Validating Performance and Simplicity of Highly Concurrent Data Structures Utilitizing the ATAC Broadcast Mechanism

by

Nicholas A. Pellegrino

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 23, 2013

Certified by .....
Armando Solar-Lezama
Associate Professor
Thesis Supervisor

Accepted by .....
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Validating Performance and Simplicity of Highly Concurrent Data Structures Utilitizing the ATAC Broadcast Mechanism

by

Nicholas A. Pellegrino

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

I evaluate the ATAC broadcast mechanism as the foundation for a new paradigm in the design of highly scalable concurrent data structures. Shared memory communication is replaced, alleviating the contention that prevents data structures from achieving high performance on the next generation of manycore computers. The alternative model utilizes thread local memory and relies on the ATAC broadcast for inter-core communication, thus avoiding the complicated protocols that contemporary data structures use to mitigate contention. I explain the design of the ATAC barrier and run benchmarking to validate its high performance relative to existing barriers. I explore several concurrent hash map designs built using the ATAC paradigm and evaluate their performance, explaining the memory access patterns under which they achieve scalability.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor

Acknowledgments

A big thanks to my adviser Armando Solar-Lezama for his guidance throughout this project. Thanks to Jim Psota for his tireless mentoring, and to teammates Seo Jin Park and Jeremy Sharpe.

This project wouldn't have been possible without insightful design advice from Nir Shavit and Graphite tutoring by George Kurian, Tony Giovinazzo, and George Bezerra.

Thank you to my family, for all your love and support over the years.

And last but not least, a big shout-out to everyone who has made this past year so incredibly awesome, especially: Boki, Speedy, Pinto, my Good Friend Karine, APenn, my third floor buddies, the ACs, Jingyun, Luquinhas, Mr. G, Bujoon, and C.J.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Background and Motivation

Speedup in individual processors used to be a reliable means to increasing computational performance. When the growth in clock cycle frequency stalled due to physical limitations of the hardware, parallel computing became the new standard for increasing instruction throughput. Therefore the performance of contemporary software relies on effectively leveraging multiprocessor hardware.

The use of multiple processors has required a rethinking of the hardware models used to access memory. Bus architectures scale poorly with the number of cores due to contention on the bus, which motivates non-uniform memory access (NUMA) architectures. NUMA systems allow a core to access a shard of local memory quickly at the expense of adding latency to remote memory access. NUMA allows a programmer to write high performance software at the cost of having to reason about the underlying hardware, resulting in an undesirable crossing of abstraction layers.

To seek the performance benefits of multiprocessor computing while writing simple, hardware-oblivious software, the All-to-All Computing (ATAC) system is introduced. ATAC uses an optical broadcast network to deliver a message to all cores with uniform latency, and scales up to 64 (and possibly more) cores. ATAC allows simultaneous broadcasting on the channel through wavelength-division multiplexing. ATAC is designed to reduce bottlenecks caused by sharing data between cores, thereby en-

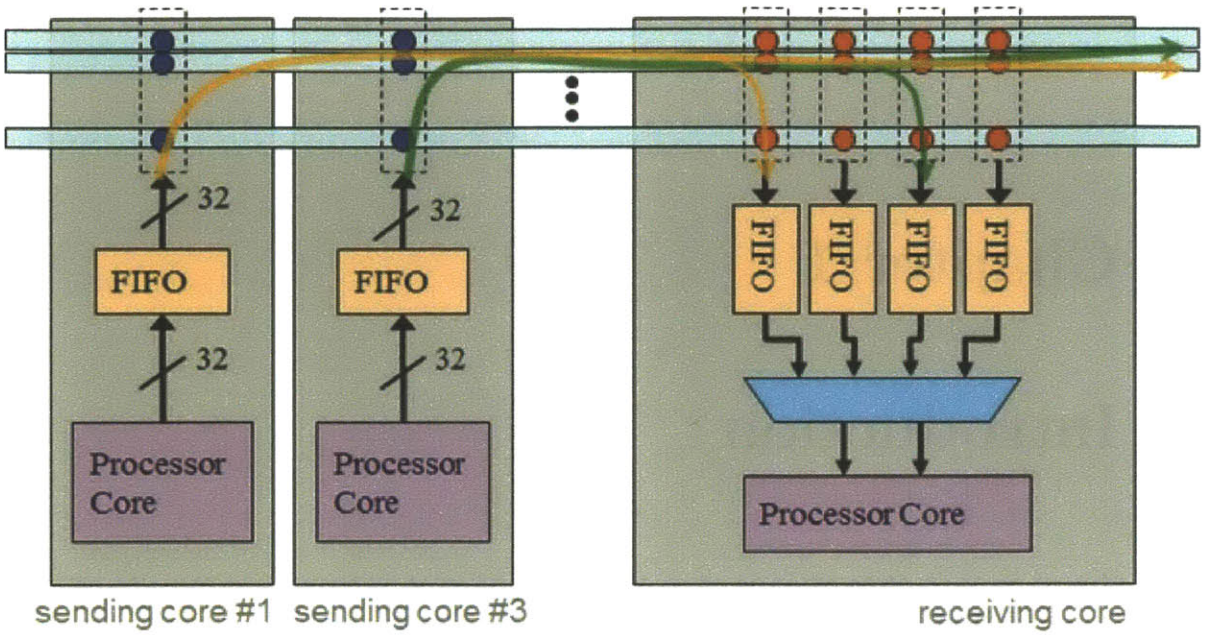

Figure 1-1: Each core broadcasts on a different wavelength to avoid contention. The receiving core processes the incoming packets simultaneously, and enqueues messages in first-in, first-out buffers before being handled by software [13].

abling the design of simple, high-performance software independent of the assignment of threads to cores [13]. Programs using ATAC are run on the Graphite simulator to obtain performance results [11].

1.2 Previous Work

ATAC has been used to implement an efficient cache coherence protocol named ACKwise [8]. ACKwise is intended for general purpose applications.

My colleague Seo Jin Park implemented a barrier using ATAC. The barrier interface is analogous to existing barriers, and is used without requiring any knowledge of the underlying implementation. Furthermore, the implementation is simple, in contrast to other high performance barriers that have complex implementations. Prior to this work, the performance characteristics of this new barrier had not been evaluated empirically [12].

1.3 Thesis Scope

This thesis will demonstrate the performance and programmability benefits enabled by ATAC, by evaluating existing and new data structures. Chapter 2 describes the design of VCBarrier, which relies on ATAC to achieve a simple implementation. Chapter 3 describes benchmarks that test the performance of VCBarrier and analyzes the results to demonstrate its high performance relative to other existing barrier implementations. Chapter 4 details the exploration of the design space of hash tables that achieve consistency using the broadcast mechanism. Chapter 5 evaluates the performance of these hash tables. Chapter 6 describes related work, and Chapter 7 provides concluding remarks and avenues for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Barrier Design

2.1 The Barrier Problem

The purpose of a barrier is to divide a program into sequential stages, while adding minimal performance overhead. Implementing a barrier requires providing a mechanism to notify all threads once the last thread completes the stage of execution.

The longer it takes for this notification to propagate to a thread, the longer the thread will be unnecessarily stalled. For example, a sense-reversing barrier relies on a shared counter to determine once all threads have reached the barrier. Contention on the shared counter means that this protocol, which is simple and practical for small numbers of cores, scales poorly [6].

Furthermore, barriers that busy-wait on shared memory are vulnerable to non-uniform exit latencies when run on NUMA systems. Consider applications that involve repeated sequential stages. The time for one stage to complete is lower bounded by the time it takes the slowest thread to complete this stage. If the application programmer manages to balance a stage's work evenly across threads, then one thread experiencing a higher exit latency than all other threads will cause the entire stage to take additional time to complete. The more sequential stages in the application, the greater the performance impact. Existing implementations for a high-performance barrier on a large number of cores require complex protocols (see Section 6.2).

