp* field that holds the return value.

Instead of letting threads invoke insert or remove operations on an FSet object, we adopt an alternative style where modifications are performed via the Invoke and GetResponse interface. The Invoke operation attempts to apply an insert or a remove operation *op* on a FSet object *b*. The operation *op* is executed only if *b* is mutable (*b.ok*) and *op* was not applied before ($\neg op.done$). In case *op* is successfully applied, it is marked as done, with the return value written in its *resp* field. The

Invoke operation returns true if *op* is (or was already) applied; otherwise, the operation returns false, in which case *b* is immutable and *op* is not applied.

The HasMember operation tests whether the given key is in the FSet object. The Freeze operation marks the given FSet object as immutable by setting its *ok* bit to false, and returns all elements of the set. The GetResponse operation returns the *resp* field of the given FSetOp object.

The Freeze operation renders an FSet permanently immutable, and also returns the final state of an FSet object. This plays prominently in nonblocking resize operations: when resizing, a thread first freezes the buckets (which are implemented using FSet objects) that will be merged or split; thereafter, keys in the frozen buckets can be safely migrated into new buckets without loss or duplication.

The role of the FSetOp's *done* bit is to ensure that every modification is applied *at most* once. This is critical to our wait-free hash set design, where threads announce operations and help each other to make progress. Using *done*, we can be sure that helping does not cause an operation to execute multiple times.

4.2 A Lock-free, Dynamic-Sized Hash Set Algorithm

Figure 4.2 presents a lock-free hash set algorithm. The hash set object provides three operations: Insert adds a key value to the set and returns true if the key was not in the set, Remove removes a key value from the set and returns true if the key was in the set, and Contains returns whether the given key is in the set.

We assume the availability of a nonblocking implementation of the FSet object. In particular, all Invoke, HasMember and Freeze operations performed on a FSet object must be lock-free with respect to the object. We also require the implementation of GetResponse to be wait-free.

Our hash set is a linked list of HNode (Hash Table Node) objects, where an HNode represents a version of the set (a new version is installed during a Resize operation). An HNode object consists of an array of FSets (*buckets*), with the

```
record HNode
  buckets   : FSet[ ]
  size      : $\mathbb{N}$
  pred      : HNode

shared variables
  head  : HNode

initially
  head ← new HNode⟨new FSet[1], 1, nil⟩
  head.buckets[0] ← new FSet⟨$\emptyset$, true⟩

Insert(k : $\mathbb{N}$) : $\mathbb{B}$
  resp ← Apply(INS, k)
  if ⟨heuristic-policy⟩ then
    Resize(true)
  return resp

Remove(k : $\mathbb{N}$) : $\mathbb{B}$
  resp ← Apply(REM, k)
  if ⟨heuristic-policy⟩ then
    Resize(false)
  return resp

Contains(k : $\mathbb{N}$) : $\mathbb{B}$
  t ← head
  b ← t.buckets[k mod t.size]
  if b = nil then
    s ← t.pred
    if $s \neq$ nil then b ← s.buckets[k mod s.size]
    else b ← t.buckets[k mod t.size]
  return HasMember(b, k)

Resize(grow : $\mathbb{B}$)
  t ← head
  if $t.size > 1 \vee grow$ then
    for i from 0 to $t.size - 1$ do
      InitBucket(t, i)
    t.pred ← nil
    size ← grow ? $t.size * 2$ : $t.size/2$
    buckets ← new FSet[size]
    $t'$ ← new HNode⟨buckets, size, t⟩
    CAS(&head, t, $t'$)

Apply(type : {INS, REM}, k : $\mathbb{N}$) : $\mathbb{B}$
  op ← new FSetOp⟨type, k, false, −⟩
  while true do
    t ← head
    b ← t.buckets[k mod t.size]
    if b = nil then
      b ← InitBucket(t, k mod t.size)
    if Invoke(b, op) then
      return GetResponse(op)

InitBucket(t : HNode, i : $\mathbb{N}$) : FSet
  b ← t.buckets[i]
  s ← t.pred
  if $b = \text{nil} \wedge s \neq \text{nil}$ then
    if $t.size = s.size * 2$ then
      m ← s.buckets[i mod s.size]
      set ← Freeze(m) $\cap$ {x | x mod t.size = i}
    else
      m ← s.buckets[i]
      n ← s.buckets[i + t.size]
      set ← Freeze(m) $\cup$ Freeze(n)
    $b'$ ← new FSet⟨set, true⟩
    CAS(&t.buckets[i], nil, $b'$)
  return t.buckets[i]
```

Figure 4.2: A Lock-free Dynamic-Sized Hash Set Implementation

array length stored in the *size* field, and a *pred* pointer that points to a predecessor HNode object. A shared pointer *head* points to the head of the HNode list. For simplicity, we make the following assumptions:

(1) A Resize operation either doubles (grows) or halves (shrinks) the size of the bucket array.

(2) The use of modular arithmetic ($index = k$ **mod** $size$) for the hash function is acceptable.

The key challenge in our algorithm is to coordinate the resizing mechanism (embodied in the RESIZE operation) with the set operations (INSERT, REMOVE, and CONTAINS).

A RESIZE operation takes a boolean parameter that indicates whether the caller intends to grow or shrink the hash table. The thread must first ensure that all the logical key values of the set are *physically* stored in the buckets of the head HNODE. This is achieved by invoking INITBUCKET on each bucket (line 23), which migrates to t those key values stored in t's predecessor but not yet in t. After the migration is complete, we set $t.pred$ to **nil** (line 24) to allow the predecessor (which is now immutable) and its buckets to be garbage collected. The thread then allocates a new HNODE t' with t as the predecessor, and uses a CAS instruction to make t' the new head HNODE (line 28). The operation does *not* initialize entries of the new bucket array: these entries are initialized lazily as they are later accessed by INSERT and REMOVE operations.

The INITBUCKET operation initializes the i-th bucket of a given HNODE t, by merging or splitting the corresponding buckets of t's predecessor HNODE (s), if s exists. The operation compares the sizes of t and s to determine whether t is growing or shrinking with respect to s, and then, freezes the corresponding bucket(s) of s before copying the elements to t. If t doubles the size of s, then (roughly) half of the elements in the (i **mod** $s.size$)-th bucket of s migrate to the i-th bucket of t (line 44). Otherwise, t halves the size of s, in which case the i-th and $(i + t.size)$-th buckets of s are merged to form the i-th bucket of t (line 48). Note that a new FSET object is allocated to store the merged or split bucket (line 49), and a CAS instruction is used to prevent races with a helping thread (line 50).

Both INSERT and REMOVE operations delegate their work to the APPLY operation (lines 2 and 7), which applies the modification to the appropriate bucket. APPLY first allocates an FSETOP object to represent the modification request (line 30), and then repeatedly attempts to apply the request to the corresponding bucket b (line 36). If b is **nil**, the thread must invoke INITBUCKET to initialize the bucket (line 35) before applying the modification. After the modification is successfully applied (line 36 returns true), the operation receives its return value via

$$\mathrm{AbsSet} \;\equiv\; \mathrm{NodeSet}(\mathit{head})$$

$$\mathrm{NodeSet}(t) \;\equiv\; \bigcup_{i\,=\,0}^{t.\mathit{size}-1} \mathrm{BuckSet}(t,i)$$

$$\mathrm{BuckSet}(t,i) \equiv \begin{cases} \mathrm{Elems}(t,i) & \text{if } \; t.\mathit{buckets}[i] \neq \mathbf{nil} \\ \mathrm{Split}(t,i) & \text{if } \; t.\mathit{buckets}[i] = \mathbf{nil} \;\wedge\; t.\mathit{pred.size} * 2 = t.\mathit{size} \\ \mathrm{Merge}(t,i) & \text{if } \; t.\mathit{buckets}[i] = \mathbf{nil} \;\wedge\; t.\mathit{pred.size} \;/\; 2 = t.\mathit{size} \end{cases}$$

$$\begin{aligned} \mathrm{Elems}(t,i) \;&\equiv\; t.\mathit{buckets}[i].\mathit{set} \\ \mathrm{Split}(t,i) \;&\equiv\; \mathrm{Elems}(t.\mathit{pred},\; i \;\mathbf{mod}\; t.\mathit{pred.size}) \;\cap\; \{x \mid x \;\mathbf{mod}\; t.\mathit{size} = i\} \\ \mathrm{Merge}(t,i) \;&\equiv\; \mathrm{Elems}(t.\mathit{pred},\; i) \;\cup\; \mathrm{Elems}(t.\mathit{pred},\; i + t.\mathit{size}) \end{aligned}$$

Figure 4.3: Refinement Mapping from Concrete Hash Sets to Abstract Sets

GetResponse (line 37).

A modification may trigger the resizing mechanism according to heuristic policies. Since the choice of policy is orthogonal to the algorithm, we leave it unspecified in our presentation (lines 3 and 8). As typical heuristics, Insert might approximate the bucket size by the number of elements it visits, and grow the hash table if the cost exceeds some threshold; upon completing a Remove, the thread may sample the sizes of randomly selected buckets and shrink the hash table if their sizes all fall below some threshold.

A Contains operation starts by searching the given key value in the corresponding bucket (b) of the head HNode. If b is not **nil**, the thread simply searches the bucket (line 18) to determine if the key value is in the set. Otherwise, the thread must trace back to s (line 15) and perform the search there. There is one troublesome interleaving, which occurs when s is resized concurrently with the Contains. Thus we must double-check (line 16) if s has become **nil** between lines 13 and 15, in which case we re-read the corresponding bucket of t (line 17), which must have become initialized prior to s becoming **nil**, and perform the search in it.

4.3 Correctness

The major goal of the linearizability proof is to show that a concrete hash set object refines an abstract set object (AbsSet), with respect to the mapping function in Figure 4.3. Intuitively, the abstract set object is defined as the union of all bucket sets of the head HNode. A bucket set (BuckSet(t, i)) is defined as the elements of the bucket (Elems(t, i)) if the bucket pointer is not **nil**, or otherwise, the union (Merge(t, i)) or intersection (Split(t, i)) of the corresponding buckets of the predecessor HNode.

We define the linearization points of Insert, Remove, and Contains as follows: An Insert or a Remove operation by thread p linearizes at an Invoke operation (line 36) that returns true (which logically sets $op_p.done$ to true). A Contains operation linearizes at the HasMember operation (line 18) if b is mutable ($b.ok$ is true) when the operation is performed; otherwise, b must have been made immutable by some concurrent Freeze operation, in which case we let the Contains operation linearize at the Freeze operation that sets $b.ok$ to false, or at p's step at line 12, whichever happens later.

To prove lock-freedom, we show that from any reachable configuration, some Insert, Remove or Contains operation completes in a bounded number of steps. First, note that a Contains operation cannot delay indefinitely between lines 12 and 17, and the final call to HasMember is lock-free by definition. An Insert or a Remove operation consists of a call to Apply and a potential call to Resize. We show that an Apply operation takes the back edge of the while loop at line 31 only if another Resize operation (called by an Insert or Remove) completes. Since we maintain the invariant that every bucket of the head HNode is mutable, for an Invoke operation of thread p to fail, p must encounter an immutable bucket. Since a bucket is made immutable only by a Resize, then for T threads, if p's Apply fails more than T times, then it means that even if $T-1$ threads were all in Resize when Apply was called, the T-th failure of p's Apply indicates that some thread must have finished its Resize, then called Apply again, indicating that it succeeded in another Insert or Remove.

Suppose S is the maximum size of the head HNode during execution. Then a Resize operation contains at most $2S$ FSet operations. Each iteration of the while loop in Apply includes at most 3 FSet operations (at most 2 in InitBucket, and one in Invoke). Therefore, at least one Insert, Remove or Contains operation must complete upon the completion of $(3T+2S)\cdot T$ FSet operations, and hence, the hash set implementation is lock-free by the assumption that the FSet operations are lock-free.

4.4 A Wait-free Hash Set Algorithm

We extend the lock-free implementation to obtain a wait-free hash set algorithm. Our wait-free hash set algorithm assumes that a *wait-free* FSet implementation is available. We start with an illustration of the main challenge of achieving wait-freedom. Recall that in our lock-free hash set algorithm, FSet objects are only required to be lock-free. Now suppose a wait-free FSet implementation is given. Does the algorithm immediately become wait-free? The answer is negative: as demonstrated in the following example, an Apply operation may take an unbounded number of steps to complete, due to concurrent Resize operations performed on the hash set object.

Let thread p attempt to insert some key value k into the hash set, and stall at the Invoke operation at line 36. Let t be the head of the HNode list and let b be the corresponding bucket where p wishes to perform the insertion. Now let another thread q complete an Insert operation that triggers a Resize operation on the hash set, after which, a new object t' becomes the head of the HNode list. Now suppose q inserts the same key value k into the hash set, and since all buckets of t' are **nil**, q invokes InitBucket to initialize the corresponding bucket of t', which freezes b, the corresponding bucket of its predecessor t. When thread p resumes, its Invoke operation will fail since b is frozen (immutable), and p will repeat the while loop in the Apply operation. The above process can repeat forever, by alternating removals and insertions of k by q, so that p's Apply operation never completes.

```
record WFOp extends FSetOp
  prio    : $\mathbb{N}$

additional shared variables
  A          : WFOp[THREADS]
  counter    : $\mathbb{N}$

initially
    counter ← 0
    for tid ← 0 to (THREADS − 1) do
        A[tid] ← new WFOp⟨−, −, −, −, ∞⟩
```

```
52 Apply(type : {INS, REM}, k : $\mathbb{N}$) : $\mathbb{B}$
53   prio ← FAI(&counter)
54   myop ← new WFOp⟨type, k, false, −, prio⟩
55   A[threadid] ← myop
56   for tid ← 0 to (THREADS − 1) do
57       op ← A[tid]
58       while op.prio <= prio do
59           t ← head
60           b ← t.buckets[op.key mod t.size]
61           if b = nil then
62               b ← InitBucket(t, op.key mod t.size)
63           if Invoke(b, op) then
64               break
65   return GetResponse(myop)
```

Figure 4.4: A Wait-free Implementation of Apply

We present a wait-free implementation of the Apply operation in Figure 4.4. The basic idea is to let threads help each other to complete their Apply operations instead of constantly competing to change and/or freeze the buckets. Our helping mechanism is similar to the doorway stage of Lamport's bakery algorithm [43].

In the wait-free algorithm, an insert or remove operation is represented using a WFOp object which adds a *prio* field to the FSetOp object. The *prio* field represents the "priority" of an operation, which dictates the operation's precedence in the helping mechanism: an operation with smaller *prio* has precedence over one with larger *prio*. The priorities of operations are generated from a strictly increasing counter (initially 0), implemented using an atomic fetch-and-increment instruction (line 53).

In Apply, thread p first allocates an WFOp object for its modification operation, associated with a unique priority, and then announces the object (line 55) in a shared array (namely A), indexed by p's thread id. Then p iterates through A and for any operation *op* announced by thread q (including p itself), if *op.prio* is smaller than (or equal to) p's most recent priority, p helps q complete (lines 59 to 64). Finally, p invokes GetResponse to get the return value of its own operation (line 65).

To prove wait-freedom, we first observe that CONTAINS is wait-free, as it does not have any loops, and the call to HASMEMBER is wait-free. To demonstrate that INSERT and REMOVE are wait-free, we show that for T threads, the inner while loop at line 58 executes at most T iterations. For thread p whose INVOKE at line 63 returns false, the head HNODE must be changed between p's line 59 and line 63, since no bucket of head can be in a frozen state. The change of head must be made by a step at line 28 of some RESIZE, indicating the completion of the outer INSERT or REMOVE operation. After T iterations of the while loop, some thread must have completed at least 2 INSERT or REMOVE operations, where the second one must have a lower priority (larger *prio*) than the priority of p. Thus, *op.done* must have been set to true, and p's INVOKE will return true in the next iteration. Therefore, APPLY operations are wait-free, since each contains at most $O(T^2)$ FSET operations, which are wait-free by assumption.

4.5 A Specialized Lock-free FSet Implementation

Figure 4.5 presents a lock-free FSET implementation specialized for the lock-free hash set in Figure 4.2. It exploits the property that in the lock-free hash set algorithm, every FSETOP object can only be applied to a bucket FSET by the allocating thread, due to absence of helping, and thus, we need not keep an actual *done* field in the FSETOP object.

The idea of the implementation is straightforward: we keep the underlying FSET objects (namely FSETNODEs) immutable, and let all updates be performed in a copy-on-write manner. We maintain a pointer *node* that points to the current FSETNODE object, which consists of the elements of the set (*set*) and a bit (*ok*) indicating whether the set is mutable. Any update to an FSET, either via an INVOKE or a FREEZE operation, must first allocate a new FSETNODE object (cloned from the current FSETNODE), then apply its change, and then finalize the modification with a CAS instruction that points *node* to the new FSETNODE.

