2015

# The Design, Implementation, and Refinement of Wait-Free Algorithms and Containers

Steven Feldman
*University of Central Florida*

# THE DESIGN, IMPLEMENTATION, AND REFINEMENT OF WAIT-FREE ALGORITHMS AND CONTAINERS

by

STEVEN DOUGLAS FELDMAN
B.S. University of Central Florida, 2012
M.S. University of Central Florida, 2013

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2015

Major Professor: Damian Dechev

# ABSTRACT

My research has been on the development of concurrent algorithms for shared memory systems that provide guarantees of progress. Research into such algorithms is important to developers implementing applications on mission critical and time sensitive systems. These guarantees of progress provide safety properties and freedom from many hazards, such as dead-lock, live-lock, and thread starvation. In addition to the safety concerns, the fine-grained synchronization used in implementing these algorithms promises to provide scalable performance in massively parallel systems.

My research has resulted in the development of wait-free versions of the stack, hash map, ring buffer, vector, and a multi-word compare-and-swap algorithms. Through this experience, I have learned and developed new techniques and methodologies for implementing non-blocking and wait-free algorithms. I have worked with and refined existing techniques to improve their practicality and applicability. In the creation of the aforementioned algorithms, I have developed an association model for use with descriptor-based operations. This model, originally developed for the multi-word compare-and-swap algorithm, has been applied to the design of the vector and ring buffer algorithms.

To unify these algorithms and techniques, I have released *Tervel*, a wait-free library of common algorithms and containers. This library includes a framework that simplifies and improves the design of non-blocking algorithms. I have reimplemented several algorithms using this framework and the resulting implementation exhibits less code duplication and fewer perceivable states. When reimplementing algorithms, I have adapted their *Application Programming Interface* (API) specification to remove ambiguity and non-deterministic behavior found when using a sequential API in a concurrent environment.

To improve the performance of my algorithm implementations, I extended OVIS's Lightweight Distributed Metric Service (LDMS)'s data collection and transport system to support performance monitoring using perf_event and PAPI libraries. These libraries have provided me with deeper insights into the behavior of my algorithms, and I was able to use these insights to improve the design and performance of my algorithms.

To my wife, family, and friends, none of whom are likely to read this.

# ACKNOWLEDGMENTS

I would like to begin by thanking all of my friends and family, without whom I would not have been able to make it this far. I would also like to thank the University of Central Florida for seven and a half years of education. As an undergraduate, I became involved in undergraduate research sponsored by the EXCEL and YES programs. My time in these programs provided the foundation upon which my graduate career was built.

To my advisor Damian, you inspired me to get involved in research as an undergrad. As a result of that, I have gained a considerable amount of experience and knowledge, I have contributed to the scientific community, I have worked at the National labs, and I have earned a Ph.D. For all that and more, I am very grateful.

To my friends and colleagues who have spent time proof reading many of my publications, including this very dissertation, your service is now over. I would like to specifically recognize, in alphabetical order: Matti Perez-Cubas, Damian Dechev, Joshua Katz, Pierre LaBorde, Carlos Valera-Leon, and Marianne Naniong. They have each spent numerous hours helping me with my grammar issues, improving my sentence structure, and overall being great friends. To each of you, you have my eternal gratitude and appreciation.

To my mother who has always been there for me when I needed her, who has always provided me with encouragement and support when I needed it, I love you and I could not have ask for a better mother.

Finally to my very supportive wife, whose patience knows no bounds, we are finally able to start the next big adventure of our lives.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1: INTRODUCTION

Over the past decade, the number of systems featuring multi-core processors have exploded. Almost every new laptop, desktop, and even cellphone features a multi-core processor. Unfortunately, more often than not, the software and applications used on these devices are not designed to take advantage of this processor advancement. For decades, the majority of software architects and application developers have developed software that is executed sequentially on a single core processor. In this sequential programming paradigm, instructions are executed one after another. Certain instructions, such as loading from a disk drive, take significantly longer to execute than others. These instructions often become a performance bottleneck.

Modern processor designs incorporate features that can overcome the bottleneck caused by these long running instructions. Since 1985, processors have used pipelining to overlap the execution of instructions and improve performance [Her08]. Various techniques, such as dynamic scheduling, branch prediction, re-order buffers, and more, have been used to further reduce the effect on performance that an expensive operation may have. Multi-core technology promises to further increase the performance by enabling multiple processes to execute their instructions concurrently. Unlike previous hardware advancements, which provide performance improvement without requiring changes to application design, multi-core technology requires an application to express how it should be parallelized. In addition to the normal design considerations of sequential applications, developers of concurrent applications must also consider how to express concurrency. For example, which data is shared between units of execution (threads), how this data is synchronized between threads, and how to handle interdependencies between threads.

## 1.1 Common Parallelization Methodologies

Common ways to parallelize an application include data parallelization, task parallelization, and pipe-line processing.

Data parallelization consists of dividing up the input data between the threads and having each thread process its subset of the data independently of the other threads. This design requires the threads to synchronize and/or combine their results once they have finished. Depending on the nature of the data, some threads may take significantly longer to complete than other threads. An example of this can be seen in a naive implementation of an algorithm that identifies all prime numbers in a range. If this range is divided into subranges and each subrange is assigned to a thread, then the threads with smaller integers in their ranges will complete much faster than threads with larger integers. This is because the time taken to determine the primality of an integer increases proportionally with its value. Depending on the application and properties of the data, heuristics could be used to reduce the differences in execution time between threads. For example, instead of assigning subranges to each thread, the designer could use an offset value to divide the work amongst the threads.

Task parallelization is an alternative to data parallelization. In an ideal implementation of the scheme, an application is divided into many independent tasks, and each task is ran in isolation from one another. Most applications are unable to achieve this level of task independence and must employ synchronization methods to ensure that threads with dependencies between them behave correctly. An example of task parallelization can be found in cloud host servers, which facilitate connections from many users to clients' application. A user connected to one application can be conceptualized as a task running on a cloud host server. This task would run independently from users connected to other applications. However, users connected to the same application may have dependencies between them and require synchronization. An example of this is shared document

editing, in which users editing the same document would have dependencies and users editing different documents would not.

Pipe-line processing is a scheme inspired by the hardware pipeline model. In this scheme, data is passed from one thread to another. Each thread receives the data, performs an operation, and then passes the data onto the next thread. Unlike the hardware model, which is regulated by clock cycles, it is possible for a thread to receive a new task before completing its current task. Buffers and/or additional threads can be added to address this problem and prevent bottlenecks. This scheme is generally used in the processing of stream data, which may arrive at irregular rates.

## 1.2 Correctness and Progress

Depending on how an application is parallelized, different synchronization methods are used to enable threads to communicate between each other. How these synchronization methods are implemented has a significant effect on the safety properties.

The safety properties of concurrent algorithms deal with both correctness and progress conditions. Correctness ensures that an algorithm behaves as described by its *Application Programming Interface* (API) in all scenarios of execution. APIs designed for sequential applications are often ambiguous in a concurrent environment, and they must be modified to remove these ambiguities. For example, sequential queues exhibit undefined behavior if a thread attempts to remove an element from an empty queue. To prevent a thread from doing so, the thread often checks the state of the queue before attempting an operation. This does not work in a concurrent environment because the state of the queue can change in between the checking of the state and the operation on the queue.

Behavior of an algorithm is also affected by the consistency model used to describe how operations

are ordered in respect to one another. For example, consider a first-in-first-out queue using a strict consistency model. If a thread pushes the value $A$ then the value $B$, it should not be possible for a thread to pop $B$ then pop $A$. This would break the first-in-first-out property of the queue. The consistency model may be relaxed to allow values to be reordered by a certain amount [AKY10]

In addition to correctness, meeting specified guarantees of progress is another important design constraint. The progress guarantee describes how an algorithm behaves when threads compete for resources. Concurrent algorithms can be classified by level of progress they guarantee. Some applications provide no guarantee of progress, while others guarantee that at least one thread executing in an application is always making progress. The most strict and strongest guarantee is that all threads in an application are always making progress (wait-freedom).

A thread's progress can be affected in a number of ways, and the two most common are mutually exclusive locks and loop structures. A lock may force a thread to wait an indeterminable amount of time before being able to acquire that lock. If multiple locks are used, a dead-lock scenario could occur, in which two or more threads wait on one another to release a lock. Loop structures can be used to enable a thread to re-attempt an operation in the event the operation fails. When multiple threads are attempting to modify shared resources, the actions of one thread may cause a thread's operation to fail. In the event of failure, the thread will often reattempt its operation until it is successful. It can be very challenging, and sometimes impossible, to find an upper limit on the number of times a thread can be forced to retry its operation.

## 1.3 The Problem

Application developers designing concurrent applications depend on the availability of practical and efficient synchronization methodologies that meet their functional requirements. My research

focuses on the design, implementation, practicality, correctness, and tuning of synchronization methods that provide high guarantees of progress. Through my research, I have developed many concurrent algorithms and containers that provide wait-free progress, which is the strong guarantees of progress. The design and implementation of these algorithms have led to new methodologies and techniques [FLD11a, FLD15a, FLD13b, FBL13, FLD11b, FKD12, CCF13]. These methodologies and techniques have been used in the development of other and more complex algorithms [BFD15, FVD15].

The difficulties of implementing wait-free algorithms stem from the requirement that each executing thread must always be making meaningful progress towards completing its own operation. This must hold true even in the event of a thread being continuously unscheduled at the most inopportune times. Consider the simple function `increment` shown in Figure 1.1. This function loads the value of an atomic counter and then attempts to replace that value with one value higher.

```
1: x = counter.load();
2: while True do;
3:     if counter.cas(x, x+1) then
4:         break;
5: return x
```

Figure 1.1: Increment Function

The is replacement is done using an *atomic compare-and-swap* or `cas` operation. This operation replaces the current value of a variable with a new value only if it matches an expected value. If the value was replaced, the operation returns TRUE; otherwise, the value of the expected value variable is set equal to the current value and the operation returns FALSE. The hardware guarantees that the value of variable does not change in between the loading of its current value and the storing of a

new value.

If other threads are incrementing the value with such frequency that the `cas` at line 2 always returns FALSE, then the thread will never make meaningful progress. In some use cases, heuristics can be applied to allow the function to return in the event of failure, but for cases in which they can not be applied, inter-thread helping techniques must developed. These techniques enable threads to help other threads complete their operations. The challenge in designing these techniques is to ensure that the effects of the operation occur exactly once.

There have been numerous papers that present non-blocking algorithms and containers [TBK12, MS96, Mic03, ST08, HSY10, SS03, Cli, Mic02, FH07, DPS06]. These non-blocking algorithms provide concurrent implementations of sequential algorithms with varying guarantees of progress. The design of these structures is motivated by demand of real time and embedded systems that require fault tolerant applications. Additionally, the number of cores in the processing nodes of high performance computing systems are increasing. This has led to a demand for non-blocking algorithms that exploit fine grained synchronization methodologies to achieve strong scaling properties.

Strong scaling [SDM11] is the scenario in which the total problem size stays fixed while the number of processing elements is increased. The challenge is how to synchronize the work of the processing elements in a correct and efficient manner without "wasting" too many cycles on parallelism overhead. In weak scaling, the problem size assigned to each processing element remains constant while the total problem size may increase. In this case, the main challenge is how to add new processing elements to the existing system.

Wait-Free algorithms are an important type of non-blocking algorithms that provide the strongest guarantee of progress. They provide the guarantee of system wide progress, ensuring that each process or thread is making progress in its own operation. A wait-free algorithm is immune to

dead-lock, live-lock, thread starvation, and priority inversion. These properties are very important in many applications, especially in embedded systems.

An example of this importance occurred during July of 1997 when the Mars Pathfinder mission experienced a series of anomalous system resets that resulted in loss of scientific data [Low02]. It was determined that these resets were caused by a flaw in the implementation that enabled a low-priority process to block a high-priority process (priority inversion). It was also discovered that the black box testing used at the time *would not* have been able to detect this issue. This is because the flaw only reveals itself when certain events occur in a specific frequency and it was not possible to test all interleaving of events.

# CHAPTER 2: ORGANIZATION OF DISSERTATION

This chapter provides an overview of the organization of this dissertation.

Chapter 3 presents relevant background information on the fundamental concepts and models used in concurrent programming.

Chapter 5 presents Tervel, which is the cumulation of my research work. Tervel is both a framework for implementing wait-free algorithms and a collection of algorithms implemented using this framework. This chapter discusses the benefits, features, and philosophy of the Tervel framework, and how it has affected the design of the algorithms that have been re-implemented in this framework.

Before developing Tervel, I led the development of several wait-free algorithms and containers. I present these algorithms in Chapter 4, before Tervel, to showcase the techniques and methodologies that were created or explored in the original design of these algorithms. I also discuss how these algorithms have influenced and affected the design of Tervel.

Chapter 6 discusses how I applied and extended performance monitoring tools to identify and resolve synchronization methodologies, implementation error, and other design flaws that negatively impact the performance and/or behavior of the non-blocking algorithms implemented within Tervel.

I conclude this dissertation in Chapter 7 by summarizing the work that I have completed and presenting my plans for the future.

# CHAPTER 3: BACKGROUND

This chapter introduces the fundamental concepts and models used in concurrent programming. It begins by presenting a concise overview of the various concepts that are presented and defined in this section. Then the following sections expand on these overviews, providing more technical details.

## 3.1 Overview

There are several known programming models used to implement concurrent applications. An application is said to be concurrent if two or more processes work together to perform a task. These processes can be executing on different machines, processors, cores, or something else entirely. The type of concurrency a program exploits is based on the available hardware. This work focuses on programming models designed for *shared-memory multi-processor/multi-core* (*SMM*) architectures. Section 3.2 provides an overview of notable hardware architectures and how they differ from shared SMM architectures.

In shared-memory multi-procssor(*SMM*) architectures, the processors and memory models are connected by an interconnection network. This allows processes executing on different processors to read and write to the same memory. Each processor typically maintains a local memory cache that must synchronize with the shared memory cache. The *memory consistency* problem is determining when to perform the synchronization and how to reconcile concurrent reads and write. Section 3.3 provides a formal definition of this problem and describes and compares popular consistency models.

In addition to the hardware synchronization challenges, there are also software synchronization

challenges. To achieve performance improvements and better hardware utilization, developers may use threads to parallelize part or all of an application. A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler. A developer must ensure that parallelized version of the application maintains the requirements of the sequential application. Depending on how threads access and modify shared data, this task could be far from trivial. Section 3.6 provides a detailed overview of established thread-level synchronization techniques and a comparison between them.

Using these thread-level synchronization techniques, it is possible to develop a wide variety of algorithms and containers. Depending on how these techniques are used, these implementations may be susceptible to a variety of concurrency dangers. These dangers include race conditions, deadlock, live-lock, priority version, and thread starvation. Section 3.5 provides the formal definitions of these dangers and examples of how they can occur.

Non-Blocking algorithms are a class of algorithms designed without mutual exclusive critical sections and instead use atomic hardware primitives. There has been extensive research into the design of non-blocking algorithms. These algorithms are classified by the guarantee of progress that they provide. Section 3.4 provides a formal description of these progress guarantees and the common techniques used to implement them.

## 3.2 Other Concurrent Architectures

Another method by which parallelism can be achieved is by distributing the problem across many machines. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers.

In contrast to shared memory systems, which allows processes to communicate through shared

memory, machines in a distributed system must communicate over a network. How this communication is performed often has a high impact on performance. Depending on the relative location of one machine to another, the time it takes for them to communicate varies. Machines co-located on the same network bus will have lower message latencies than those located on different network buses. Additionally, the amount of communication passing through a bus and the number of different paths through the network affect the congestion and performance of the network.

The key difference between designing distributed algorithms and shared memory algorithms is that in distributed algorithms communication is minimized and optimized, while in shared memory algorithms time spent in critical sections is minimized and access is safeguarded.

## 3.3 Correctness Conditions

Reasoning about the correctness of algorithms is a challenging task. It is often done by finding a way to equate the behavior in a concurrent environment to its behavior in a sequential environment. I used the following correctness properties when proving or reasoning about correct behavior of the algorithms that I have developed. These are formally defined and discussed in [Her08].

- Quiescent consistency: an algorithm is quiescently consistent if the effects of method calls appear to happen in a one-at-a-time sequential order and method calls separated by a period of inactivity appear to take effect in their real-time order.
- Sequential consistency: an algorithm is sequentially consistent if the effects of method calls appear to happen in a one-at-a-time sequential order and method calls appear to take effect in program order.
- Linearizability: an algorithm is linearizable if each method call appears to take effect instantaneousness some point between its invocation and response.

It is important to note that sequentially consistent algorithms are not compositional with one another, while quiescently consistent and linearizable algorithms are. Linearizability is the strongest of these three properties, and as a result it was predominately used to show the correctness of the algorithms I implemented. Showing an algorithm to be linearizable requires identifying linearization points. The linearization point is typically a single step in which the effects of a method call become visible to other threads.

Linearizability is a powerful property that allows a series of concurrent operations to be ordered by their linearization points. Given an initial state, a final state, and a set of concurrent operations, a valid sequential history can be constructed using linearization points.

## 3.4 Progress Conditions

Progress conditions are another important property of concurrent algorithms and containers. They are used to describe how concurrent method calls may affect the ability of a method call to complete. Methods can be classified either as blocking or non-blocking.

Blocking algorithms use mutual exclusion to safe guard access to critical sections. This is typically accomplished by using a lock or semaphore. The use of mutual exclusion is very common as it is easy to relate the concurrent behavior to the sequential behavior. Unfortunately, these designs have inherent dangers that are often hard to detect. Section 3.5 discusses how using mutual exclusion can introduce hazards such as dead-lock, live-lock, priority inversion, and thread starvation.

Non-blocking algorithm methods avoid the use of mutual exclusion and instead leverage hardware synchronization primitives to exploit fine grained synchronization. These primitives often operate on a single register at a time and handle coherency between the executing processes.

The most common synchronization are primitives:

- compare-and-swap: or `cas` is an operation that atomically compares the value at an address with the address's current value and replaces the current value with a new value if the current value matches an expected value.
- fetch-and-add: or `faa` is an operation that atomically increments the value at an address by a specified amount. It returns the pre-incremented value.
- store: atomically writes a value to the specified address, ensuring concurrent stores do not result in a value that is a mixture of multiple writes.
- load: atomically read a value to the specified address, ensuring that partially written values are not read.

Non-blocking algorithm methods are classified by the level of progress they guarantee. These classifications are as follows:

- Obstruction-Free: A method is obstruction-free if at any point in its execution, it executes in isolation from other methods, then it is able to finish its execution in a finite number of steps.
- Lock-Free: A method is lock-free if it guarantees that infinitely often some method call finishes its execution in a finite number of steps.
- Wait-Free: A method is wait-free if it guarantees that every method call finishes its execution in a finite number of steps.

Each classification is a superset of the previous, wait-free being the strongest and most desirable property.

## 3.5 Concurrent Hazards

In addition to logic and implementation hazards known to sequential programmers, concurrent programmers face even more hazards. Depending on how synchronization is implemented, algorithms may be prone to or contain hard to detect scenarios in which undefined behaviors may occur. These dangers are often undetectable by conventional testing methodology, as they only occur under very specific conditions.

In addition to correctness, there are other hazards that may affect the liveness of algorithms. For example, dead-lock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. Similar to a deadlock, live-lock is a situation in which the states of the processes involved constantly change with regard to one another, and thus none progresses. And finally, thread starvation is when a thread is perpetually denied a resource, preventing it from making progress in its own operation.

When designing algorithms for mission critical and embedded systems, it is important to ensure that the algorithms are free from these dangers. Otherwise, significant harm and/or damage may occur as a result of the system or part of the system becoming unresponsive. Wait-Free algorithms are free from all three of the aforementioned dangers, and as such are highly desirable for such systems.

Another danger is the ABA problem, which can be caused by the value of an address changing to a value previously held at that address. This could allow a thread to incorrectly update the value of that address. For example, let a reference to an object, $A$, be stored at an address and let some thread attempt to replace $A$ with a reference to $Z$ using a compare-and-swap operation (`cas`). However, before executing the `cas` operation, some other thread replaces $A$ with a reference to some object $B$ and returns $A$ to the allocator. If $A$ is a reallocated and a reference to $A$ replaces $B$,

the first thread will replace the reference to $A$ without knowing that its state has changed.

Hazard pointer [Mic04] and reference counting [DMM01] are two common memory management techniques used to address this danger.

## 3.6 Non-Blocking Synchronization Techniques

There are many techniques used in the development of non-blocking algorithms. This section discusses two common techniques and the motivation behind them. More advanced techniques and those that I have developed are discussed in later sections.

One common technique is to use a compare-and-swap(`cas`) operation to conditionally change the value at an address. By examining the returned results, developers can determine if another thread modified the address. If the returned value matches the expected value, then the value was changed by the invoking thread. Otherwise, the value was changed by another thread. By including logic to handle both cases, developers have constructed complex algorithms.