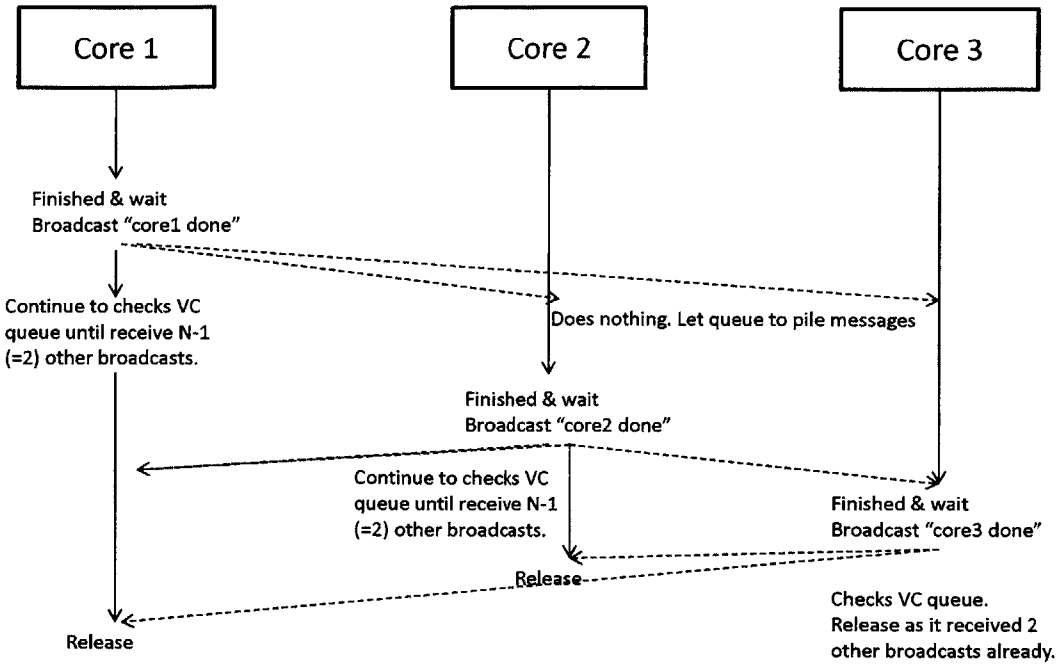

Figure 2-1: VCBarrier protocol [12].

2.2 The VCBarrier Design

The Virtual Channel Barrier (VCBarrier for short) is implemented using the Virtual Channel interface, a layer of abstraction built on top of the ATAC broadcasting system [12]. VCBarrier is designed to achieve high performance relative to existing barriers through a simple implementation.

VCBarrier works as follows on a N-threaded program. When a thread reaches a barrier, it broadcasts out to all threads a message indicating that the barrier has been reached. The thread then checks its message queue for messages from other threads, and busy-waits until it has received $N - 1$ messages (see Figure 2-1).

2.2.1 Performance Considerations

VCBarrier leverages broadcast to avoid waiting on shared memory. VCBarrier simply keeps a tally of messages received, requiring only a counter in local memory. All coordination between threads occurs through the broadcast network, which avoids scalability problems without resorting to complex protocols (see Section 6.2). Since

the broadcast takes uniform time to propagate, we expect uniform exit latencies.

2.2.2 Simple Implementation

VCBarrier is implemented with a straightforward inter-thread communication protocol, which is much easier to reason about than more complex high performance barriers.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Barrier Performance Evaluation

3.1 Hypothesis

By relying on local memory and the fast ATAC broadcast network, I expect VCBarrier to exhibit lower exit latencies on multicore systems than barriers that rely on shared memory. By employing a uniform latency messaging system, I expect VCBarrier to exhibit uniform exit latencies across threads. As a consequence of the above two conjectures, I expect the end-to-end runtime of benchmarks in which the barrier is a major source of overhead to improve when conventional barriers are replaced by VCBarrier.

3.2 Benchmark Methodology

Here I provide a justification for the approaches used to measure and compare the performance of VCBarrier against existing barrier implementations.

3.2.1 Conventional Barriers for Comparison

I run the benchmarks using VCBarrier as well as the standard Linux Pthreads barrier and the sense-reversing barrier. The Pthreads barrier belongs to the POSIX standard. It is a commonly used barrier, but because it switches to the operating system, its

invocation incurs the cost of context switching. Therefore, the benchmarks not only measure the cost of the barrier implementation, but also operating system effects. Since VCBarrier runs in user space, it avoids context switching. Therefore, we also compare VCBarrier to a common user level barrier, but choose to include the results of the Pthreads barrier to give as a reference point a pervasively used barrier.

We include an implementation of the sense-reversing barrier as a example of a user level barrier. The sense-reversing barrier is bottlenecked by contention on a counter in shared memory, so we can test whether the paradigm of using the ATAC broadcast with local memory can outperform a shared memory barrier of similar implementation complexity. Although sophisticated protocols exist which rely on shared counters, employing similar principles as the sense-reversing barrier while suffering from less contention, these protocols are much more complicated to implement (see Section 6.2). We seek to show that we can provide performance and simple programmability in implementing concurrent data structures that scale to large numbers of cores by employing the ATAC programming paradigm.

3.2.2 Data Collection Framework

I utilize Graphite's CAPI library to take timing measurements within a custom framework for data collection. I call the CarbonGetTime method, which returns a timestamp of the thread specific clock time. In Graphite, threads are represented by separate processes in the simulator, which are each responsible for separately maintaining a representation of the clock cycle count. While the timestamp is not globally cycle accurate, threads synchronize their clocks whenever threads interact, thus maintaining the cycle count within a level of accuracy adequate for measuring barrier exit latencies [11].

In order to gather data on exit latencies, I implemented wrapper classes for each barrier. The wrapper classes all inherit from a common base class, which records the timestamp as each thread enters and leaves the barrier. The subclasses are responsible for supplying an implementation of the method that calls the actual barrier being benchmarked. After the benchmark completes running, I write the entry and exit

timestamps for each thread to a file. A separate Python script post-processes the data to calculate the exit latency of each thread by taking that thread's exit timestamp and subtracting the last entry time of the threads at the corresponding barrier stage.

In order to calculate the end-to-end runtimes of each benchmark, I simply record the timestamp at the beginning and end of each benchmark and take the difference. I measure exit latencies and end-to-end runtimes in separate executions of each benchmark, so that overhead from recording barrier entry and exit times does not contribute to the end-to-end runtime statistics. From a software engineering perspective, I use the same wrapper classes to invoke the barriers but simply pass a boolean at barrier initialization which tells the wrapper class whether or not barrier entry and exit timestamps should be recorded.

3.3 Synthetic Benchmark - Barrier in a Loop

3.3.1 Explanation

This benchmark simply consists of a loop that contains a wait call to the barrier, iterated over M times. It is designed to test the performance of the barrier in isolation.

3.3.2 Results

Running this benchmark demonstrates that VCBarrier achieves a lower exit latency than the Pthreads barrier. VCBarrier especially outperforms the sense-reversing barriers as the number of cores scales (see Figure 3-1).

End-to-end performance demonstrates that the faster VCBarrier speeds up total program runtime (see Figure 3-2).