The immutable nature of FSETNODE objects allows us to implement the inner

```
record FSetNode
  set : Set of $\mathbb{N}$
  ok  : $\mathbb{B}$

record FSet
  node : FSetNode

record FSetOp
  type : $\{INS, REM\}$
  key  : $\mathbb{N}$
  resp : $\mathbb{B}$

Freeze($b$ : FSet) : Set of $\mathbb{N}$
  $o \leftarrow b.node$
  while $o.ok$ do
    $n \leftarrow$ new FSetNode$\langle o.set, \mathbf{false}\rangle$
    if $CAS(\&b.node, o, n)$ then
      break
    $o \leftarrow b.node$
  return $o.set$

Invoke($b$ : FSet, $op$ : FSetOp) : $\mathbb{B}$
  $o \leftarrow b.node$
  while $o.ok$ do
    if $op.type = INS$ then
      $resp \leftarrow op.key \notin o.set$
      $set \leftarrow o.set \cup \{op.key\}$
    else if $op.type = REM$ then
      $resp \leftarrow op.key \in o.set$
      $set \leftarrow o.set \setminus \{op.key\}$
    $n \leftarrow$ new FSetNode$\langle set, \mathbf{true}\rangle$
    if $CAS(\&b.node, o, n)$ then
      $op.resp \leftarrow resp$
      return true
    $o \leftarrow b.node$
  return false

HasMember($b$ : FSet, $k$ : $\mathbb{N}$) : $\mathbb{B}$
  $o \leftarrow b.node$
  return $k \in o.set$

GetResponse($op$ : FSetOp) : $\mathbb{B}$
  return $op.resp$
```

Figure 4.5: A Specialized Lock-free FSet Implementation

set using any sequential algorithm. Since each bucket of a hash table tends to contain only a small number of elements, one appealing option is an unsorted array: it exploits better cache locality than list-based alternatives, and affords the compiler an opportunity to employ wide vector operations.

We also note that some simple (but useful) optimizations are elided in the pseudo code to avoid clutter. In our implementation, we let an insert (or remove) operation exit early if the key value is (or is not) in the set, thereby avoiding an unnecessary update (allocation, CAS).

4.6 A Cooperative Wait-free FSet Implementation

Figure 4.6 presents an FSet implementation designed for our wait-free hash set algorithm, where the FSet implementation "cooperates" with the helping mechanism

```
record FSetNode
  set  : Set of $\mathbb{N}$
  op   : FSetOp $\cup \{\bot\}$

record FSet
  node : FSetNode
  flag : $\mathbb{B}$

record FSetOp
  type : $\{INS, REM\}$
  key  : $\mathbb{N}$
  resp : $\mathbb{B}$
  prio : $\mathbb{N}$

Invoke($b$ : FSet, $op$ : FSetOp) : $\mathbb{B}$
  while $b.node.op \neq \bot \wedge op.prio \neq \infty$ do
    if $b.flag$ then
      DoFreeze($b$)
      break
    $o \leftarrow b.node$
    if $o.op =$ nil then
      if $op.prio \neq \infty$ then
        if $CAS(\&o.op, \textbf{nil}, op)$ then
          HelpFinish($b$)
          return true
    else
      HelpFinish($b$)
  return $op.prio = \infty$

Freeze($b$ : FSet) : Set of $\mathbb{N}$
  $b.flag \leftarrow$ true
  return DoFreeze($b$)

HasMember($b$ : FSet, $k$ : $\mathbb{N}$) : $\mathbb{B}$
  $o \leftarrow b.node$
  $op \leftarrow o.op$
  if $op \neq \textbf{nil} \wedge op \neq \bot \wedge op.key = k$ then
    return $op.type = INS$
  return $k \in o.set$

GetResponse($op$ : FSetOp) : $\mathbb{B}$
  return $op.resp$

DoFreeze($b$ : FSet) : Set of $\mathbb{N}$
  while $b.node.op \neq \bot$ do
    $o \leftarrow b.node$
    if $o.op =$ nil then
      if $CAS(\&o.op, \textbf{nil}, \bot)$ then
        break
    else
      HelpFinish($b$)
  return $b.node.set$

HelpFinish($b$ : FSet)
  $o \leftarrow b.node$
  $op \leftarrow o.op$
  if $op \neq \textbf{nil} \wedge op \neq \bot$ then
    if $op.type = INS$ then
      $resp \leftarrow op.key \notin o.set$
      $set \leftarrow o.set \cup \{op.key\}$
    else if $op.type = REM$ then
      $resp \leftarrow op.key \in o.set$
      $set \leftarrow o.set \setminus \{op.key\}$
    $op.resp \leftarrow resp$
    $op.prio \leftarrow \infty$
    CAS($\&b.node, o,$ new FSetNode$\langle set, \textbf{nil}\rangle$)
```

Figure 4.6: A Cooperative Wait-free FSet Implementation

(Figure 4.4) to achieve wait-freedom.[1]

We inherit the immutable design of the underlying set objects as in the previous lock-free FSet implementation. Now, however, the wait-free implementation must prevent duplicate execution of operations due to the presence of helping. This is achieved by leveraging the *prio* field: for any FSetOp object *op*, its abstract *done* field is **true** if *op.prio* is set to ∞, and we maintain an invariant that *op* is performed only if *op.prio* is *not* ∞.

[1]The implementation is lock-free by itself.

The key to the protocol is to let contending threads synchronize at the *op* field of an FSetNode object. To perform an FSetOp (*op*), a thread first attempts to change *node.op* from **nil** using CAS. Subsequently, the thread invokes HelpFinish to compute the return value of *op*, marks *op* as done, and replaces the current FSetNode with the result set (a new object) using CAS.

A Freeze operation first announces its intention by setting $flag$ to **true**, and invokes DoFreeze to set *node.op* to $\perp$. To show a Freeze operation is wait-free, it is sufficient to show that DoFreeze completes in a bounded number of steps. For any thread p in an DoFreeze operation, we notice that the CAS at line 123 can fail only because *node.op* is changed by a concurrent CAS at line 123 or line 102. In the former case, *node.op* is set to $\perp$ and p's while loop will terminate in the next iteration. In the latter case, any subsequent Invoke operation will see that $flag$ is set, and invoke DoFreeze. Thus, either p's CAS succeeds in its next iteration, or a concurrent DoFreeze sets *node.op* to $\perp$, which forces p to terminate its while loop in the following iteration.

An HasMember operation must first check if there exists a linearized insert or remove operation by inspecting the *op* field. This is necessary because an Invoke operation on a FSet object b may return **true** without invoking HelpFinish on b (in cases where *op* is performed by a concurrent thread), leaving b in an intermediate state. Neglecting to check the *op* field can cause a subsequent Contains operation to erroneously miss the immediately preceding insert or remove operation, which violates linearizability.

4.7 Performance Evaluation

We evaluate the performance of our hash tables via a stress-test microbenchmark. The experiments were run on a Niagara2 system with one 1.165 GHz, 64-way Sun UltraSPARC T2 CPU with 32 GB of RAM, running Solaris 10. The Niagara2 has eight cores, each eight-way multi-threaded, for a total of 64 threads. We used the Oracle JDK version 1.7.0_13. We also run experiments on an x86 system with one

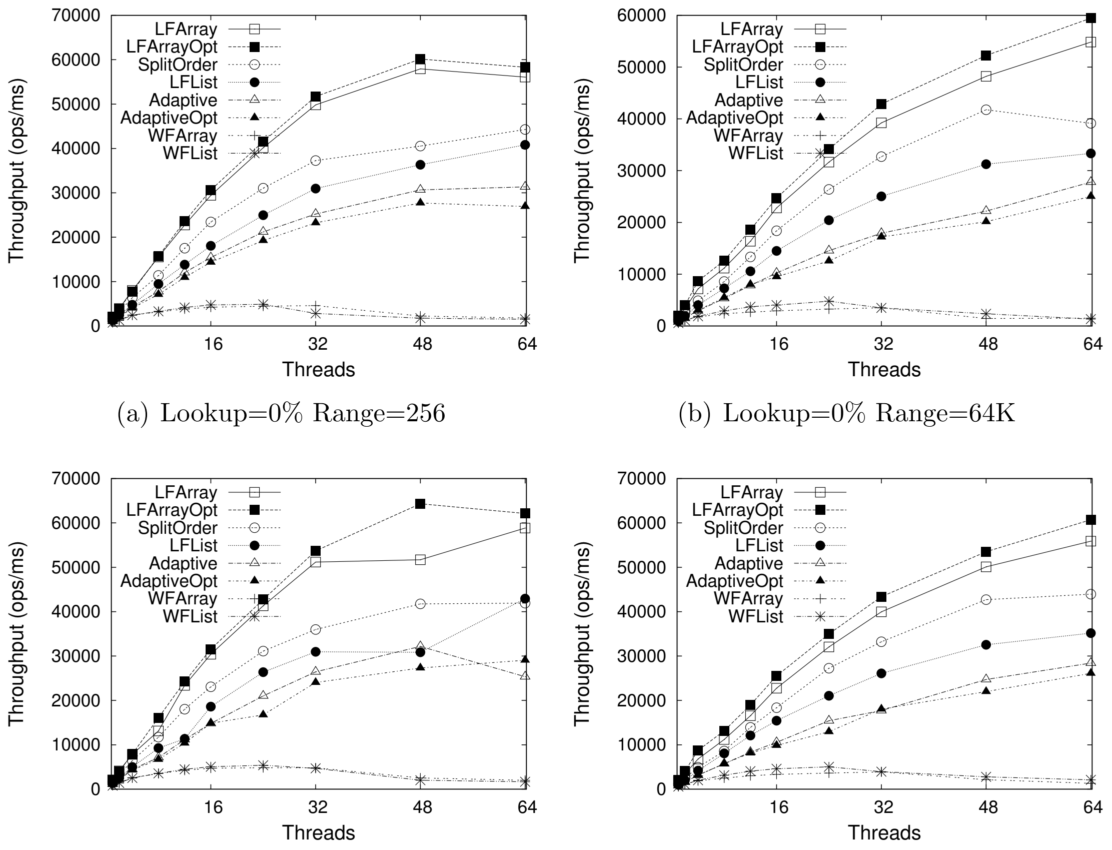
2.66GHz Intel Xeon X5650 processor and 6GB of RAM, running Linux kernel 3.11. The processor has six cores, each two-way multi-threaded, for a total of 12 threads. On the x86, we used OpenJDK version 1.7.0_51.

We compare eight implementations:

- **SplitOrder** is the algorithm proposed by Shalev and Shavit [65] and is used as our baseline. Our Java implementation of SplitOrder used the latest C++ version as a reference, to ensure a faithful implementation. To the best of our knowledge, this is the best-performing algorithm for implementing an extensible (but not shrinkable) hash table. Furthermore, the implementation optimized its configuration of the directory structure (using a two-level tree) for the size of each experiment. This ensures a minimal bucket size for the duration of each experiment. Unlike the baseline, the remaining seven algorithms were run with support for dynamic resizing.
- **LFArray** is our lock-free hash table, in which per-bucket freezable sets are implemented as arrays of unsorted elements (Chapter 4.5).
- **LFArrayOpt** removes a level of indirection from LFArray by pointing buckets directly to array elements, rather than FSET markers.
- **LFList** is similar to **LFArray**, except it uses an unsorted list implementation (Chapter 3) for its freezable sets. In addition to the above lock-free implementations, we consider four wait-free implementations, which use a wait-free FSET (Chapter 4.6).
- **WFArray** and **WFList** employ the straightforward wait-free APPLY from Figure 4.4 to make the LFArray and LFList algorithms wait-free.
- **Adaptive** applies the Fastpath/Slowpath technique [42] to reduce the overhead of WFArray.
- **AdaptiveOpt** applies the optimizations from LFArrayOpt to Adaptive. All adaptive algorithms used a threshold of 256 consecutive failures to trigger a switch to the slow path.

All implementations (to include SplitOrder) were optimized using techniques from the `java.util.concurrent` package.

Our main focus is performance in the absence of resizing operations. To this

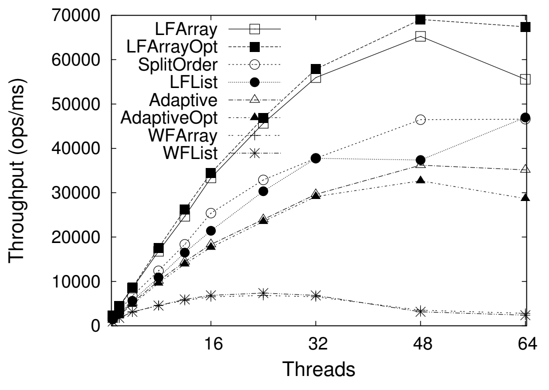


(a) Lookup=0% Range=256 (b) Lookup=0% Range=64K

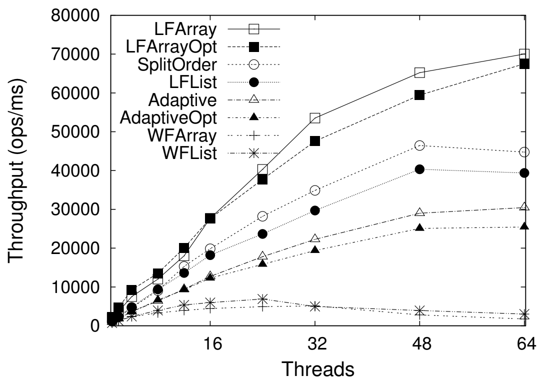
(c) Lookup=10% Range=256 (d) Lookup=10% Range=64K

Figure 4.7: Microbenchmark Performance on Niagara2 (Write-Heavy)

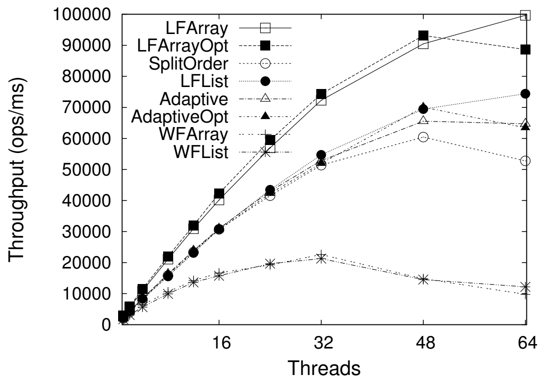
end, for a given experiment we begin by pre-populating each hash table to hold half of the experiment's key range. For a lookup ratio L, we randomly select operations such that insert and remove are chosen with equal probability $(1 - L)/2$. Thus while operations and keys are randomly selected, the number of elements in the table remains steady. We report the average of five 5-second trials. Variance was negligible.