If atomic stores were used instead, a developer would not be able to know what value was overwritten. For example, if incrementing a counter, a thread reads the value $0$, but before the thread stores $1$, another thread changes the value to $1$. The first thread, not knowing the value to have already been incremented, will overwrite the value.

Another technique is the use of descriptor objects [Bar93]. These objects encapsulate logic concerning a pending operation and enable inter-thread helping. In general, a thread will attempt to place a reference to a descriptor object into a shared variable and then call the descriptor object's helping routine. If a reference to a descriptor object already exists in that variable, the thread will call that object's helping routine and try again.

The helping routine is important to ensure progress of algorithms, as without it threads must wait until the object has been removed. Since it is unknown when the object will be removed, it breaks any progress guarantees. Another challenge faced, which I discuss in Section 5.3.3, is ensuring a thread will be able to place a descriptor object. Other threads may place references with such frequency that one or more other threads starve.

# CHAPTER 4: RESEARCH DESIGN AND METHODOLGY

My research has resulted in the development of several wait-free algorithms and containers. In the design and implementation of these algorithms, I have discovered and generalized several techniques and methodologies for implementing concurrent algorithms. The remainder of this section presents the implementation, derived techniques, and evaluation of the wait-free algorithms developed throughout my research.

## 4.1 Design: Wait-Free Hash Map

A hash map is a container that facilitates the storage, retrieval, and updating of key-value pairs. These operations are usually performed in $\mathcal{O}(1)$ complexity. Hash maps are used in a wide variety of applications ranging from databases, image processing, web services, and more.

The hash map that I led the development of was the first to provide a wait-free progress guarantee for all operations on it. Other implementations are prone to live-lock and/or thread starvation in the following scenarios:

- When the capacity of the container has been reached and a resize is triggered.
- When concurrent insertion of key-value pairs may cause a thread to be continually delayed in its own operation.
- When concurrent insertion of key-value pairs may prevent a thread from ascertaining if a key is present in the container.

The main goal of the design was to provide both *safety* and *high performance* for multi-processor

applications. In experimental evaluations, this hash map design performs, on average, 7 times faster than a traditional blocking design. Additionally, it outperforms the best available alternative non-blocking designs in a large majority of cases, typically by a factor of 15 or higher.

### *4.1.1 Related Work*

At the time of development, there were no existing wait-free hash maps in the literature. As a result, the design was compared to the lock-free implementations presented in [Mic02], [GGH04], and [SS03].

In [Mic02], the authors present a lock-free hash map which uses linked-lists to resolve collisions. It differs from the one I designed in that it does not guarantee constant-time for operations after a resize is performed [SS03] [Mic02].

In [GGH04], Gao et al. present an openly-addressed hash map that is *almost* wait-free; however, it degrades in performance to lock-free during a resize.

In [SS03], Shalev and Shavit present a linked-list structure that uses pointers as shortcuts to logical buckets that allow the structure to function as a hash table. In contrast to my wait-free hash map design, the work by Shalev and Shavit does not present a hash map and it is lock-free.

There was a single claim of a wait-free hash map that appeared as a presentation by Cliff Click [Cli]; the author now claims lock-freedom. Moreover, the work by Click was not published.

A popular concurrent hash map that is part of Intel's Threading Building Blocks (TBB) [Int] library is claimed to be lock-free, but is also unpublished.

### *4.1.2 Implementation Overview*

The key challenge that was faced while developing a wait-free hash map is how to increase the capacity of the hash map while facilitating concurrent operations. If a blocking synchronizations method were used, all threads would be forced to wait until the thread performing the resize has copied the key-value pairs from the old hash map to a larger hash map.

Because of this, the design of this hash map stores key-value pairs across multiple arrays, thereby removing the need for a lengthy copy-over. Each position on an array may hold a reference either to another array or to a key-value pair. The implementation uses a primary array length and secondary array length, in which the primary array length is often much larger than the secondary. Using this structure, key-value pairs are placed based on the binary representation of their keys.

Each key is conceptually divided into subsets, in which the decimal value of each subset indicates the position on an array to place the key-value pair. The number of bits in the first subset corresponds is $log_2(\text{primary array length})$ and the number of bits in the following subsets is $log_2(\text{secondary array length})$.

In the following sections, I present a high level description of each hash map operation.

### *4.1.3 Traversal*

Algorithm 4.1 describes how the traversal of the hash map is performed. The decimal value of the first subset is used to determine the position on the primary array to examine. If the position holds a reference to an array, the next subset is used to determine the position on that array to examine. Once a non-array value is found, the traversal is complete.

```
1: procedure TRAVERSE(node, array, pos, subsets, depth)
2:     node = array[pos].load();
3:     while IsArray(node) do
4:         array = array[pos];
5:         pos = subsets[depth++];
6:         node = array[pos].load();
7:         if isMarked(node) then
8:             node = expand(node, array, pos, depth);
```

Figure 4.1: Hash map traversal procedure

```
1: function FIND(key)
2:     array = primaryArray;
3:     subsets = getSubsets(key);
4:     depth = 0;
5:     pos = subsets[depth];
6:     fcount = 0;
7:     while True do
8:         traverse(node, array, pos, subsets;
9:         if node == null OR node.key != key then
10:            return null;
11:        else
12:            return node.value;
```

Figure 4.2: Hash map find operation

### *4.1.4 Find*

Algorithm 4.2 describes how the `find` operation is performed. Once the traversal has identified a position holding a non-array value, it acts based on that value.

- If the position holds a reference to a key-value pair whose key matches the passed key, the value member is returned.

- Otherwise null is returned.

### *4.1.5 Insertion*

Algorithm 4.3 describes how the `insert` operation is performed. Once the traversal has identified a position holding a non-array value, it acts based on that value.

- If the position is empty, a `cas` operation is used to attempt to replace it with a reference to the key-value pair.
- If the position holds a reference to a key-value pair whose key matches the key being inserted, false is returned indicating that the key already exists in the hash map.
- Otherwise, the position holds a reference to a key-value pair whose key does not match the key being inserted. In this case, an expansion (Algorithm 4.4) is performed to resolve the hash collision.

If an expansion is performed or a `cas` operation fails, the position is re-examined.

### *4.1.6 Removal*

Algorithm 4.5 describes how the `remove` operation is performed. Once the traversal has identified a position holding a non-array value, it acts based on that value.

- If the position holds a reference to a key-value pair whose key matches the passed key, a `cas` operation is used to attempt to replace it with a null reference. If successful, true is returned, otherwise the position is re-examined.
- Otherwise, false is returned, indicating the key is not present.

```
1:  function INSERT(key, value)
2:      array = primaryArray;
3:      subsets = getSubsets(key);
4:      depth = 0;
5:      pos = subsets[depth];
6:      fcount = 0;
7:      while True do
8:          traverse(node, array, pos, subsets);
9:          if node == null then
10:             if array[pos].cas(node, new pair(key,value)) then
11:                 return true;
12:         else if node.key == key then
13:             return false;
14:         else
15:             expand(node, array, pos, depth);
16:         if fcount++ == MAX_FAIL then
17:             array[pos].atomicOR(0x1);
18:             expand(node, array, pos, depth);
19:             fcount = 0;
```

Figure 4.3: Hash map insertion operation

```
1:  function EXPAND(node, array, pos, depth)
2:      newArray = new SecondaryArray;
3:      subsets = getSubsets(node.key);
4:      newArray[subsets[depth]] = unMark(node);
5:      array[pos].cas(node, newArray);
6:      return array[pos].load();
```

Figure 4.4: Hash map expand function

```
1:  function REMOVE(key)
2:      array = primaryArray;
3:      subsets = getSubsets(key);
4:      depth = 0;
5:      pos = subsets[depth];
6:      fcount = 0;
7:      while True do
8:          traverse(node, array, pos, subsets;
9:          if node == null OR node.key != key then
10:             return false;
11:         else
12:             if array[pos].cas(node, null) then
13:                 return true;
14:         if fcount++ == MAX_FAIL then
15:             array[pos].atomicOR(0x1);
16:             expand(node, array, pos, depth);
17:             fcount = 0;
```

Figure 4.5: Hash map remove operation

### *4.1.7 Wait-Freedom*

As described, thread-starvation could occur in the event of keys being repeatedly inserted and removed from the same position. This use case could cause a thread's `cas` operation to continually fail. To prevent this, an atomic bit marking technique is used to force an expansion to occur at the contended position.

When a fail counter reaches a user defined threshold, the thread uses an atomic `bitwise OR` operation to place a mark on the least significant bit of the value at the position. Threads that see this bit mark must replace the reference with an array containing an unmarked version of that reference. This is shown on Line 17 Algorithm 4.3 and Line 7 Algorithm 4.1.

Given enough hash collisions and failures, the hash map will expand to a depth equal to the number

of subsets. A property of the structure of the hash map is that no hash collisions can occur at this depth. This is because the entire key has been used in the determining of this position. Because of this property, the hash map's operations use specialized logic when operating at this depth.

If an `insert` operation's `cas` failed, it implies another thread inserted that key and that it should return, indicating that the key already exists in the hash map. Likewise, if a `remove` operation's `cas` failed, it implies another thread removed that key and that it should return, indicating that the key does not exist in the hash map.

To show wait-freedom, first examine the two looping structures:

- The for loop is bounded by the number of subsets in the key, which is based on the finite size of the key data type.
- The while loop is bounded by the fail count. When it is reached, bit marking ensures that the next iteration loads a reference to an array.

Since the functions contain calls to only wait-free functions and contain only bounded loops, they are also wait-free.

### *4.1.8 Performance Evaluation*

To test the performance of this design, a test was constructed to determine the time it took to execute one million operations. The distribution of operations and the number of threads executing these operations were varied. This variation showed how the hash map's performance varies across use cases.

The performance of this hash map and the following hash maps were compared:

- C++11 standard template library hash map protected by an optimized global lock (Lock-STL) [ISO11]
- Split-Ordered Lists (Split-Ordered) [SS03]
- Michael's lock-free hash map (Michael) [Mic02]
- Click's hash map [Cli]
- Intel TBB's implementation (TBB) [Int]

The operation distributions are based on the following use cases:

- The first distribution is based on a reported typical operation mix for hash maps [SS03](88% `get`, 10% `insert`, 2% `remove`).
- The second distribution is an inversion of the first distribution (10% `get`, 88% `insert`, 0% `update` 2% `remove`).
- The third distribution consists of an even mix of operations (`update`: 25% `get`, 25% `insert`, 25% `update` 25% `remove`).

Based on the Figures 4.6, 4.7, and 4.8 on average, the wait-free algorithm outperforms the traditional blocking design by a factor of 7 or more. It performs faster than the lock-free algorithms typically by a factor of 15. The lack of scalability of the blocking solution is a result of the fact that the lock is applied to all operations, not only those that conflict. Both lock-free solutions scale well; however, they perform worse when more `insert` operations are performed, because the `insert` operations trigger more global resizes. Due to the incremental approach in resizing the hash map, there is performance improvements over the other designs in the tested scenarios except for TBB. The other lock-free designs show an average of a 17.5 times performance decrease when

compared to Intel's TBB implementation. In contrast, this approach is competitive with only a 14% loss in performance to provide the stronger progress guarantee of wait-freedom.

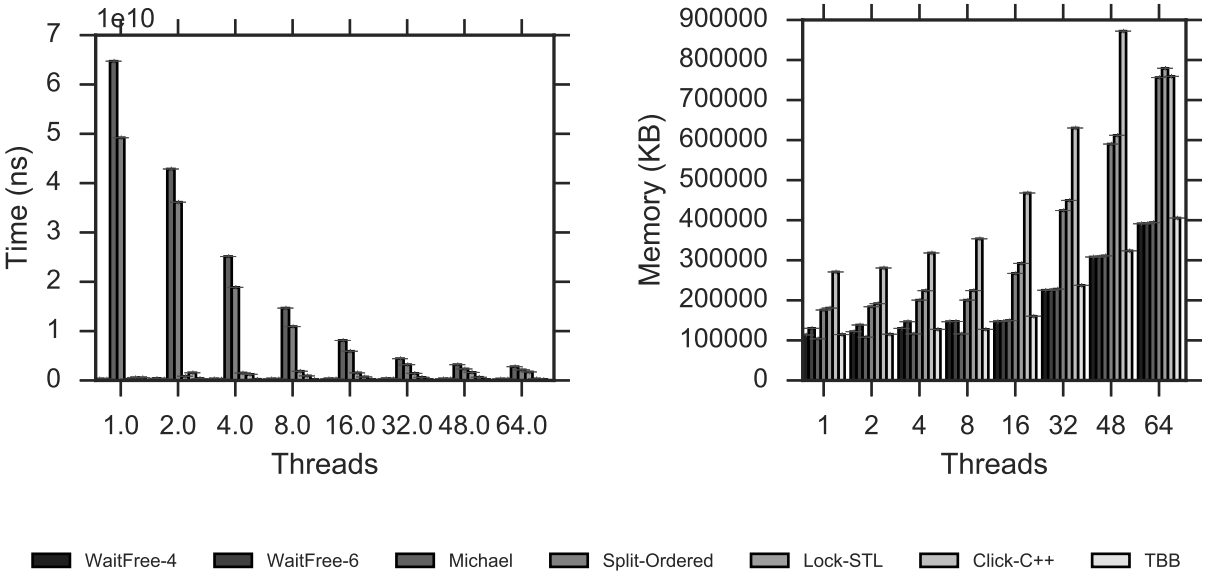



Figure 4.6: 10% Get, 88% Insert, 0% Update, 2% Remove

### *4.1.9 Tervel Implementation*

This design was re-implemented in the Tervel framework. This reimplementation incorporates a new API and the application of memory reclamation. Section 5.4.3 describes these changes and the motivations behind them.

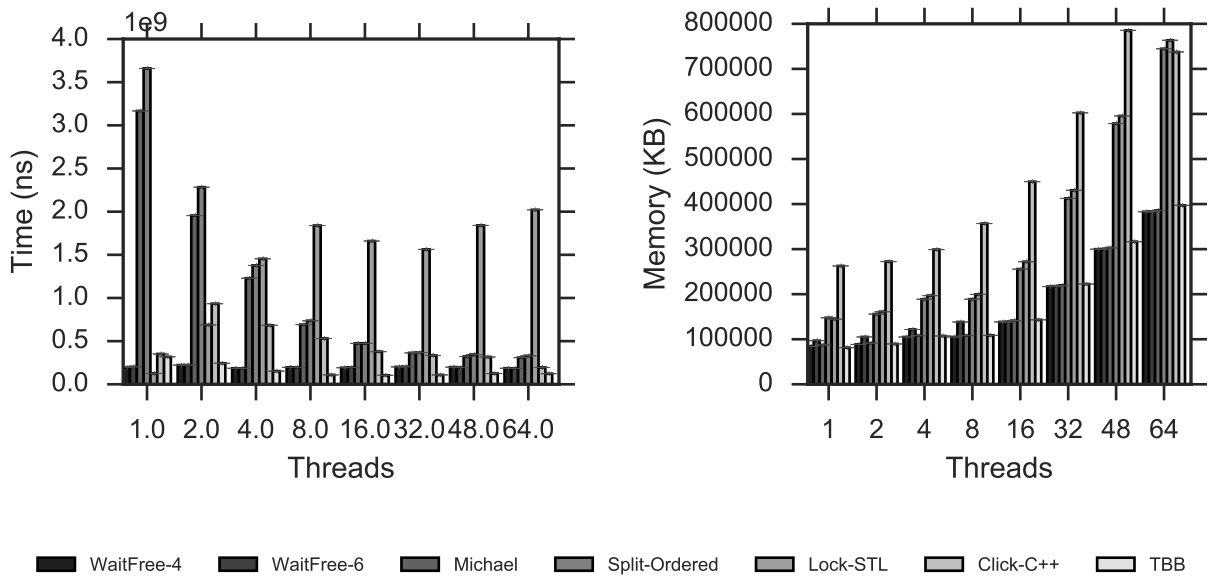



Figure 4.7: 34% Get, 33% Insert, 0% Update, 33% Remove

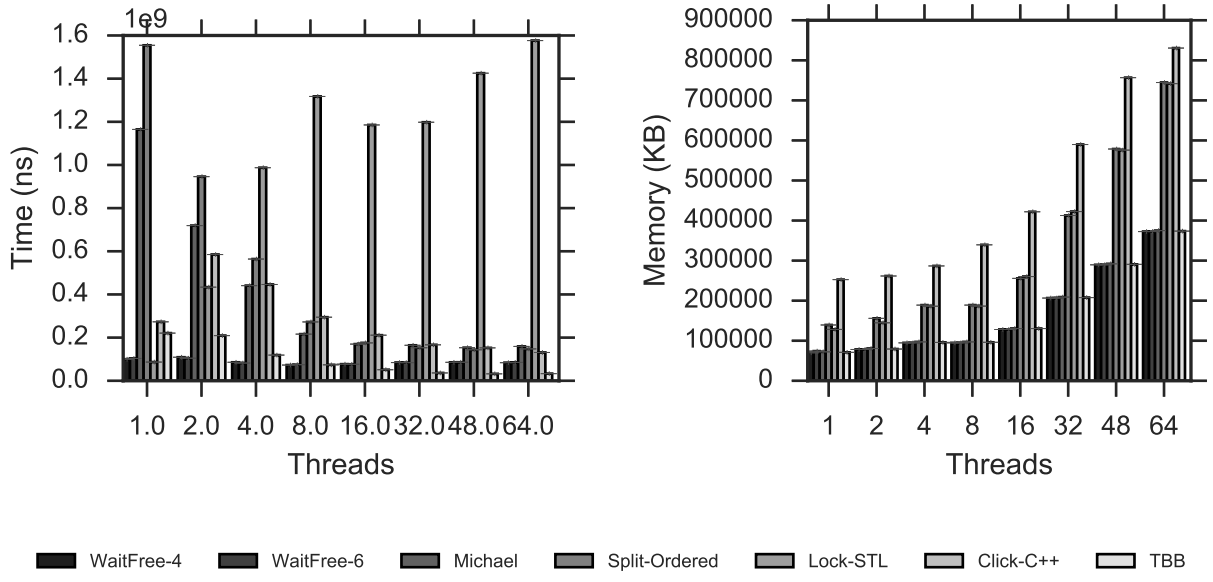



Figure 4.8: 88% Get, 10% Insert, 0% Update, 2% Remove

## 4.2 Design: Wait-Free Multi-Word Compare-and-Swap

A multi-word compare-and-swap (MCAS) is an algorithm used to conditionally update one or more memory addresses [HFP02]. This update occurs only if the value at each address matches an expected value. For correctness, this update must provide the appearance of atomicity, such that a thread would be unable to read a newer value and then read an older value. There are several concurrent algorithms and containers that depend on the ability to perform an MCAS operation. How these algorithms use MCAS varies. Common use cases include:

- Updating objects that are larger than a machine word. For example in [PH05], an MCAS is used to allow a non-blocking hash table to support multi-word length key and value data types. Unlike the wait-free hash map [FLD13b], this design focuses on high data locality and lower memory utilization.

- Use in an array based lock-free priority queue [LS12] to percolate higher priority elements to the front of the queue.

- A concurrent binary search tree [FH07] that uses MCAS to maintain the tree's balance. The properties of the MCAS algorithm ensure that concurrent modifications do not introduce incorrect behavior.

- A fall back path for systems that use hardware transaction memory. When operations exceed the supported size of the hardware transaction memory, MCAS can be used instead. This is because MCAS does not have a limit on the number of locations, while most HTM proposals limit the number of locations [SAH06].

### *4.2.1 Related Work*

At the time of its development, I was aware of the following other MCAS algorithms that provide similar functionality.

Israeli et al. presents a lock-free and disjoint-access parallel MCAS algorithm [IR94]. This algorithm requires a thread identifier to be stored alongside the value of a memory address, limiting the number of bits available to the value. This thread identifier is used to access a set of global variables that contain information about the operations that are currently executing in the system. Israeli et al.'s design does not support the ability to perform a read through to retrieve the current value for an address. Rather they require the thread to help complete the pending operation before proceeding with its own operation. This algorithm is dependent on the LL/VL/SC primitive[1], which is not provided by any contemporary system.

Anderson et al. demonstrates a wait-free MCAS algorithm that is disjoint-access parallel and supports read through parallelism [ARJ97]. In contrast to [IR94], their design requires that each memory word that contains a value to be followed by an additional memory word containing auxiliary information. This information may include the identification of a thread performing an operation at the address and the information needed to help complete the operation. Using a non-redundant helping scheme, this design chooses not to perform recursion to help complete a conflicting operation. Instead it causes the conflicting operation to be restarted. Like [IR94], this design requires the availability of the LL/VL/SC primitive. A simplified lock-free version of this algorithm was presented by Moir [Moi97]. Attiya et al. [AH11] have also presented improvements upon this design.

Harris et al. in [FH07] propose a lock-free MCAS algorithm that is disjoint-access parallel, sup-

[1]Load-link, Validate, Store Conditional; used to ensure the value at an address has not been unknowingly modified.

ports read through parallelism, and does not depend on LL/VL/SC. Rather this design uses a CAS operation to replace the expected value at an address with a reference to a descriptor object. This design reserves the two lowermost bits of each address to distinguish between values and descriptor objects. To ensure correct behavior of the MCAS algorithm and to prevent the ABA problem, Harris et al. designed a "double compare single swap" algorithm. Compared with [IR94] and [ARJ97], their design shows a significant increase in performance and portability.