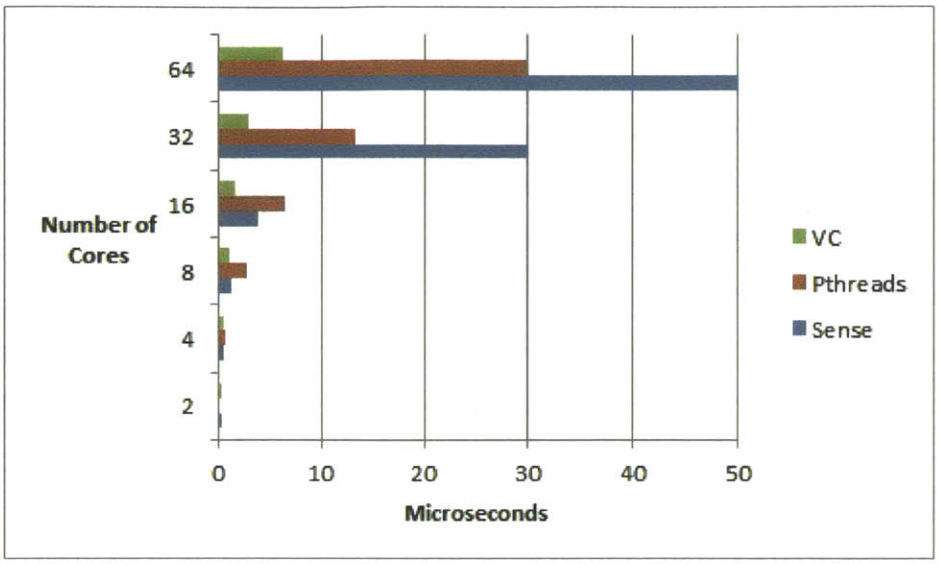

Figure 3-1: Median exit latency for benchmark Barrier in a Loop, $M = 1000$. Sense-reversing with 64 cores takes 154 microseconds.

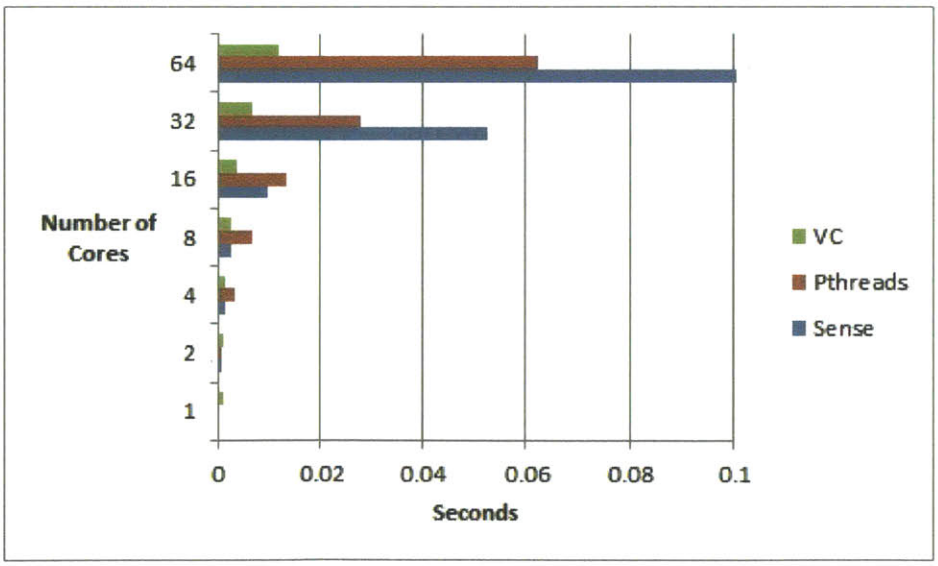

Figure 3-2: End-to-end runtime for benchmark Barrier in a Loop, $M = 1000$. Sense-reversing with 64 cores takes 0.252 seconds.

3.4 Application Benchmark - PARSEC Streamcluster

3.4.1 Explanation

The Princeton Application Repository for Shared-Memory Computers (PARSEC) consists of a series of benchmarks designed to evaluate the performance capabilities of multiprocessor architectures. We consider the Streamcluster benchmark, an online algorithm for clustering a stream of input points that suffers from high barrier overhead, and run the algorithm with the three aforementioned barrier types [3]. Streamcluster is run with a range of input sizes, to identify how end-to-end performance of the algorithm with different barrier implementations scales in both input size and number of cores. Larger input sizes than those evaluated exist, but take much longer to simulate (the largest presented here takes days to run on all barrier and core combinations) and are therefore omitted due to the scalability constraints of the simulator.

3.4.2 Results

The exit latency of VCBarrier scales better than the Pthreads and sense-reversing barrier when run on this real-world benchmark, just as it did in the synthetic benchmark (see Figure 3-3).

The end-to-end performance of Streamcluster scales better for higher numbers of cores, regardless of input size. However, the difference is more pronounced for smaller inputs (see Figure 3-4). This can be explained because there is more work to be done for larger inputs, so the barrier overhead takes a smaller fraction of the total program runtime, and by Amdahl's law, speeding up the barrier will have less of an impact on total program performance.

Notably, increasing the number of cores reduces end-to-end runtime for all input sizes when VCBarrier is used. On the other hand, both the Pthreads and sense-reversing barriers exhibit worse performance once the number of cores crosses a certain

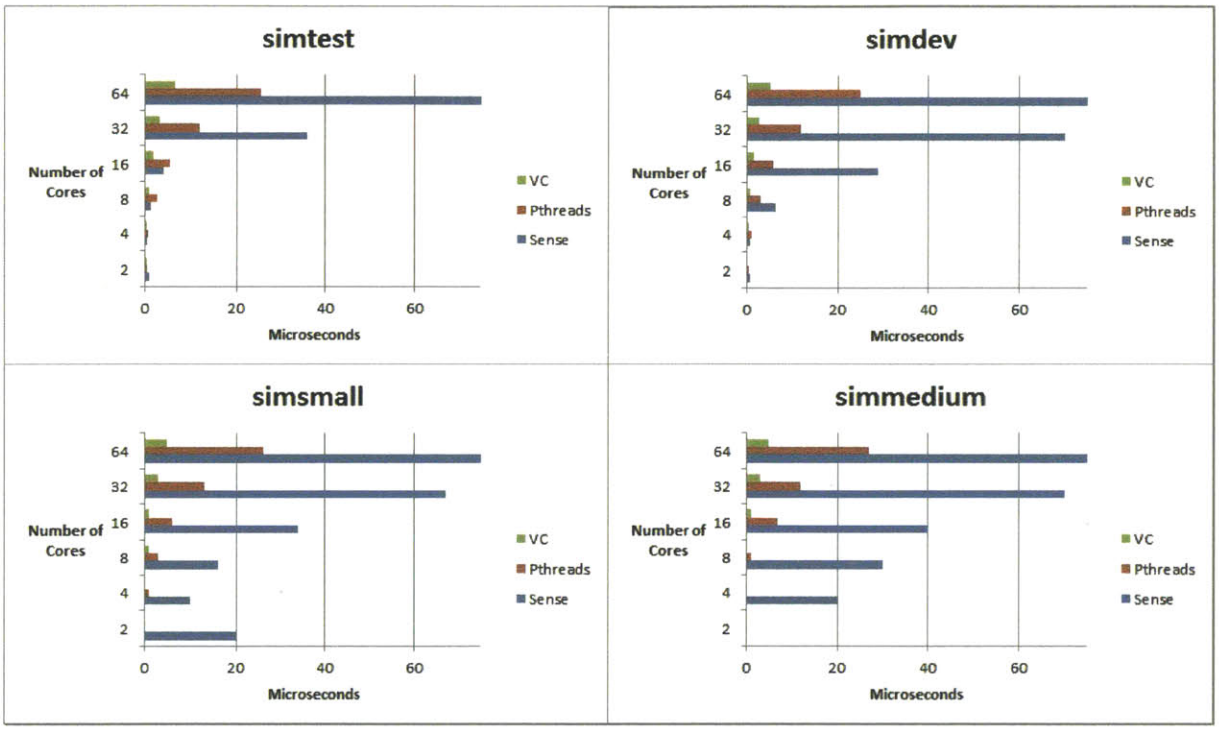

Figure 3-3: Median exit latency for benchmark Streamcluster, on four input sizes: simtest, simdev, simsmall, and simmedium (in increasing order). Sense-reversing with 64 cores on simtest has value 175 microseconds, on simdev has value 158 microseconds, on simsmall has value 140 microseconds, and on simmedium has value 150 microseconds.