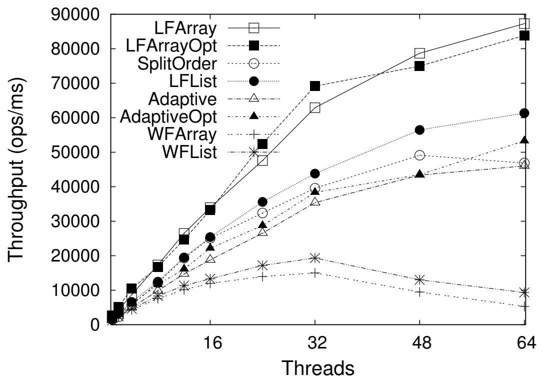
(a) Lookup=34% Range=256



(b) Lookup=34% Range=64K



(c) Lookup=80% Range=256



(d) Lookup=80% Range=64K

Figure 4.8: Microbenchmark Performance on Niagara2 (Read-Heavy)

4.7.1 SPARC Performance

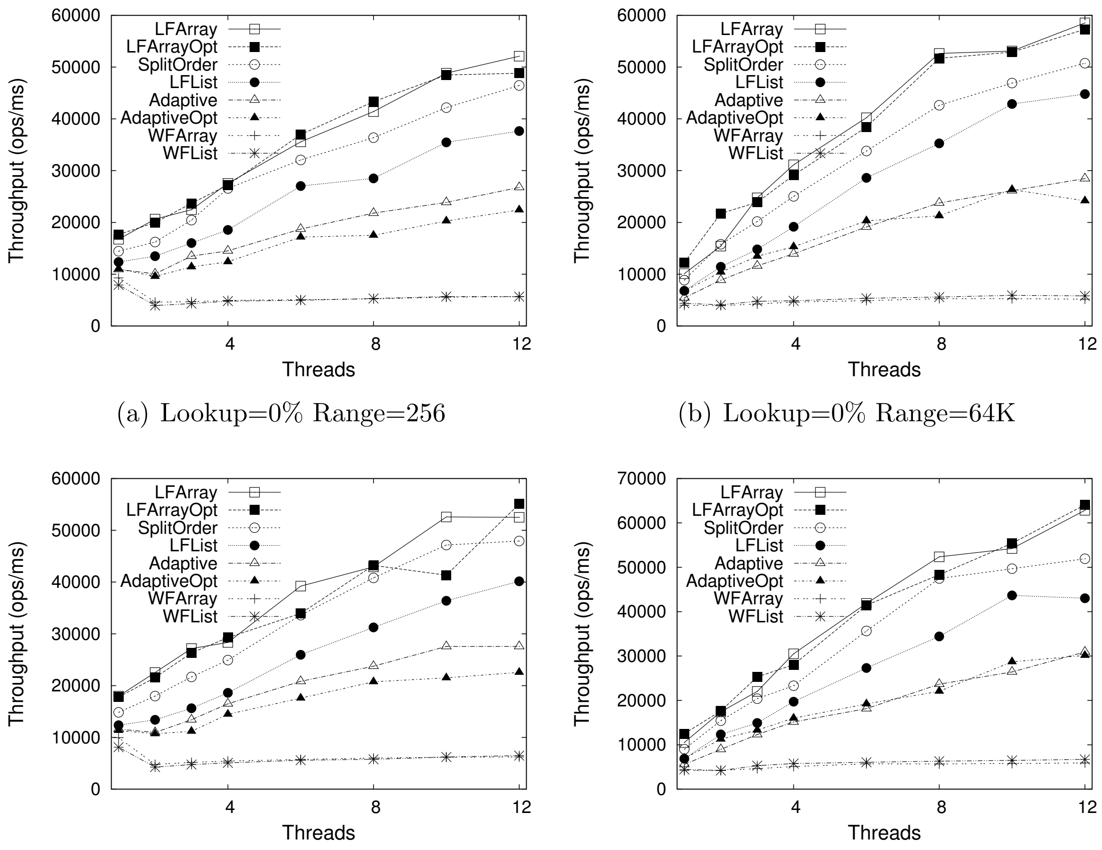
Figures 4.7 and 4.8 present performance on the Niagara2. The Niagara2 has simple in-order cores with per-core L1 caches and a shared L2 cache. While memory access latencies are low, there is no out-of-order execution to hide the latency of memory accesses: during a cache miss, another hardware thread is scheduled. For all but the smallest key range, the random distribution of keys ensures that every access will incur an L1 cache miss to dereference the bucket, regardless of the implementation. Since SplitOrder uses a sorted list, whereas LFList uses an unsorted list, SplitOrder should traverse half as many pointers, on average. However, our efficient mechanism for finding buckets keeps the gap below 2×.

When we implement each per-bucket FSet as an array, a new trade-off is introduced. On the one hand, all pointer chasing within a bucket is eliminated; on the other, insert and remove operations must copy the entire array. The high memory bandwidth of the Niagara2, coupled with the absence of pointer chasing within each bucket, result in superior performance for LFArray and LFArrayOpt. The most important factor in hash table performance on Niagara2 appears to be pointer chasing.

The WFArray and WFList implementations show limited scaling, except when lookups dominate. The poor performance is due to the cost of announcing every operation: incrementing a single global shared counter is a bottleneck, as is the use of the WFOp array for announcing operations and finding threads to help. The Fastpath/Slowpath technique does much to recover this cost. However, even though our threshold of 256 failures virtually guarantees no fallbacks to the slow path, wait-free FSet operations still carry a cost due to extra memory indirection and allocation.

4.7.2 x86 Performance

In experiments on a 6-core/12-thread Intel Xeon 5650, our algorithms behaved similarly to the Niagara2. The key differences were that there was little difference between LFArray and LFArrayOpt, and that, at high lookup ratios, the adaptive algorithms were able to close much of the gap with LFList. Given the much different microarchitecture of the X5650, the lack of significant difference between these results and those reported for the Niagara2 give confidence that the behaviors we observed were consequences of the fundamental characteristics of our algorithms, rather than architecture-specific anomalies. In particular, the locality afforded by implementing each bucket as an array enables LFArray to outperform SplitOrder in most cases.



(a) Lookup=0% Range=256

(b) Lookup=0% Range=64K

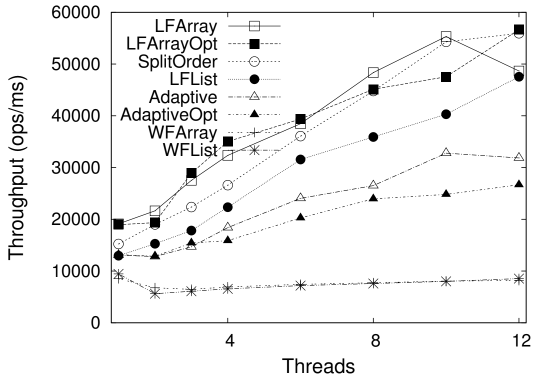
(c) Lookup=10% Range=256

(d) Lookup=10% Range=64K

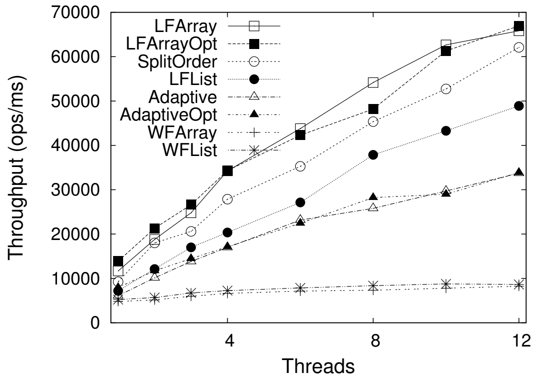
Figure 4.9: Microbenchmark Performance on x86 (Write-Heavy)

4.8 Summary

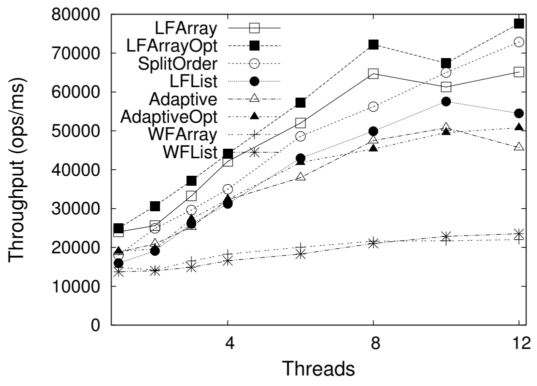
Our resizable hash table implementations differ from prior efforts in that they allow keys to be moved among buckets during a resize, without sacrificing throughput or progress. Our technique allows the hash table's buckets to be implemented as arrays, which increases locality and reduces pointer chasing. Our practical wait-free algorithms also demonstrate the resilience of the fast-path-slow-path technique [42]: it took little effort to make our lock-free algorithms wait-free, and the result was dramatically faster than a naive wait-free solution.



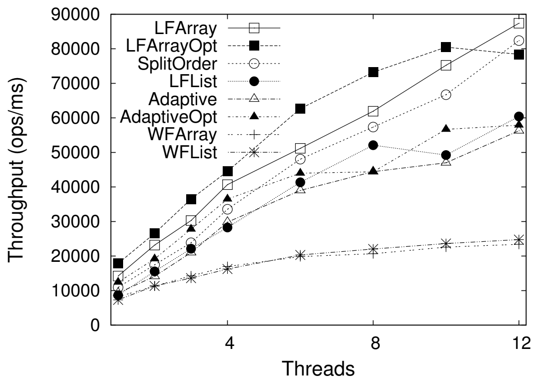
(a) Lookup=34% Range=256



(b) Lookup=34% Range=64K



(c) Lookup=80% Range=256



(d) Lookup=80% Range=64K

Figure 4.10: Microbenchmark Performance on x86 (Read-Heavy)

The FSet objects used within the hash table leaves great opportunity for improvement. Most significantly, we show (in Chapter 6) that our copy-on-write based implementation can be accelerated by hardware transactions, which significantly improves performance and preserves the lock-free/wait-free progress guarantee.

Chapter 5

Array-Based Concurrent Priority Queues

Our third contribution is a practical concurrent priority queue implementation based on a heap-like structure, which we call the "mound". The original paper was published in Proceedings of the 41st International Conference on Parallel Processing (ICPP 2012) [47].

A mound is a tree of sorted lists that can be used to construct linearizable, disjoint access parallel priority queues that are either lock-free or lock-based. Like skiplists, mounds achieve balance, and hence asymptotic guarantees, using randomization. However, the structure of the mound tree resembles a heap. The benefits of mounds stem from the following novel aspects of their design and implementation:

- While mound operations resemble heap operations, mounds employ randomization when choosing a starting leaf for an Insert. This avoids the need for insertions to contend for a mound-wide counter, but introduces the possibility that a mound tree will not be perfectly balanced.
- The use of sorted lists avoids the need to swap a leaf into the root position during ExtractMin. Combined with the use of randomization, this improves disjoint-access parallelism. Asymptotically, ExtractMin is $O(log(N))$, with roughly the same overheads as the Hunt heap [37].

```
record LNODE
  value   : $\mathbb{N}$         // value stored in this list node
  next    : LNODE     // next element in list

record MNODE
  list    : LNODE     // sorted list of values
  dirty   : $\mathbb{B}$         // false if mound property holds
  c       : $\mathbb{N}$         // incremented on every update

initially
  depth ← 1
  for i ← 0 to N do
    tree[i] ← new MNODE⟨nil, false, 0⟩
```

Figure 5.1: Mound Datatypes

- The sorted list also obviates the use of swapping to propagate a new value to its final destination in the mound INSERT operation. Instead, INSERT uses a binary search along a path in the tree to identify an insertion point, and then uses a single writing operation to insert a value. The INSERT complexity is $O(log(log(N)))$.
- The mound structure enables several novel uses, such as the extraction of multiple high-priority items in a single operation, and extraction of elements that are likely to have high priority.

5.1 Mound Algorithm Overview

A mound is a rooted tree of sorted lists. For simplicity of presentation, we consider an array-based implementation of a complete binary tree, and assume that the array is always large enough to hold all elements stored in the mound. The array structure allows us to locate a leaf in $O(1)$ time, and also to locate any ancestor of any node in $O(1)$ time. We focus on the operations needed to implement a lock-free priority queue with a mound, namely EXTRACTMIN and INSERT. We permit the mound to store arbitrary non-unique, totally-ordered values, and ∞ is the maximum value. We reserve ∞ as the return value of an EXTRACTMIN on an empty mound, to prevent the operation from blocking.

Figure 5.1 presents the basic data types for the mound. A mound consists of an array-based binary tree of MNODE objects (namely *tree*) and a *depth* field. The MNODE type describes nodes that comprise the mound's tree. Each node consists of a pointer to a sorted list, a boolean field, and a sequence number (unused in the locking algorithm). We define the value of a MNODE based on whether its list is **nil** or not. If the MNODE's list is **nil**, then its value is ∞. Otherwise, the MNODE's value is the value stored in the first element of the list, i.e., *list.value*. The VAL function is a shorthand for this computation. A mound is initialized by setting every element in the tree to $\langle \textbf{nil}, \textbf{false}, 0 \rangle$. This indicates that every node has an empty *list*, and hence a logical VAL of ∞.

In a traditional min-heap, the heap invariant only holds at the boundaries of functions, and is stated in terms of the following relationship between the values of parent and child nodes:

$$\forall p, c \in [1, N] : (\lfloor c/2 \rfloor = p) \Rightarrow \textsc{Val}(\mathit{tree}[p]) \leq \textsc{Val}(\mathit{tree}[c])$$

Put another way, a child's value is no less than the value of its parent. This property is also the correctness property for a mound *when there are no in-progress operations*. In other words, when *dirty* is not set, a node's value is no greater than the value of either of its children. Precisely, when an operation is between its invocation and response, we employ the *dirty* field to express a more localized mound property:

$$\forall p, c \in [1, N] : (\lfloor c/2 \rfloor = p) \wedge (\neg \mathit{tree}[p].\mathit{dirty})$$

$$\Rightarrow \textsc{Val}(\mathit{tree}[p]) \leq \textsc{Val}(\mathit{tree}[c])$$

Insert When inserting a value v into the mound, the only requirement is that there exist some node index c such that $\textsc{Val}(\mathit{tree}[c]) \geq v$ and if $c \neq 1$ (c is not the root index), then for the parent index p of c, $\textsc{Val}(\mathit{tree}[p]) \leq v$. When such a node is identified, v can be inserted as the new head of $\mathit{tree}[c].\mathit{list}$. Inserting v as the head of $\mathit{tree}[c].\mathit{list}$ clearly cannot violate the mound property: decreasing $\textsc{Val}(\mathit{tree}[c])$ to v does not violate the mound property between $\mathit{tree}[p]$ and $\mathit{tree}[c]$, since $v \geq \textsc{Val}(\mathit{tree}[p])$. Furthermore, for every child index c' of c, it already holds that $\text{VAL}(tree[c'])\geq\text{VAL}(tree[c])$. Since $v\leq\text{VAL}(tree[c])$, setting $\text{VAL}(tree[c])$ to v does not violate the mound property between $tree[c]$ and its children.

The INSERT method operates as follows: it selects a random leaf index l and compares v to $\text{VAL}(tree[l])$. If $v\leq\text{VAL}(tree[l])$, then either the parent of $tree[l]$ has a VAL less than v, in which case the insertion can occur at $tree[l]$, or else there must exist some node index c in the set of ancestor indices $\{\lfloor l/2\rfloor,\lfloor l/4\rfloor,\ldots,1\}$, such that inserting v at $tree[c]$ preserves the mound property. A binary search is employed to find this index. Note that the binary search is along an ancestor chain of logarithmic length, and thus the search introduces $O(log(log(N))$ overhead. The leaf is ignored if $\text{VAL}(tree[l])<v$, since the mound property guarantees that every ancestor of $tree[l]$ must have a $\text{VAL}<v$, and another leaf is randomly selected. If too many unsuitable leaves are selected (indicated by a tunable *THRESHOLD* parameter), the mound is expanded by one level. Note that INSERT is bottleneck-free. Selecting a random leaf avoids the need to maintain a pointer to the next free leaf, which would then need to be updated by every INSERT and EXTRACTMIN. Furthermore, since each node stores a sorted list, we do not need to modify a leaf and then swap its value upward, as in heaps. The number of writes in the operation is $O(1)$.

ExtractMin EXTRACTMIN resembles its analog in traditional heaps. When the minimum value is extracted from the root, the root's VAL changes to equal the next value in its list, or ∞ if the list becomes empty. This behavior is equivalent to the traditional heap behavior of moving some leaf node's value into the root. At this point, the mound property may not be preserved between the root and its children, so the root's *dirty* field is set true.

To restore the mound property at N, a helper function (MOUNDIFY) is used. It analyzes the triangle consisting of a dirty node and its two children. If either child is dirty, it first calls MOUNDIFY on the child, then restarts. When neither child is dirty, MOUNDIFY inspects the VAL of $tree[n]$ and its children, and determines which is smallest. If $tree[n]$ has the smallest value, or if it is a leaf with no children, then the mound property already holds, and the $tree[n].dirty$ field is set to false. Otherwise, swapping $tree[n]$ with the child having the smallest VAL is guaranteed

```
INSERT($v : \mathbb{N}$)
  while true do
    $c \leftarrow$ FINDINSERTPOINT($v$)
    $C \leftarrow tree[c]$
    if VAL($C$) $\geq v$ then
      $h \leftarrow$ new LNODE$\langle v, C.list \rangle$
      $C' \leftarrow$ MNODE$\langle h, C.dirty, C.c + 1 \rangle$
      if $c = 1$ then
        if $CAS(\&tree[c], C, C')$ then return
      else
        $P \leftarrow tree[c/2]$
        if VAL($P$) $\leq v$ then
          DCSS($\&tree[c], C, C', \&tree[c/2], P$)
    delete($h$)

EXTRACTMIN() : $\mathbb{N}$
  while true do
    $R \leftarrow tree[1]$
    if $R.dirty$ then
      MOUNDIFY(1)
    else if $R.list =$ nil then
      return $\infty$
    else
      $R' \leftarrow$ MNODE$\langle R.list.next$, true, $R.c + 1 \rangle)$
      if $CAS(\&tree[1], R, R')$ then
        $v \leftarrow R.list.value$
        delete($R.list$)
        MOUNDIFY(1)
        return $v$

VAL($N$ : MNODE) : $\mathbb{N}$
  if $N.list =$ nil then
    return $\infty$
  else
    return $N.list.value$

RANDLEAF($d : \mathbb{N}$) : $\mathbb{N}$
  return $random\ i \in [2^{d-1}, 2^d - 1]$;

FINDINSERTPOINT($v : \mathbb{N}$) : $\mathbb{N}$
  while true do
    $d \leftarrow depth$
    for $attempts \leftarrow 1$ to THRESHOLD do
      $leaf \leftarrow$ RANDLEAF($d$)
      if VAL($leaf$) $\geq v$ then
        return BINARYSEARCH($leaf, 1, v$)
    CAS($\&depth, d, d + 1$)
```

Figure 5.2: The Lock-free Mound Algorithm

to restore the mound property at $tree[n]$, since VAL($tree[n]$) becomes less than or equal to the value of either of its children. However, the child involved in the swap now may not satisfy the mound property with its children, and thus its *dirty* field is set true. In this case, MOUNDIFY is called recursively on the child. Just as in a traditional heap, $O(log(N))$ calls suffice to "push" the violation downward until the mound property is restored.