Sundell presents a wait-free MCAS algorithm based on a greedy helping scheme [Sun11]. Like Harris et al.'s design, his design is disjoint-access parallel, supports read through parallelism, and does not depend on LL/VL/SC. In the first phase of the greedy helping scheme, a thread attempts to place a reference to its MCAS operation at as many addresses in its operation as it can. In the next phase, if another MCAS operation holds some of those addresses, then one of the operations will steal addresses from the other. Unlike [FH07], Sundell makes no claim that his algorithm is ABA-free, and when examined, his algorithm can exhibit undefined behaviors in certain cases caused by the ABA-problem[2]. In Sundell's algorithm, thread starvation can occur in the case in which the result of a CAS operation consistently causes a thread to reattempt that CAS operation. Because of this, the algorithm is lock-free and not wait-free.

### *4.2.2 Implementation Overview*

The MCAS algorithm that I led the design of was the first design to provide wait-free progress guarantees and the ABA-free safety property [FLD15a]. In synthetic tests performed with 64 threads on a 64 core workstation, the design completes on average 67.8% more MCAS operations than other comparable designs. On average, it improves performance by 8.6% over all tested scenarios.

[2]See Sec. 3.5 for more details.

The implementation of the MCAS algorithm is based on using helper descriptor objects, MCasHelper, to hold the logical value of an address constant until the MCAS operation has completed. The MCasHelper object is composed solely of a reference to an MCAS operation record.

An MCAS operation record (MCasDescriptor) contains the following for each address:

- An expected value.
- A new value to replace the expected value.
- An atomic reference to a descriptor object that is initially null.

### *4.2.3 Association Model*

The atomic reference is part of an association model that was developed to overcome correctness issues related to concurrent helping and the ABA problem. This association model enables a thread to distinguish between a descriptor object that was placed during the operation from one that was placed after the operation was completed. The latter could occur in the event that a thread is suspended from execution just before placing a descriptor. If it resumes after the operation has been completed and the value at the address matches the expected value, the thread will incorrectly place the descriptor. If not handled properly, this could allow the value of an address to be changed twice by a single operation.

### *4.2.4 Performing an MCAS*

To perform an MCAS, a thread first constructs an MCasDescriptor operation record and then calls its execute function. This function iterates through each address, and if the value currently held

at the address matches the expected value, an attempt is made to replace the value with a reference to an MCasHelper descriptor. If the value currently held at the address does not match the expected value, the MCasDescriptor's state is set to a constant indicating failure, and the operation is complete.

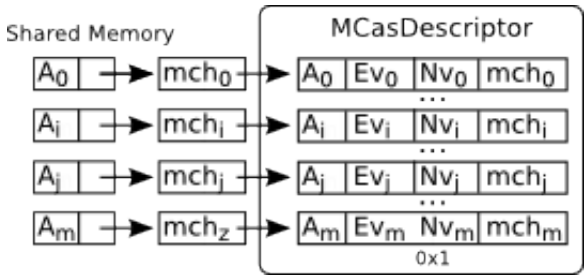



Figure 4.9: Visualization of the MCAS operation

If it was unsuccessful with placing an MCasHelper due to concurrent modification, the current value is re-read. If the MCasHelper was successfully placed, the next step would be to attempt to associate the MCasHelper with the MCasDescriptor.

If this association failed, it implies that some other MCasHelper was used to complete the operation. It also implies that the operation has been completed. If the association was successful, the function will move onto the next address.

After replacing each expected value with a MCasHelper object, a second iteration is performed to replace each MCasHelper with its logical value.

Figure 4.9 presents a visual representation on the dependencies between the two types of objects. The left side is shared memory and the right side is the MCasDescriptor. In this figure, the value of each address has already been replaced by an MCasHelper. The association between the objects is represented by the fact that each row's atomic MCasHelper reference refers to a corresponding MCasHelper.

### 4.2.5 *Performance Evaluation*

To test the performance of this design, a test was constructed to determine the number of operations that were executed over a period of time. The distribution of operations and the number of threads executing these operations were varied. This variation showed how MCAS's performance varies across use cases.

The performance of this MCAS was compared to the lock-free MCAS (LFMCAS) presented by Harris et al [FH07]. Unfortunately, when tested, Sundell's MCAS [Sun11] exhibited behavior that produced inconsistencies in the testing methodology, which invalidated the test results. All implementations were provided by their respective authors.

In a multi-word object benchmark, each thread repeatedly tries to increment the value of each word in the object by 16[3].

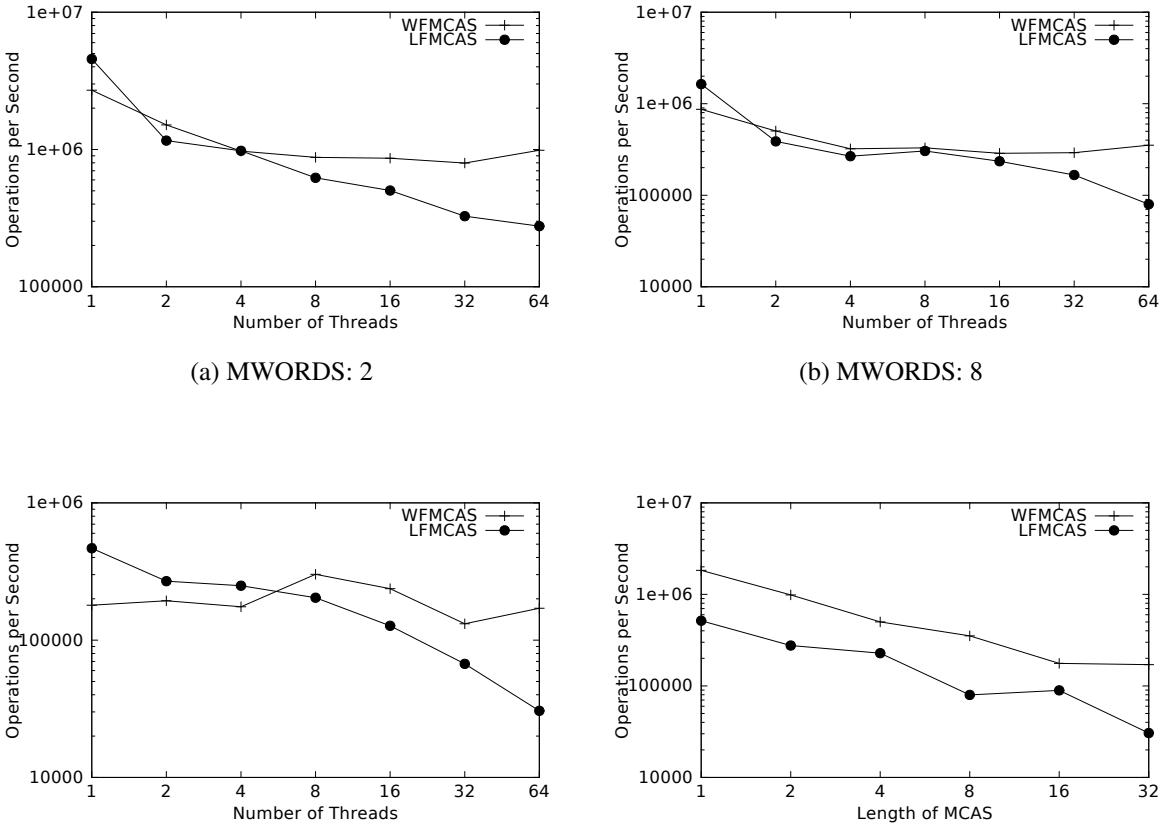
Figure 4.10 presents a set of representative graphs based on this benchmark. Graphs 4.10a, 4.10b, and 4.10c depict the effects of increasing the number of threads updating a shared multi-word object. The WFMCAS performs on average 10% more operations per second when compared to the LFMCAS in this benchmark. When the number of threads are 16, 32 and 64, the WFMCAS performs on average 35.4%, 50.3%, and 77.1%, respectively, more operations per second.

In Graph 4.10d, the number of threads is held constant at 64, and the size of the updated object is varied. In this test, the WFMCAS performs on average 67.8% more operations than the LFMCAS.

Figure 4.11 presents the performance of each MCAS algorithm when used in the construction of a *sorted doubly-linked list* container. Threads executing this test perform a higher amount of work outside of the MCAS algorithm, thereby exhibiting less thread contention. Threads performing

[3]Incrementing by 16 ensures that the two least significant bits are always 0.

this benchmark will repeatedly try to insert and delete elements from the list. The ratio of these operations to one another were varied.




(a) MWORDS: 2

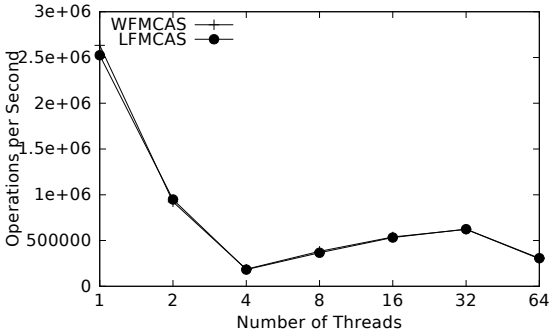
(b) MWORDS: 8

(c) MWORDS: 32

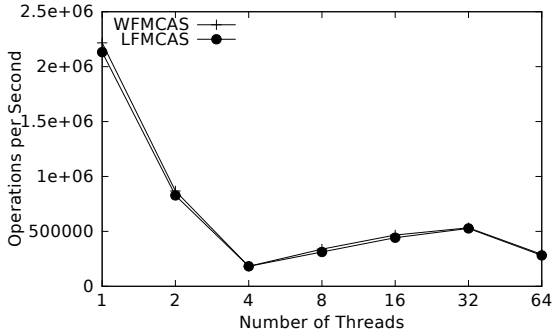
(d) Threads: 64

Figure 4.10: Multi-word Test Results (log scale)

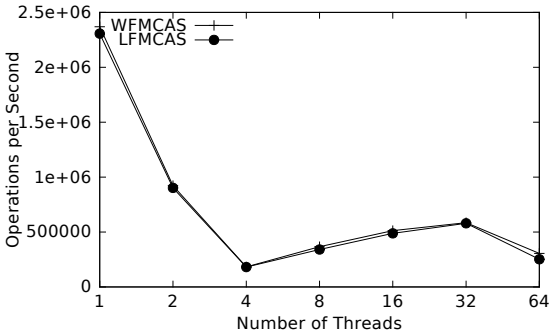
In this benchmark, both implementations scaled equally well. Over all tests, the WFMCAS performs on average 2% more operations per second than the LFMCAS. This insignificant difference in performance can be attributed to the necessity of searching the list for the location to perform an operation. This search was found to consume 84% of the execution time. These benchmarks revealed that when implemented in a practical algorithm, this approach achieves wait-freedom without sacrificing performance. A more thorough examination of this performance comparison is presented in [FLD15a]



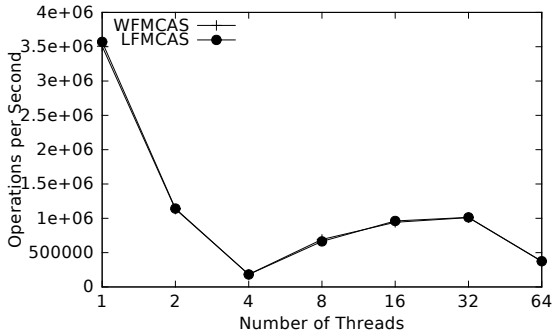

(a) 25% Insertion, 75% Deletion




(b) 50% Insertion, 50% Deletion




(c) 75% Insertion, 25% Deletion




(d) 100% Insertion, 0% Deletion

Figure 4.11: Sorted Doubly-Linked List

### *4.2.6 Tervel Implementation*

In [FLD15b], a new implementation of the MCAS algorithm is described that includes memory management, a re-organization of how internal objects are represented in memory, and how the helping routines are defined. This new implementation takes advantage of the latest features of C++11 and a more refined methodology for implementing helping routines. Section 5.4.2 describes these changes and the motivations behind them.

## 4.3 Design: Wait-Free Ring Buffer

The Ring Buffer is a staple data structure in computer science. It is excellent for high demand applications such as multimedia, network routing, and trading systems. In [BFD15], I oversaw the development of a new wait-free ring buffer design. To improve performance and scalability, this design diffuses thread contention using sequence counters. On average, the design completes 15% more than TBB's concurrent bounded queue [Int], 10% more than the lock-free approach presented by Krizhanovsky [Kri13], and 140% more than the cycle queue by Tsigas [TZ01].

### *4.3.1 Related Work*

At the time of the development, I was aware of the following other ring buffers that provide similar functionality:

A lock-free ring buffer by Tsigas et al. [TZ01] that uses a `cas` operation to apply operations. An enqueue is performed by determining the tail of the buffer and then using a `cas` operation to enqueue an element. A dequeue is performed by determining the head of the buffer and then using a `cas` operation to dequeue an element. In the event a thread is unable to successfully apply its operation, starvation could occur. Another issue with this design is that it does not diffuse thread contention. The competitive nature leads to congestion and poor scalability.

Krizhanovsky [Kri13] presents a lock-free and high performance ring buffer. This implementation relies on the atomic fetch-and-add operation to increment head and tail counters. The returned value of this operation determines the index to perform an operation. This diffuses contention and increases scalability, but also increases the complexity of the design. This increase in complexity comes form the additional logic needed to prevent incorrect behavior from occurring when positions are reused. These designs require each thread to maintain a separate tail and head value of the

last completed enqueue or dequeue. The smallest of all threads' *local* head and tail values is used to determine the head and tail value at which all threads have completed their operations. These values prevent an attempt to enqueue or dequeue at a location where a previously invoked thread has yet to complete its operation. A consequence of this design choice is that one thread's inaction could prevent all other threads from making progress. As a result, this design is not thread-death safe.

Intel Thread Building Blocks (TBB) [Int] provides a concurrent bounded queue which utilizes a fine-grained locking scheme. The algorithm uses an array of micro queues to alleviate contention on individual indexes. Upon starting an operation, threads are assigned a ticket value which is used to determine the sequence of operations in each micro queue. Threads will wait until their ticket is valid, which may take a while, in the event other threads are delayed.

In addition to the above, I also implemented a naive implementation of a ring buffer using the wait-free multi-word compare-and-swap described in Section 4.2. In this design, head and tail markers are moved along the buffer. The act of moving a marker requires the completion of six hardware compare-and-swap operations, which results in poor performance.

### *4.3.2 Implementation Overview*

The design presented in [BFD15] uses sequence counters to diffuse contention and reduce forced dependencies. Before performing an operation, a thread atomically increments the relevant sequence counter and uses the result (*seqid*) to complete its operation. The monotonic nature of a counter ensures each thread has a unique value.

In general, it is unlikely that the actions of a thread may interfere with another thread. However, certain use cases may significantly increase the probability of interference. Using the sequence

numbers, I helped design logic to handle the case in which threads interfere with one another. Scenarios that could cause this include:

- An enqueue thread operating on an empty position.
- A dequeue thread operating on a non-empty position.
- A position holding a sequence ID that is less or greater than the expected sequence ID.
- A `cas` operation continually returns failure.

These scenarios are often the result of inopportune context switches and/or thread delay.

### *4.3.3 Implementation of Enqueue*

The following describes the procedure a thread uses to perform an enqueue operation. Pseudo code from Figure 4.12 is referenced to clarify the explanation.

The enqueue function returns after a element has been enqueued (Line 24) or if it has been determined that the buffer is full (Line 28).

If the buffer is not full, the tail sequence counter is incremented and the returned result is used to determine the position at which to enqueue (Lines 5, 6). Each position on the buffer holds a *node* type that internally has a sequence ID and is either an *EmptyNode* or *DataNode*.

If the position holds a node type whose sequence number is less than expected, a back-off routine is used to allow delayed threads to complete their operation.

```
1:  procedure ENQUEUE(val)
2:      tryHelpAnother()
3:      fails = 0
4:      while isNotFull() do
5:          seqid = nextTailSeq()
6:          pos = getPosition(seqid)
7:          newNode = ElemNode(seqid, val)
8:          while true do
9:              if fails++ == MAXFAILS then
10:                 op = EnqueueOp(val)
11:                 makeAnnouncement(op)
12:                 return op.result()
13:             node = buffer[pos].load()
14:             if node.op then
15:                 node.op.associate(node, &(buffer[pos]))
16:                 continue
17:             else if isSkipped(node) then
18:                 break
19:             else if node.seqid < seqid then
20:                 backoff()
21:                 if node != buffer[pos].load() then
22:                     continue
23:             if node.seqid <= seqid and isEmptyNode(node) then
24:                 if buffer[pos].cas(node, newNode) then
25:                     return true
26:             else if node.seqid > seqid or isElemNode(node) then
27:                 break
28:     return false
```

Figure 4.12: Ring buffer enqueue procedure

If the position holds an *EmptyNode* whose sequence number is less than or equal to the expected sequence number, the *EmptyNode* is replaced with a new node. If the replacement is successful, the function returns. Otherwise, the value at the position is re-examined.

If the position holds a node whose sequence number is greater than expected or is a *DataNode* type,

then a new sequence number is needed. The dequeue function contains logic to handle skipped sequence numbers. The ability to skip sequence numbers is important to prevent one thread from blocking others.

In the event a thread is unable to complete its operation, an association model and progress assurance scheme [FLD15b] are used to allow any thread to complete another thread's operation. The associate function at Line 15 prevents the effects of the operation from occurring multiple times. Internally, this function removes an incorrectly placed object. Section 5.3.3 describes the association model and progress guarantee in detail.

### *4.3.4 Implementation of Dequeue*

The following describes the procedure a thread uses to perform an enqueue operation. Pseudo code from Figure 4.13 is referenced to clarify the explanation.

If the buffer is not empty, the head sequence counter is incremented and the returned result is used to determine the position at which to dequeue (Lines 5, 6).

Like the enqueue operation, if the position holds a node type whose sequence number is less than expected, a back-off routine is used to allow delayed threads to complete their operation.

If the position holds a *DataNode* whose sequence number is equal to the expected value, that node is replaced with an *EmptyNode* containing the next sequence number for that address.

Unlike an enqueue, a dequeue can not remove a *DataNode* whose sequence number is less than expected. If this were allowed, it would break the FIFO property of the ring buffer. To maintain FIFO property and allow a dequeue to skip a sequence ID, I employ a bit marking scheme.

```
1:  procedure DEQUEUE(&result)
2:      tryHelpAnother()
3:      fails = 0
4:      while isNotEmpty() do
5:          seqid = nextHeadSeq()
6:          pos = getPosition(seqid)
7:          newNode = EmptyNode(seqid + getCapacity())
8:          while true do
9:              node = buffer[pos].load()
10:             if fails++ == MAXFAILS then
11:                 op = DequeueOp()
12:                 makeAnnouncement(op)
13:                 return op.result(result)
14:             else if node.op then
15:                 node.op.associate(node, &(buffer[pos]))
16:             else if isSkipped(node) and isEmptyNode(node) then
17:                 if buffer[pos].cas(node, newNode) then
18:                     break
19:             else if seqid > node.seqid then
20:                 backoff()
21:                 if node == buffer[pos].load() then
22:                     if isEmptyNode(node) and buffer[pos].cas(node, newNode) then
23:                         break
24:                     else
25:                         setSkipped(&buffer[pos])
26:             else if seqid < node.seqid then
27:                 break
28:             else
29:                 if isElemNode(node) then
30:                     if isSkipped(node) then
31:                         newNode = setSkipped(newNode)
32:                     if buffer[pos].cas(node,newNode) then
33:                         *result = node.value
34:                         return true
35:                 else
36:                     backoff()
37:                     if node == buffer[pos].load() and buffer[pos].cas(node, newNode) then
38:                         break
39:     return false
```

Figure 4.13: Ring buffer dequeue procedure

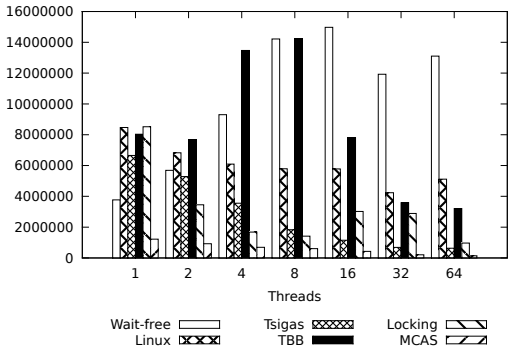
Before requesting a new sequence number, a dequeue operation will ensure that the current value at the position is marked to be skipped (Line 25). After marking, the position is rechecked to ensure that it has not changed in between the last checking and the marking.

If the position is a marked *DataNode* and its sequence number matches the expected value, it is replaced by a marked *EmptyNode*. This marked node will be replaced by an unmarked *EmptyNode* containing a higher sequence number (Line 16). In [BFD15], the correctness and nuances of this approach are explained in detail.