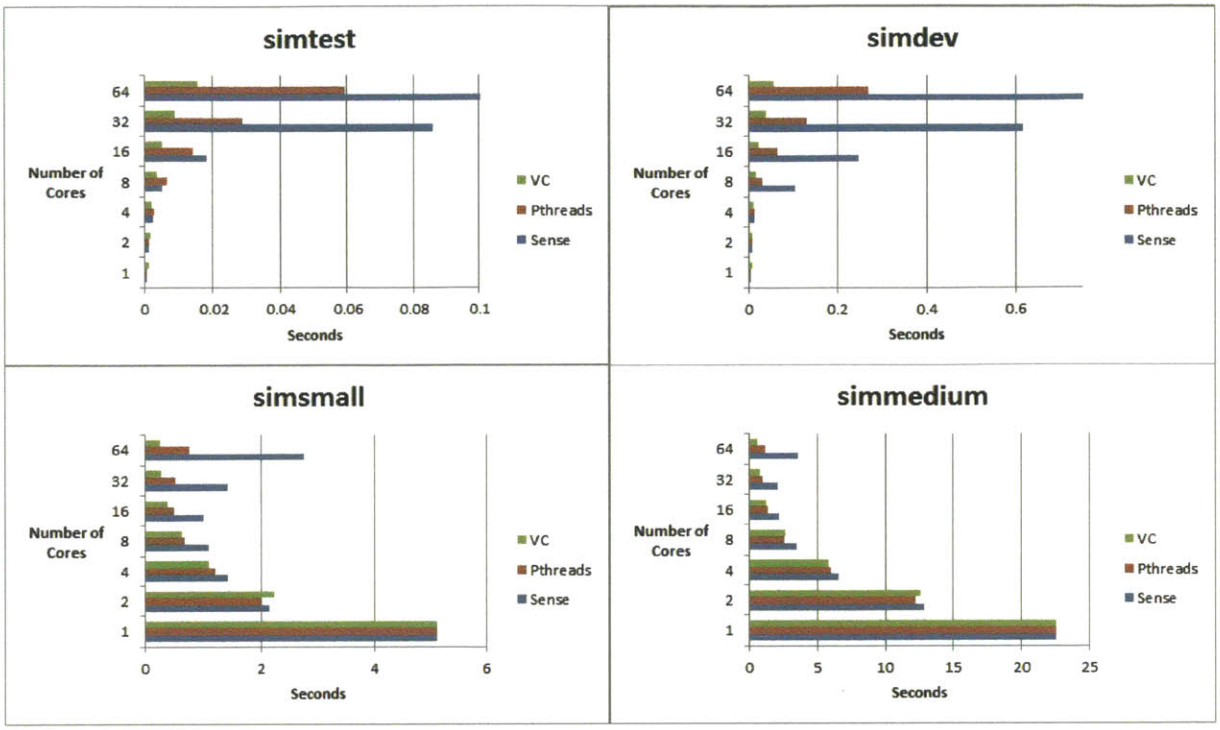

Figure 3-4: End-to-end runtime for benchmark Streamcluster, on four different input sizes. Sense-reversing with 64 cores on simtest takes 0.31 seconds and sense-reversing with 64 cores on simdev takes 1.4 seconds.

threshold. This provides real-world evidence that VCBarrier allows the benefits of parallelism to extend to a higher number of cores than possible with the Pthreads and sense-reversing barriers.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Hash Map Design

4.1 Introduction to Concurrent Hash Maps

A hash map supports amortized constant time insertion and lookup of key-value pairs. A concurrent hash map is accessible by multiple threads while providing specified correctness guarantees. Correctness guarantees are divided into two parts: safety (an invalid execution history never happens) and liveness (the system makes progress). A rule of thumb is that the average application performs 1 insertion for every 10 lookups, so a high performance hash map must support fast lookups [6].

The design of concurrent hash maps benefits from disjoint-access-parallelism, the property that most accesses to the hash map occur in isolation and therefore enable a high degree of parallelization [7]. This potential performance benefit is counterbalanced by the complication that resizing a hash map requires either coarse grained locking, and hence serialization, or complicated protocols to handle resizing (see Section 6.3).

Implementing concurrent versions of serial data structures requires rethinking definitions of correctness. Since method calls on the object can no longer be reasoned about serially, method definitions must take into account the interleaving of instructions [16]. Whereas the barrier has a simple usage model (calls to the barrier block until all threads have invoked the barrier) a concurrent hash map needs to support much more complex combinations of interleavings, where multiple insert and lookup

operations may happen concurrently. Fortunately, as data structure designers it is only necessary to provide correctness guarantees as strong as the application requires. Therefore we may tradeoff correctness guarantees for performance benefits.

4.2 High-Level Approach

We seek to explore the design space of distributed hash maps that leverage ATAC. We target applications that exhibit thread locality and focus on optimizing lookups. Therefore we design hash maps that operate only on thread local memory. Each thread has in local memory an instantiation of the standard C++ library hash map [2], and all inter-core communication occurs via ATAC broadcast.

We relax consistency guarantees in order to improve performance. We try to limit use of broadcast wherever possible so threads can perform method calls indepedently. Insertion only occurs on a local hash map. Lookup first accesses the local hash map, returning the key-value entry if it is found locally, and only communicates with other threads if the key-value entry is not found locally. This reduces instances of latency introduced by round-trip broadcasting, and furthermore, optimizes lookups for applications when access patterns exhibit thread locality. Resizing only takes place on the local hash maps, so resizing only introduces serialization if the core on which resizing takes place receives a concurrent lookup request.

4.3 Hash Map Protocols

I implement a series of concurrent distributed hash maps to explore the properties of the design space made feasible by ATAC. There are two dimensions upon which all hash map implementations vary: the get-request protocol used to lookup a key-value pair remotely when it is not found locally and the replication protocol used to insert remote key-value pairs into the local hash map. Each dimension contains 3 protocols, so all combinations of protocols generate 9 hash map implementations.

		Get-Request Protocol		
		AllAck	**NoAck**	**TimeoutNoAck**
Replication Protocol	**NoReplication**	NoReplication-MapAllAck	NoReplication-MapNoAck	NoReplication-MapTimeout-NoAck
	LazyReplication	LazyReplication-MapAllAck	LazyReplication-MapNoAck	LazyReplication-MapTimeout-NoAck
	LazyBroadcastReplication	LazyBroadcast-ReplicationMap-AllAck	LazyBroadcast-ReplicationMap-NoAck	LazyBroadcast-ReplicationMap-TimeoutNoAck

Table 4.1: Listing of all hash map implementations

4.3.1 Get-Request Protocols

AllAck is the basic get-request protocol; NoAck and TimeoutNoAck are successive optimizations.

AllAck

AllAck is the simplest of the get-request protocols. A thread performing a lookup executes the following steps:

1. Executing thread checks the local hash map for the key-value pair.
2. If found, returns the requested key-value pair. Else,
3. Broadcasts for the key.
4. Upon receiving request, other cores enter a callback routine: check their local hash map for the key-value pair, and reply with a point to point ATAC message to the requesting core.
5. Requesting thread receives all replies. If one or more replies contain a found value, the requesting thread returns the first value that it received; else it returns that no key-value pair was found.

AllAck has the property that if duplicate keys are never inserted into the hash map, then the hash map is linearizable. If duplicate values for the same key are

inserted into the map, then priority will be given to the key-value pair existing in the local hash map.

NoAck

NoAck is a slightly more complex optimization of AllAck. It follows essentially the same protocol, except when a remote lookup is performed, it will stop waiting to receive subsequent replies after receiving the first positive reply, as all subsequently processed replies are inconsequential to the return value of the lookup. NoAck provides the same correctness guarantees as AllAck.