5.2 The Lock-Free Mound

An appealing property of mounds is their amenity to a lock-free implementation. The pseudocode for our lock-free algorithm appears in Figures 5.2 and 5.3. As is

```
MOUNDIFY($n : \mathbb{N}$)
  while true do
      $N \leftarrow tree[n]$
      $d \leftarrow depth$
      if $\neg N.dirty$ then
          return
      if $n \in [2^{d-1}, 2^d - 1]$ then
          $N' \leftarrow$ MNODE$\langle N.list,$ false$, N.c + 1\rangle$
          if $CAS(\&tree[n], N, N')$ then
              return
          continue
      $L \leftarrow tree[2 * n]$
      $R \leftarrow tree[2 * n + 1]$
      if $L.dirty$ then
          MOUNDIFY($2 * n$)
          continue
      if $R.dirty$ then
          MOUNDIFY($2 * n + 1$)
          continue
      if VAL($L$) $\leq$ VAL($R$) $\wedge$ VAL($L$) $<$ VAL($N$) then
          $N' \leftarrow$ MNODE$\langle L.list,$ false$, N.c + 1\rangle$
          $L' \leftarrow$ MNODE$\langle N.list,$ true$, L.c + 1\rangle$
          if $DCAS(\&tree[n], N, N', \&tree[2 * n], L, L')$ then
              MOUNDIFY($2 * n$)
              return
      else if VAL($R$) $<$ VAL($L$) $\wedge$ VAL($R$) $<$ VAL($N$) then
          $N' \leftarrow$ MNODE$\langle R.list,$ false$, N.c + 1\rangle$
          $R' \leftarrow$ MNODE$\langle N.list,$ true$, R.c + 1\rangle$
          if $DCAS(\&tree[n], N, N', \&tree[2 * n + 1], R, R')$ then
              MOUNDIFY($2 * n + 1$)
              return
      else
          $N' \leftarrow$ MNODE$\langle N.list,$ false$, N.c + 1\rangle$
          if $CAS\&(tree[n], N, N')$ then
              return
```

Figure 5.3: The Lock-free Mound Algorithm: Moundify

common when building lock-free algorithms, we require that every shared memory location be read atomically. We perform updates to shared memory locations using compare-and-swap (CAS), double-compare-and-swap (DCAS), and optionally double-compare-single-swap (DCSS) operations. We assume that these operations atomically read/modify/write one or two locations, and that they return a boolean indicating if they succeeded. We assume that CAS, DCAS, and DCSS do not fail spuriously. We also assume that the implementations of these operations are at least lock-free. Note that lock-free DCAS and DCSS can be simulated using CAS primitives. Given these assumptions, the lock-free progress guarantee for our algorithm is based on the observation that failure in one thread to make forward progress must be due to another thread making progress. To avoid the ABA problem, every mutable shared location (e.g., each MNode) is augmented with a counter (c). The counter is incremented on every CAS/DCAS/DCSS.

Lock-Free Moundify If no node in a mound is marked *dirty*, then every node satisfies the mound property. In order for $tree[n]$ to become *dirty*, either (a) $tree[n]$ must be the root, and an ExtractMin must be performed on it, or else (b) $tree[n]$ must be the child of a dirty node, and a Moundify operation must swap lists between $tree[n]$ and its parent in the process of making the parent's *dirty* field false.

Since there is no other means for a node to become *dirty*, the algorithm provides a strong property: in a mound subtree rooted at n, if n is not *dirty*, then Val($tree[n]$) is at least as small as every value stored in every list of every node of the subtree. This in turn leads to the following guarantee: for any node $tree[p]$ with children $tree[l]$ and $tree[r]$, if $tree[p]$ is dirty and both $tree[l]$ and $tree[r]$ are not *dirty*, then executing Moundify(p) will restore the mound property at $tree[p]$.

In the lock-free algorithm, this guarantee enables the separation of the extraction of the root's value from the restoration of the mound property, and also enables the restoration of the mound property to be performed independently at each level, rather than through a large atomic section. This, in turn, allows the recursive cleaning Moundify of one ExtractMin to run concurrently with another ExtractMin.

The lock-free Moundify operation retains the obligation to clear any *dirty* bit that it sets. However, since the operation is performed at one level at a time, it is possible for two operations to reach the same *dirty* node. Thus, Moundify(n) must be able to *help* clean the *dirty* field of the children of $tree[n]$, and must also detect if it has been helped (in which case $tree[n]$ will not be *dirty*).

The simplest case is when the operation has been helped. In this case, the read on line 46 discovers that the parameter is a node that is no longer *dirty*. The next simplest case is when Moundify is called on a leaf: a CAS is used to clear the *dirty* bit.

The third and fourth cases are symmetric, and handled on lines 63-74. In these cases, the children *tree*[*r*] and *tree*[*l*] of *tree*[*n*] are read and found not to be *dirty*. Furthermore, a swap (by DCAS) is needed between *tree*[*p*] and one of its children, in order to restore the mound property. Note that a more expensive "triple compare double swap" involving *tree*[*n*] and both its children is not required. Consider the case where *tree*[*r*] is not involved in the DCAS: for the DCAS to succeed, *tree*[*n*] must not have changed since line 46, and thus any modification to *tree*[*r*] between lines 56 and 66 can only lower $\text{Val}(tree[r])$ to some value $\geq \text{Val}(tree[n])$.

In the final case, *tree*[*n*] is dirty, but neither of its children has a smaller Val. A simple CAS can clear the *dirty* field of *tree*[*n*]. This is correct because, as in the above cases, while the children of *tree*[*n*] can be selected for Insert, the inserted values must remain $\geq \text{Val}(tree[n])$ or else *tree*[*n*] would have changed.

Lock-Free ExtractMin The lock-free ExtractMin operation begins by reading the root node of the mound. If the node is *dirty*, then there must be an in-flight Moundify operation, and it cannot be guaranteed that the Val of the root is the minimum value in the mound. In this case, the operation helps perform Moundify, and then restarts.

There are two ways in which ExtractMin can complete. In the first, the read on line 17 finds that the node's *list* is **nil** and not *dirty*. In this case, at the time when the root was read, the mound was empty, and thus ∞ is returned. The linearization point is the read on line 17.

In the second case, ExtractMin uses CAS to atomically extract the head of the list. The operation can only succeed if the root does not change between the read and the CAS, and it always sets the root to *dirty*. The CAS is the linearization point for the ExtractMin: at the time of its success, the value extracted was necessarily the minimum value in the mound.

Note that the call to Moundify on line 27 is not strictly necessary: Extract-Min could simply return, leaving the root node *dirty*. A subsequent ExtractMin would inherit the obligation to restore the mound property before performing its own CAS on the root. Similarly, recursive calls to MOUNDIFY on lines 67 and 73 could be skipped.

After an EXTRACTMIN calls MOUNDIFY on the root, it may need to make several recursive MOUNDIFY calls at lower levels of the mound. However, once the root is not *dirty*, another EXTRACTMIN can remove the new minimum value of the root.

Lock-Free Insert The simplest technique for making INSERT lock-free is to use a k-Compare-Single-Swap operation, in which the entire set of nodes that are read in the binary search are kept constant during the insertion. However, the correctness of INSERT depends only on the insertion point $tree[c]$ and its parent node $tree[p]$.

First, we note that expansion only occurs after several attempts to find a suitable leaf fail: In INSERT, the RANDLEAF and FINDINSERTPOINT functions read the *depth* field once per set of attempts to find a suitable node, and thus *THRESHOLD* leaves are guaranteed to all be from the same level of the tree, though it may not be the leaf level at any point after line 38. The CAS on line 43 ensures expansion only occurs if the random nodes were, indeed, all leaves.

Furthermore, neither the FINDINSERTPOINT nor BINARYSEARCH method needs to ensure atomicity among its reads: after a leaf is read and found to be a valid starting point, it may change. In this case, the binary search will return a node that is not a good insertion point. This is indistinguishable from when binary search finds a good node, only to have that node change between its return and the return from FINDINSERTPOINT. To handle these cases, INSERT double-checks node values on lines 4 and 11, and then ensures the node remains unchanged by updating with a CAS or DCSS.

There are two cases for INSERT: when an insert is performed at the root, and the default case.

First, suppose that v is being inserted into a mound, v is smaller than the root value (VAL($tree[1]$)), and the root is not *dirty*. In this case, the insertion must occur at the root. Furthermore, any changes to other nodes of the mound do not affect the correctness of the insertion, since they cannot introduce values $<$ VAL($tree[1]$). A CAS suffices to atomically add to the root, and serves as the linearization point (line 9). Even if the root is *dirty*, it is acceptable to insert at the root with a CAS, since the insertion does not increase the root's value. The insertion will conflict with any concurrent MOUNDIFY, but without preventing lock-free progress. Additionally, if the root is dirty and a MOUNDIFY operation on the root is concurrent, then either inserting v at the root will decrease $\textsc{Val}(tree[1])$ enough that the MOUNDIFY can use the low-overhead code path on line 77, or else it will be immaterial to the fact that line 66 or 72 is required to swap the root with a child.

This brings us to the default case. Suppose that $tree[c]$ is not the root. In this case, $tree[c]$ is a valid insertion point if and only if $\textsc{Val}(tree[c]) \geq v$, and for $tree[c]$'s parent $tree[p]$, $\textsc{Val}(tree[p]) \leq v$. Thus it does not matter if the insertion is atomic with respect to all of the nodes accessed in the binary search. In fact, both $tree[p]$ and $tree[c]$ can change after FINDINSERTPOINT returns. All that matters is that the insertion is atomic with respect to some reads that support $tree[c]$'s selection as the insertion point. This is achieved through reads on lines 4 and 11, and thus the reads performed by FINDINSERTPOINT are immaterial to the correctness of the insertion. The DCSS on line 13 suffices to linearize the INSERT.

Note that the *dirty* fields of $tree[p]$ and $tree[c]$ do not affect correctness. Suppose $tree[c]$ is *dirty*. Decreasing the value at $tree[c]$ does not affect the mound property between $tree[c]$ and its children, since the mound property does not apply to nodes that are *dirty*, and cannot affect the mound property between $tree[p]$ and $tree[c]$, or else FINDINSERTPOINT would not return c. Next, suppose $tree[p]$ is *dirty*. In this case, for line 13 to be reached, it must hold that $\textsc{Val}(tree[p]) \leq v \leq \textsc{Val}(tree[c])$. Thus the mound property holds between $tree[p]$ and $tree[c]$, and inserting at $tree[c]$ will preserve the mound property. The *dirty* field in $tree[p]$ is either due to a propagation of the *dirty* field that will ultimately be resolved by a simple CAS (e.g., $\textsc{Val}(tree[p])$ is $\leq$ the VAL of either of $tree[p]$'s children), or else the *dirty* field will be resolved by swapping $tree[p]$ with $tree[c]$'s sibling.

5.3 Additional Features

Since the mound uses a fixed tree as its underlying data structure, it is amenable to two nontraditional uses. The first, "probabilistic ExtractMin", is also available in a heap: since any `MNode` that is not dirty is, itself, the root of a mound, ExtractMin can be executed on any such node to select a random element from the priority queue. By selecting with some probability shallow, nonempty, non-root `MNode`*s* instead of the root, ExtractMin can lower contention by probabilistically guaranteeing the result to be close to the minimum value.

It is possible to execute an "ExtractMany" operation, which returns several elements from the mound. In the common case, most `MNode`s in the mound will be expected to hold lists with a modest number of elements. Rather than remove a single element, ExtractMany returns the entire list from a node, by setting the *list* pointer to **nil** and *dirty* to true, and then calling Moundify. This technique can be used to implement prioritized work stealing. Finally, mounds can be used in place of bag data structures, by executing ExtractMin or ExtractMany on any randomly selected non-null node. While lock-free bag algorithms already exist [66], this use demonstrates the versatility of mounds.

5.4 Performance Evaluation

In this chapter, we evaluate the performance of mounds using targeted microbenchmarks. Experiments labeled "Niagara2" were collected on a 64-way Sun UltraSPARC T2 with 32 GB of RAM, running Solaris 10. The Niagara2 has eight cores, each eight-way multithreaded. On the Niagara2, code was compiled using gcc 4.3.2 with –O3 optimizations. Experiments labeled "x86" were collected on a 12-way HP z600 with 6GB RAM and a Intel Xeon X5650 processor with six cores, each two-way multithreaded, running Linux 2.6.32. The x86 code was compiled using gcc 4.4.3, with –O3 optimizations. On both machines, the largest level of the cache hierarchy is shared among all threads. The Niagara2 cores are substantially simpler than the x86 cores, and have one less level of private cache.

We implemented DCAS using a modified version of the technique proposed by Harris et al [28]. The resulting implementation resembles an inlined nonblocking software transactional memory [27]. We chose to implement DCSS using a DCAS. Rather than using a flat array, we implemented the mound as a 32-element array of arrays, where the n^{th} second-level array holds 2^n elements. We did not pad `MNode` types to a cache line. This implementation ensures minimal space overhead for small mounds, and we believe it to be the most realistic for real-world applications, since it can support extremely large mounds. We set the THRESHOLD constant to 8. Changing this value did not affect performance, though we do not claim optimality.

Since the x86 does not offer non-faulting loads, we used a per-thread object pool to recycle `LNode`*s* without risking their return to the operating system. To enable atomic 64-bit reads on 32-bit x86, we used a lightweight atomic snapshot algorithm, as 64-bit atomic loads can otherwise only be achieved via high-latency floating point instructions.

5.4.1 Effects of Randomization

Unlike heaps, mounds do not guarantee balance, instead relying on randomization. To measure the effect of this randomization on overall mound depth, we ran a sequential experiment where 2^{20} INSERTs were performed, followed by $2^{19} + 2^{18}$ EXTRACTMINs. We measured the fullness of every mound level after the insertion phase and during the remove phase. We also measured the fullness whenever the depth of the mound increased. We varied the order of insertions, using either randomly selected keys, keys that always increased, or keys that always decreased. These correspond to the average, worst, and best cases for mound depth. Lastly, we measured the impact of repeated insertions and removals on mound depth, by initializing a mound with 2^8, 2^{16}, or 2^{20} elements, and then performing 2^{20} randomly selected operations (an equal mix of INSERT and EXTRACTMIN).

Table 5.1 describes the levels of a mound that have nodes with empty lists after 2^{20} insertions. For all but the last of these levels, incompleteness is a consequence of the use of randomization. Each value inserted was chosen according to one of three

Insert Order	% Fullness of Non-Full Levels
Increasing	99.96% (17), 97.75% (18), 76.04% (19), 12.54% (20)
Random	99.99% (16), 96.78% (17), 19.83% (18)

Table 5.1: Incomplete mound levels after 2^{20} insertions. Incompleteness at the largest level is expected.

Initialization	Ops	Non-Full Levels
Increasing	524288	99.9% (16), 94.6% (17), 61.4% (18), 17.6% (19), 1.54% (20)
Increasing	786432	99.9% (15), 93.7% (16), 59.3% (17), 17.6% (18), 2.0% (19), 0.1% (20)
Random	524288	99.7% (16), 83.4% (17), 14.7% (18)
Random	786432	99.7% (15), 87.8% (16), 38.9% (17), 3.6% (18)

Table 5.2: Incomplete mound levels after many ExtractMins. Mounds were initialized with 2^{20} elements, using the same insertion orders as in Table 5.1.

policies. When each value is larger than all previous values ("Increasing"), the worst case occurs. Here, every list has exactly one element, and every insertion occurs at a leaf. This leads to a larger depth (20 levels), and to several levels being incomplete. However, note that the mound is still only one level deeper than a corresponding heap would be in order to store as many elements. [1]

When "Random" values are inserted, we see the depth of the mound drop by two levels. This is due to the average list holding more than one element. Only 56K elements were stored in leaves (level 18), and 282K elements were stored in the 17th level, where lists averaged 2 elements. 179K elements were stored in the 16th level, where lists averaged 4 elements. The longest average list (14 elements) was at level 10. The longest list (30) was at level 7. These results suggest that mounds should produce more space-efficient data structures than either heaps or skiplists, and also confirm that randomization is an effective strategy.

We next measured the impact of ExtractMin on the depth of mounds. In

[1]The other extreme occurs when elements are inserted in decreasing order, where the mound organizes itself as a sorted list at the root.