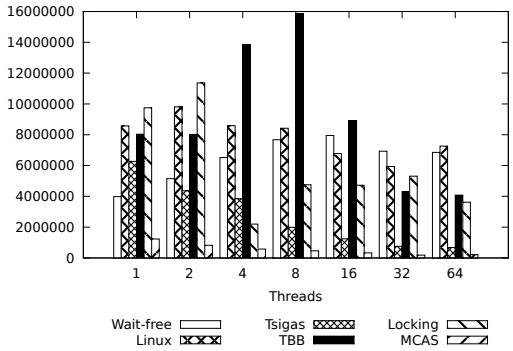
### *4.3.5 Performance Evaluation*

In evaluating the performance of this ring buffer design, I compared its throughput to the throughput of the aforementioned related designs. This comparison was performed by having a main thread initialize the buffer and then spawn a set of worker threads. These threads, executing for a predefined amount of time, perform enqueue and dequeue operations based on the use case being tested. The following graphs depict the throughput of each buffer under different use cases. The y-axis represents the number of operations completed during the testing period.

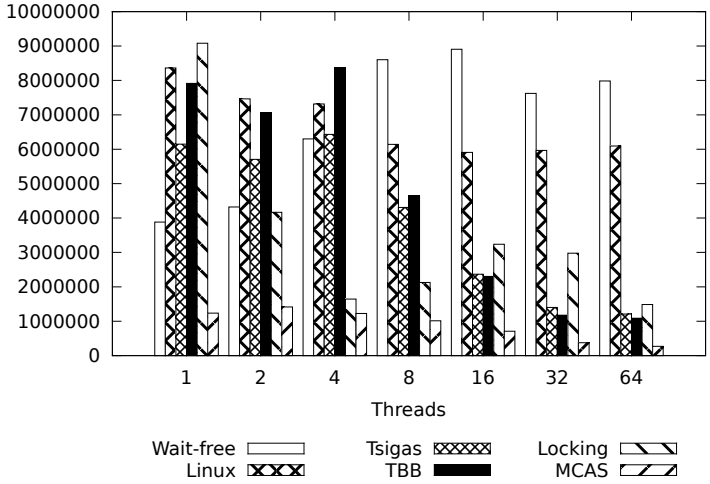
Figure 4.14a explores the performance of the buffers when only enqueues occur on an empty buffer. Figure 4.14b is similar in that only dequeues are being performed on a full buffer. The wait-free buffer performs at least 25% more enqueue operations than other locking and non-blocking approaches, but 8% fewer dequeue operations than TBB and 18% fewer than the Linux buffer. However, at 64 threads, the wait-free buffer performs 68% more dequeue and 150% more enqueue operations than TBB.



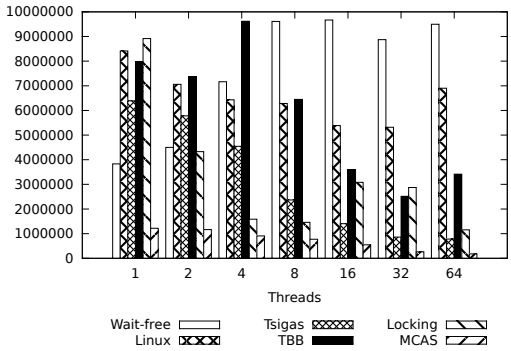

(a) 100% Enqueue



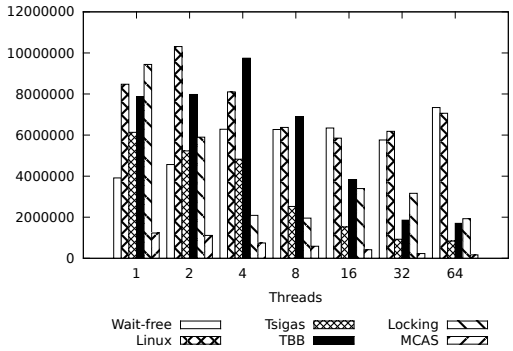

(b) 100% Dequeue




(c) 50% Enqueue, 50% Dequeue




(d) 80% Enqueue, 20% Dequeue




(e) 20% Enqueue, 80% Dequeue

Figure 4.14: Performance comparison

Figure 4.14c shows the throughput of the buffers when there is an even ratio of enqueue to dequeue operations. In this use case, the wait-free buffer performs, on average, 46% more operations than other buffers except for the Linux Buffer, in which there is only a 1% improvement.

Figures 4.14d and 4.14e present the performance in use cases in which there is an imbalance in the enqueue and dequeue operations. When there are more enqueues than dequeues, the wait-free buffer performs, on average, 16% more operations than other approaches. This performance difference increases as thread count increases, highlighting the design's scalability. When there are more dequeues, the wait-free buffer performs, on average, 45% more operations than Tsigas, Locking, and the MCAS approaches. When compared to TBB, there was an average of 2% improvement, and when compared to the Linux buffer, it performs, on average, 18% fewer operations. When comparing the performance in which there are 64 executing threads, the wait-free buffer averages a 280% performance improvement over all algorithms and a 4% improvement over the Linux Buffer.

### *4.3.6 Tervel Implementation*

This design was originally implemented using the Tervel framework. Its final implementation is available in the current release of the Tervel library.

## 4.4 Design: Wait-Free Vector

The vector is a fundamental data structure, which provides constant-time access to a dynamically-resizable range of elements. In general, vectors provide the following operations:

- pushBack: appending an element.

- popBack: removing the last element.
- random access: reading and modifying elements based on their index.
- insertAt: inserting an element at a specified position.
- erastAt: removing an element from a specified position.

The vector that I led the development of was the first to provide wait-free progress guarantee for all operations [FVD15] and the first non-blocking design to support contiguous elements.

Compared to the previously discussed algorithms, vectors support a wider range of operations and exhibit stricter memory representation requirements. It was discovered that by supporting the full vector API, performance was significantly reduced. To overcome this loss in performance, I explored the idea of using alternative implementations. I focused on the tail operations, `pushBack` and `popBack`, as a significant portion of the code is related to handling concurrent tail operations. In [FVD15], I presented two alternative implementations that improve performance significantly, at the cost of reduce functionality. For example, one `pushBack` implementation is safe to execute alongside other `pushBack` and `random access` operations, but not `popBack` operations. These alternatives were designed to execute when specific other operations were not executing, enabling a user to balance functionality and performance. Section 4.4.4 describes the restrictions, use case, and design of these models in detail.

The following were main goals of the wait-free vector project:

- To develop a wait-free vector that provides more of the standard API than the existing non-blocking approach.
- To explore the practicality and versatility of the association model.

- To identify ways to reduce the bottlenecks related to synchronization.
- To implement the design using the Tervel framework, extending and modifying the framework as necessary.

The implementation of the vector had a substantial influence on the Tervel framework. Its use of descriptors and association model shaped the provided abstract descriptor classes. Its memory management requirements led to a call back centric design pattern of Tervel's memory reclamation scheme. Using this design pattern removed the need to include code to handle a significant number of edge cases. I was able to encapsulate this code into the abstract classes. Section 5 describes the Tervel framework, abstract classes, and other features it provides.

### *4.4.1 Related Work*

There was one other non-blocking concurrent vector in literature [DPS06]. It provides a lock-free progress guarantee and supports concurrent `read`, `write`, `pushBack`, and `popBack` operations. Its design is based on using a single shared object to serialize `popBack` and `pushBack` operations. To complete either operation, a thread must first acquire the shared object. If this object is already owned, the thread must execute a helping procedure.

In contrast to the presented approach, this design can not guarantee a tail operation will complete if new operations are continually added to the system. Random access `read` and `write` operations are able to execute concurrently without acquiring this shared object; however, there is no mechanism to prevent a thread from accessing a position that is not in bounds. This can lead to the case where a thread reads or writes to a position that is out of bounds resulting in undefined behavior. This lock-free approach puts the burden of bounds checking on the user, and it is unclear how a user can safely perform bounds checking. For example, if a thread were to check the size

of the vector, another thread can immediately pop one or more elements. The first thread would be unaware of this and could access an out of bounds position. Further, this design is prone to the ABA problem [DPS06, Section 3.5], which may lead to elements not being stored contiguously.

A fine-grain locking approach is presented in Intel's Thread Building Blocks (TBB) library [Int]. This vector supports semi-bound checked random access operations and the `pushBack` operation, but does not support the `popBack` operation. `PushBack` is performed by fetching and adding one to the size variable and writing the value at the fetched position. In contrast to the presented approach, their methodology does not provide a mechanism to distinguish between a position thats holds a value and a position that has been assigned but not written to, allowing for the case where a thread may operate on obsolete data or a partially written value.

Both of these designs use an array segment model to increase the capacity of the vector. This model avoids the copy-over problem by placing references to arrays into a statically sized global array. This causes the elements to be distributed across several arrays. In [FVD15], the underlying memory model is abstracted, and either the contiguous or segmented model maybe used.

### *4.4.2 Implementation Overview*

The following sections present a design overview of the operations supported by the vector. For simplicity, certain details that relate to edge cases and progress conditions have been omitted, and a full description of these operations is presented in [FVD15].

### *4.4.3 Tail Operations*

The vector's tail operations, `pushBack` and `popBack`, are used to add and remove elements from the tail of the vector. In a sequential vector, the index of the vector's tail is equal to the size of the

vector. In this vector, the tail of the vector is the position following the last element of the vector. The vector's contiguous element property guarantees that there is only one position that has this property. To avoid data races between concurrent operations, this implementation modifies both the tail of the vector and the last element.

To perform a `popBack`, a thread places a helper object at the tail of the vector and then replaces the preceding value with another helper object. Next it creates an association between the two objects, ensuring correct behavior, and replaces both objects with their logical values.

The `pushBack` operation is performed by placing a helper object at the tail of the vector and then reading the value of the preceding position. Based on the read value, the state of the helper is set either passed or failed, and the helper object is replaced by its logical value.

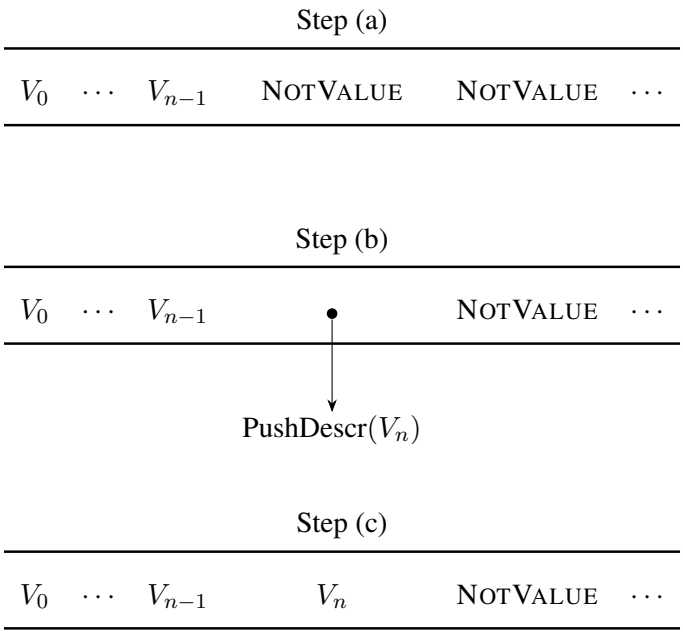



Figure 4.15: `pushBack`($V_n$)

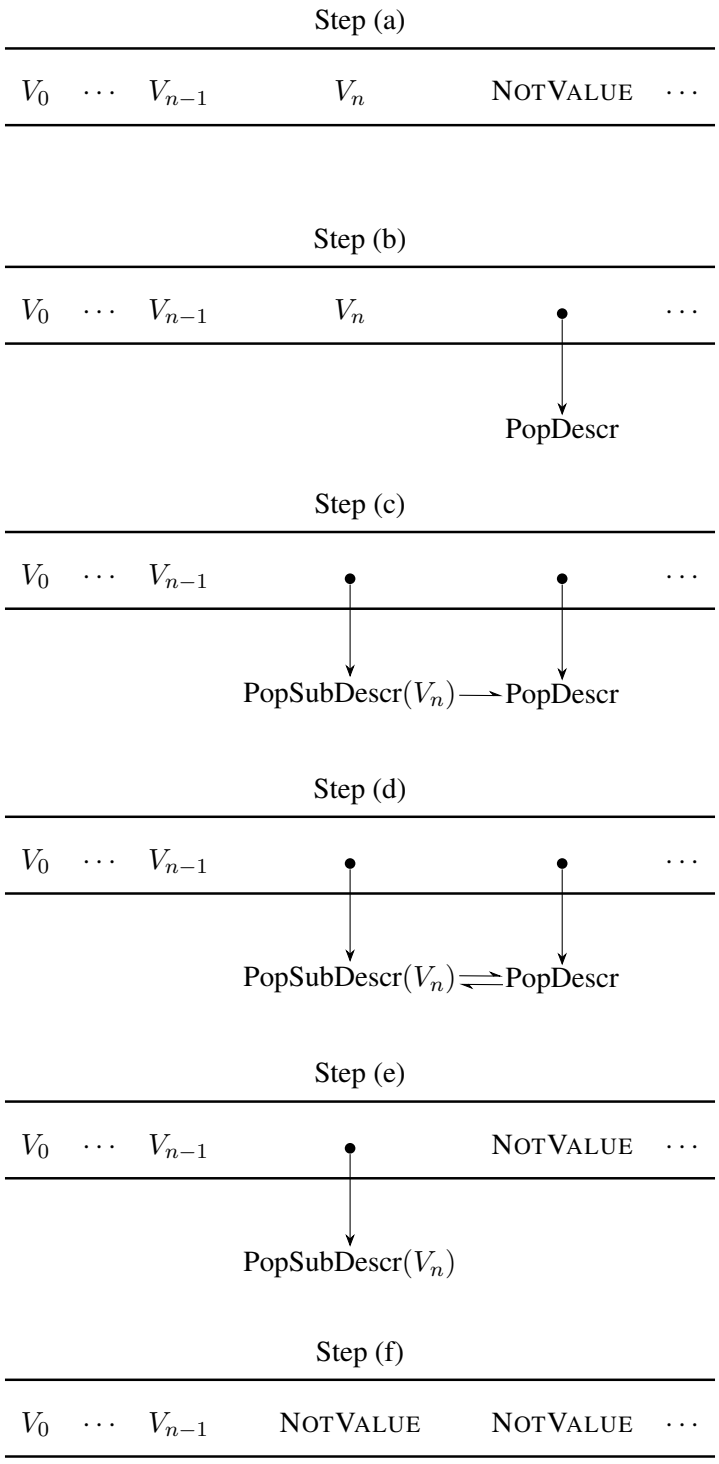



Figure 4.16: `popBack`

A helper's logical value is based on whether or not the operation was successful. If the operation failed, the logical value is the value which the helper objects replaced; otherwise, it is the result of the operation. A visualization of these steps is shown in Figures 4.16 and 4.15.

### *4.4.4 Optimized Tail Operations*

The design of the above tail operations includes logic to handle concurrent operations; however, by restricting the types of concurrent operations, two alternative implementations were constructed. In [FVD15], two alternative models are presented that achieve higher performance by assuming either `pushBack` or `popBack` are used. Users of the vector can freely switch between the models as long as there is not overlapping executions of conflicting operations.

The first alternative uses the `fetch-and-add` operation to assign a position to a thread to perform its operation. When this version is being used, either `pushBack` or `popBack` may be executed on the vector, and no other vector operations are allowed. Threads performing a fetch-and-add based `pushBack` will perform a `fetch-and-add` on the size member, incrementing it by one, and then store the value they are appending at the returned position. Threads performing a fetch-and-add based `popBack` will perform a `fetch-and-add` on the size member, decrementing it by one, read the value at the specified position, and then replace it with NULL. This design diffuses thread contention by creating disjoint parallelism, which significantly increases performance at the cost of functionality.

The second alternative uses a competitive `cas` model to perform a tail operation. When this version is being used either `pushBack` or `popBack` operations may execute concurrently with other vector operations. Threads performing a `cas pushBack` or `cas popBack` will compete to change the value at the tail from NULL to the result of their operation. If a thread fails at changing a value, it will move onto the next position.

### *4.4.5 Random Access Operations*

In contrast to previous implementations [DPS06], the random access operations I developed supports bounds checking. There are safeguards in place to prevent incorrect behavior caused by accessing a position that is beyond the size of the vector.

Any attempt to access a position that is not within the bounds of the vector causes the function to return false. A position is not within bounds if the specified position is greater than or equal to the capacity or size of the vector. It is also not within bounds if it holds a value, NOTVALUE, indicating the position is out of bounds.

The random access read operation, `at`, is used to return the value at a specified position. Because the vector uses temporary helper objects, this function includes logic to return the logical value of a helper object instead of the helper object itself. Depending on the complexity of concurrent operations, the `at` operation may take longer than $\mathcal{O}(1)$ complexity to complete.

In [FVD15], the random access write operation has been replaced by a conditional write operation, `cwrite`, which behaves nearly identically to the hardware `cas` operation. The difference between the two is that `cwrite` contains logic to detect if a value is a reference to a helper object. If a reference is read, the helper object is removed by calling its help complete function, and the operation must retry.

Both operations could loop indefinitely if they continuously read references to helper objects and are unable to safely de-reference them. To address this danger, the [FVD15] uses the aforementioned progress assurance scheme (Section 5.3.3) and the association model [FLD15a]) to place a limit on the loop.

### *4.4.6 Multi Position Operations*

Multi-position operations alter or read the values at multiple different indexes. For modifications caused by this type of operation to be linearizable, the values at each address must appear to change at the same time. This restriction guarantees that if a thread reads a newer value at an address, it will not read an older value at another address.

To the best of my knowledge, I am not aware of any other non-blocking vector to support this type of operation.

The three common multi-position operations are: `insertAt`, `eraseAt`, and `map`. The first operation, `insertAt`, replaces the value at each index greater than a specified index with the value of the preceding index. Then it replaces the value at the specified index with a new value. The second operation, `eraseAt`, shifts the value at each index greater than or equal to a specified index with the value at the next index. The last operation, `map`, takes a function and set of indexes as arguments. For each index, it calls the function, passing as an argument the value at that index, and then replaces the value at the index with the result of the function.

To perform the above operations correctly is very high, and this functionality is provided for the sake of completeness of the vector's API.

In general, a multi-position operation replaces the value at each index with a reference to a helper object. After placing a helper, the thread associates it with the previously placed helper object, forming a doubly-linked list. A doubly-linked list was chosen because the number of positions to operate on could be unknown. After it has been determined that the operation is complete, each helper object is replaced by its logic value.

It is important to consider the order in which helper objects are placed. If care is not taken, then

a cyclic depend-icy may occur, resulting in dead-lock. To guard against this, helper objects are placed in ascending order of index. Ascending order, as opposed to descending order, was chosen to prevent a lengthy operation from significantly delaying a tail operation. If descending order was chosen, then the entire multi-position operation must be completed before any further tail operations could begin.

A complication does arise when multi-position helper objects and tail operation helper objects meet. This danger is avoided by including type checking in the complete function of a multi-position operation's descriptor. If it encounters a descriptor object known to have a cyclic dependency, it will perform special logic to resolve the dependency before calling complete.

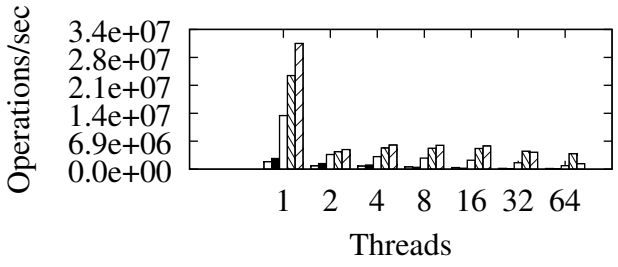
### *4.4.7 Performance Evaluation*

This section compares the performance of the wait-free vector (WFvec) to that of the lock-free vector (LFvec) [DPS06] and TBB's vector (TBBvec) [Int].

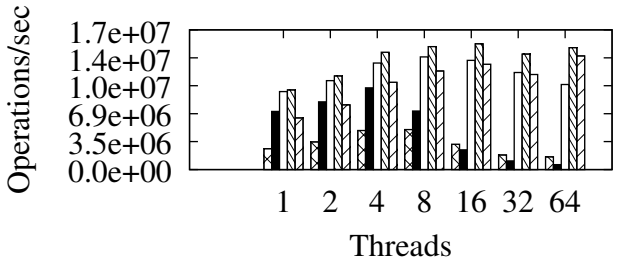
The performance comparison is a measurement on the number of operations completed in each test. For each test, a main thread constructs a vector with capacity of one million and then inserts ten thousand elements. Next, a set of worker threads are constructed, and once all are ready, they execute operations for 5 seconds. The operations executed are selected randomly based on a distribution. The following graphs show the performance of each vector as the number of threads, type of operations, and ratio of operations are varied.