Implementing NoAck requires adding a timestamp to each core's request so the requesting thread can distinguish between replies that are relevant to the current remote lookup and old reply messages that may have been delayed due to a core responding late. Checking that messages contain the correct timestamp and updating the thread's timestamp are expected to introduce extra overhead.

By processing fewer packets before returning, we expect a performance benefit; on the other hand, the timestamp overhead adds extra steps to perform. The performance of NoAck is evaluated empirically in Section 5.4.2.

TimeoutNoAck

TimeoutNoAck is an extension to the NoAck protocol. It contains the extra step of saving the clock cycle when a get-request is sent, and prematurely terminating the process of waiting for additional reply messages once a specified time has passed. If the time threshold is high enough on a non-congested system, the protocol will function the same as NoAck; otherwise it aborts slow get-requests.

While TimeoutNoAck provides no guarantees that remote entries will be found, it will generally produce the same resuls as NoAck if the time threshold is tuned for the underlying system. For example, increasing the number of cores will increase the total number of messages that a requesting core receives, and therefore will require raising the threshold so all messages are processed in the typical case.

4.3.2 Data Replication Protocols

Three different data replication protocols are introduced. Replication may enable more lookups to hit the thread local hash map. The choice of replication protocol does not affect the correctness guarantees of the hash map, but does affect performance by changing the number of packets sent and processed and by changing the frequency of local lookup hits. Whether the cumulative performance effect yields speedup or slowdown is expected to depend on the memory access patterns of the particular application.

NoReplication

The simplest data replication protocol, NoReplication, does not replicate any data. Therefore, every time a thread performs a lookup on a value that was not inserted locally, it will perform a remote get-request according to the protocol being used.

LazyReplication

LazyReplication inserts a key-value pair locally if a reply returns a positive result.

LazyBroadcastReplication

LazyBroadcastReplication functions similiarly to LazyReplication, except when a thread replies to a remote get-request, it broadcasts the reply instead of replying to the requester over a point to point ATAC message. All cores receive the reply, and if the thread local hash map does not already contain an entry for that key, it inserts the key-value pair locally.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Hash Map Performance Evaluation

5.1 Hypothesis

I measure the performance of the ATAC hash maps by evaluating the throughput of insertions and lookups.

We expect insertions to fully exploit parallelism, and therefore exhibit near linear speedup across all hash map implementations.

Our hypothesis for lookup performance comes in two parts. Regarding the different get-request protocols, we hope for a constant factor improvement from AllAck to NoAck, in the case that the extra complications of the NoAck protocol do not overwhelm the expected performance gains.

Our hypothesis regarding the different replication protocols is more sophisticated. Predicting performance with the hash map is much more complicated than the barrier, because there are many different possible memory access patterns for the hash map. I choose only a few of the many different memory access patterns for the hash map, in order to test the following predictions. First, I expect LazyReplication to be slightly more efficient than NoReplication when the replication mechanism results in hitting locally replicated key-value pairs, and to be less efficient if it replicates key-value pairs without locally hitting them again. The greater the ratio of replicated key-value pairs that are hit locally, the greater I expect LazyReplication to outperform NoReplication. I expect LazyBroadcastReplication to be slowed down relative to LazyReplication due

to the overhead of additional packets in the system, but to exhibit better performance when sufficiently many replicated key-value pairs are hit locally.

Since I expect TimeoutNoAck, with an adequate timeout threshold such that few or no lookups are aborted, to have approximately the same performance as NoAck, the results for TimeoutNoAck are omitted here to simplify the presentation.

5.2 Benchmark Methodology

Since the performance of the hash map is highly dependent on the memory access pattern, and hence the sequence of operations performed, I chose to implement simple benchmarks to test the specific properties conjectured in Section 5.1. Each benchmark has the same approach: the concurrent hash map is initialized, the threads are spawned, the starting time is recorded (using the same timing framework as outlined in Section 3.2.2), all threads are run to completion, and the ending time is recorded. The difference between the starting and ending time is reported as the end-to-end runtime of the benchmark. All benchmarks are deterministic, to make it easy to debug and reason about performance.

As a reference point, I include a standard C++ library hash map [2] that is serialized with a global lock. Although this is a very rudimentary implementation of a concurrent hash map that exhibits no parallelism, it is included as a simple sanity check on the performance of the ATAC hash maps. In figures it is referred to as SequentialMap.

5.3 Insertion Benchmark

5.3.1 Explanation

The insertion benchmark performs a fixed number of insertions I into the hash map, divided between a variable number of threads N. I measure the end-to-end time at which all threads have completed their I/N insertions.

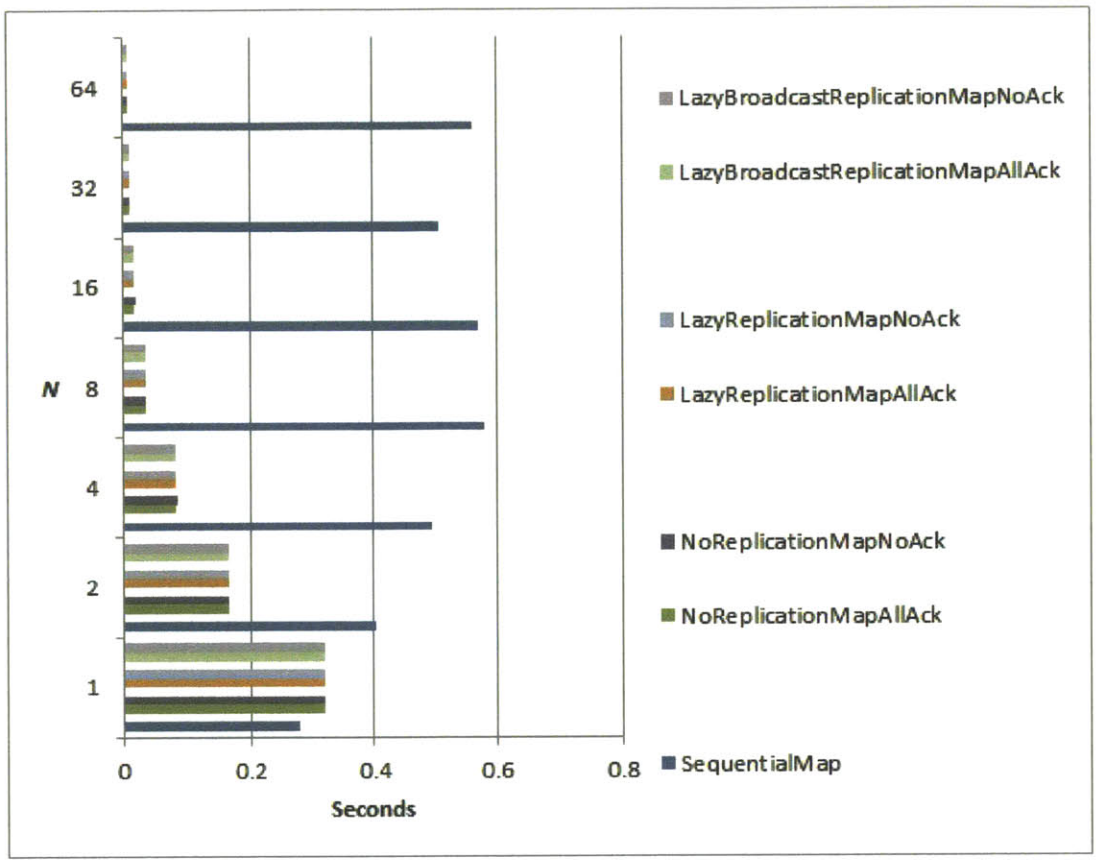

Figure 5-1: Insertion benchmark results.

5.3.2 Results

The results are reported in Figure 5-1. Speedup across all ATAC hash maps is observed up as the number of cores increases.