Initial Size	Incomplete Levels
2^{20}	99.9% (16), 99.4% (17), 74.3% (18)
2^{16}	99.7% (13), 86.1% (14)
2^{8}	95.3% (6), 68.8% (7)

Table 5.3: Incomplete mound levels after 2^{20} random operations, for mounds of varying sizes. Random initialization order was used.

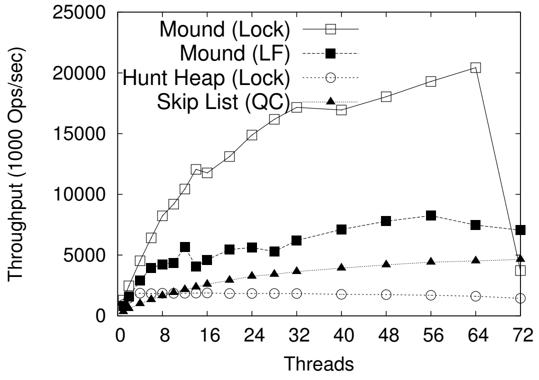
Table 5.2, we see that randomization leads to levels remaining partly filled for much longer than in heaps. After 75% of the elements have been removed, the deepest level remains nonempty. Furthermore, we found that the repeated ExtractMin operations decreased the average list size significantly. After 786K removals, the largest list in the mound had only 8 elements.

To simulate real-world use, we pre-populated a mound, and executed 2^{20} operations (an equal mix of Insert and ExtractMin), using randomly selected keys for insertions. The result in Table 5.3 shows that this usage does not lead to greater imbalance or to unnecessary mound growth. However, the incidence of removals did reduce the average list size. After the largest experiment, the average list size was only 3.

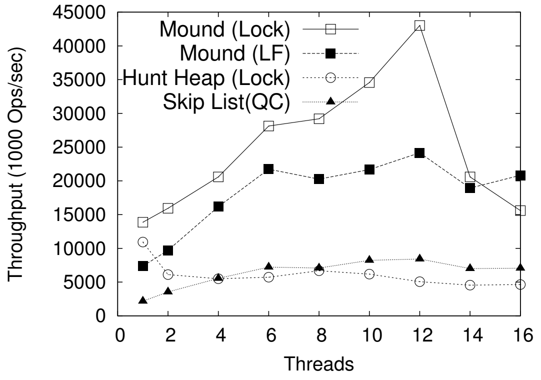
5.4.2 Insert Performance

Next, we evaluate the latency and throughput of Insert operations. As comparison points, we include the Hunt heap [37], which uses fine-grained locking, and a quiescently consistent, skiplist-based priority queue [50]. Each experiment is the average of three trials, and each trial performs a fixed number of operations per thread. We conducted additional experiments with the priority queues initialized to a variety of sizes, ranging from hundreds to millions of entries. We present only the most significant trends.

Figure 5.4 presents Insert throughput. The extremely strong performance of the fine-grained locking mound is due both to its asymptotic superiority, and its



(a) Niagara2 Insert



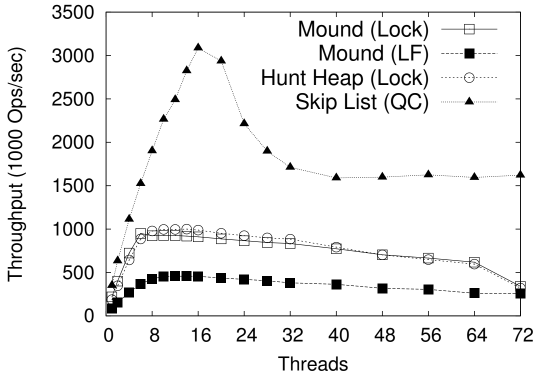
(b) x86 Insert

Figure 5.4: Insert Test: Each thread inserts 2^{16} randomly selected values.

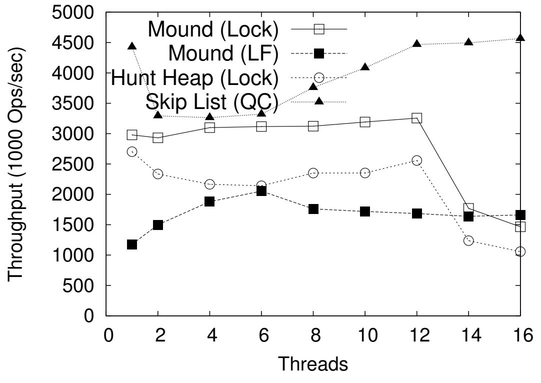
low-overhead implementation using simple spinlocks. In contrast, while the lock-free mounds scale well, they have much higher latency. On the Niagara2, CAS is implemented in the L2 cache; thus there is a hardware bottleneck after 8 threads, and high overhead due to our implementation of DCAS with multiple CASes. On the x86, both 64-bit atomic loads and DCAS contribute to the increased latency. As previously reported by Lotan and Shavit, insertions are costly for skip lists. The hunt heap has low single-thread overhead, but the need to "trickle up" causes Inserts to contend with each other, which hinders scalability.

5.4.3 ExtractMin Performance

In Figure 5.5, each thread performs 2^{16} ExtractMin operations on a priority queue that is pre-populated with exactly enough elements that the last of these operations will leave the data structure empty. The skip list implementation is almost perfectly disjoint-access parallel, and thus on the Niagara2, it scales well. To extract the minimum, threads attempt to mark (CAS) the first "undeleted" node as "deleted" in the bottom level list, and keeps searching if the marking failed. On successfully marking a node as "deleted", the thread performs a subsequent physical removal of the marked node, which mitigates further contention between operations.



(a) Niagara2 ExtractMin



(b) x86 ExtractMin

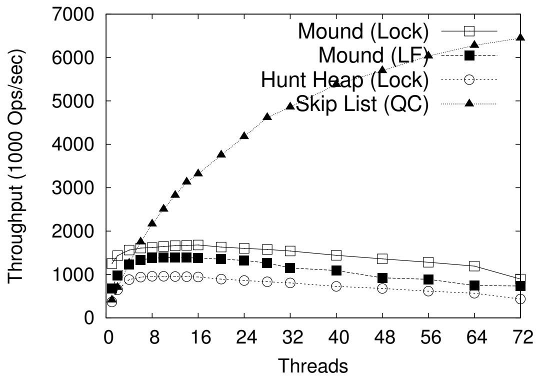
Figure 5.5: ExtractMin Test: Each thread performs 2^{16} operations to make the priority queue empty.

On the x86, the deeper cache hierarchy results in a slowdown for the skiplist from 1–6 threads, after which the use of multithreading decreases cache misses and results in slight speedup.

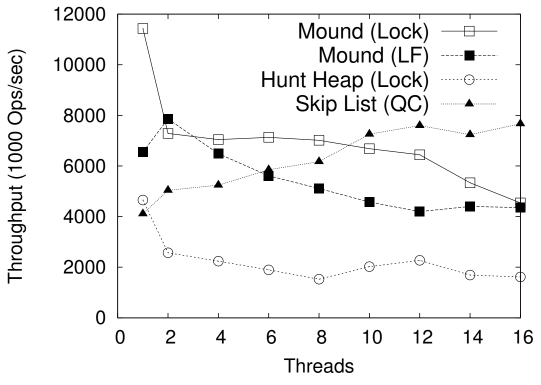
The algorithms of the locking mound and the Hunt queue are similar, and their performance curves match closely. Slight differences on the x86 are largely due to the shallower tree of the mound. However, in both cases performance is substantially worse than for skiplists. As in the Insert experiment, the lock free mound pays additional overhead due to its use of DCAS. Since there are $O(log(N))$ DCASes, instead of the single DCAS in Insert, the overhead of the lock free mound is significantly higher than the locking mound.

5.4.4 Scalability of Mixed Workloads

The behavior of a concurrent priority queue is expected to be workload dependent. While it is unlikely that any workload would consist of repeated calls to Insert and ExtractMin with no work between calls, we present such a stress test microbenchmark in Figure 5.6 as a more realistic evaluation than the previous single-operation experiments.



(a) Niagara2 Mixed



(b) x86 Mixed

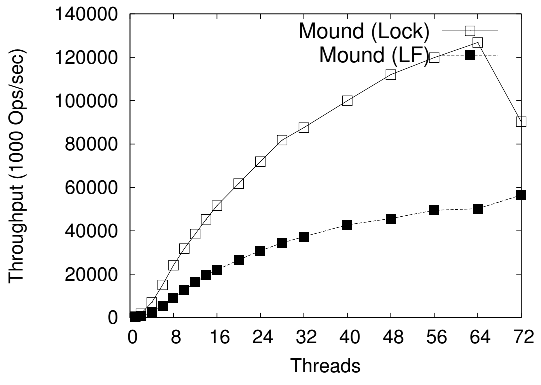
Figure 5.6: Mixed Test: Equal mix of random Insert and ExtractMin operations are performed on a queue initialized with 2^{16} random elements.

In the mixed workload, we observe the mounds provide better performance at lower thread counts. On the x86, the locking mound provides the best performance until 10 threads, but suffers under preemption. The lock-free mounds outperform skiplists until 6 threads. As in the ExtractMin test, once the point of hardware multithreading is reached, the large number of CASes becomes a significant overhead.

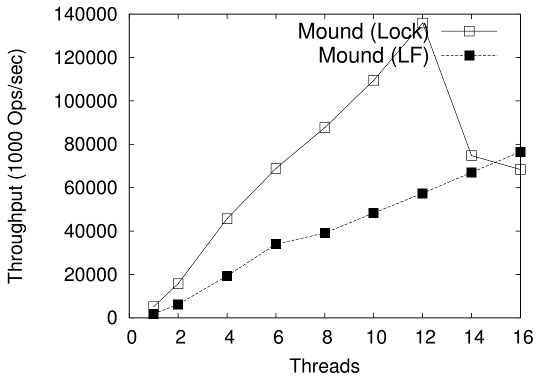
5.4.5 ExtractMany Performance

One of the advantages of the mound is that it stores a collection of elements at each tree node. As discussed in Chapter 5.3, implementing ExtractMany entails only a simple change to the ExtractMin operation. However, its effect is pronounced. As Figure 5.7 shows, ExtractMany scales well.

This scaling supports our expectation that mounds will be a good fit for applications that employ prioritized or probabilistic work stealing. However, there is a risk that the quality of data in each list is poor. For example, if the second element in the root list is extremely large, then using ExtractMany will not provide a set of high-priority elements. Table 5.4 presents the average list size and average value



(a) Niagara2 ExtractMany



(b) x86 ExtractMany

Figure 5.7: ExtractMany Test: The mound is initialized with 2^{20} elements, and then threads repeatedly call ExtractMany until the mound is empty.

Level	List Size	Avg. Value	Level	List Size	Avg. Value
0	12	52.5M	9	15.46	367M
1	15.5	179M	10	13.81	414M
2	21.75	215M	11	12.33	472M
3	21.75	228M	12	10.57	538M
4	21.18	225M	13	8.80	622M
5	20.78	263M	14	7.22	763M
6	19.53	294M	15	5.47	933M
7	18.98	297M	16	3.67	1.14B
8	17.30	339M	17	2.14	1.45B

Table 5.4: Average list size and list value after 2^{20} random insertions.

of elements in a mound after 2^{20} insertions of random values. As desired, extracted lists are large, and have an average value that increases with tree depth. Similar experiments using values from smaller ranges are even more pronounced.

5.5 Summary

We presented the mound, a new data structure for use as concurrent priority queues. The mound combines a number of novel techniques to achieve its performance and progress guarantees. Chief among these are the use of randomization and the employment of a structure based on a tree of sorted lists. Linearizable mounds can be implemented in a highly concurrent manner using either pure-software DCAS or fine-grained locking. Their structure also allows several new uses. We believe that prioritized work stealing is particularly interesting.

In our evaluation, we found mound performance to exceed that of the lock-based Hunt priority queue, and to rival that of skiplist-based priority queues. The performance tradeoffs are nuanced, and will certainly depend on workload and architecture. Workloads that can employ ExtractMany or that benefit from fast Insert will benefit from the mound. The difference in performance between the x86 and Niagara2 suggests that deep cache hierarchies favor mounds.

The lock-free mound is a practical algorithm despite its reliance on software DCAS. We believe this makes it an ideal data structure for designers of new hardware. In particular, we demonstrate (in Chapter 6) that the effectiveness of new concurrency primitives, such as hardware transactional memory, will be easier to address given algorithms like the mound, which can serve as microbenchmarks and demonstrate the benefit of faster hardware multiword atomic operations.

Chapter 6

Transactional Acceleration of Concurrent Data Structures

Our fourth contribution is to introduce the Prefix Transaction Optimization (PTO), a set of techniques that employ hardware transactional memory to accelerate existing concurrent data structures. The original paper was published in Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2015) [49].

We explore how HTM might benefit the design and implementation of nonblocking concurrent data structures. Specifically, we propose a methodology, called Prefix Transaction Optimization (PTO), by which HTM can be used to accelerate an existing implementation. There are three components of PTO, which vary in terms of the degree to which they can be automated, the amount of implementation-specific knowledge needed, and the potential gain. In the first step, we create a prefix transaction to execute a sequence of steps in the existing implementation, which uses HTM but may fail. In the second step, we mechanically optimize this prefix through strength reduction and elimination of corner cases [60], and other classic compiler optimizations. In the third step, we modify the original algorithm so as to introduce minimal "overhead" while affording more aggressive optimization of the prefix transaction.

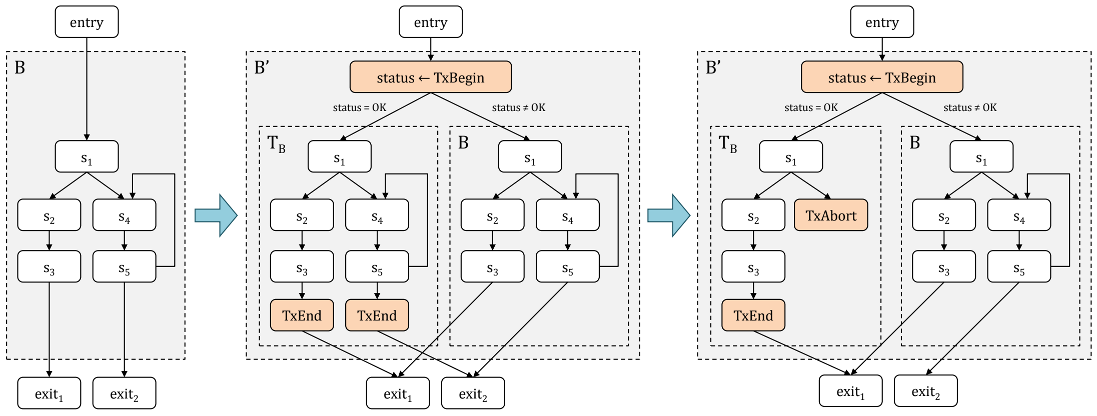


Figure 6.1: Prefix Transaction Transformation

PTO offers many compelling properties. It preserves the progress guarantees of the original algorithm, which is an improvement over many approaches to transactional acceleration of concurrent programs. It is also a composable technique, which can be applied at multiple levels of granularity. PTO optimizations can be linked together, and the benefits of doing so are (more or less) additive. Lastly, and most significantly, PTO can dramatically improve performance. We observe speedups of up to 1.5x at one thread, and up to 3x at 8 threads, on state-of-the-art nonblocking data structures.

6.1 Prefix Transaction Optimization

In this chapter, we present the algorithm-agnostic aspects of the Prefix Transaction Optimization (PTO) technique. Enhancements and modifications specific to a single data structure or class of data structures are discussed in Chapter 6.2.

6.1.1 Model

The PTO technique is applicable to concurrent objects implemented in shared memory using read/write registers and common synchronization primitives (e.g.

compare-and-swap, fetch-and-add, etc). The object interface defines a set of invocable operations.

We adopt the *control flow graph* representation [59] for each operation (and its sub-operations), where a node represents a step in the algorithm and an edge represents a transition in the control flow. We assume each operation has a single *start* node. For two nodes a and b in a control flow graph, *a dominates b* if any code path from start to b includes a. A *superblock* is a connected sub-graph where all nodes are dominated by a single *entry* node. An edge from a node within a superblock to one outside is called an *exit* edge.

We assume that HTM is supported by the architecture via three instructions: `TxBegin` starts a transaction, `TxEnd` commits the transaction, and `TxAbort` causes the transaction to abort. The `TxBegin` instruction can return more than once: if a transaction cannot commit for any reason, then the effects of the transaction are undone, and control returns to the point of `TxBegin` with a return value indicating the cause of the inability to commit. A return value of `OK` indicates that the code is running as a transaction.

HTM is assumed to provide strong atomicity [6]. When a hardware transaction is running, none of its effects are visible to any concurrent code; all effects become visible atomically at `TxEnd`. During the execution of the transaction, if any concurrent transaction performs a conflicting access, the HTM will choose (at least) one transaction to abort. If any nontransactional code performs a conflicting access, the transaction will immediately abort. Upon any abort, control will return to `TxBegin`, where the program can decide whether to attempt the transaction again.