These tests were conducted on a 64-core ThinkMate RAX QS5-4410 server running Ubuntu 12.04 LTS. It is a NUMA system with four AMD Opteron 6272 CPUs (16 cores per chip @2.1 GHz) and 314 GB of shared memory. For each algorithm tested, a separate executable was compiled using GCC 4.8 and the options -std=c++11 and -O3. The presented numbers are an average over

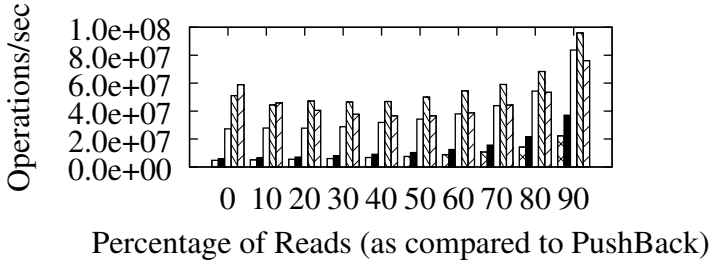
ten executions.



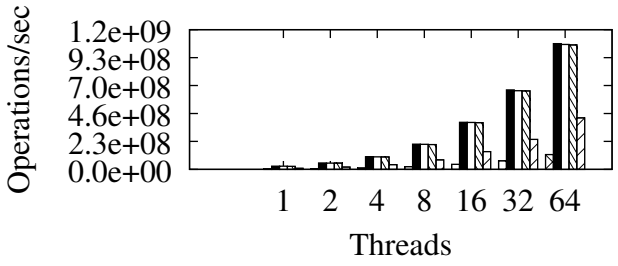

(a) 100% PushBacks




(b) 90% Read, 10% PushBacks




(c) 64 Executing Threads




(d) 100% Read

Figure 4.17: Random Access Reads and PushBack Operations

Figure 4.17 shows how the performance of each algorithm changes as the ratio of random access to `pushBack` operations is varied. TBBvec and WFfaa both outperform other implementations when there are solely `pushBack` operations, and this is shown in graph 4.17a. This use pattern is typical for applications that accumulate elements for later processing. On average, WFfaa performs 1.16 times as many operations per second as TBBvec, and 13.38 times as many as LFvec. Compared to WFcas and WFvec, WFfaa performs 2.3 and 22.12 times as many operations per second respectively.

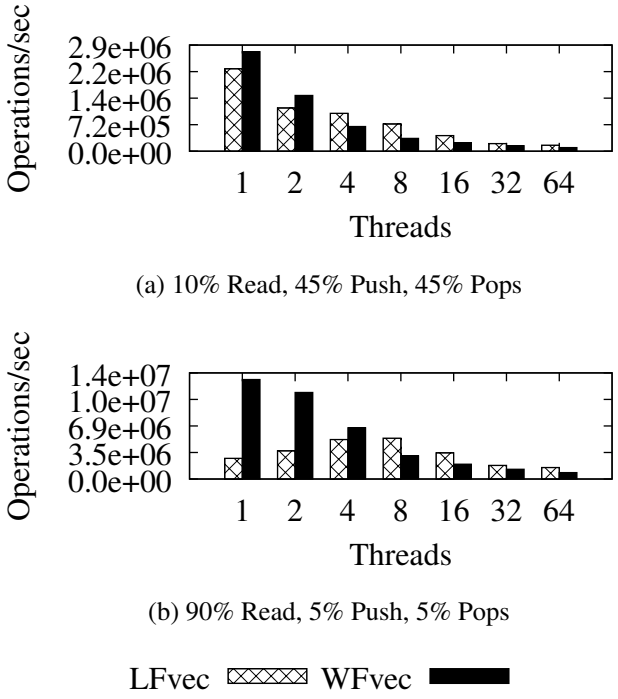
The poor scaling of the WFvec and LFVec were attributed to the cost of supporting conflicting operations. The WFcas scales more favorably than either the WFvec or LFVec due to its simpler design and smaller critical section. However, it can not distribute thread contention as effectively as the `faa` based approaches.

Graph 4.17b shows the performance when 10% of operations are `pushBack` and the majority of operations are random access reads. This corresponds to the scenario when a majority of threads are processing elements while a few are still adding elements. Each implementation scales similarly up until 8 to 16 threads, after which they lose performance. WFfaa and TBBvec maintain scalability the longest, followed by WFcas, WFvec, and LFvec. This loss in performance is attributed to the fact that there are 16 cores in each processor. Systems with a higher core per processor count may exhibit more favorable scaling.

Graph 4.17c shows the performance when the ratio of `read` to `pushBack` operations is varied and the number of threads is 64. Each algorithm scales similarly, with WFfaa performing best followed by TBBvec and WFcas. The number of operations completed increases significantly as the percent of `read` operations increase. This is attributed to the fact that the `read` operation is less costly than `pushBack`.

Graph 4.17d compares the performance of the vector's `read` operation. The WFvec performs

on average 9.28 times as many operations as LFvec and 2.66 times as many as TBBvec. This was unexpected as all three models were using the segmented array model. After analysis, this difference in performance was attributed to how the array segments are accessed in the WFvec. This approach uses non-atomic loads to access the array segment. If a NULL reference is read, an atomic load is used to ensure the value is current. This procedures is safe because once a non-null value is placed at that address, it will not be changed.




(a) 10% Read, 45% Push, 45% Pops

(b) 90% Read, 5% Push, 5% Pops

Figure 4.18: Random Access Reads and Tail Operations

Fig. 4.18 presents the performance of the vectors when there are interleaved `pushBack`, `popBack`, and `read` operations. TBBvec, WFfaa, and WFcas are not included in this test as they do not support concurrent `pushBack` and `popBack`. In this scenario, LFvec outperforms WFvec by a factor of 1.46, with both approaches scaling poorly as the number of threads increases. The methodology used by LFvec to support both operations has significant safety concerns ( [DPS06]), and I believe that the marginal performance benefit achieved by such a design does not justify the risk.

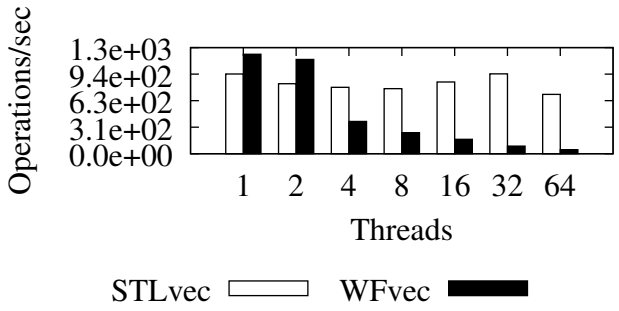



Figure 4.19: 50% insertAt, 50% eraseAt

To the best of my knowledge, there are no other concurrent vector that supports `insertAt` or `eraseAt` operations. Figure 4.19 compares performance of the WFvec's operations to that of the C++ Standard Library (STLvec) with a global lock. STLvec outperforms the WFvec by a factor of 5.28.

The WFvec's `insertAt` and `eraseAt` operations become serialized at the lowest shared index. Whichever operation places a descriptor object at that index first, must occur first. This is effectively the same as locking the hash map, except that it is significantly less efficient.

Regardless of the performance, these operations are important for two reasons: First, the design can be adapted to perform other types of vector operations, such as the map function. Second, these operations are used infrequently and supporting them does not increase the design complexity of other operations.

Table 4.1 depicts the performance difference between STLvec and the WFVec when there are three groups executing different types of operations. The left side of this table indicates the number of threads assigned to perform a specified class of operation. The right side displays the factor by which the number of operations increase (+) or decrease (-) when using WFvec as compared to STLvec.

Table 4.1: Thread group comparison to STLvec

| Threads: | | | Performance Improvement Factor: | | |
|---|---|---|---|---|---|
| Shift | Tail | RA | Shifts | Tails | RAs |
| 16 | 24 | 24 | -7.02x | +448.00x | +494,886.71x |
| 16 | 48 | 0 | -9.80x | +226.47x | N/A |
| 2 | 31 | 31 | -1.33x | +336.12x | +653,188.12x |
| 4 | 16 | 44 | -2.39x | +205.81x | +414,960.92x |
| 4 | 30 | 30 | -2.82x | +300.03x | +299,140.57x |
| 4 | 44 | 16 | -1.91x | +215.70x | +162,612.65x |
| 4 | 60 | 0 | -3.63x | +238.50x | N/A |

Even though the WFvec performs significantly fewer *shift* operations, it performs massively more *tail* and *Random access (RA)* operations. If a developer is only using shift operations, they should be using a linked-list. Otherwise, based on this table, the WFvec will most likely provide better performance than the STLvec.

### *4.4.8 Tervel Implementation*

This design was originally implemented using the Tervel framework. Its final implementation is available in the current release of the Tervel library.

## 4.5 Application: Dedup

In [FBL13], I study the effective use of non-blocking containers in a data deduplication application.

### *4.5.1 Overview of Dedup*

PARSEC's deduplication application performs a large number of concurrent compression operations on a data stream using the pipeline parallel processing model. The purpose of this study was to determine the practical implications of integrating non-blocking algorithms into real-world applications.

In this study, I manually re-factored the code, replacing the conventional lock-based synchronization mechanisms with non-blocking methodologies. Specifically, I integrated my wait-free hash map [FLD13b] and boost's lock-free queue implementation [Boo] in an attempt to increase the degree of concurrency of the application.

The goals of this project was to understand the necessary modifications needed to convert a lock-based multi-thread application to a non-blocking application and to identify scenarios in which non-blocking constructs improve the application's performance.

### *4.5.2 Contributions*

My contributions to this work was the analysis of the design of the non-blocking containers and detailing of the steps needed to integrate them within the duplication application. I then studied their behavior in a wide variety of inputs and configurations of the application. I designed a micro-benchmark that mimicked the application's behavior, and using this benchmark, I was able to validate and confirm my results.

Additionally, this work compared the semantic differences between the API of traditional sequential containers and containers designed for use in concurrency. This distinction is important, as many applications use containers that have a sequential API in an concurrent environment, which

complicates the process of replacing these containers with non-blocking versions.

### *4.5.3 Performance Evaluation*

Figures 4.20a and 4.20b reveal that the performance differences between the hash map implementations are minor. In figure 4.20c, a much larger input file is used, and as a result, I see that the wait-free hash map performs between 7 and 21% better.

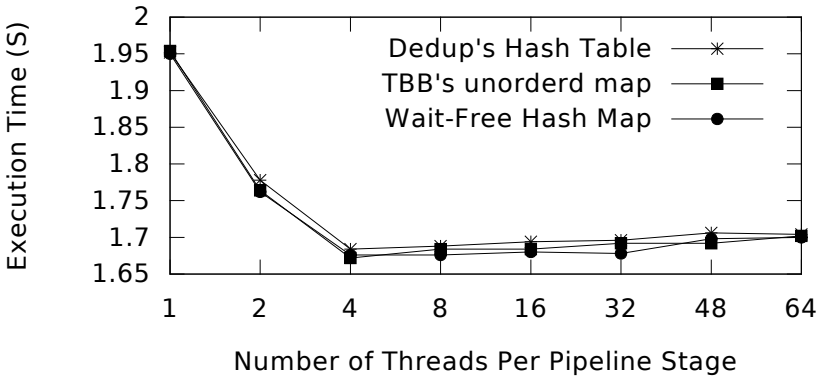
This performance improvement is attributed to the increased amount of work completed by each pipelines stage. Performance peaks at 16 threads per stage, with four stages, this means that all cores of our system are being utilized.

It's theorized [FBL13] that the wait-free hash map performs better than other designs because of its collision management scheme and how it diffuses thread contention across shared memory.

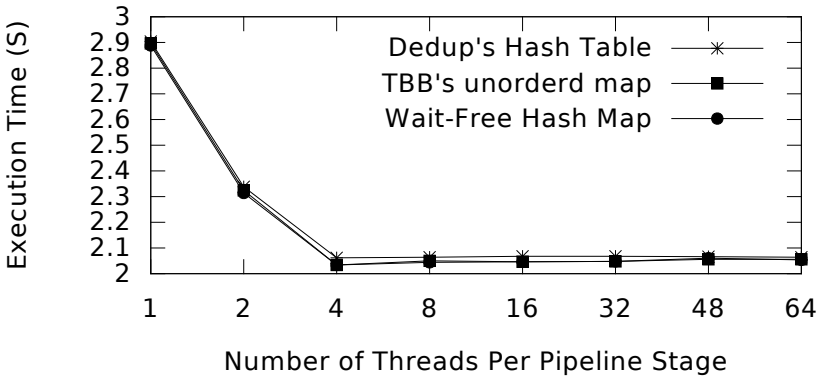
Figure 4.21 compares how performance is affected by replacing the queue. In contrast to the hash map, the queue focuses contention on the head and tail memory locations, decreasing opportunities to exploit parallelism. The results in these graphs confirm this as there is no significant change in performance across implementations.

### *4.5.4 Summary of Dedup*

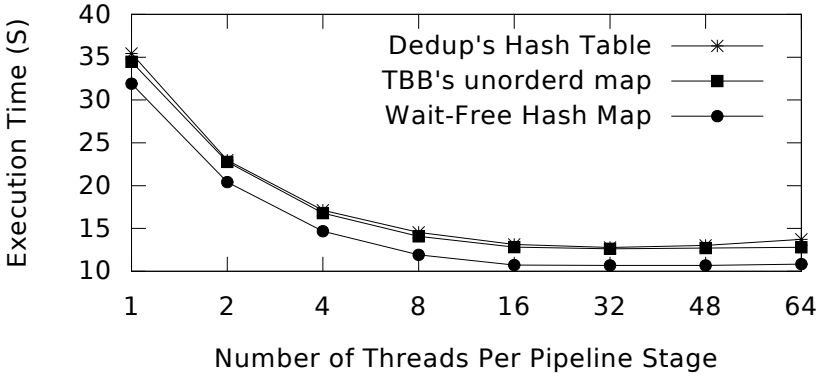
The conversion and performance analysis of the *Dedup* application resulted in a collaborator of mine exploring automatic code generation of concurrent containers. Using this code generator, they hope to simplify the process of developing, using, and testing of concurrent containers.




(a) Input: Simsmall

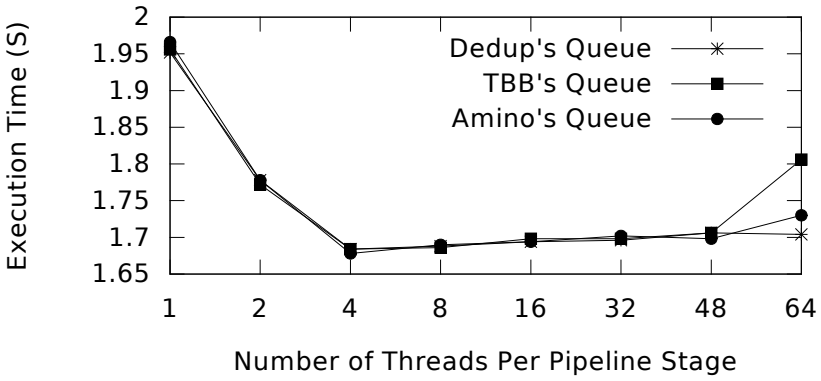



(b) Input: Simmedium

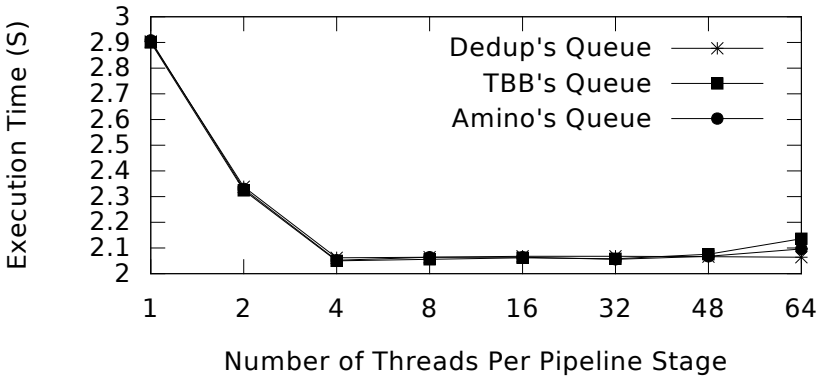



(c) Input: SimLarge

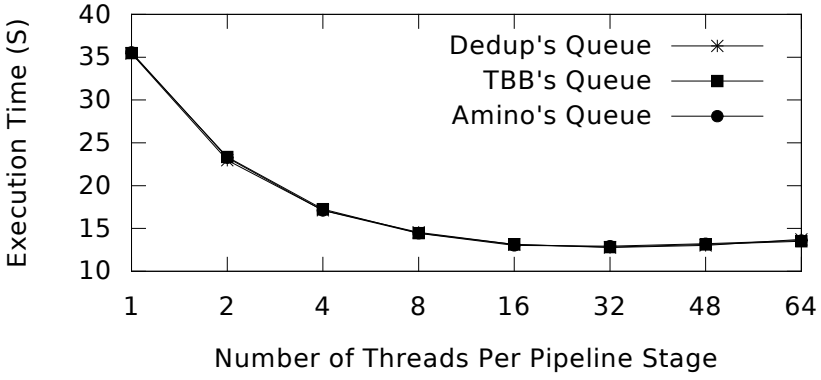
Figure 4.20: Performance of *Dedup* when using different hash map implementations




(a) Input: SimSmall




(b) Input: Simmedium




(c) Input: Simlarge

Figure 4.21: Performance of *Dedup* when using different queue implementations

# CHAPTER 5: RESULTS

The cumulation of my research has resulted in the creation of a new library of wait-free algorithms and containers. This library, Tervel, also contains a powerful framework that facilitates efficient implementation of wait-free algorithms.

## 5.1 Tervel Overview

Tervel consists of a framework for implementing wait-free algorithms and the collection of algorithms implemented using this framework.

Tervel's framework consists of three core components:

- Inter-thread helping techniques (Section 5.3.1)
- Progress assurance scheme (Section 5.3.3)
- Memory management constructs (Section 5.3.5)

In addition to these core components, it also provides several other useful features, such as management of recursive helping and accessor objects.

Tervel's framework was developed in parallel with the development of the wait-free vector [FVD15] with the goal being to implement the vector and future algorithms in a more systematic and less error prone way.

In addition to a wait-free vector, Tervel also includes wait-free stack, ring buffer, hash map, and multi-word compare-and-swap algorithms.

## 5.2 Tervel Related Work

I am aware of several concurrent libraries that focus on fine-grained synchronization and progress guarantees. Below is a brief summary of them and key differences between them and Tervel.

The C++ Standard Template Library (STL) provides several sequential containers; none are concurrent.

Amino Concurrent Building Blocks (Amino) is an open source software project [ami10]. Its goal is to develop concurrent libraries or building blocks that can be used by programmers. It provides several implementations of lock-free algorithms, but does not include any wait-free algorithms. This library was last updated on April 14th, 2010.

Boost [Boo] provides a lock-free queue and a lock-free stack algorithm. These algorithms are implemented based on the designs described in [Her08]. Like Amino, it does not provide any wait-free algorithms.

LibCDS [lib14] is a collection of lock-free and lock-based fine-grained algorithms such as maps, queues, lists, etc. The library contains implementation of well-known algorithms and memory reclamation schemas for modern processor architectures. While it provides more functionality and algorithms than the initial library release, I believe that over time the library will grow to support these functionalities. A key difference between Tervel and LibCDS is that the goal of Tervel is to ensure that every component is wait-free.

STAPL (the Standard Template Adaptive Parallel Library) [TBF11] is a framework for developing parallel programs in C++. It is designed to work on both shared and distributed memory parallel computers. STAPL includes a run-time system, design rules for extending the provided library code, and optimization tools. Its goal is to allow the user to work at a high level of abstraction

and hide many details specific to parallel programming, and to allow a high degree of productivity, portability, and performance.

The differences between all of these approaches are summed up in Table 5.1.

Table 5.1: Library Features by Degree (none, low, some, high)

| | STL | Tervel | Amino | Boost | CDS | STAPL |
|---|---|---|---|---|---|---|
| Reliance on a Runtime System | none | none | low | none | low | high |
| Non-blocking Algorithms | none | high | some | low | some | none |
| Non-blocking Memory Reclamation | none | high | low | low | some | none |
| Source Code Availability | high | high | high | high | high | none |

## 5.3 Tervel Framework

Tervel framework is a collection of descriptor-based techniques for non-blocking synchronization. It provides implementations of inter-thread helping techniques, a progress assurance scheme, and memory management.

### *5.3.1 Inter-Thread Helping Techniques*

To support descriptor-based helping techniques, Tervel provides abstract classes to guide the implementation of descriptor objects. Objects extending these classes must provide implementations of the `complete` and `value` member functions.

The complete function is used to remove a descriptor object from an address, and the value function returns the logical value of the descriptor object. This is either the value that the descriptor object

replaced or a value determined by the operation that placed the descriptor. The latter is returned if the operation has been completed, but the descriptor has not yet been removed.

These functions enable a developer to more readily reason about the correctness of two concurrent operations. The developer must only need to consider the case in which a thread calls the complete function of a descriptor object at an arbitrary point in time. Without such functions, the developer may have to consider more complex interactions; e.g., two or more different operations executing concurrently.