5.4 Lookup Benchmark

5.4.1 Explanation

The lookup benchmark has the same series as steps as the insertion benchmark, except that during the initialization phase, the same insertions are made as in the insertion benchmark itself. During the measured part of the benchmark, lookup operations are performed.

To understand the significance of the results, it is necessary to understand the

memory access pattern of this benchmark. I insertions are made by the N threads. The insertions follow a deterministic pattern: thread i inserts key-value pair with key=$(i + j * N)$ for all $0 \leq j < I/N$. Therefore, each thread inserts exactly one key in the range $[0, N-1]$ and so on for every N keys. The lookups also follow a deterministic pattern: thread i performs lookups traversing the set of keys $(i/N + x)$ mod I for $0 \leq x < R * I/N$, where R is the ratio of number of lookups performed to number of insertions performed. When $R = 1$, every key is looked up by exactly one thread, and when $R = 2$, every key is looked up by exactly two threads.

The value of R is important in determining which replication protocol has better predicted performance. When $R = 1$, replication serves no benefit to performance, because a key is never accessed more than once. When $R > 1$, LazyBroadcastReplication begins to hit local replicas. Assuming threads progress at the same rate, when $R > 2$, all lookups after the first $2I$ lookups will hit local replicas. Therefore, when $R \gg 2$, we expect LazyBroadcastReplication to outperform NoReplication. LazyReplication, on the other hand, will only hit local replicas after having fully traversed the set of keys, i.e. when $R > N$, and therefore is expected to outperform NoReplication only when $R \gg N$.

5.4.2 Results

The results are reported in Figures 5-2 through 5-6. LazyReplication and LazyBroadcastReplication exhibit the expected patterns of performing worse than NoReplication when replication is underutilized (i.e., a thread does not perform multiple lookups on the same key) and perform better than NoReplication when replicated is adequately utilized. LazyReplication is moderately slower than NoReplication when $1 \leq R \leq 10$, which falls under the regime of no replica hits, but outperforms NoReplication at $R = 100$, which exhibits replica hits. LazyBroadcastReplication performs much worse than the other two replication protocols at $R = 1$, which makes sense because it is experiencing the overhead of broadcasting replies without the benefit of any replica hits. LazyBroadcastReplication performs better relative to the other protocols at $R = 2$, at which point it has started to benefit from replica hits, outperforms the

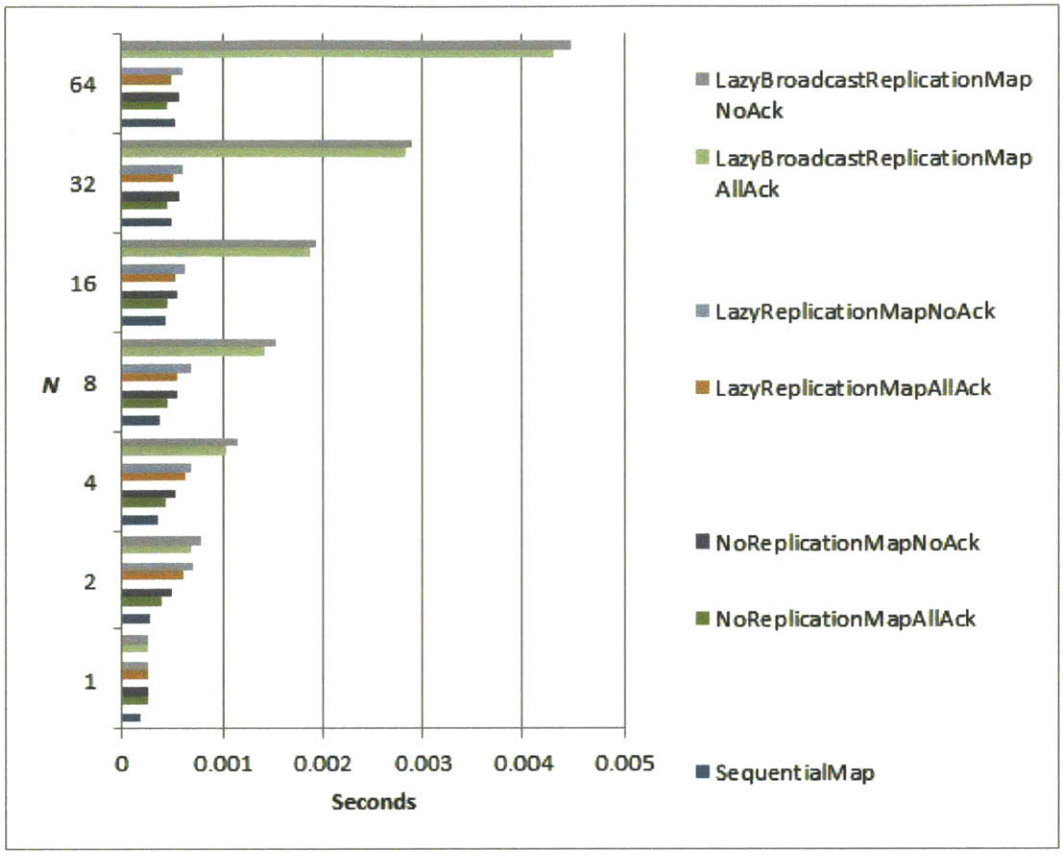

Figure 5-2: Lookup benchmark results, $R = 1$.

other protocols for small N at $R = 5$, and is able to outperform the other protocols for small and large numbers of N at $R = 10$ and especially at $R = 100$.

NoAck generally performs similarly or slightly slower than AllAck. I suspect that this is due to the extra steps required to perform the intended optimization. Fortunately the optimization was designed with a model in mind where packets sent to a particular core are serviced concurrently, so the callbacks had to be supported to run concurrently. Since callbacks actually happen serially on Graphite, the optimization can be simplified and expensive operations (compare-and-swap) can be replaced with less costly read-write operations. I leave this as a future optimization.

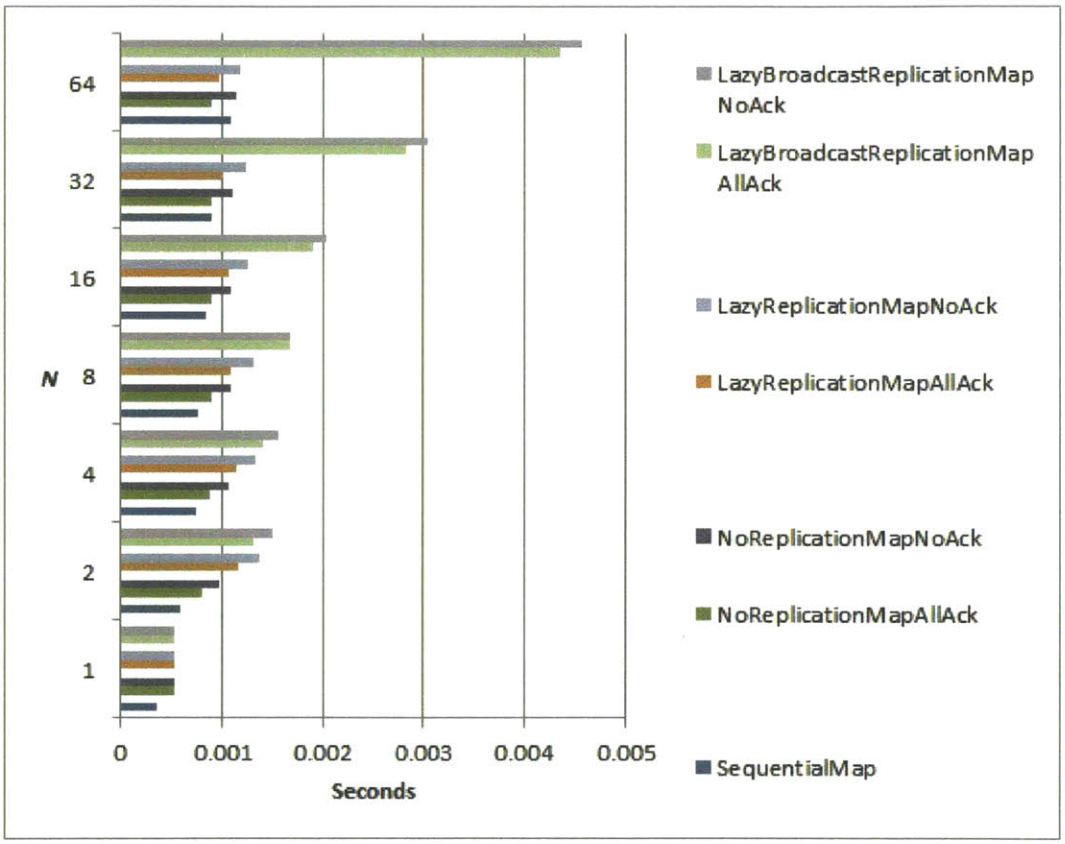

Figure 5-3: Lookup benchmark results, $R = 2$.