6.1.2 The Prefix Transaction Transformation

Given a superblock B in a control flow graph G, the Prefix Transaction Transformation (illustrated in Figure 6.1) is constructed by attempting to execute the superblock using a hardware transaction. If that attempt fails, then the original version of code is invoked, without the use of a transaction. More precisely:

Definition 1. *Let B be a superblock of a control flow graph G. The Prefix Transaction Transformation is a function $\mathcal{T}_{\mathcal{B}}(G)$ that maps G to G', a copy of G with B replaced by B', such that:*

- *T_B is a copy of T where a* ***`TxEnd`*** *instruction is inserted at each exit edge, and zero or more* ***`TxAbort`*** *instructions are inserted at any edges of T except the exit edges;*
- *S is a* ***`TxBegin`*** *instruction with a branch to the dominator of T_B if the return value is* `OK` *and a branch to the dominator of B otherwise;*
- *Let B' be the superblock dominated by S with all nodes of T_B and B included.*

Given a transformation $\mathcal{T}_{\mathcal{B}}(G)$ defined in Definition 1, we say B' is the *optimized superblock*. Inside B', we say T_B is the *prefix transaction* of B, and B is the *fallback*.

The following theorems capture basic properties of the Prefix Transaction Transformation. We first prove the correctness of our transformation, by constructing a refinement mapping [3] from the transformed implementation to the original (Theorem 2). We then prove the progress guarantee of the original implementation is preserved by the transformation (Theorem 3). Finally, implied by the theorems, we observe that the program may choose to explicitly abort a transaction at any point (within the transaction) without compromising correctness or progress conditions.

Theorem 2 (Refinement)**.** *Let G be the control flow graph of some operation of an implementation I, and let I' be the implementation with $\mathcal{T}_{\mathcal{B}}(G)$ applied to I. I' refines I.*

sketch. The mapping of states is simply an identical function that maps the states of I' to the states of I. For a process p taking a step in I', if the step is not a transactional instruction or access, we let p take a corresponding step in I. For a `TxBegin`, `TxAbort`, or a transactional access step in I', p takes no step in I. For a `TxEnd` step in I', let k be the number of steps p has taken in-between the `TxEnd` the last `TxBegin` step, we let process p take k steps in I. □

Theorem 3 (Progress Preservation)**.** *Let G be the control flow graph of some operation of a lock-free (or wait-free) implementation I, and let I' be the implementation with $\mathcal{T}_{\mathcal{B}}(G)$ applied to I. I' provides lock-free (or wait-free) progress.*

sketch. Suppose I is lock-free. Then some operation in I completes if process p takes a bounded number of steps. For a given configuration c of I, let k be this bound. Then at most k steps are spent in superblock B before some operation completes, and since B contains at least one step, it can be executed no more than k times before some operation completes.

Let f be the refinement mapping constructed in Theorem 2. For a configuration c' in I' where $c = f(c')$, process p can spend at most k steps in a transaction (excluding the `TxBegin`, `TxAbort` and `TxEnd` steps) before some operation completes. A committed or aborted transaction takes at most $(k+3)$ steps including the `TxBegin`, `TxAbort` and `TxEnd` steps. In case the transaction aborts, at most $(2k+3)$ steps are spent to execute the optimized superblock. Hence, we know in configuration c', some operation completes within $k \cdot (2k+3)$ steps taken by process p.

Proving the preservation of wait-free progress employs the similar technique. □

6.1.3 Optimizing Prefix Transactions

We now turn our discussion to how to optimize the prefix transaction. We first present optimizations that can be easily identified and performed by a compiler using canonical static analyses.

Eliminating Synchronization: Correctness proofs of concurrent data structures often assume sequential consistency [44]. Implementations, in turn, must entail memory fences to enforce explicit ordering on architectures with weaker memory models.

Within a prefix transaction, memory fences can be elided, since they are subsumed by the implicit memory fences of `TxBegin` and `TxEnd` instructions, and atomic synchronization primitives, such as compare-and-swap and read-modify-write operations, can be replaced with their corresponding loads, stores, and branches.

Eliminating Redundant Loads: *Double-checking* is a technique used in many concurrent data structures [17, 57]. Implementations employ double-checking to ensure a consistent view of multiple memory locations. In the prefix transaction, a single read to a shared location suffices, since the second read will always return the same value (given the transaction does not perform a write to the location in-between the reads); any conflicting write to the location will cause the transaction to abort.

For implementations that use the atomic compare-and-swap primitive, the compare-and-swap is usually attempted after a preceding read to the location. Since in a transaction we convert the compare-and-swap to a read followed by a conditional write, the read (produced by the conversion) can coalesce with the former read.

We also observe that many search data structures [17,71] employ a *search phase*, followed by an *update phase* that performs its writes after validating selected locations accessed in the search phase. These implementations are likely to benefit from the elimination of redundant loads enabled by our transformation.

Eliminating Redundant Stores: Nonblocking data structures often exploit *intermediate states* during an update operation to allow helping from concurrent threads. The size of intermediate states may vary from unused bits embedded in the data fields [26,46] to complex, dynamically-allocated auxiliary structures [17,61]. Fundamentally, these intermediate states are introduced to overcome the difficulty that traditional synchronization primitives can update only a single word at a time.

It is commonly seen in nonblocking algorithms [17,47,61,64] that operations first attempt to change several locations from a clean state to some intermediate state, and then restore them back to a clean state. Given that an update is performed within a transaction, and all stores to a location appear atomically, the temporary change to intermediate states can be eliminated. Furthermore, if dynamic memory allocations are involved to create intermediate objects, these allocations can be eliminated together with the silent stores, mitigating pressure on the shared allocator object.

In concurrent data structures using hazard pointers [55] or reference counts [67] to manage dynamic memory, intermediate updates to the hazard lists (i.e., insertion followed by removal) or to the reference counters (i.e., increment followed by decrement) can be safely eliminated as redundant stores in the prefix transaction.

6.1.4 Avoiding Helping in Prefix Transactions

Although helping is the key idea behind many nonblocking concurrent data structures, it tends to increase contention among threads in some cases [29, 42, 47, 58].

When a prefix transaction observes states in which it must perform helping to make progress, it may be preferable to simply abort the transaction and switch to executing the lock-free fallback. The rationale governing such decision is twofold: First, when a prefix transaction determines to help, the situation suggests a concurrent operation is accessing locations touched by the transaction (and vice versa) and is likely to create a conflict that causes the transaction to abort. Thus, the explicit abort can serve as an ad-hoc backoff mechanism to avoid the contention in the first place. Second, if the prefix transaction is optimized (as discussed in Chapter 6.1.3) so that it does not introduce intermediate states, it can be desirable to maximally avoid helping (which introduces intermediate states) in the prefix transaction for the sake of improving total throughput.

From a pragmatic perspective, we argue that it is fairly straightforward for a concurrent data structure designer to identify the helping code paths in the algorithm, and decide whether to replace them with explicit aborts in the prefix transactions. Examples of how to make such choices are discussed in subsequent chapters. On the other hand, we found that in most nonblocking algorithms, a helping code path can be defined as an unreachable sub-path in the control flow graph of a single-threaded execution. A trivial example is the code to handle a failed compare-and-swap operation. Using this definition as a heuristic, an optimizing compiler can collect information from a single-threaded profile run, and (approximately) identify the helping paths for making optimization decisions.

6.1.5 Recursive Optimizations

Prefix Transaction Transformation is a *local* optimization, which means it can be applied to a whole operation or to individual components (superblocks) of the operation. More importantly, the optimization can be repeatedly applied on optimized code until achieving the best performance.

The simplest example of an recursive optimization is to allow an aborted prefix transaction to retry before attempting the fallback. For instance, the following transformation attempts the same prefix transaction T_B twice before switching to the fallback:

$$\mathcal{T}_{\mathcal{B}}(\mathcal{T}_{\mathcal{B}}(G))$$

A more powerful use of recursive optimization is to compose optimizations in a hierarchical structure. Suppose that in the control flow graph G of some operation, superblock B is a sub-graph of superblock A. The following transformation:

$$\mathcal{T}_{\mathcal{B}}(\mathcal{T}_{\mathcal{A}}(G)) \text{ where } B \subset A$$

first attempts the prefix transaction T_A, and in the fallback path of T_A, the program can still benefit from the optimizations of $\mathcal{T}_{\mathcal{B}}(G)$.

Hierarchical composition has an important impact in practice: Applying the transformation on larger superblocks maximizes the opportunity for eliminating redundancy (i.e. loads, stores, and fences), but makes it harder for transactions to make progress under contention, and thus, hurts scalability. Applying the transformation on smaller superblocks facilitates making progress, but reduces the opportunity to reduce latency. Composing optimizations makes it possible to achieve low latency and high scalability at the same time. We also notice that, by Theorem 3, applying the transformation for a bounded number of times preserves the progress guarantees of the original implementation.

6.2 Applying Prefix Transaction Optimization

The PTO technique presented in Chapter 6.1 does not require much algorithm-specific knowledge, though a programmer with knowledge about expected common paths may insert explicit aborts to increase optimization opportunities. We now turn our attention to the technical details of applying PTO to specific concurrent data structures, including additional algorithm-specific optimizations.

Mindicators We first consider the Mindicator data structure [46]. Like SNZI [18] and the f-array [40], the Mindicator is a static-sized tree that computes a function over a set of values, where each thread offers at most one value as an input to the function. The original Mindicator algorithm uses a marking phase to traverse from a per-thread leaf up to some point in the tree, and unmarks nodes as it traverses back to the leaf. Unlike f-array, not all operations must traverse to the root; unlike SNZI, additional functions (min, max) are supported in addition to 0/1 saturating addition.

The application of PTO to the Mindicator did not make use of any algorithm-specific optimizations, primarily because the tree is static and hence there is no memory allocation. By applying PTO, the marking and unmarking steps could be coalesced: marking and unmarking were previously both implemented as increments to a per-node counter; with PTO, the counter is incremented once, by two. This, in turn, eliminated the downward traversal entirely. After applying PTO, we tuned the threshold for retries before PTO falls back to the lock-free slow path. A choice of three attempts yielded the best performance.

Mounds We also applied PTO to the mound, a heap-like data structure that implements a priority queue (discussed in Chapter 5). Like the Mindicator, the mound is a tree-shaped data structure. However, it is a tree of sorted lists, where each list is only modified at its head. We did not choose the mound because it is the best nonblocking heap or priority queue. We chose it instead for the value it adds when evaluating PTO. Specifically, the mound employs double-compare-and-swap (DCAS) and double-compare-single-swap (DCSS) operations throughout its implementation, to perform atomic updates on up to two locations. This afforded an opportunity to evaluate the impact of applying PTO locally, e.g., to individual DCAS and DCSS operations.

In the mound, insertion consists of a search, followed by a double-compare-single-swap (DCSS), which is implemented in software through a sequence of CAS instructions. Removal entails performing a CAS to remove the top of the heap, and then several DCAS operations to restore invariants at the root and then on its children, recursively. Insertions can barely benefit from PTO, because they are streamlined and contention-free already: the heap itself is a static tree, obviating memory management overheads, and the insertion entails a log-log-depth traversal and just one simulated DCSS. Similarly, employing PTO on the entire removal operation is not effective at any level of concurrency, since all concurrent removals contend at the top of the heap. However, it is profitable to use PTO on a sub-operation of insert and removal, namely the DCAS/DCSS operations.

Skip Lists Lock-free skip lists [22] are a widely used search data structure to implement concurrent maps and sets. In the skip list algorithm, an update operation first locates the predecessor and successor nodes of a given key value, and then uses a sequence of compare-and-swap operations to link/unlink the nodes into the hierarchy of lists.

We experimentally determined that local application of PTO was the only promising technique. We proceeded to apply PTO only to the insert and remove operations. In an insert operation, we use a prefix transaction to update the next pointers of the predecessors. Similarly, in a remove operation, we attempt to mark the deleted node's next pointers using a single transaction, instead of performing individual compare-and-swap operations.

Nonblocking Binary Search Trees We now discuss our experience with applying PTO to the nonblocking binary search tree (BST) algorithm created by Ellen et al. [17]. The algorithm implements a set object with insert, remove, and lookup operations. To achieve lock-freedom, the algorithm employs a "marking" technique to coordinate concurrent updates to the BST. During an insert or a remove operation, the thread first traverses down the tree (the search phase) to locate an appropriate position to perform the update. Then in the update phase, the thread allocates an operation descriptor (Info record) that contains sufficient information to allow helping from other threads. The descriptor is installed at nodes involved in the update, using compare-and-swap operations: one node is marked in an insertion and two are marked in a removal. An operation linearizes if it successfully marks all nodes involved in the update. Upon completion, some of the nodes are restored to a clean state.

We identify two opportunities to apply PTO in the binary search tree algorithm. The first is to put the entire update operation inside a transaction. The second is to use a prefix transaction to execute the update phase, leaving the search phase out of the transaction. In both choices, we can eliminate the allocation of the descriptor for an insert operation, because the node is restored to a clean state at the end of the transaction. For a remove operation, since the algorithm does *not* restore one of the updated nodes to a clean state, we cannot safely eliminate its descriptor. However, we can use a unique, statically-allocated dummy descriptor in place of a dynamically allocated one: When all updates are performed in a transaction, there is no need for helping if the transaction commits, and the dummy descriptor is simply ignored by subsequent operations.

Dynamic-Sized Hash Tables The final data structure we studied is a nonblocking resizable hash table (discussed in Chapter 4). The algorithm employs a "freezable set" abstraction to achieve nonblocking size adjustments. In the hash table, each bucket is a pointer to a freezable set object, which is implemented as an unsorted array of elements. All updates to the array are performed via copy-on-write, that is, by creating an updated version of the array to replace the old one, and then using a compare-and-swap on the bucket pointer.

A straightforward application of PTO on the hash table appears barely helpful, since the algorithm is streamlined: In the common case, an insert or a remove operation on the hash table consists of a single allocation and an uncontended compare-and-swap on the bucket pointer. To improve performance, we changed the algorithm by removing the copy-on-write within transactions.

The idea of our optimization is to perform *speculative in-place writes* to the array objects, so that allocations could be avoided in the common case. When making this change, we attached a counter to the bucket pointers, so that a transactional update could increment the counter and modify the bucket in place. Unfortunately, this can affect the correctness of a concurrent lookup to the bucket. To prevent errors, we degrade the progress of lookups from wait-free to lock-free, by requiring lookups to double-check the bucket pointer counter after they search the bucket.

6.3 Evaluation

In this chapter, we evaluate the effectiveness of PTO in accelerating concurrent data structures. We consider five data structures as discussed in Chapter 6.2, which affords us the ability to look at the various aspects of PTO in detail.

6.3.1 Microbenchmarks

We use three microbenchmarks in our experiments:

setbench evaluates of set implementations which support insert, remove, and lookup operations. Each thread repeatedly invokes a lookup or an update operation (with equal chance of being an insert or a remove) with some random value within range.

pqbench evaluates priority queue implementations where each thread repeatedly invokes a INSERT with some random value or a EXTRACTMIN; the EXTRACTMIN returns a null value if the queue is empty.

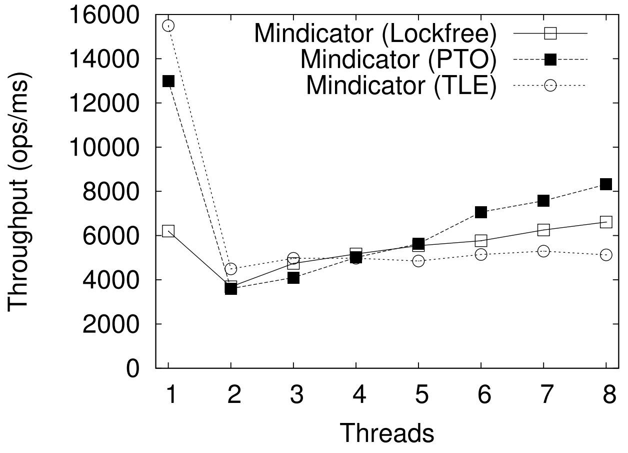


Figure 6.2: Mindicator Microbenchmarks

mbench evaluates Mindicator objects where each thread repeatedly invokes an arrive operation with some random value, followed by a depart operation.

All experiments were conducted on an machine equipped with an Intel Core i7-4770 CPU running at 3.40GHz, with 8 GB of RAM. The i7-4770 supports Intel's Restricted Transactional Memory (rtm) interface. There are 4 cores, each 2-way multi-threaded, for a total of 8 hardware threads. The software stack included Ubuntu 14.04.1 and GCC 4.8.2. All experiments were run in 32-bit mode, and data points are the average of 5 trials.