If the algorithm's operations are linearizable, then it can be shown that two concurrent descriptor-based operations, operating on overlapping address spaces, are ordered by whichever placed a descriptor object at a common address first. The other will either see the descriptor and help, or the operation will be completed first. Cyclic dependencies may arise if an operation uses multiple descriptor objects. However, this can be prevented by placing descriptor objects in an ascending or descending address order.

When working with multiple dependent descriptors, an association model can be used to ensure correct behavior. In this model, one object is a *parent* and the rest are *children*. The parent contains atomic reference(s), which are initially NULL and set using a `cas` operation. The children contain a reference to the parent object. A child and a parent are said to be *associated* if the child references the parent and the parent references the child. When using this model, it is necessary to include specific logic in the `on_watch` function of a child descriptor object to ensure that it returns TRUE only if the child is associated with its parent. In general, the `on_watch` function attempts to acquire a watch on the parent object, and if successful, it attempts to associate them. If this association fails, the descriptor object is replaced by its logical value before returning FALSE Otherwise, the function returns TRUE.

The watching of the parent object is an important step. Consider the case in which a thread attempts

to associate a child object with a parent object. However, just before the `cas` operation is invoked, the parent object is freed and reused. When the `cas` operation executes, the application could experience undefined behavior.

By encapsulating this logic in the `on_watch` function, the number of places in which an implementation error may occur is reduced. Instead of having each operation include logic to handle an object's specialized logic, the `on_watch` function ensures that if an object is watched, it is also associated.

### *5.3.2 Recursive Helping*

Recursive helping has not been discussed significantly in the literature, but its presence may lead to a scenario in which a thread consistently sees new descriptor objects that it must remove before being able to finish executing its current operation. Tervel provides two mechanisms by which to detect such an event.

The first is that each thread tracks the number of operations it is currently helping. If this number exceeds the number of executing threads, the thread will return back to its own operation. For a thread to have gotten to this point, it implies that at least one of the operations the thread believes it is helping has completed. If this is the case, a dependency between the thread's own operation and the one it is currently helping no longer exists.

The second mechanism has each thread store, in a thread-local variable, the address of a control word. When the value of this control word is no longer NULL, it implies the thread's operation is complete. This allows a thread to detect if some other thread has completed their operation while it was performing a helping routine. For algorithms that use the association model, the control word is often the atomic reference to the child member inside the parent descriptor object.

### 5.3.3 Progress Assurance Scheme

Progress assurance allows the construction of wait-free algorithms by preventing scenarios of live-lock in the event a thread is continually preempted by other threads. The majority of the wait-free algorithms implemented in Tervel depend on this for their progress guarantees. Tervel's progress assurance scheme is constructed from an announcement table [Her91, KP12], descriptor objects [Bar93], and the association model [FLD15a]. The progress assurance scheme works as follows:

Before a threads begins an operation, it will check for an announcement (Algorithm 5.3 Line 2). The `check_for_announcement` function contains two THREAD_LOCAL integers, $checkDelay$ and $checkPos$, which are used to reduce the cost of calling this function. In general, the cost of including this scheme is one atomic load for every $maxDelay$ calls to an algorithm or container's member function, where $maxDelay$ is a compile time constant chosen by the user. If $checkDelay$ is greater than zero, it is decremented and the function returns. Otherwise, $checkDelay$ is set equal to the $maxDelay$ constant, $checkPos$ is incremented, and the thread checks the $checkPos$ of the announcement table. The announcement table is an array of references to `OpRecord` type objects[1]. If the checked position contains a reference to an `OpRecord`, the `OpRecord`'s `help_complete` function will be called. The `help_complete` function has the requirement that the operation described in the `OpRecord` must be complete upon its return.

If while executing an operation the thread determines that is has live-lock (Algorithm 5.3 Line 4), it will create an `OpRecrd` that describes its operation and call the `make_announcement` function (Algorithm 5.3 Line 18). When the function returns. it guarantees that the operation is complete.

[1] See Section 5.3.4 for more details on `OpRecords`

Internally, this function will place a reference to the `OpRecord` in the announcement table, call its `help_complete` function, and then remove the reference.

### *5.3.4 Tervel Operation Records*

An operation record or `OpRecord` is a type of descriptor object that contains the information necessary for an arbitrary thread to execute an entire operation.

For simple operations, the *fast-path-slow-path* [KP12] design methodology is used. In this methodology, a thread examines the state of an `OpRecord` and executes it if its state is not in the complete state. When designing more complex operations, this design may allow the ABA problem or data races to occur. For example, if it is uncertain which memory words will be affected by an operation, the same operation may be successfully executed on multiple memory locations and values could be reused, leading to the ABA problem.

To avoid these problems, the association model is used when implementing complex operations. In general, a thread will replace a value with references to a child descriptor object that contains a copy of the value. Then a thread will attempt to associate the child descriptor with its parent (in this case, an `OpRecord`). If successful, the reference to the child descriptor is replaced by the result of the operation. Otherwise, it is replaced by the value contained within.

Section 5.4.1 provides an example of an operation record used to achieve wait-freedom in a linked-list based stack.

### *5.3.5 Memory Protection*

To handle memory reclamation, Tervel provides implementations of hazard pointers [Mic04] and reference counting [DMM01] to ensure objects are not re-used or freed while a thread is operating on them. The API for these implementations has been expanded to allow for their use with objects that have complex dependencies.

A number of papers that present concurrent algorithms suggest the use of either hazard pointers or reference counting to support reusing memory; however, they omit the necessary implementation details [FLD13b, FLD15a, TBK12, FH07]. Tervel provides a comprehensive interface by which developers can add either hazard pointer (HP) or reference counting (RC) protection to shared memory or objects. Throughout this paper, I refer to the act of applying memory protection as *watching*, the act of removing memory protection as *unwatching*, and an object that has memory protection as *watched*.

The standard procedure by which objects are protected is as follows:

- First read a value from an address.
- Next apply memory protection.
- Check to see if the value at the address has changed.

If the value has not changed, the object is considered watched.

To provide memory protection to objects with complex dependencies, the developer can define additional steps that are performed during the applying of memory protection. These steps are encapsulated by an `on_watch` member function, which is called by the `watch` function if the object was successfully watched. If the `on_watch` function returns FALSE, the `watch` function

removes the watch on the object and also returns FALSE. In addition to the `on_watch` function, Tervel also provides `on_unwatch` and `on_is_watched` functions. Section 5.4.2 shows how these functions are leveraged in Tervel's re-implementation of the multi-word compare-and-swap.

### *5.3.6 Memory Reclamation*

In order to safely reuse objects or return memory to the system, Tervel provides both thread-local and shared memory pools. When an object is no longer needed, the *object's owner* will call a specialized free function based on how the object was allocated. An object's owner is a thread responsible for freeing that object. An object must be owned by only one thread or it may result in an object being freed by multiple threads. In general, an object's owner is determined as follows:

- An object is initially owned by the thread it was allocated to.
- A thread takes ownership of an object that it removed all references to it.
- An object's ownership transfers to a thread if it becomes associated with that thread's operation.

The last point is necessary for objects that contain references to other objects. For these objects, it is usually the case that none of the objects can be freed while any are watched. An object is freed only if the call to `is_watched` returns FALSE. This function internally calls the `on_is_watched` member function of the passed object. This allows a developer to encode logic to prevent an object from being freed prematurely. It does require a *root* object to be identified and have that object's destructor call the appropriate free function for each object referenced by the root object.

When freeing an `HPElement` object, a thread adds the object to its thread-local `HPElement` memory pool. Then the `is_watched` function is called on each element in the pool. If the

function returns FALSE, the object is removed from the pool, and it is returned to the system allocator.

An `RCElement` cannot be returned to the allocator; instead, it is moved from an *unsafe* memory pool to a *safe* memory pool. This is because it is possible for the reference count member of the object to be incremented at any point, making it unsafe to return the object to the system. When allocating a `RCElement`, the thread will attempt to get an object from the following sources in order: thread-local safe pool, thread-local unsafe pool, shared safe pool, and finally system allocator. To prevent a single thread from accumulating too many objects, a load balancing scheme is used. If a thread contains too many objects, it offloads the excess to the shared pool.

To simplify management of subclasses of `RCElement` that exhibit varying sizes, the allocated size of these objects is restricted to be a multiple of the system cache. A separate pair of unsafe and safe pools are used for each size. This implementation improves memory utilization of an application by allowing all instances of all algorithms to share a common set of memory pools. This is in contrast to algorithms that contain their own independent reclamation scheme.

## 5.4 Wait-Free Algorithms and Containers

Tervel provides a number of abstract classes and structures to guide a developer who is implementing non-blocking or wait-free algorithms. The following section presents excerpts from several wait-free algorithms implemented in Tervel. These excerpts were selected because they showcase the expressiveness, functionality, and conciseness of the framework. For full implementation details, please visit: `cse.eecs.ucf.edu/tervel`

### 5.4.1 Wait-Free Stack

In addition to the published algorithms, I have also implemented a wait-free stack to illustrate how a developer may user Tervel to implement a wait-free container. The design uses a competitive `compare-and-swap` model in which threads compete to modify a single shared variable. To guard against a possible live-lock scenario, I used Tervel's progress assurance scheme to implement operation records.

```
template<typename T>
class Stack {
  class Node; /* Used to store values */
  class Accessor; /* Used to safely dereference objects*/
  Stack() {};
  ~Stack() {};
  bool push(T v);
  bool pop(T &v);

  std::atomic<Node *> head_{nullptr};
};
```

Figure 5.1: Wait-Free Stack Class

```
class Stack::Node :
    public util::memory::hp::Element {
  Node(T &v) : _val(v) {};
  ~Node() {};
  T value() { return _val; };
  void value(T &v) { _val = v; };
  void next(Node *n) { next_ = n; };
  Node *next() { return next_; };

  T _val;
  Node *next_ {nullptr};
};
```

Figure 5.2: Stack Node Object Class

This stack implementation has a concurrent oriented API (Figure 5.1), uses a linked-list structure to store elements (Figure 5.2), and provides *last-in-first-out* (LIFO) ordering of elements. Its `pop`

operation returns a boolean indicating whether or not an element was popped, and if so that value is also returned. Its `push` operation always returns true in this implementation. Other implementations may limit the number of elements stored, and in these implementations it may return false in some cases.

Operations on the stack are executed by using a `cas` operation to change the head of the linked-list. Figures 5.3 and 5.4 show how the two stack operations are completed.

```
bool pop(T& v) {
  util::ProgressAssurance::check_for_announcement(); {
  util::ProgressAssurance::Limit progAssur;
  while (!progAssur.isDelayed()) {
    Accessor access(&head_);
    Node *cur;
    if (access.load(&cur)) {
      if (cur == nullptr) return false;
      Node *next = cur->next();
      if (head_.compare_exchange_strong(cur, next)) {
        v = cur->value();
        cur->safe_delete();
        return true;
      }
    }
  }
  PopOp *op = new PopOp(this);
  util::ProgressAssurance::make_announcement(op);
  bool res = op->result(v);
  op->safe_delete();
  return res;
};
```

Figure 5.3: Wait-Free Stack Pop Operation

```
bool push(T v) {
  util::ProgressAssurance::check_for_announcement();
  util::ProgressAssurance::Limit progAssur;
  Node *elem = new Node(v);
  while (!progAssur.isDelayed()) {
    Accessor access(&head_);
    Node *cur;
    if (access.load(cur)) {
      elem->next(cur);
      if (head_.compare_exchange_strong(cur, elem))
        return true;
    }
  }
  delete elem;
  PushOp *op = new PushOp(this);
  util::ProgressAssurance::make_announcement(op);
  op->safe_delete();
  return true;
}
```

Figure 5.4: Wait-Free Stack Push Operation

In general, the `push` operation is completed when a thread successfully replaces the value of `head_` with a node object whose next pointer is equal to that value (Figure 5.4 Line 10). Similarly, the `pop` operation is completed by replacing the value of `head_` variable with `head_->next()` (Figure 5.3 Line 10).

In both operations, an `Accessor`(Figure 5.5) object is used to encapsulate logic related to memory protection. The `Accessor's load` function reads the current value of the address and attempts to apply hazard pointer memory protection. If successful, the function returns TRUE and assigns the read value to the pass-by-reference variable $v$. Otherwise, FALSE is returned. Upon its destruction, the hazard pointer protection is released.

If an operation record is being used to complete an operation, it is possible to read a value that is a helper type(Figure 5.5 Line 8). In this implementation, the `on_watch` function of the helper type always returns FALSE. This is because internally it replaces the helper with its logical value

(Figure 5.7 Line 10).

```
class Stack::Accessor {
  typedef util::memory::hp::HazardPointer hp_t;
  Accessor(std::atomic<Node *> *a) : a_(a){};
  ~Accessor() { hp_t::unwatch(hp_t::SlotID::SHORTUSE); };
  bool load(Node * &v) {
    v = a_->load();
    if (v == nullptr) return true;
    if (util::is_1st_lsb_1(v)) {
      Helper * h = util::get_1st_lsb_0(v);
      hp_t::watch(hp_t::SlotID::SHORTUSE, h, a_, v);
      return false;
    }
    return hp_t::watch(hp_t::SlotID::SHORTUSE, v, a_, v);
  };
  std::atomic<Node *> *a_;
};
```

Figure 5.5: Stack Accessor Class

```
class StackOp : public util::OpRecord {
  StackOp(Stack *s) : ds_(s) { };
  ~StackOp() {
    Helper *h = helper_.load();
    if (h != fail_val_)
      delete h;
  }
  bool on_is_watched() {
    if (helper_.load() != fail_val_)
      return hp_t::is_watched(helper_.load());
    return false;
  };

  virtual bool associate(Helper *h) {
    Helper *temp = nullptr;
    helper_.compare_exchange_strong(nullptr, h);
    return helper_.load() == h;
  };

  ...
  Stack * ds_;
  std::atomic<Helper *> helper_{nullptr};
};
```

Figure 5.6: Stack Operation Record Class

```
class Helper : public util::memory::hp::Element {
  Helper(StackOp *op) : op_(op) {}
  bool on_watch(std::atomic<void *> *a, void *e) {
    if (hp_t::watch(hp_t::SlotID::SHORTUSE2, op_, a, e)) {
      finish(a,e);
      hp_t::unwatch(hp_t::SlotID::SHORTUSE2);
    }
    return false;
  };
  void finish(std::atomic<Node *> *address, Node *n) {
    if (op_->associate(this))
      address->compare_exchange_strong(n, n_value_);
    else
      address->compare_exchange_strong(n, o_value_);
  };
  StackOp * const op_;  Node * o_value_; Node * n_value_;
};
```

Figure 5.7: Stack Helper Class

Without using the progress assurance scheme, it is possible for a thread to indefinitely execute the while loops in the `pop` and `push` algorithms. Figure 5.6 presents the general stack operation record, `StackOp`, which consists of a pointer to the stack container and an atomic reference to a helper object. Note how the `on_is_watched` checks whether or not an associated helper object is watched and how the destructor function frees the helper object.

```
void help_complete() {
  Helper * h = new Helper(this);
  Node *h_lsb = util::get_1st_lsb_1(h);
  while (this->helper_ == nullptr) {
    Node *cur; Accessor access(&ds_->head_);
    if (access.load(cur) == false) continue;
    if (cur == nullptr) { StackOp::fail(); break;}
    h->o_value_ = cur;
    h->n_value_ = cur->next();
    if (ds_->head_.compare_exchange_strong(cur, h_lsb)) {
      h->finish(&ds_->head_, h_lsb);
      if (this->helper_ != h) h->safe_delete();
      return; }
  }
  delete helper;
};
```

Figure 5.8: Stack PopBack Operation Help Complete Function

Figure 5.8 shows the `help_complete` function of the `pop` operation record. For brevity, the `help_complete` function of the `push` operation has been omitted. This `help_complete` function is similar to the `pop` algorithm from Figure 5.3 with the following differences:

- The while loop terminates when the value of the `helper_` variable is no longer NULL.
- The function places a helper object, which is then replaced by the result of the operation, as opposed to placing the result of the operation in the first place. This is important to prevent the effects of an operation from occurring multiple times.

After a helper object has been placed, its finish function is called, which replaces it with its logical value. This function, shown in Figure 5.7, includes logic to check the association between the helper and its operation. If the operation is not associated with a helper, an attempt will be made to associate them. If they are associated, then the helper is replaced with its `n_value_` member; otherwise, it is replaced with its `o_value_` member.

```
class MCASop<T> : public util::OpRecord {
  ...
  ~MCASop() {
    for helper in helpers {
      util::memory::rc::free_descriptor(helper, true);
  }
  bool on_is_watched() {
    for helper in helpers
      if (helper == MCAS_FAIL_CONST) {
        break;
      } else if (util::memory::rc::is_watched(helper)) {
        return true;
      }
    }
    return false;
  }
  void help_complete(x=0) {
    for (; x < helpers.length; x++) {
      if (place\_helper(x) == false) {
        return;
      }
    }
    state.cas(undecided, pass);
  }
  ...
  atomic<State> state;
  T* addresses[];
  T expected_values[];
  T new_values[];
  atomic<Helper *> helpers[];
```

Figure 5.9: MCAS operation record

### *5.4.2 Multi-Word Compare-And-Swap*

The wait-free Multi-Word Compare-and-Swap [FLD15a] algorithm was reimplemented using Tervel's design patterns. Compared to the original implementation, there was fewer redundancy and better encapsulation of helping routines. For example, the `on_watch` function associates objects, removing the need to handle unassociated or incorrectly placed objects from within each function. Instead the handling of these events is done purely within `on_watch`.

```
class Helper : public util::Descriptor {
  ...
  bool on_watch(address, cur) {
    typedef util::memory::hp::HazardPointer hp_t;
    if (hp_t::watch(hp_t::SlotID::SHORTUSE, cur, address, op)) {
      bool res = this.associate();
      hp_t::unwatch(hp_t::SlotID::SHORTUSE);
      return res;
    }
    return false;
  }
  ...
  T value() {
    if (op.state == passed)
      return op.new\_values[idx];
    else
      return op.expected\_values[idx];
  }
  void complete(address, cur) {
    op->help\_complete(idx+1);
    if (op.state == passed)
      temp = op.new\_values[idx];
    else
      temp = op.expected\_values[idx];
    address.cas(cur,value())
  }
  ...
  bool associate() {
    op->helpers[idx].cas(null,this)
    if (op->helpers[idx].load() != this) {
      temp = op.expected_values[idx];
      op->addresses[idx].cas(this, temp);
      return false;
    } else { return true;}
  }
  ...
  const MCASop *op;
  const int idx;
```

Figure 5.10: MCAS helper descriptor object

The MCAS design uses two types of descriptor objects, which are partially described in Figure 5.9 and 5.10. It is performed by iteratively replacing the expected value at each address with a reference to an MCasHelper. To prevent the ABA problem from occurring, this design uses the association model (Section 4.2.3). After placing an MCasHelper, the next step is to associate it

with its MCasOp. The association model is expressed by defining an `associate` function (Figure 5.10 Line 28). This function uses a `cas` operation to assign a child reference to the address of the MCasHelper. It returns whether or not the child references the MCasHelper. If it references some other MCasHelper, the function removes the MCasHelper. It is important to quickly remove incorrectly placed objects to prevent other threads from accessing them.

Figure 5.10 Line 19 presents the complete function that is called by a thread to remove an MCasHelper placed by a different thread. Because a thread calls the complete function after it has acquired a watch on the object, it does not have to consider the case in which a descriptor has been placed in error. For example, if an MCasHelper was placed in error, its `on_watch` function (which is called by the watch function) would have removed it when the call to associate returned false.

In order for an MCasHelper to be removed, the MCAS operation that placed it must be completed. This is accomplished by calling the MCASop's `help_complete` function. Upon its return, the state of the MCASop will have been changed from undecided to either passed or failed. Once the state has been decided, the MCasHelpers may be replaced with their logical values. The logical value of an MCasHelper is determined by calling its *value* function.

The MCAS implementation uses Tervel's memory management features to safeguard the reclamation of descriptor objects. It uses hazard pointers to protect MCASop objects and reference counting to protect MCasHelper objects. The MCASop object is responsible for freeing all MCasHelper objects referenced by it. As such, the thread that owns the MCASop also acquires ownership of those MCasHelpers. This ownership is expressed in the MCASop's `on_is_watched` and destructor function.

### *5.4.3 Wait-Free Hash Map*

The API of the wait-free hash map described in [FLD13b] mirrors that of the sequential hash map, but it allows concurrent operations. To safeguard access to key-value pairs, it is required that a thread acquire hazard pointer protection on an object before dereferencing. Unfortunately, this restriction breaks the wait-free guarantee of the algorithm. The authors describe a mechanism by which they use the `atomic bitwise OR` to force the table to expand in the event a memory address is experiencing heavy contention. However, I do not believe that this mechanism can be adapted to address possible live-locks introduced by applying hazard pointers.