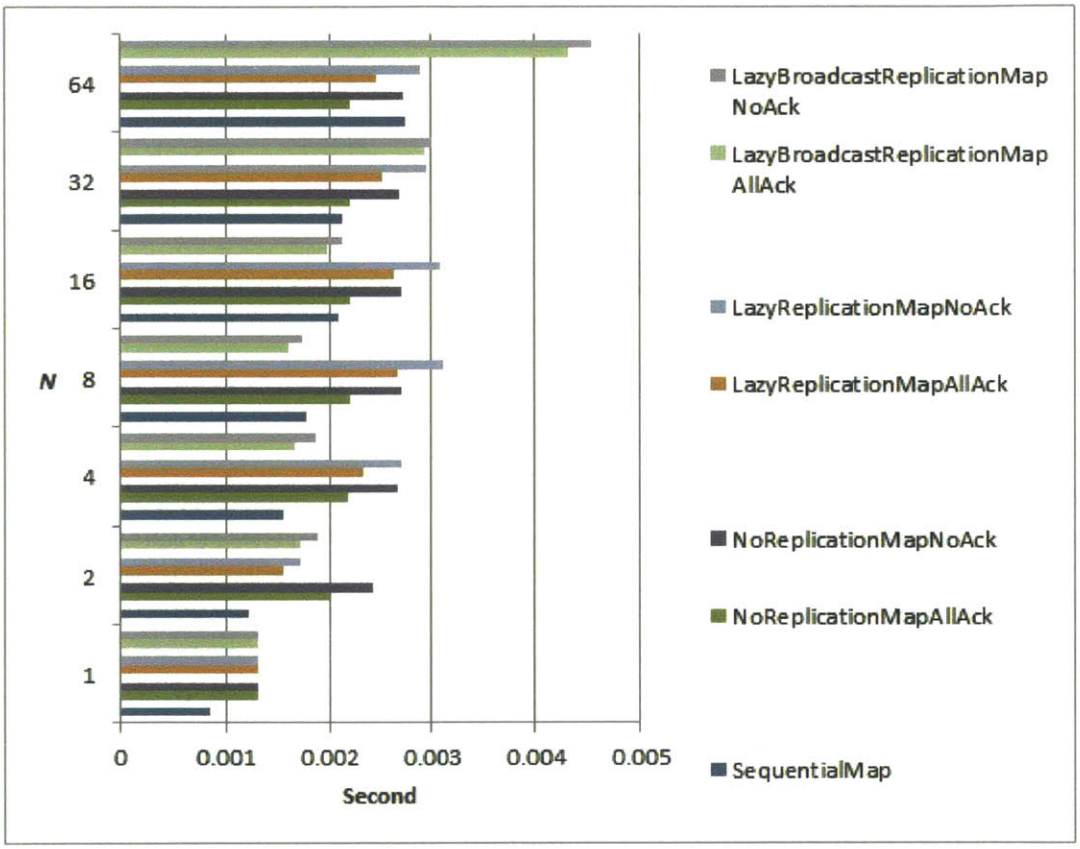

Figure 5-4: Lookup benchmark results, $R = 5$.

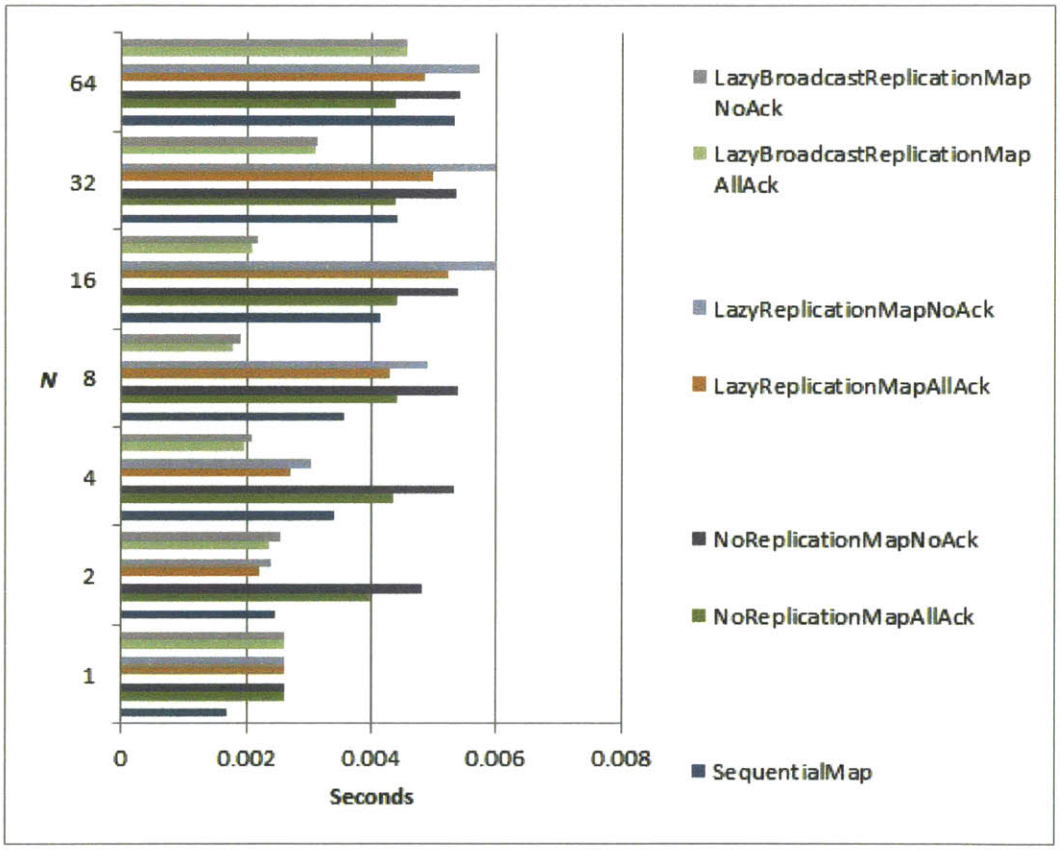

Figure 5-5: Lookup benchmark results, $R = 10$.