6.3.2 Latency and Scalability Improvement

Figure 6.2 contrasts the performance of the PTO Mindicator with the original lock-free implementation. We also compare to a version in which the Mindicator is protected by a coarse-grained lock, and transactional lock elision (TLE) [63] is employed to allow concurrency. In the experiment, threads repeatedly insert and then remove a randomly-chosen value; this ensures that some operations must traverse to the top of the tree. We configured the Mindicator as a binary tree with 64 leaves, and used the default mapping, where threads were assigned to leaves from left to right.

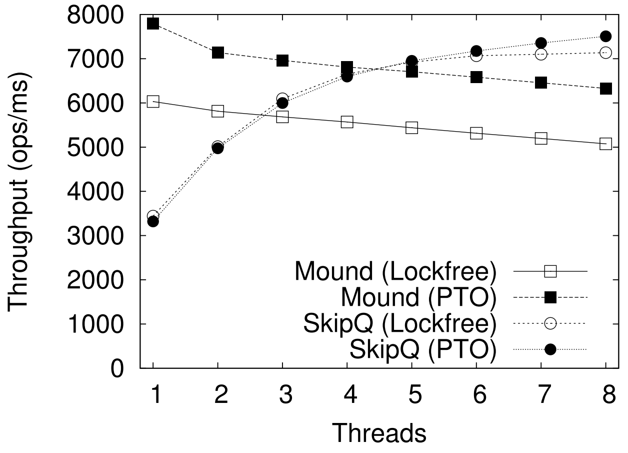
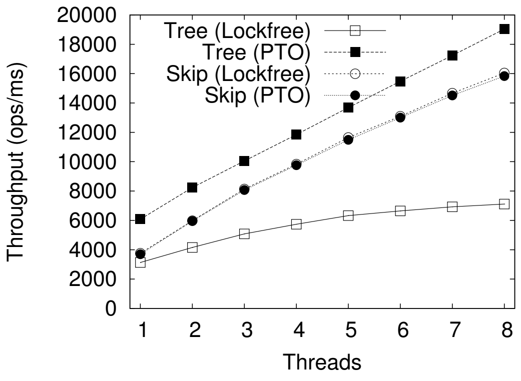


Figure 6.3: Priority Queue Microbenchmarks

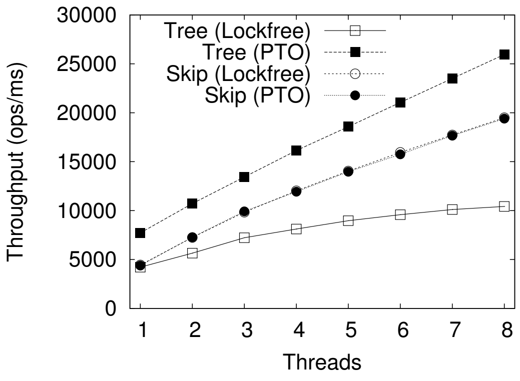
There are two important trends: First, we see that at a single thread, PTO provides latency that is nearly as good as TLE, which does not have marking, unmarking, or helping phases. Thus we can conclude that PTO can provide near-optimal single-thread performance. Second, we see that whereas TLE scales poorly, due to its locking fallback, PTO scales comparably to the original lock-free code. Thus in all cases, PTO is on par with the best performing algorithm. Furthermore, beyond 4 threads we see that PTO scales better than the lock-free code. This is a natural consequence of the workload: when using random keys, as the number of threads increases, the likelihood that any thread must traverse to the root decreases. As fewer threads traverse to the root, the likelihood of conflicts for any thread also decreases, and the prefix transaction becomes more likely to succeed.

Figure 6.3 shows performance for a workload with an even mix of insert and removeMin operations on the mound, using random keys. Using PTO, we were able to replace up to five CAS operations with a single transaction for each of the DCAS and DCSS operations. We encapsulated the DCAS in a function, and tuned the retry parameter once, ultimately settling on a value of four. This value was used for all DCASes, whether at the (high contention) root of the mound, or at leaves.

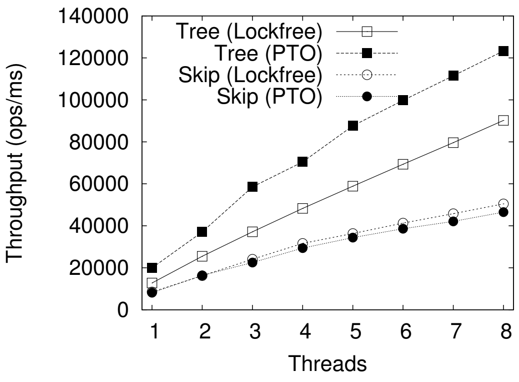
The main benefit of PTO for the mound was in removing latency from each



(a) Lookup=0% Range=512



(b) Lookup=34% Range=512



(c) Lookup=100% Range=512

Figure 6.4: Logarithmic Search Data Structure Microbenchmark

DCAS. This result is similar to the finding of Yoo et al. [72], that coarsening atomic regions via hardware transactions can amortize some of the costs of atomicity. In terms of concurrent data structure design, the lesson is that thinking in terms of DCAS and other simpler primitives remains useful: assuming the availability of DCAS allowed the mound to split ExtractMin into multiple atomic operations, thereby limiting the duration of contention on the mound root.

6.3.3 Impacts on Relative Performance

We next turn our attention to skiplists. We evaluate skiplists in two settings: as a search data structure (Figure 6.4) and as a priority queue (Figure 6.3).

We began with Gramoli's skiplist implementation [24]. To create a skiplist priority queue, we employed a modified version of the Lotan-Shavit technique [50], and made it linearizable by disallowing a pop operation from traversing through an marked node.

While we expected to observe a similar decrease in latency to the mound, due to the reduced latency for coarsened atomic operations, such benefit did not manifest. There are two drivers of this result. First, the main source of latency is not silo maintenance, but accessing locations that are not in the cache, during the traversal stage. According to the criteria in [10], the skiplist implementation is already close to optimal with respect to concurrency. Thus at one thread, there was little to gain. The second impediment to speedup at higher thread counts is that as a silo maintenance operation traverses the silo, it becomes increasingly prone to conflicts with concurrent readers. Intel TSX employs a requester-wins [7] conflict detection strategy, and thus any read to the write set of an PTO operation causes the PTO operation to fail.

6.3.4 Additive Benefits in Recursive PTO

PTO is a compositional technique, and can be used to optimize an entire operation, as well as a portion of its fallback path. To assess this property, we evaluated the nonblocking BST created by Ellen et al. [17]. We transliterated the code from Java to C++, replacing `volatile` variables with sequentially consistent `std::atomic` variables. We also employed an epoch-based memory reclamation policy, to ensure that locations were not reclaimed while a concurrent thread held a reference to them.

We identified two applications of PTO to the BST, which we refer to as PTO1 and PTO2. In PTO1, the entire insert, remove, and lookup operations are transformed using PTO. By optimizing the lookup phase, we are able to remove code that double-checks the values of reads. We were also able to replace sequentially

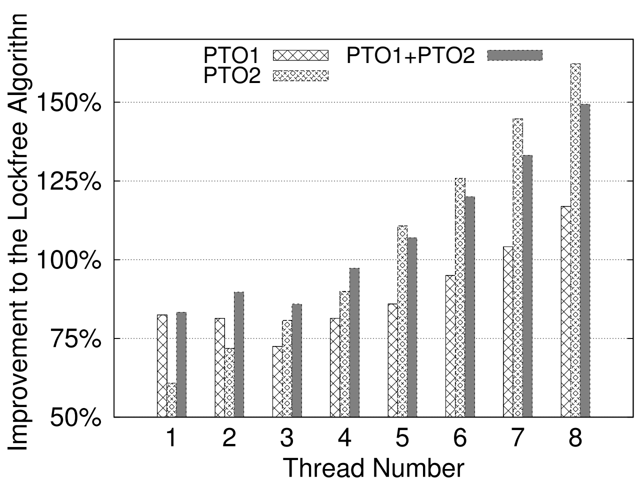
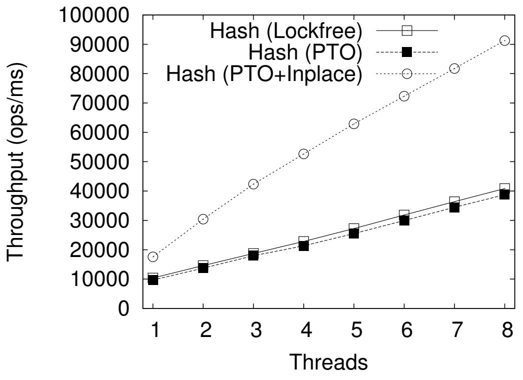


Figure 6.5: Composition of PTO on a Binary Search Tree

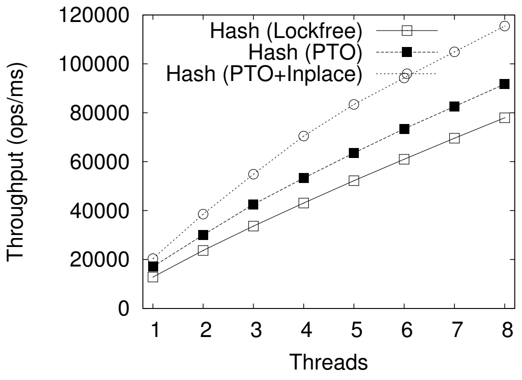
consistent `std::atomic` accesses with relaxed accesses, which may avoid processor and compiler fences on some architectures.

As Figure 6.5 shows, PTO1 results in more than 75% higher throughput at low thread counts. In contrast, PTO2 only optimizes the update phase of the insert and remove operations. While it also offers an improvement at all thread counts, the effect is much less at low levels of concurrency, where search overhead dominates, but much higher as concurrency increases. The improvement at higher thread counts is a consequence of a smaller contention window: since the traversal is not part of the hardware transaction, there are fewer opportunities to conflict with concurrent transactions. However, the lookup phase no longer runs in a hardware transaction, and thus must incur the overheads of double-checking and fences.

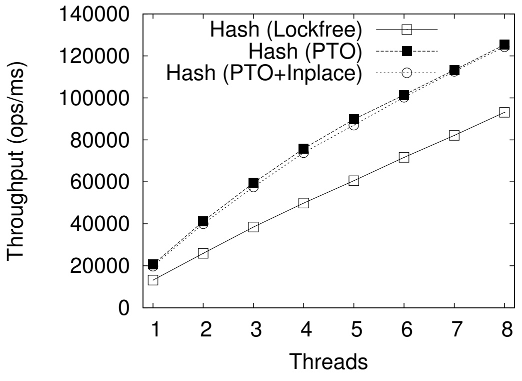
In PTO1+PTO2, we employ PTO1, and then use PTO2 within the fallback path. To fall back all the way to the original lock-free algorithm, an operation must first fail 2 times in PTO1, and then 16 times in PTO2. This composition achieves close to the best of both approaches. Even more remarkably, the composition of PTO+PTO2 boosts the BST performance to a constant factor higher than the skiplist set. As Figure 6.4 shows, the optimized BST provides the same scalability as the skiplist, but with lower latency.



(a) Lookup=0% Range=64K



(b) Lookup=80% Range=64K



(c) Lookup=100% Range=64K

Figure 6.6: Hash Table Microbenchmark

6.3.5 Fast Speculative In-place Updates

We ported the hash table from Java to C++, again using an epoch-based memory reclamation policy. We applied PTO to each of the insert, lookup, and remove operations, and then performed algorithm-specific optimizations to eliminate copy-on-write.

The simple application of PTO does little to benefit updates, since their overhead is dominated by the cost of allocating a new bucket, copying the old bucket's values, and applying the corresponding insert or removal. However, lookup operations show a decrease in latency. When PTO is applied to the lookup, all interaction with the

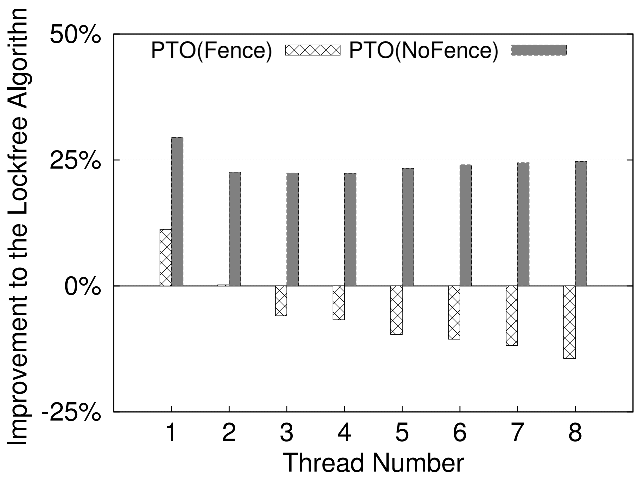


Figure 6.7: Fence Elimination on Mound

epoch-based reclaimer can be elided. This eliminates two memory fences and two stores. Given the streamlined code path, there is a noticeable impact on latency.

Figure 6.6 presents the performance improvement for this optimization. In a write-only workload, we observe more than 2x speedup at 8 threads, and 1.8x speedup at one thread. The improvement is a consequence of the elimination of copying, and reduced interaction with the allocator. Since the allocator can require system calls, and its metadata can present a bottleneck, the benefits increase at higher thread counts.

6.4 What Makes PTO Fast?

Our evaluation shows some dramatic improvements in performance, particularly for the BST and hash table. However, it also shows some more modest gains, and fails to improve the skiplist at all. While the methodology does simplify the task of accelerating a concurrent data structure, it is still beneficial to be able to analyze an algorithm and predict whether it will benefit from PTO.

Generalizing our above experiments, we believe that there are four principal sources of latency that PTO can eliminate:

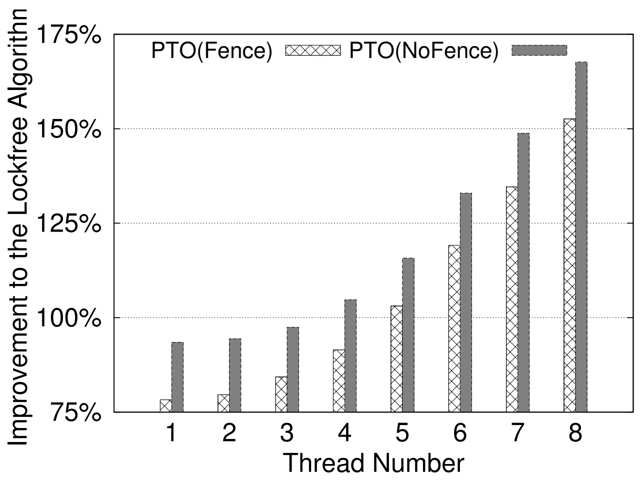


Figure 6.8: Fence Elimination on Binary Search Tree

Memory Fences: Figure 6.7 and 6.8 present additional results for the mound and BST, showing the impact when we did not elide memory fences within hardware transactions. For both the mound, where the placement of fences was hand-optimized, and the BST, where the placement of fences mirrored their placement in the equivalent (and necessarily conservative) Java code, we see that the elimination of fences contributed significantly to savings in latency. For the mound, the impact of removing fences was the sole source of improvement. For the BST, fences were a component of a suite of techniques that decreased latency.

Double-Checking Reads: Double-checked reads introduce two costs: not only do they add more instructions to the operation, but they also introduce branches, for when the check fails. In Figure 6.8, we break down sources of reduced latency for the BST write-only experiment. While fence removal plays a significant role, the baseline improvement comes without it. At low thread counts, where the entire operation is enclosed in a transaction, the credit is largely due to eliminating double-checking of reads.

Redundant Stores: In the Mindicator and mound, the process of marking and unmarking nodes during an update or DCAS creates unnecessary work. Eliminating this work was the primary driver of improved latency in the mound.

Allocation: The ability to replace copy-on-write in the hash table was single-handedly responsible for more than 2x speedup on the write-dominated workload. This improvement directly followed from the reduced interaction with the allocator. A similar benefit arose in the BST, where we were able to avoid allocating descriptors, but did not affect the mound, where descriptors are reused from one operation to the next.

6.5 Rethinking Concurrent Data Structure Design and Implementation

We highlight two implications of PTO on concurrent data structure design.

Optimization on Strengthened Invariants: We first observe that the use of a hardware transaction can strengthen some of the invariants of the original data structure. The most straightforward example is that within a hardware transaction, the intermediate states of an operation are not visible to other threads. In many nonblocking data structures, operation descriptors are installed by the operation to indicate that a certain objects are involved in an operation, and those descriptors are removed during the clean-up phase of the operation, after its linearization point. In many algorithms, it will be possible to avoid not only the installation and removal of descriptors, but also their allocation and deallocation.