To address this and other limitations of this concurrent design, the following adaptations were made:

- Operation records are used to apply an operation in the event live-lock is detected.
- The `get` and `update` operations were combined into an `access` operation.
- The `insert` operation now returns TRUE if a key-value pair was inserted and FALSE if it already exists in the hash map.
- The `remove` operation now returns a value indicating one of three possible results:
  - The key-value pair was removed.
  - The key-value pair was not removed because it is currently being accessed.
  - The key-value pair is not in the hash map.

The `access` operation takes as arguments the key to find and a Tervel accessor object, and returns a boolean indicating whether or not the key was found in the hash map. The accessor object removes a lot of ambiguity that may occur in the original API where multiple updates and/or

deletes may occur on the same key. A simplified example of this is presented in Figure 5.11. For brevity, implementation details of the `searchForKey` function have been omitted. In short, `searchForKey` searches the hash map for the passed key and returns the following:

- found: This indicates whether or not the key was found.
- array: The array on which the key is or would be stored.
- pos: The position on the array at which the key is or would be.
- pair: A value loaded from array[pos]. If it is a reference to an object, the object was successfully watched.

```
typedef util::memory::hp::HazardPointer hp_t;

bool access(Key, Accessor *access) {
  found, array, pos, pair = searchForKey(key);
  if (found) {
    res = pair.increment_access();
    hp_t::unwatch(hp_t::SlotID::SHORTUSE);
    if (res >= 0) {
      access->init(pair);
      return true;
    }
  }
  return false;
}
```

Figure 5.11: Hash map access operation

If the `access` operation returns TRUE, the accessor is used to read and modify a key's value. Each key-value pair contains an atomic counter, and internally the `access` operation performs a fetch-and-add on this counter. If the fetch-and-add returns a non-negative value, a reference to a pair is stored within the accessor. Otherwise, a negative result indicates that the object has been deleted. When the accessor is deleted, its destructor decrements the counter of the pair within.

```
bool remove(Key key) {
  util::ProgressAssurance::check_for_announcement();
  found, array, pos, pair = searchForKey(key);
  if(found == false) { return not_found; }
  else if (pair->logically_delete()) {
    removeReference(array, pos, pair);
    return key_removed;
  else { return key_in_use; }
}
```

Figure 5.12: Hash map remove operation

```
bool insert(Key key, Value value) {
  util::ProgressAssurance::check_for_announcement();
  util::ProgressAssurance::Limit progAssur;
  pair = hashmap_make_pair(key, value);

  while (!progAssur.isDelayed()) {
    found, array, pos, cur = searchForKey(key);
    if (found) {
      hashmap_return_pair(pair);
      return false;
    } else if (cur == nullptr) {
      if (array[pos].cas(cur, pair))
        return true;
    } else {
      expand_map(array, pos, cur);
    }
  }
  HashMapInsertOp *op = new HashMapInsertOp(this, pair);
  util::ProgressAssurance::make_announcement(op);
  res = op->result();
  op->safe_delete();
  return res;
}
```

Figure 5.13: Hash map insert operation

A `remove` operation attempting to delete a key-value pair will first attempt to logically delete it (Figure 5.12 Line 5). Internally, the `logicalDelete` function attempts to change the atomic counter from 0 to $-1 * number\_of\_threads$. If successful, this prevents other threads from accessing the key-value pair. If it fails, the `remove` operation returns a value indicating that a thread is accessing the specified key.

Figure 5.13 presents the `insert` operation, and Figure 5.14 presents its operation record. For brevity, only the `insert` operation's operation record is included.

```
class HashMapInsertOp : public util::OpRecord {
  ...
  T result() { return helper_.load() != fail_value_; }

  bool associate(address, helper) {
    helper_.cas(null, helper);
    if (helper_.load() == helper) { address.cas(helper, pair); }
    else { address.cas(helper, nullptr); }
  }
  ...
  void help_complete() {
    h = new HashMapHelper(this);
    while (helper_.load() == nullptr) {
      found, array, pos, cur = searchForKey(key);
      if (found) {
        helper_.cas(nullptr, fail_value_);
        delete h;
        return;
      } else if (cur == nullptr) {
        if (array[pos].cas(cur, h)) {
          this->associate(h);
          h->safe_delete();
          return;
        }
      } else { expand_map(array, pos, cur); }
    }
  }
  ...
  HashMap *map;
  HashMapPair *pair;
  atomic<HashMapHelper *> helper_;
}
```

Figure 5.14: Hash map insert operation record

This design places a helper object on an array, associates it with an operation record, and then replaces it with a reference to a key-value pair. The helper object allows the thread that made the operation record to determine if the insert was successful or not.

```
class HashMapHelper : public util::Descriptor {
  ...
  bool on_watch(address, cur) {
    typedef util::memory::hp::HazardPointer hp_t;
    if (hp_t::watch(hp_t::SlotID::SHORTUSE, cur, address, op)) {
      bool res = op->associate(address, this);
      hp_t::unwatch(hp_t::SlotID::SHORTUSE);
    }
    return false;
  }
  ...
  HashMapOp *op;
}
```

Figure 5.15: HashMapHelper descriptor object

An alternative to using the helper object would be to include an additional variable in the key-value pair. However, I believe that such a design will require additional conditional statements in the key-value pair's `on_watch` function. I choose not to go with this approach, because I believe that the conditions necessary for an operation record to be used are highly unlikely.

## 5.5 Tervel Summary

In summary, the Tervel library provides a framework that developers can use to streamline the implementation of their non-blocking algorithms. It includes inter-thread helping techniques, memory management constructs, and a progress assurance scheme.

Application developers can take advantage of the hash map, vector, stack, and other algorithms already implemented within Tervel in their concurrent applications. The API of these algorithms have been adapted to provide the developer with ability to more accurately reason about correctness and behavior of these algorithms within a concurrent environment. Additionally, compared to many other concurrent algorithm designs, these implementations often provide stronger safety properties and exhibit increased scalability.

By invitation, I presented the Tervel library to developers at SRI International[2]. This talk focused on the initial release, features, and applicability of the library. The Tervel library was also presented by Dr. Dechev at the International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS) and by Dr. Dechev and me as part of LockHeed Martin's webinar series on embedded computing. These presentations focused on the novelties and methodologies used in Tervel and the algorithms implemented within the framework. The Tervel library is open-source and available at the following site: ucf-cs.github.io/Tervel/

[2]Formally, Standford Research Institute International

# CHAPTER 6: PERFORMANCE METRICS

Selecting concurrent algorithms that operate efficiently on multi-core shared memory systems is a major challenge. Developers usually rely on the results of micro benchmarks or small scale tests of an application to select between two or more algorithm implementations. It has been my experience that such tests lead to false assumptions about the performance of concurrent algorithms. I found that the throughput[1] of algorithms can vary greatly across use cases and system architectures. This has motivated me to seek out new ways to compare concurrent algorithms to enable developers to better predict behavior and applicability.

Like sequential algorithms, hardware characteristics can have a significant effect on algorithm behavior. However, hardware features such as the number of processors and cores, hyper-threading, and cache characteristics, affect concurrent algorithms differently than sequential algorithms. For example, increasing the size of the cache line usually results in better performance; however, it may increase false-sharing in a concurrent algorithm, which would decrease performance [Gra15].

The performance of concurrent algorithms is also significantly affected by how it is used. Different workload and operation distributions (read-dominated vs. write-dominated) may change the scalability of an algorithm [AA14]. Often times performance is the measurement of the number of operations completed in a period of time; however, this can be misleading if a significant portion of the operations have no effect. For example, if a billion *pop* operations were performed on a stack with a hundred elements, then it would be a fallacy to judge its performance solely on operations completed. This is because after the first hundred pop operations the stack would become empty and the rest would not have an effect on the data structure.

[1] The number of operations completed in a period of time

This chapter describes how I utilized hardware and software metrics to gain deeper insights into how different non-blocking synchronization techniques impact algorithm performance. It is composed of two parts: the first describes my work supervising the extension of Tervel's framework to provide information on the behavior of its algorithms, and the second describes my work enhancing and extending OVIS's Lightweight Distributed Metric Service (LDMS) [AAB14].

## 6.1 Tervel Software Metrics

To gain insights into behavior of non-blocking designs, I supervised the instrumentation of the Tervel framework to support the tracking of key behaviors such as memory management, inter-thread helping, and progress assurance.

Specifically, the framework now supports monitoring of the following events:

- announcement_count: The number of times a thread determines it has been delayed and attempts to recruit other threads to help complete its operation.
- helped_announcement: The number of times a thread has been recruited to help a delayed thread.
- limit_value: The mean value of how close a thread comes to making an announcement before completing its operation.
- rc_is_descr: The number of times a thread checks if a value is a reference to a descriptor object.
- rc_remove_descr: The number of times a thread has to remove another thread's descriptor object.

- rc_offload: The number of times a thread must offload extra reference count protected objects from its memory pool to the shared pool.
- rc_watch_fail: The number of times a thread fails to acquire reference count protection on an object.
- hp_watch_fail: The number of times a thread fails to acquire hazard pointer protection on an object.
- max_recur_depth_reached: The number of times a thread reaches the maximum recursive depth and must return back to its operation.

The announcement_count, helped_announcement, and limit_value metrics provide insights into the effects of Tervel's progress assurance scheme and the likelihood of thread starvation if it were not used. Other metrics provide information relating to the inter-thread helping techniques and the memory management constructs. Using these metrics, it is possible to ascertain how threads affect one another when operating on overlapping address spaces.

To support the testing and comparison of various algorithms in various use cases, I expanded and refined the testing procedure that I have previously used when evaluating algorithm performance. This testing procedure and instrumentation has been integrated into the latest release of Tervel.

## 6.2 OVIS's Lightweight Distributed Metric Service

Using a software package called Lightweight Distributed Metric Service (LDMS) [AAB14] that was developed as part of a suite of scalable HPC monitoring tools called OVIS [BDG08], I explored techniques to evaluate the effectiveness of using periodic performance counter data collection for the analysis of distributed multi-core applications and algorithms. The advantage of developing

this type of utility is that it can be used to inform code users and developers of inefficiencies and changes in efficiency over the life of a system due to system software and hardware updates and application code changes.

My contribution to this tool was the extension and development of new hardware performance counter data collection modules for LDMS. In particular my specific contributions were the enhancement of LDMS's perf_event [Wea13] sampler and implementation of a new sampler for the PAPI [BDG00] library. These enabled the monitoring of both hardware and software events associated with distributed application execution. By analyzing the results of experiments monitored in this fashion, I identified ways to characterize and improve the performance of the associated multi-core algorithms.

Analysis of performed experiments using these samplers has identified patterns that may explain certain performance behavior of multi-core applications. It is hoped that by applying this information to algorithm design, it will enable developers to overcome performance limitations.

In the following two sections I describe the LDMS sampler modules I enhanced and implemented in order to enable scalable system wide measurement and analysis on HPC systems.

### *6.2.1 Sampler: perf*

*Linux's perf tools*, also referred to as perf_event [Wea13], is a tool that provides access to CPU performance counters through a generalized abstraction layer that removes the need to modify code when moving from one architecture to another architecture that supports similar metrics. Events can be tracked globally or limited to events triggered by a specified process, and they can be further refined to events that occur on a specified core. Because this tool can be utilized by *root* for the monitoring of any supported events, it can be used for global periodic monitoring as

a system service. The monitored information, taken in conjunction with scheduler and resource manager logs, can provide valuable insight into how a user application is utilizing node level resources on a per-core/per-subsystem granularity and how this varies across the user application's node allocation.

```
ldmsctl$ load   name=perfevent
ldmsctl$ config name=perfevent action=init component_id=<int> set=<string>
ldmsctl$ config name=perfevent action=add  pid=<int> cpu=<int> type=<int>
                id=<int> metricname=<string>
ldmsctl$ start  name=perfevent interval=<int>
```

Figure 6.1: How to use perf_event sampler

A perf_event sampler was already present in LDMS; however, the user interface for configuration was difficult to use and lacked the ability to monitor the *uncore* counters. My enhancement consisted of a simplified interlace and an extension to support *uncore* counters. Figure 6.1 describes how a user can instantiate, configure, and start the improved sampler. If the developer specifies a cpu core value of $-1$, it will track the specified process across all cpu cores and if a pid of $-1$ is specified, all processes on a single cpu core will be tracked. The number of events and processes which can be tracked by this sampler is only limited by the number supported by the perf_event library, which may vary on the hardware architecture. Perf_event provides a utility program, *perf list*, that displays a list of supported events for the current architecture.

### *6.2.2 Sampler: PAPI*

*The Performance API* or PAPI project is aimed at developing a standard programming interface by which hardware performance counters are accessed [BDG00]. One of PAPI's most significant

features is its portability; source code which uses its interfaces can be run on multiple different architectures with minimal concern for compatibility. Additionally, PAPI provides tools to determine the availability and compatibility of various hardware counter events supported on a particular system. One of PAPI's limitations, however, is that it can only be programmed by a user to collect information related to that users processes and their children. It does not allow user *root* to monitor globally and thus cannot be used to provide system wide monitoring.

```
ldmsctl$ load   name=spapi
ldmsctl$ config name=spapi action=init component_id=<int> set=<string>
ldmsctl$ config name=spapi action=add pid=<pid> event=<string>
                metricname=<string>
ldmsctl$ start  name=spapi interval=<int>
```

Figure 6.2: How to use PAPI sampler

Figure 6.2 describes how a user can instantiate, configure, and start the improved sampler. The API to PAPI differs from that of perf_event in two regards. The first is that it does not require a numerical event code; instead a user is able to use a string to identify the event to track. The second is that it does not allow event tracking to be limited to a specific core.

The number of events and processes which can be tracked by this sampler is only limited by the number supported by the PAPI library, which may vary based on architecture. PAPI provides two utility programs, *papi_avail* and *papi_component_avail*, that display a list of supported events for the current architecture.

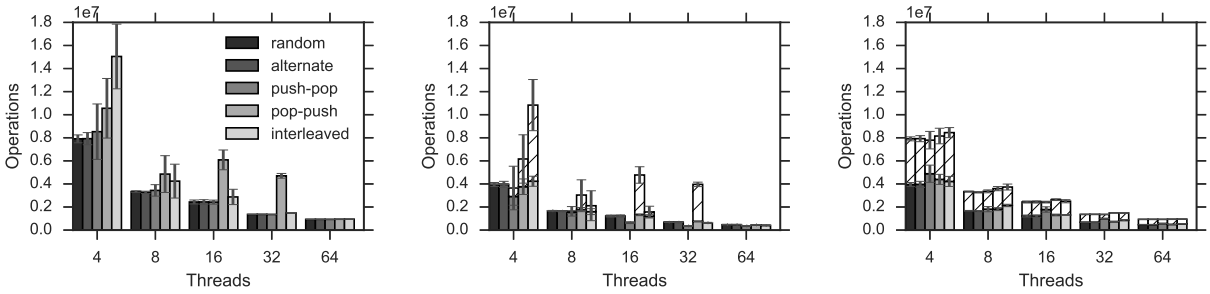
PAPI is capable of automatically monitoring all threads of a forked process, but not of an attached process, which is how the sampler uses PAPI to monitor an application. To overcome this, a user can explicitly configure the sampler to track each child process. For applications that use a large

number of threads or for applications that create and destroy threads, this is not an applicable solution.

## 6.3 Insights Gained

This section describes the analysis of the values reported by the aforementioned metrics during the testing of *Tervel*'s *Stack* and *Vector* data structures.

### *6.3.1 Similar Use-Cases*




(a) All Operations (b) Pop Operations (c) Pass Operations

Figure 6.3: Similar Use Cases of a Stack

Figures 6.3a-6.3c present the throughput of *Tervel*'s wait-free stack in several use cases designed to have similar numbers of pop and push operations completed. The design of this stack's pop and push operations appears to be computationally equivalent, and as such the cost to perform them should be similar as well. Before testing, 16384 elements were *pushed* onto the stack to decrease the likelihood of the stack from becoming empty.

The following use cases are shown in Figures 6.3a-6.3c:

- *random*: threads select with equal probability between the two operations.
- *alternate*: threads alternate between the two operations.
- *push-pop*: the first half of spawned threads perform only push and the second half perform only pop.
- *pop-push*: the first half of spawned threads perform only pop and the second half perform only push.
- *interleaved*: every other spawned thread performs only pop and the rest perform push.

Figure 6.3a depicts the total number of operations completed during testing. Use cases in which each thread executes just one type of operation perform significantly better. At $4$ threads, *interleaved* executes 90 percent more operations than *random*. However, this significant difference fades out as the level of concurrency increases, and at $64$ threads there is minimal difference between use cases.

Figure 6.3a also reveals an unexpected insight, that the order of thread creation affects throughput, even though all threads block until receiving a signal to begin the test. Except for the highest level of concurrency, *pop-push* exhibits significantly higher throughput than *push-pop*. Observing these behavioral differences in highly similar use cases motivates the need for a diverse set of metrics for the analysis of concurrent algorithms.

To understand why these very similar use cases produce different throughput results, Figure 6.3b depicts only the number of pop operations performed. In this graph the bottom portion of each bar is the number of successful operations and the top half is the number of failed operations. A failed pop operation occurs when attempting to perform a pop operation on an empty stack. As the number of threads increase, the failed pop operations decrease, and at $64$ threads there are no failed operations.

The presence of failed pop operations reveals that the time complexity of the pop operation must be less than that of the push operation, or else the stack would not be able to become empty. Interestingly, this imbalance is more pronounced in *interleaved* and *pop-push* use cases, which further strengthens my observation that thread creation order impacts algorithm throughput.

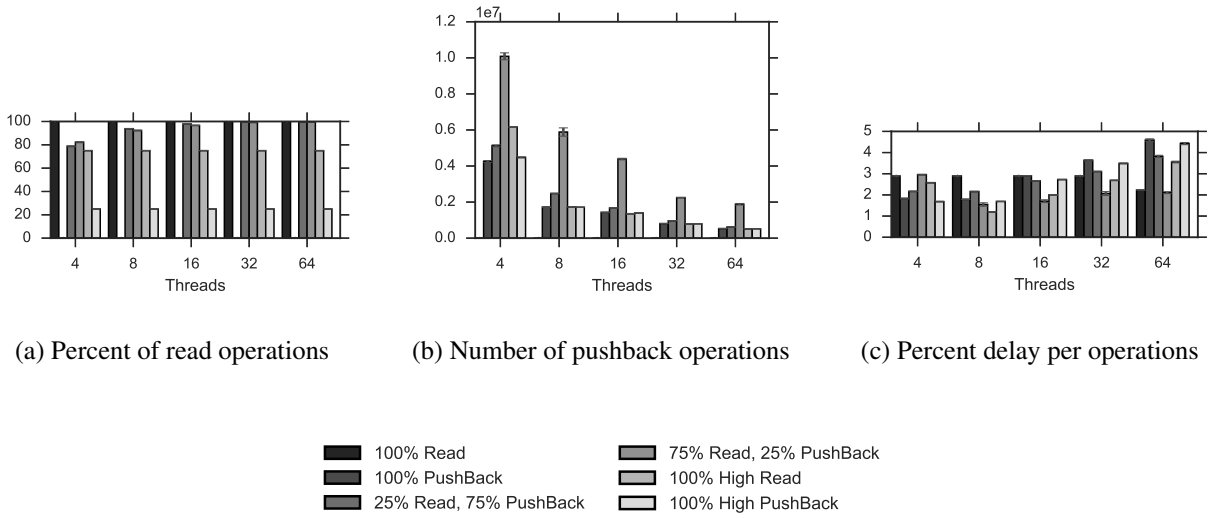
Figure 6.3c controls for failed operations by only counting successful stack operations performed. The lower half of each bar is the number of push operations and the upper half is the number of pop operations. In this figure the performance of similarly distributed use case is now much closer to each other, but at lower threads there are still clear differences. While the *random* and *alternate* use cases exhibit little throughput difference between the number of push and pop operations (less than $0.01\%$), for other use cases this difference is significantly higher. For example in the *pop-push* use case, there are $8.5\%$ more push operations than pop operations and in the *push-pop* case, there are $73\%$ more pop operations than push operations.

It is hypothesized that the pop operation takes less time to complete than the push operation; because of this, threads dedicated to performing pop are able to perform more operations than threads performing push. When threads perform both operations, this difference appears to average out.

