Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures

by

Jonathan M. Eastep

B.S., University of Texas at Austin (2004)
S.M., Massachusetts Institute of Technology (2007)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by . . . .
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures

by

Jonathan M. Eastep

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

As multicores become prevalent, the complexity of programming is skyrocketing. One major difficulty is efficiently orchestrating collaboration among threads through shared data structures. Unfortunately, choosing and hand-tuning data structure algorithms to get good performance across a variety of machines and inputs is a herculean task to add to the fundamental difficulty of getting a parallel program correct. To help mitigate these complexities, this work develops a new class of parallel data structures called Smart Data Structures that leverage online machine learning to adapt themselves automatically. We prototype and evaluate an open source library of Smart Data Structures for common parallel programming needs and demonstrate significant improvements over the best existing algorithms under a variety of conditions. Our results indicate that learning is a promising technique for balancing and adapting to complex, time-varying tradeoffs and achieving the best performance available.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis is dedicated to my parents, my girlfriend, and my mentor in the early days, Michael Taylor. You stood by me throughout the PhD process when I needed you most, and I would not have finished without your support and belief in me.

I also owe a big debt to my collaborators David Wingate and Marco Santambrogio who helped take Smart Data Structures from a concept to a living, breathing system. I feel privileged to have worked with such kind, devoted, and talented researchers.

To my colleagues at MIT, thank you for the opportunity to grow and learn together. To Jason Miller, Henry Hoffman, and Harshad Kasture especially, thank you for always making the time to bounce ideas around with me. It significantly contributed to my intellectual growth at MIT.

To my research adviser Anant Agarwal, thank you for supporting me and patiently nurturing my development as a researcher. Thank you for re-teaching me how to think about a problem and how to communicate clearly with others. It is the most valuable thing I have ever learned.

Contents

1 Introduction **19**

1.1 Programming Problems 19

1.2 Self-Aware Computing 20

1.3 Smart Data Structures Introduction 20

1.4 Smart Data Structures Design Overview 23

1.4.1 Design Challenges 23

1.4.2 Application Interfaces 23

1.4.3 Learning and Reward Architecture 23

1.5 Key Contributions 25

1.6 Thesis Overview 27

2 Background and Related Work **29**

2.1 Background on Concurrent Data Structures 29

2.1.1 Historical Evolution of Spin-Locks 30

2.1.2 Historical Evolution of Flat Combining 33

2.2 Adaptive Data Structures 36

2.2.1 Auto-Tuned Libraries 36

2.2.2 Adaptive Programming Frameworks 38

2.3 Smart Data Structures Contributions Summary 39

2.3.1 Novel Online Learning Adaptation Methodology 39

2.3.2 Novel Lock Acquisition Scheduling Optimizations 40

2.3.3 Closed-Loop, Dynamic Decision-Making 40

2.3.4 Simplified Extensibility via Model-Free Learning 41

2.4 Additional Learning-Based Self-Aware Systems 42
2.4.1 Resource Allocation 42
2.4.2 Scheduling and Load-Balancing 42
2.4.3 Libraries and Optimization 43

3 Smart Data Structures Design 45
3.1 Smart Data Structures Architecture 45
3.1.1 Implementation Strategy 46
3.1.2 Optimization Methodology 47
3.2 Smart Data Structures Prototype Library 49
3.2.1 Supported Data Structures 49
3.2.2 Application Interfaces 50
3.2.3 Data Structure Implementations 50
3.2.4 Library Extensibility and Other Features 58

4 Learning Design and Challenges 61
4.1 Design Challenges 61
4.2 Learning Architecture 62
4.3 Learning Engine Algorithm 64
4.4 Learning Thread Tradeoffs 67

5 Performance Results 69
5.1 Experimental Setup 70
5.2 Performance of Existing Alternatives 71
5.3 Scancount Sensitivity 73
5.4 Performance of Smart Data Structures 75
5.5 Adaptivity of Smart Data Structures 78
5.6 Application Case Studies 81
5.6.1 Application Descriptions 82
5.6.2 Smart Data Structures Versus Previous Work 84
5.6.3 Smart Data Structures Versus Performance Bounds 90

5.6.4 Usage Guidelines 94

6 Scalability Results **97**

6.1 Introduction 97

6.2 Concurrency Demands 99

6.2.1 Data Structure Communication Bottlenecks 100

6.2.2 Data Structure Algorithm Bottlenecks 102

6.2.3 Summary of Data Structure Concurrency Constraints 109

6.2.4 Smart Data Structures Communication Bottlenecks 110

6.2.5 Smart Data Structures Reward Bottlenecks 114

6.2.6 Smart Data Structures Learning Bottlenecks 118

6.2.7 Summary of Smart Data Structures Concurrency Constraints . 129

6.3 Multi-Data-Structure Demands 131

6.3.1 Multi-Data-Structure Incremental Bottlenecks 132

6.3.2 Multi-Data-Structure Scaling Constraints 136

6.4 Multi-Optimization Demands 137

6.4.1 Multi-Optimization Incremental Bottlenecks 137

6.4.2 Multi-Optimization Scaling Constraints 140

6.5 Case Studies 142

6.5.1 Combined Constraints 143

6.5.2 Case Study Applications 144

6.5.3 Overall Scaling Results 146

7 Smart Locks Performance Results **151**

7.1 Experiment Overview 151

7.2 Dynamic Overclocking Experiment 152

7.2.1 Experimental Setup 152

7.2.2 Results 153

7.3 SPLASH-2 Static Heterogeneity Experiment 156

7.3.1 Experimental Setup 157

7.3.2 Results 158

7.4 Smart Locks Usage Guidelines 160
7.4.1 Self-Optimizing Data Structures 160
7.4.2 Learning Thread Sensitivity Analysis 161