Similarly, some algorithms employ hazard pointers to prevent objects from being made unreachable during critical periods in a method's execution. When the method is executed within a hardware transaction, there is an invariant that memory accessed by the transaction will not change due to external events. Thus it is not possible for an object accessed by a transaction T to become unreachable before T commits. While T must respect the hazard pointers reserved by concurrent (non-transactional) threads, T need not guard locations via hazard pointers during its own operation. In an analogous manner, hardware transactions do not need to update memory management epochs [22, 52]. This latter case clearly cannot be handled by the compiler, since epochs are represented with monotonically increasing counters. For short operations, such as those on hash tables, epoch operations and their corresponding memory fences can be a significant contributor to latency; for read-only operations, epochs can again be a significant cost, due to their introduction of memory fences.

Progress vs. Optimization Trade-off: A more aggressive opportunity lies in weakening the progress guarantees of the original algorithm to increase the opportunity for fast-path optimization. There exist algorithms [29,48] in which read-only lookup operations are wait-free. Reducing the progress of lookups to lock-free can have non-local benefits by increasing the opportunity to optimize the PTO fastpath of inserts and removals.

In the hash table case, we see a PTO insertion or removal can modify the array in-place, as long as it increments the counter within its hardware transaction. Doing so ensures that concurrent lookups will not miss a value concurrently removed and inserted, at the cost of the operation retrying when there is concurrency. If concurrency between modifications and lookups is rare, or if modifications are, themselves, frequent, the optimization may outweigh the added overhead (and reduced progress guarantees) of the modified set. Modifications of this technique can be applied to algorithms that use copy-on-write, marking, descriptors, simulated DCAS, and indirection-based versioning of data.

In summary, we see significant potential to (re)design concurrent data structures to be PTO-friendly. If the prefix succeeds with high probability, then common costs, especially those related to memory management (reference counts, hazard pointers, epochs, indirection), become less significant. A slow-path that bears these costs, coupled with an unencumbered fast-path, may provide a “sweet spot” for algorithm designers. When these techniques cease to be performance bottlenecks, they may be employed to more rapidly develop novel concurrent data structures.

6.6 Summary

We introduced a methodology for accelerating concurrent data structures by using hardware transactional memory. Our technique involves creating a fast-path transaction that succeeds or fails in bounded time, and a set of optimizations that can be applied to that fast-path to eliminate latency. In evaluation on five data structures, we saw performance benefits ranging from 50% to 3x for the hash table and binary search tree. Even when the methodology did not improve performance, we did not observe any significant slowdowns.

Apart from performance, our methodology offers many other benefits: It relies upon, and hence confirms the value of, strongly atomic hardware transactions. It preserves nonblocking progress, despite the absence of progress guarantees for current hardware transactional memory. Our technique is oblivious to the capacity of the underlying HTM. Lastly, it is both local and compositional. This last point is crucial, as it allows data structure designers to use existing mechanisms, such as lock-free DCAS, and then transactionally accelerate them.

Chapter 7

Conclusions

7.1 Summary

In this dissertation, we created nonblocking implementations of unordered linked lists, resizable hash tables, and array-based priority queues.

The lock-free and wait-free unordered linked list algorithms introduced in this dissertation are shown to scale well across a variety of benchmarks, making them suitable for use both as standalone lists, and as the foundation for wait-free stacks and non-resizable hash tables.

The nonblocking hash table algorithms support resizing in both directions: shrinking and growing. The heart of the table, a freezable set abstraction, greatly simplifies the task of moving elements among buckets during a resize. Furthermore, the freezable set abstraction makes possible the use of highly optimized implementations of individual buckets, including implementations in which a single flat array is used for each bucket, which improves cache locality. In performance evaluation, we find that our lock-free implementation is consistently better than the current state-of-the-art split-ordered list, and that performance for the adaptive wait-free algorithm is compelling across microbenchmark configurations.

The mound priority queues combine a number of novel techniques to achieve their performance and progress guarantees. Chief among these are the use of randomization and the employment of a structure based on a tree of sorted lists. Linearizable mounds can be implemented in a highly concurrent manner using DCAS. Their structure also allows several new uses such as the probabilistic ExtractMin and ExtractMany operations. In performance evaluation, we show that the lock-free mound is a practical algorithm despite its reliance on software DCAS.

We also introduced the Prefix Transaction Optimization (PTO) to use HTM to accelerate existing concurrent data structures. The technique provides significant performance boost to a variety of state-of-the-art data structures while preserving their strong progress guarantees.

7.2 Future Research Directions

Systematic approaches to efficient wait-free computation. Herlihy [32] demonstrated the existence of universal constructions for wait-free concurrent objects. It remains an open problem whether all such objects can be made practical, because wait-free data structures implemented from universal constructions [33] tend to incur significant overhead. Although many lock-free data structures have been proposed, practical wait-free implementations are relatively rare. There exist techniques [42] to construct adaptive wait-free implementations whose performance approximates their lock-free versions, however, there are still considerable latency gaps between the best lock-free implementation and the best sequential implementation.

HTM can potentially bridge the aforementioned performance gap, and ultimately, create a systematic approach to construct efficient and scalable wait-free data structures. In many wait-free data structures, operation descriptors are installed to indicate that certain objects are involved in an operation, and those descriptors are removed during the clean-up phase of the operation. Within a hardware transaction, the intermediate states of an operation are not visible to other threads, and thus, in these algorithms, it will be possible to avoid not only the installation and removal of descriptors, but also their allocation and deallocation. As demonstrated in Chapter 6, HTM can also be used to create speculative fast paths that avoid, in the common case, the overhead of the wait-free code paths.

Concurrent implementations of STL containers. The C++ Standard Template Library (STL) provides a set of general-purpose data structures such as list, set, and map objects. These data structures are designed and implemented for sequential use, and require external synchronization (i.e., locks) when concurrent accesses are demanded. Since C++ is widely-used in programming concurrent systems (OS and web servers), it is desirable to provide a standardized, alternative version of STL that is optimized for concurrency, similar to the approach adopted by the Java standard libraries [1].

Despite the potential impact on real-world applications, there are several research challenges in this direction. For example, most data structures proposed in research literature provide relatively narrow interfaces, while the STL tends to include extra functionality, such as iteration and statistics methods. Ensuring linearizability of all operations is challenging, due to performance costs and verification. Hence, it may be attractive to provide relaxed specifications. A possible approach is to promise linearizability only to a limited set of operations, and to provide weaker consistency for the rest of the operations. The techniques needed to compose such hybrid specifications, as well as their impacts on practicality, remain interesting open questions. One can leverage recent progress in concurrent data structure research to facilitate this research, such as memory management, iterators, and HTM-based acceleration.

In-memory databases. In recent years, researchers have made significant progress to improve the performance and capacity of non-volatile memory systems. The availability of low-latency, non-volatile random-access memory has great potential to create disruptions in database management system (DMBS) implementations. In fact, the DBMS industry is currently trying to leverage shared-memory concurrent data structures to build highly efficient and scalable in-memory database solutions [4].

The concurrent data structures presented in this dissertation can be used to improve in-memory DBMS implementations. We see the potential shift to non-volatile memory technology as an opportunity to unify the research of disk-optimized data structures (i.e. optimized for sequential accesses) and shared-memory concurrent data structures. Specifically, we believe the pragmatic importance of using scalable synchronization techniques will increase along with the adoption of non-volatile memory systems: Since the latency gap between accessing persistent storage and accessing memory is shrunk if not eliminated, the DBMS implementation may no longer assume the use of mutual exclusion locks (or "latches") is comparatively cheap, and hence, one must carefully contemplate the trade-off among various synchronization options. Shared-memory concurrent data structures, which provide high scalability and low latency, are a promising response to such technology trends.

7.3 Concluding Remarks

Our work showed that concurrent data structure design remains a fertile research area. With careful investigation of the properties of specific data structures, we were able to gain new insights to further improve their performance, and to strengthen their progress guarantees (from lock-freedom to wait-freedom).

We demonstrated the potential of using HTM to accelerate concurrent data structure implementations. It is clear that HTM has changed our toolboxes as well as our fundamental performance assumptions in concurrent data structure design. A refactoring of existing implementations to exploit the power of HTM is likely to create significant performance benefits. An even more promising research direction is to investigate how much improvement one can obtain by changing the way in which data structures are designed, with the availability of HTM in mind.

Bibliography

[1] Java Concurrency Utilities. http://docs.oracle.com/javase/7/docs/index.html.

[2] Memcached: A Distributed Memory Caching System. http://memcached.org/.

[3] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[4] Adam Prout. The Story Behind MemSQL's Skiplist Indexes, 2014. http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/.

[5] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, 2011.

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

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

[8] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory.

In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, AB, Canada, Aug. 2014.

[9] P. Chuong, F. Ellen, and V. Ramachandran. A Universal Construction for Wait-Free Transaction Friendly Data Structures. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[10] T. David, R. Guerraoui, T. Che, and V. Trigonakis. Designing ASCY-compliant Concurrent Search Data Structures. Technical Report EPFL-REPORT-203822, Ecole Polytechnique Federale de Lausanne, 2014.

[11] D. Dice, T. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of Lazy Subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory*, Paris, France, July 2014.

[12] D. Dice, Y. Lev, V. Marathe, M. Moir, M. Olszewski, and D. Nussbaum. Simplifying Concurrent Algorithms by Exploiting Hardware TM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[13] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):569, 1965.

[14] Doug Lea. Java Concurrent Skip List Map Implementation. http://www.java2s.com/Code/Java/Collections-Data-Structure/ConcurrentSkipListMap.htm.

[15] K. Dragicevic and D. Bauer. Optimization Techniques for Concurrent STM-Based Implementations: A Concurrent Binary Heap as a Case Study. In *Proceedings of the 23rd International Symposium on Parallel and Distributed Processing*, Rome, Italy, May 2009.

[16] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proceedings of the*

30th ACM Symposium on Principles of Distributed Computing, San Jose, CA, June 2011.

[17] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, Zurich, Switzerland, July 2010.

[18] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the Twenty-Sixth ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[19] P. Fatourou and N. D. Kallimanis. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[20] S. Feldman, P. LaBorde, and D. Dechev. Concurrent Multi-Level Arrays: Wait-Free Extensible Hash Maps. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Samos Island, Greece, July 2013.

[21] M. Fomitchev and E. Ruppert. Lock-Free Linked Lists and Skip Lists. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

[22] K. Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, University of Cambridge, Sept. 2003.

[23] H. Gao, J. F. Groote, and W. H. Hesselink. Almost Wait-Free Resizable Hashtable. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, Apr. 2004.

[24] V. Gramoli. More Than You Ever Wanted to Know about Synchronization. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming*, San Francisco, CA, Feb. 2015.

[25] M. Greenwald. Two-Handed Emulation: How to Build Non-Blocking Implementation of Complex Data-Structures using DCAS. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.

[26] T. Harris. A Pragmatic Implementation of Non-Blocking Linked Lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, Oct. 2001.

[27] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[28] T. Harris, K. Fraser, and I. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing*, Toulouse, France, Oct. 2002.

[29] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*, Pisa, Italy, Dec. 2006.

[30] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[31] D. Hendler, N. Shavit, and L. Yerushalmi. A Scalable Lock-free Stack Algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM, 2004.

[32] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[33] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[34] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

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

[36] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[37] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Information Processing Letters*, 60:151–157, Nov. 1996.

[38] Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). Feb. 2012.

[39] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th International Symposium On Microarchitecture*, Vancouver, BC, Canada, Dec. 2012.

[40] P. Jayanti. f-arrays: Implementation and applications. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, Monterey, California, July 2002.

[41] A. Kogan and E. Petrank. Wait-Free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, Feb. 2011.

[42] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-Free Data Structures. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012.

[43] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[44] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, Sept. 1979.

[45] Y. Lev and J.-W. Maessen. Split Hardware Transactions: True Nesting of Transactions Using Best-Effort Hardware Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[46] Y. Liu, V. Luchangco, and M. Spear. Mindicators: A Scalable Approach to Quiescence. In *Proceedings of 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, July 2013.

[47] Y. Liu and M. Spear. Mounds: Array-Based Concurrent Priority Queues. In *Proceedings of the 41st International Conference on Parallel Processing*, Pittsburgh, PA, Sept. 2012.

[48] Y. Liu, K. Zhang, and M. Spear. Dynamic-Sized NonBlocking Hash Tables. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, July 2014.

[49] Y. Liu, T. Zhou, and M. Spear. Transactional Acceleration of Concurrent Data Structures. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, Portland, OR, June 2015.

[50] I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May 2000.

[51] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, CA, June 2003.

[52] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels.* PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[53] J. M. Mellor-Crummey and M. L. Scott. Scalable Reader-writer Synchronization for Shared-memory Multiprocessors. *ACM SIGPLAN Notices*, 26(7), 1991.

[54] M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, Aug. 2002.

[55] M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[56] M. Michael and M. Scott. Correction of a Memory Management Method for Lock-Free Data Structures. Technical report, Department of Computer Science, University of Rochester, Rochester, NY, USA, 1995.

[57] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, May 1996.

[58] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 103–112. ACM, 2013.

[59] S. S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann, 1997.

[60] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[61] A. Prokopec, N. Bronson, P. Bagwell, and M. Odersky. Concurrent Tries with Efficient Non-Blocking Snapshots. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, Feb. 2012.

[62] C. Purcell and T. Harris. Non-blocking Hashtables with Open Addressing. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[63] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*, Austin, TX, Dec. 2001.

[64] N. Shafiei. Non-blocking Patricia Tries with Replace Operations. In *Proceedings of 33rd International Conference on Distributed Computing Systems*, Philadelphia, PA, July 2013.

[65] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, 2006.

[66] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A Lock-free Algorithm for Concurrent Bags. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures, SPAA*, volume 11, pages 335–344, 2011.

[67] H. Sundell and P. Tsigas. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *Journal of Parallel and Distributed Computing*, 65:609–627, May 2005.

[68] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-Free Linked-Lists. In *Proceedings of the 16th International Conference on Principles of Distributed Systems*, Rome, Italy, Dec. 2012.

[69] J. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, Ottowa, Ontario, Canada, Aug. 1995.

[70] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[71] L. Xiang and M. L. Scott. Compiler Aided Manual Speculation for High Performance Concurrent Data Structures. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming*, Shenzhen, China, Feb. 2013.

[72] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High Performance Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, CO, Nov. 2013.

[73] D. Zhang and P.-Å. Larson. LHlf: Lock-Free Linear Hashing (Poster Paper). In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, New Orleans, LA, Feb. 2012.

[74] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear. Practical Non-Blocking Unordered Lists. In *Proceedings of the 27th International Symposium on Distributed Computing*, Jerusalem, Israel, Oct. 2013.

Curriculum Vitae

Name

Yujie Liu

Education

2013, M.Sc. in Computer Science, Lehigh University

2010, B.Eng. in Software Engineering, Tianjin University

Professional Experience

06/2013 $\sim$ 08/2013, Intern, Oracle Labs

06/2012 $\sim$ 08/2012, Intern, Oracle Labs

12/2009 $\sim$ 05/2010, Intern, Alibaba Cloud Computing

Selected Publications

Transactional Acceleration of Concurrent Data Structures. Yujie Liu, Tingzhe Zhou, and Michael Spear. 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). Portland, OR, USA, June 2015.

Dynamic-Sized Nonblocking Hash Tables. Yujie Liu, Kunlong Zhang, and Michael Spear. 33rd ACM Symposium on Principles of Distributed Computing (PODC). Paris, France, July 2014.

Transaction-Friendly Condition Variables. Chao Wang, Yujie Liu, and Michael Spear. 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). Prague, Czech Republic, June 2014.

Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Salt Lake City, UT, USA, March 2014.

Practical Non-blocking Unordered Lists. Kunlong Zhang, Yujiao Zhao, Yajun Yang, Yujie Liu, and Michael Spear. 27th International Symposium on Distributed Computing (DISC). Jerusalem, Israel, October 2013.

Mindicators: A Scalable Approach to Quiescence. Yujie Liu, Victor Luchangco, and Michael Spear. 33rd IEEE International Conference on Distributed Computing Systems (ICDCS). Philadelphia, PA, USA, July 2013.

Using Hardware Transactional Memory to Correct and Simplify a Readers-Writer Lock Algorithm. David Dice, Yossi Lev, Yujie Liu, Victor Luchangco, and Mark Moir. 18th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP). Shenzhen, China, February 2013.

On the Platform Specificity of STM Instrumentation Mechanisms. Wenjia Ruan, Yujie Liu, Chao Wang, and Michael Spear. 11th IEEE/ACM International Conference on Code Generation and Optimization (CGO). Shenzhen, China, February 2013.

Mounds: Array-Based Concurrent Priority Queues. Yujie Liu and Michael Spear. 41st International Conference on Parallel Processing (ICPP). Pittsburgh, PA, USA, September 2012.

Delegation and Nesting in Best-effort Hardware Transactional Memory. Yujie Liu, Stephan Diestelhorst, and Michael Spear. 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). Pittsburgh, PA, USA, June 2012.