In summary, this analysis of the stack algorithm showcases a critical fallacy that may occur when using *throughput* instead of *work completed* as a selection criteria. Even when using *work completed* to ignore operations that do not contribute work, subtle differences between the tested

### *6.3.2 Read and Write Dominated Use-Cases*

This section uses *Tervel*'s wait-free vector to gain insights into the behavior of read and write dominated use cases. Specifically, Figures 6.4a-6.5d use the following use cases:

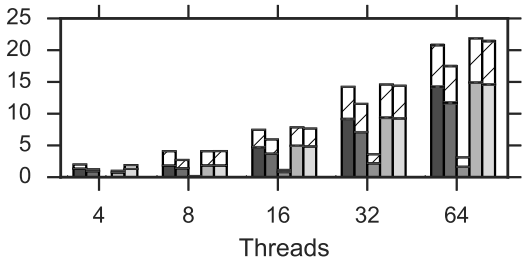



(a) Percent of read operations (b) Number of pushback operations (c) Percent delay per operations

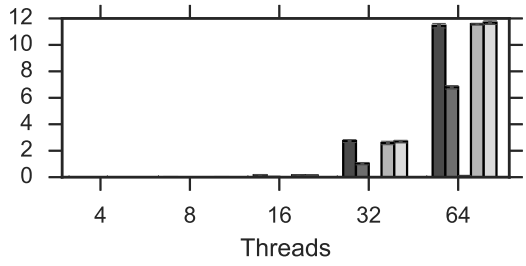
(d) Legend

Figure 6.4: Vector Metrics

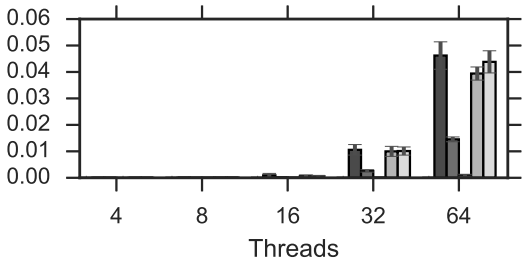
- *100% Read*: all threads perform read operations.
- *100% PushBack*: all threads perform pushback operations.
- *25% Read, 75% PushBack*: 25% of threads perform read, 75% perform pushback operations.
- *75% Read, 25% PushBack*: 75% of threads perform read, 25% perform pushback operations.
- *100% High Read*: all threads perform read with a 75% chance and pushback with a 25% chance.
- *100% High PushBack*: all threads perform read with a 25% chance and pushback with a 75% chance.



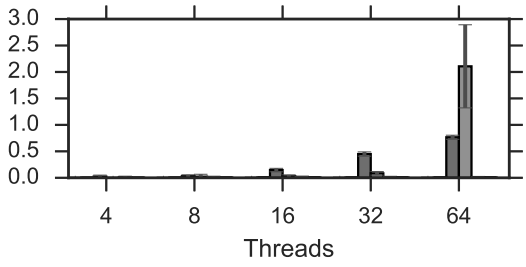

(a) Average interference per pushback




(b) Percent PushBack that made an announcement




(c) Percent of operations an announcement was helped




(d) Average threads helping an announcement


(e) Legend

Figure 6.5: Tervel Metrics

Figure 6.4a shows the percent of read operations performed during the test. One might anticipate that the read operations will account for roughly $25\%$ of operations in the *25% Read, 75% PushBack* use case; however, the percent of read operations is, on average, closer to $94\%$ of the total operations. Interestingly the *75% Read, 25% PushBack* use case also exhibits similar behavior as well, with on average $94\%$ of operations being read operations. This behavior continues to show even with a higher severity as the level of parallelism increases. When there are $32$ or more threads,

read operations account for over $99\%$ of the throughput. In the other use cases in which each thread performs the same distribution of operations (e.g. *100% High Read*), the ratio of read operations to pushback operations corresponds to the specified distribution.

It is well known that read dominated use cases perform more operations than write dominated ones, and this difference is often attributed to a higher number of read operations occurring in the system. Figure 6.4b shows the number of pushback operations completed and reveals that the *75% Read, 25% PushBack* use case consistently completes more pushback operations than all other use cases, even in cases in which all threads are responsible for performing pushback operations. This comparison reveals that not only are more read operations occurring but also more write operations.

Not only does this use case perform more pushback operations then other use cases at each degree of concurrency, but it also does so when controlling for the number of threads that perform pushback. Table 6.1 shows the change in the number of pushback operations for different use cases as the number of threads performing pushback increases. Comparing *75% Read, 25% PushBack* at $16$ threads to *100% PushBack* at $4$ threads, both having four threads dedicated to performing pushback, there is a $3\%$ increase in pushback operations. This rate further increases to $32\%$ when comparing *75% Read, 25% PushBack* at $64$ threads and *100% PushBack* at $16$ threads.

In general, the number of pushback operations decrease as the number of pushback threads increase; however, comparing $75\%$ *Read,* $25\%$ *PushBack* at $64$ threads to *100% PushBack* at $8$ threads, the number of operations increase by $9\%$. It is possible that the presence of the reader threads affect the core assignment, thread scheduling, and/or memory access pattern in such away that it overcomes the typical performance loss.

Table 6.1: Comparison of PushBack Threads

| Use Case | Total Threads | PushBack Threads | PushBack Operations |
|---|---|---|---|
| *75% Read, 25% PushBack* | 4 | 1 | 10084994 |
| *75% Read, 25% PushBack* | 8 | 2 | 5885725 |
| *25% Read, 75% PushBack* | 4 | 3 | 5118916 |
| *75% Read, 25% PushBack* | 16 | 4 | 4385139 |
| *100% PushBack* | 4 | 4 | 4251912 |
| *25% Read, 75% PushBack* | 8 | 6 | 2476079 |
| *75% Read, 25% PushBack* | 32 | 8 | 2248981 |
| *100% PushBack* | 8 | 8 | 1723122 |
| *75% Read, 25% PushBack* | 64 | 16 | 1879421 |
| *100% PushBack* | 16 | 16 | 1426616 |
| *25% Read, 75% PushBack* | 32 | 24 | 963851 |
| *100% PushBack* | 32 | 32 | 806628 |
| *25% Read, 75% PushBack* | 64 | 48 | 620496 |
| *100% PushBack* | 64 | 64 | 529983 |

### *6.3.3 Tervel Metrics*

This section uses an instrumentation of *Tervel* to provide insights into the observed behaviors of the vector. Figures 6.4c-6.5d present several metrics that reveal the interaction between threads within *Tervel*'s non-blocking algorithms.

Figure 6.5a shows, on average, the number of times the actions of a thread interferes with the progress of another thread. The bottom half of each bar represents the number of times per push-

back operation that a thread fails to acquire memory protection on an object. The top half represents the number of times per pushback operation that a thread helps complete a partially completed operation.

In this graph, the number of times a thread fails to acquire memory protection is significantly higher than the number of times a thread helps complete a partially completed operation. When a thread fails to acquire memory protection, it implies that a partially completed operation was observed, but before being able to perform a helping routine, the operation was completed. This scenario is more ideal than the opposite, which would lead to redundant helping and higher thread congestion; however, it still increases the number of retries in an operation.

The interference among threads increases as the number of threads increase, with *75% Read, 25% PushBack* increasing significantly slower than other use cases. This may explain why this use case has a higher pushback throughput compared to other use cases(Figure 6.4b).

Figure 6.5b shows the percent of pushback operations that were completed through an announcement. When the number of executing threads is less than $32$, very few threads use an announcement to complete an operation. However, when the number of threads increased to $32$ and $64$, on average $12\%$ of operations were completed through the use of an announcement. Threads executing in the *75% Read, 25% PushBack* use case rarely use an announcement to complete an operation. This enables threads to avoid performance penalties that can occur by using an announcement, and as result it exhibits more favorable throughput. An explanation for the lack of announcements can be found in Figure 6.5a, which shows that there are fewer instances of thread interference for this use case compared to others. Conversely, the *100% High PushBack* use case exhibits a high level of thread interference and a high amount of announcements.

Figure 6.5c shows the percent of operations in which a thread must first help to complete another thread's announcement before beginning its own, revealing the impact that the progress assurance

scheme has on an average operation. On average, less than $0.05\%$ of operations must first complete an announced operation. Higher performance might be gained by reducing the number of times helping routines are executed. This can be accomplished by increasing the value of a user defined constant that controls the frequency by which threads check for announcement.

Figure 6.5d shows the average number of threads that help to complete an announcement. In the majority of use cases an announcement is not helped. This happens either because of announcement checks are too infrequent or announcement are being made after too few attempts. The immediate effect is that an operation is completed without the help of another thread, but from a slower execution path. This slower execution path can increase thread interference, which often leads to performance degradation. For example in the *75% Read, 25% PushBack* use case at 64 threads, on average two threads are helping some other thread to complete an operation. This means that the majority of work performed will be redundant and/or not used for the sake of ensuring progress.

# CHAPTER 7: SUMMARY

As a researcher in the *Computer Software Engineering Scalable and Secure Systems Lab*, I led the design and implementation of numerous algorithms and containers that were the first of their kind. My research has focused on achieving high guarantees of progress and liveness within a system, while maintaining scalable performance.

My initial project, the wait-free hash map, was first presented as a poster at the 25th International Conference on Supercomputing(ICS) [FLD11b] and at the 15th annual workshop on High Performance Embedded Computing(HPEC) [FLD11a]. An improved and extended version of this work was later presented at the 13th International Conference on Embedded Computer Systems: Architectures, MOdeling, and Simulation [FLD13b], where it received the "Stamatis Vassiliadis Best Paper Award." A journal extension of this work has been accepted for publication in the International Journal of Parallel Programming(IJPP).

In my next major project, I developed methodology for performing a multi-word compare-and-swap in a wait-free and ABA-free manor. This methodology was presented at the Many-Core Architecture Research Community Symposium (MARCS) held in conjunction with ACM's SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity(Splash) 2013 [FLD13a], and an extended version of this work appeared in IJPP [FLD15a]. In this work, I developed the association model that would later become the building block upon which significantly more complex algorithms were developed.

The wait-free vector, published in IEEE Transactions on Parallel and Distributed Systems [FVD15], and the wait-free ring buffer, presented at ACM SAC [BFD15], are two examples of complex algorithms in which I expanded upon the principals, techniques, and methodologies developed by others and myself in earlier work.

My research has culminated in the release of the Tervel library and framework. To the best of my knowledge, there is currently no other library devoted to providing wait-free implementations of algorithms and containers. In this framework, I brought together the numerous techniques, methodologies, and programming models that I have developed and/or used. This unification enables developers to efficiently implement wait-free algorithms in a straight-forward manner. The framework part of Tervel includes inter-thread helping techniques, such as bitmarking, the association model, and descriptor objects, as well as memory management and progress assurance schemes. The library part of Tervel provides common containers such as hash map, vector, queues, stacks, and linked lists. The Tervel library has been presented to engineers at both SRI International and Lockheed Martin.

In the evaluation of Tervel, I oversaw the development of a new set of software metrics designed specifically for non-blocking algorithms. These metrics are capable of measuring the amount of interference that is caused by threads sharing a resource. Additionally, by using these metrics, I was able to identify unexpected behavior that indicated implementation error. These implementation errors did not introduce incorrect behavior, but caused higher level thread congestion and operation retries.

## 7.1 Future Work

I spent the summer of 2014 interning at Lawrence Livermore National Laboratory(LLNL) where I was tasked with solving challenges related to the scalability and performance of distributed graph algorithms. My primary task was the reimplementation of their delegate graph partitioning algorithm to increase the maximum supported graph scale. Their original implementation was limited to graphs of scale $2^{30}$ or less, my final implementation was capable of constructing graphs as large as scale $2^{40}$, a substantial improvement. Using my implementation, LLNL's Catalyst cluster tied

for second, by problem scale, on the graph500's November 2014 benchmark report, and as of the July 2015 report, it is still tied.

My time at LLNL showed how narrow my field of study has been, and though I have made significant contribution in my area, I recognize that I need to expand my experience and knowledge. To gain this knowledge and experience, I am pursuing a role within *Google's Site Reliability Engineering's Bandwidth Authority* team where I will be exposed to a variety of hardware architectures, parallel systems, and programming paradigms. I plan to use my time there to identify new ways in which I can make contributions in the realm of parallel computing.

## 7.2 Concluding Remarks

I would like to thank once again my friends, family, colleagues, and the University of Central Florida for supporting me while I pursued my research. I look forward to identifying and exploring new areas of research interest both within the parallel paradigm and outside of it. After gaining industry experience and a first hand understanding of the needs and challenges faced within the community, I plan to resume academic research orientated to the betterment of it all. At the same time I plan to devote a portion of my time towards maintaining and enhancing my existing work.

# LIST OF REFERENCES


[AA14] Maya Arbel and Hagit Attiya. "Concurrent updates with RCU: Search tree as an example." In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pp. 196–205. ACM, 2014.

[AAB14] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp. 154–165, Piscataway, NJ, USA, 2014. IEEE Press.

[AH11] Hagit Attiya and Eshcar Hillel. "Highly concurrent multi-word synchronization." *Theor. Comput. Sci.*, **412**(12-14):1243–1262, March 2011.

[AKY10] Yehuda Afek, Guy Korland, and Eitan Yanovsky. "Quasi-Linearizability: Relaxed Consistency for Improved Concurrency." In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *OPODIS*, volume 6490 of *Lecture Notes in Computer Science*, pp. 395–410. Springer, 2010.

[ami10] "Concurrent Building Block.", March 2010.

[ARJ97] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. "Real-time computing with lock-free shared objects." *ACM Trans. Comput. Syst.*, **15**(2):134–165, May 1997.

[Bar93] Greg Barnes. "A method for implementing lock-free shared-data structures." In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, SPAA '93, pp. 261–270, New York, NY, USA, 1993. ACM.

[BDG00] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. "A portable programming interface for performance evaluation on modern processors." *International Journal of High Performance Computing Applications*, **14**(3):189–204, 2000.

[BDG08] Jim M Brandt, Bert J Debusschere, Ann C Gentile, Jackson R Mayo, Philippe P Pébay, David Thompson, and Matthew H Wong. "OVIS-2: A robust distributed architecture for scalable RAS." In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8. IEEE, 2008.

[BFD15] Andrew Barrington, Steven Feldman, and Damian Dechev. "A Scalable Multi-producer Multi-consumer Wait-free Ring Buffer." In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pp. 1321–1328, New York, NY, USA, 2015. ACM.

[Boo] Boost C++ Libraries. http://www.boost.org/ , Retrieved 02/02/2014.

[CCF13] J. Chen, A. Choudhary, S. Feldman, B. Hendrickson, C.R. Johnson, R. Mount, V. Sarkar, V. White, and D. Williams. "Synergistic Challenges in Data-Intensive Science and Exascale Computing.", March 2013.

[Cli] Cliff Click. A lock-free hash table (`http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf`). Retrieved 12/12/2012.

[DMM01] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. "Lock-free reference counting." In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, PODC '01, pp. 190–199, New York, NY, USA, 2001. ACM.

[DPS06] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. "Lock-free dynamically resizable arrays." In *Proceedings of the 10th international conference on Principles of Distributed Systems*, OPODIS'06, pp. 142–156, Berlin, Heidelberg, 2006. Springer-Verlag.

[FBL13] Steven D. Feldman, Akshatha Bhat, Pierre LaBorde, Qing Yi, and Damain Dechev. "Effective Use of Non-blocking Data Structures in a Deduplication Application." In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pp. 133–142, New York, NY, USA, 2013. ACM.

[FH07] Keir Fraser and Tim Harris. "Concurrent programming without locks." *ACM Trans. Comput. Syst.*, **25**(2):5, 2007.

[FKD12] Steven Feldman, Emile Kanhai, and Damian Dechev. "Lock-Free Concurrent Hash Tables." *Science Magazine, Special Feature: International Science & Engineering Visualization Challenge 2011*, February 2012.

[FLD11a] Steven Feldman, Pierre LaBorde, and Damian Dechev. "A Lock-Free Concurrent Hash Table Design for Effective Information Storage and Retrieval on Large Data Sets." In *Proceedings of the 15th Annual High Performance Computing Workshop (HPEC 2011)*, 2011.

[FLD11b] Steven Feldman, Pierre LaBorde, and Damian Dechev. "Facilitating Efficient Parallelization of Information Storage and Retrieval on Large Data Sets." In *Proceedings of the 25th ACM International Conference on Supercomputing*, 2011.

[FLD13a] Steven Feldman, Pierre LaBorde, and Damian Dechev. "A Practical Wait-Free Multi-Word Compare-and-Swap Operation." In *Many-Core Architecture Research Community (MARC) Symposium at SPLASH 2013*, 2013.

[FLD13b] Steven Feldman, Pierre LaBorde, and Damian Dechev. "Concurrent Multi-level Arrays: Wait-free Extensible Hash Maps." In *Proceedings of the 13th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, Samos, Greece, July 2013.

[FLD15a] Steven Feldman, Pierre Laborde, and Damian Dechev. "A Wait-Free Multi-Word Compare-and-Swap Operation." *Int. J. Parallel Program.*, **43**(4):572–596, August 2015.

[FLD15b] Steven Feldman, Pierre LaBorde, and Damian Dechev. "Tervel: A Unification of Descriptor-based Techniques for Non-blocking Programming." In *Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV)*, Samos, Greece, July 2015.

[FVD15] S. Feldman, C. Valera-Leon, and D. Dechev. "An Efficient Wait-Free Vector." *Parallel and Distributed Systems, IEEE Transactions on*, **PP**(99):1–1, 2015.

[GGH04] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. "Almost Wait-Free Resizable Hashtable." In *IPDPS*, 2004.

[Gra15] Vincent Gramoli. "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms." In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–10. ACM, 2015.

[Her91] Maurice Herlihy. “Wait-Free Synchronization.” In *Trans. on Programming Languages and Systems*, pp. 124–149. ACM, 1991.

[Her08] Maurice Herlihy. *The art of multiprocessor programming*. Elsevier/Morgan Kaufmann, Amsterdam London, 2008.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. “A Practical Multi-word Compare-and-Swap Operation.” In *Proceedings of the 16th International Conference on Distributed Computing*, DISC ’02, pp. 265–279, London, UK, UK, 2002. Springer-Verlag.

[HSY10] Danny Hendler, Nir Shavit, and Lena Yerushalmi. “A scalable lock-free stack algorithm.” *J. Parallel Distrib. Comput.*, **70**:1–12, January 2010.

[Int] Intel Corporation. Reference for Intel Threading Building Blocks (http://threadingbuildingblocks.org/). Retrieved 02/02/2014.

[IR94] Amos Israeli and Lihu Rappoport. “Disjoint-access-parallel implementations of strong shared memory primitives.” In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC ’94, pp. 151–160, New York, NY, 1994. ACM.

[ISO11] ISO/IEC 14882 Standard for Programming Language C++. *Programming languages: C++*. American National Standards Institute, September 2011.

[KP12] Alex Kogan and Erez Petrank. “A methodology for creating fast wait-free data structures.” *SIGPLAN Not.*, **47**(8):141–150, February 2012.

[Kri13] Alexander Krizhanovsky. “Lock-free multi-producer multi-consumer queue on ring buffer.” *Linux Journal*, **2013**(228):4, 2013.

[lib14] “Concurrent Data Structures (libcds).”, September 2014.

[Low02] MichaelR. Lowry. "Software Construction and Analysis Tools for Future Space Missions." In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pp. 1–19. Springer Berlin Heidelberg, 2002.

[LS12] Yujie Liu and Michael Spear. "A lock-free, array-based priority queue." *SIGPLAN Not.*, **47**(8):323–324, February 2012.

[Mic02] Maged M. Michael. "High performance dynamic lock-free hash tables and list-based sets." In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 73–82, New York, NY, USA, 2002. ACM Press.

[Mic03] Maged Michael. "CAS-Based Lock-Free Algorithm for Shared Deques." In *Euro-Par 2003: The Ninth Euro-Par Conference on Parallel Processing, LNCS volume 2790*, pp. 651–660, 2003.

[Mic04] Maged M. Michael. "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects." *IEEE Trans. Parallel Distrib. Syst.*, **15**:491–504, June 2004.

[Moi97] Mark Moir. "Transparent Support for Wait-Free Transactions." In *Proceedings of the 11th International Workshop on Distributed Algorithms*, WDAG '97, pp. 305–319, London, UK, UK, 1997. Springer-Verlag.

[MS96] Maged M Michael and Michael L Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms." In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pp. 267–275. ACM, 1996.

[PH05] Chris Purcell and Tim Harris. "Non-blocking hashtables with open addressing." In *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, pp. 108–121, Berlin, Heidelberg, 2005. Springer-Verlag.

[SAH06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. "McRT-STM: a high performance software transactional memory system for a multi-core runtime." In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pp. 187–197, New York, NY, USA, 2006. ACM.

[SDM11] John Shalf, Sudip Dosanjh, and John Morrison. "Exascale computing technology challenges." In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, pp. 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[SS03] Ori Shalev and Nir Shavit. "Split-ordered lists: lock-free extensible hash tables." In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pp. 102–111, New York, NY, USA, 2003. ACM Press.

[ST08] Håkan Sundell and Philippas Tsigas. "Lock-free deques and doubly linked lists." *J. Parallel Distrib. Comput.*, **68**:1008–1020, July 2008.

[Sun11] Håkan Sundell. "Wait-Free Multi-Word Compare-and-Swap Using Greedy Helping and Grabbing." *International Journal of Parallel Programming*, **39**:694–716, 2011. 10.1007/s10766-011-0167-4.

[TBF11] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. "The STAPL parallel container framework." In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pp. 235–246, New York, NY, USA, 2011. ACM.

[TBK12] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. "Wait-free linked-lists." *SIGPLAN Not.*, **47**(8):309–310, February 2012.

[TZ01] Philippas Tsigas and Yi Zhang. "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems." In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pp. 134–143. ACM, 2001.

[Wea13] Vincent M Weaver. "Linux perf_event features and overhead." In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, p. 80, 2013.