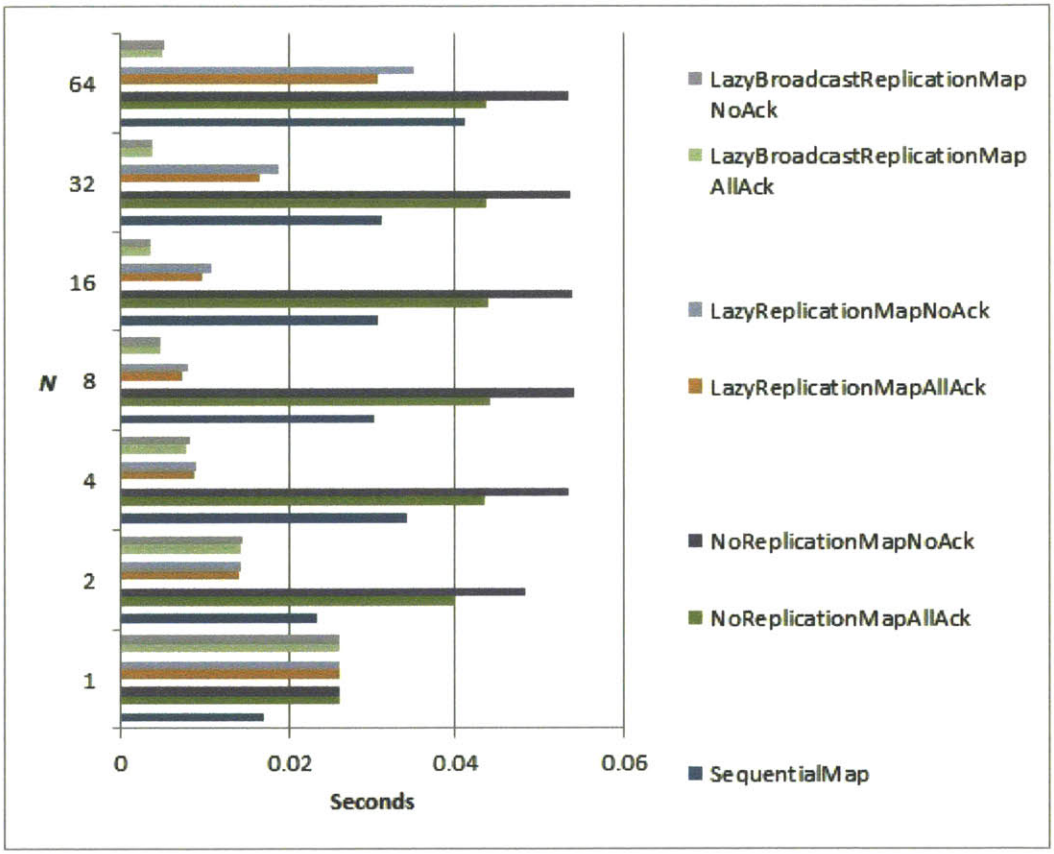

Figure 5-6: Lookup benchmark results, $R = 100$.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Related Work

6.1 On-Chip Optical Communication

The use of on-chip optical communications is a new field, with a research just beginning [13]. Kirman et al. present an optical broadcast mechanism to implement cache-coherence [5]. The mechanism has much in common with ATAC, but the functionality serves a much more specialized, and therefore limited, purpose. The Corona architecture supports a general communication mechanism, but its hardware is set up such that senders experience contention, unlike in ATAC [4].

6.2 Barrier

Software barriers can be grouped into several categories. Centralized barriers, such as the sense-reversing barrier, rely on all threads updating common data which has a certain value when threads are allowed to stop spinning and continue. This method is prone to contention. A combing barrier also waits on common data, but the updating process involves modifying data shared by only a subgroup of the threads. One thread from each subgroup is assigned the task of coordinating with other groups in a hierarchical fashion, so contention is less. In a combining system, coordinator threads are determined dynamically. A tournament barrier functions similarly, except that the coordinating threads are statically assigned. A dissemination barrier seeks

to reduce contention in a nonhierarchical manner, such that all threads perform the same operations [9]. The ATAC barrier most closely fits under the dissemination paradigm. What distinguishes the ATAC barrier from the centralized, combining, and tournament models is that all communication occurs through the broadcast network, enabling each thread to wait on local data and avoiding contention.

Some high-performance barriers are implemented using dedicated hardware [17]. Because dedicated hardware is costly, hardware-assisted software barriers have also been developed [14]. The Intel BlueGene supercomputer project is an example of a system that leverages specialized hardware to implement a high-performance barrier [18]. The ATAC barrier benefits from novel hardware, but is built on top of a system that exists for general usage.

6.3 Hash Map

Good performance can be achieved for a hash table with fixed number of bins simply by assigning a lock to each bin [10]. Data structures have been implemented that support resizing without global locking, such as the recursive split ordering (RSO) hash table [15]. RSO requires a more complicated algorithm, compared to our hash map which is implemented in a simple manner.

ACKwise is a general purpose cache coherence protocol implemented on ATAC [8]. The hash map essentially is a more specific application for coherence between threads, which allows specific tradeoffs between correctness guarantees and performance as is suitable for a given application.

Chapter 7

Conclusion

7.1 Remarks

In this thesis, I explored the design and performance of concurrent data structures implemented with ATAC broadcasts and thread local memory. I described VCBarrier and its straightforward implementation, and validated its high performance up to large numbers of cores. I explained the design of several hash maps with relaxed consistency guarantees, and examined their performance under certain memory access patterns. In all, I demonstrated that the ATAC broadcast mechanism can be used to construct highly scalable data structures while maintaining simple implementations.

7.2 Future Work

Additional performance testing of the hash maps is desirable to understand how their performance scales under other memory acccess patterns. An application benchmark, in addition to the synthetic benchmarks, would be useful to evaluate performance in a real-world setting. Work is currently being done to evaluate the performance of Memcached, a distributed memory object caching system, on Graphite with an ATAC hash map as the underlying key-value store [1].

Other broadcast-based hash map protocols exist to be explored and tested. We are interested in evaluating a distributed hash map that assigns cores to groups. Each

group divides the keyspace between its members. Individual cores are responsible for storing key-value pairs that fall within its assigned keyspace. A lookup would first query the appropriate core within the requesting core's group, and should the key-value pair not be found, then the appropriate core would query all of the other cores assigned to that keyspace, using one of the get-request protocols similar to those outlined in this report. This configuration is intended to reduce the number of broadcast packets received.

Bibliography

[1] Memcached - a distributed memory object caching system, August 2013. http://memcached.org/.

[2] unordered_map - C++ Reference, August 2013. http://www.cplusplus.com/reference/unordered_map/unordered_map/.

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[4] D. Vantrease et al. Corona: System Implications of Emerging Nanophotonic Technology. In *ISCA*, 2008.

[5] N. Kirman et al. Leveraging Optical Technology in Future Bus-based Chip Multiprocessors. In *MICRO*, 2006.

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

[7] Amos Israeli and Lihu Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160. Aug 1994.

[8] J. Miller, J. Psota, G. Kurian, N. Beckman, J. Eastep, J. Liu, M. Beals, J. Michel, L. Kimerling, and A. Agarwal. ATAC: A Manycore Processor with On-Chip Network. In *MIT Technical Report*, 2009.

[9] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

[10] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[11] J. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, January 2010.

[12] Seo Jin Park. Analyzing Performance and Usability of Broadcast-Based Inter-Core Communication (ATAC) on Manycore Architecture. Master's thesis, Massachusetts Institute of Technology, June 2013.

[13] J. Psota, J. Miller, G. Kurian, H. Hoffman, N. Beckmann, J. Eastep, and A. Agarwal. ATAC: Improving Performance and Programmability with On-Chip Optical Networks. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3325–3328, 2010.

[14] John Sartori and Rakesh Kumar. Low-Overhead, High-speed Multi-core Barrier Synchronization. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC, pages 18–34, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Ori Shalev and Nir Shavit. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM*, 53(3):379–405, May 2006.

[16] Nir Shavit. Data Structures in the Multicore Age. *Commun. ACM*, 54(3):76–84, 2011.

[17] Rajeev Sivaram, Craig B. Stunkel, and Dhabaleswar K. Panda. A Reliable Hardware Barrier Synchronization Scheme. In *IPPS*, pages 274–280. IEEE Computer Society, 1997.

[18] Robert W. Wisniewski. BlueGene/Q: Architecture, CoDesign; Path to Exascale. 2012.