8 Future Work 165
8.1 Scalability Enhancements 165
8.2 Additional Smart Data Structures 167
8.3 Additional Axes of Adaptation 167
8.4 Alternative Learning Integration Strategies 168
8.5 Applications to Other Systems 169

9 Conclusion 171

A Lazy Counter Algorithm 173

List of Figures

1-1 The Anatomy of Standard Parallel Data Structures. Data structures consist of storage, interfaces, and algorithms. The storage organizes the data, the interfaces specify how the data can be accessed and manipulated, and algorithms implement the interface operations while preserving correct concurrent semantics. Knobs parameterize the behavior of the storage and algorithms. 21

1-2 The Anatomy of Smart Data Structures. Smart Data Structures augment standard data structure interfaces, storage, and algorithms with online machine learning to internally optimize the knobs that control their behavior. 22

1-3 Smart Data Structures Internals. All Smart Data Structures share a learning thread which jointly optimizes the knobs that control their behavior. Performance feedback, the reward, drives the optimization. 24

1-4 Smart Pairing Heap Throughput vs. Post Computation. Through online learning, the Smart Pairing Heap significantly improves performance over the average static bound, achieving and exceeding the ideal static bound. 27

2-1 Properties of a Spin-Lock Algorithm 30

3-1 The Anatomy of Standard Parallel Data Structures. Data structures consist of storage, interfaces, and algorithms. The storage organizes the data, the interfaces specify how the data can be accessed and manipulated, and algorithms implement the interface operations while preserving correct concurrent semantics. Knobs parameterize the behavior of the storage and algorithms. 46

3-2 The Anatomy of Smart Data Structures. Smart Data Structures augment standard data structure interfaces, storage, and algorithms with online machine learning to internally optimize the knobs that control their behavior. 47

3-3 Flat Combining Data Structures. The Flat Combining Skip List is pictured. Flat Combining data structures consist of a *publication list*, a lock, a *scancount*, and a serial data structure. 53

3-4 The Smart Queue, Skip List, and Pairing Heap. The Smart Skip List is pictured. These Smart Data Structures augment the Flat Combining algorithm with an online machine learning engine to optimize a performance-critical knob of the algorithm called the *scancount*. . . . 54

4-1 Smart Data Structures Learning Architecture. A learning engine collects performance feedback in the form of a reward signal from a reward monitor. An internal reward monitor is provided by default; for generality, external, application-specific reward monitors are also supported. 62

5-1 Performance Characterization of the Best Existing Algorithms. The Flat Combining Queue, Skip List, and Pairing Heap substantially outperform the others at higher concurrency levels and heavier loads (lower post computation). 72

5-2 Sensitivity to the Scancount in Producer-Consumer Application Structures: Throughput vs Scancount Over a Range of Loads. The ideal scancount varies widely and depends on both the load and the data structure. 74

5-3 Smart Queue Throughput vs Post Computation: A Comparison Against Ideal and Average Static Throughput Bounds. The Smart Queue achieves near ideal static throughput for most data structure loads. . 76

5-4 Smart Skip List Throughput vs Post Computation: A Comparison Against Ideal and Average Static Throughput Bounds. The Smart Skip List achieves near ideal static throughput for most data structure loads. 77

5-5 Smart Pairing Heap Throughput vs Post Computation: A Comparison Against Ideal and Average Static Throughput Bounds. The Smart Pairing Heap achieves near ideal static throughput for most data structure loads. 77

5-6 Smart Data Structures Throughput Under Variable Load: A Comparison Against Ideal Dynamic and Average Dynamic Throughput for Different Variation Frequencies. In many cases, Smart Data Structures achieve near-ideal throughput. Throughput slowly decreases as changes in the load become more frequent. 80

6-1 Concurrency Scaling: One Flat Combining Data Structure Shared Among n Application Threads. The communication ports between the application threads and the Flat Combining components are depicted. 101

6-2 Concurrency Scaling of the Flat Combining Queue. The Flat Combining Queue is compared to the best existing queue algorithms on a SPARC T2 system. It reaches maximum performance at 24 threads but outperforms all prior queues up to 64 threads. Some time after 64 threads, the Combining Tree Queue is expected to overtake it as the highest performance queue. 103

6-3 Concurrency Scaling of the Flat Combining Skip List. The Flat Combining Skip List is compared to the best existing priority queue implementations on a SPARC T2 system. It reaches maximum performance at 4 threads but outperforms all prior priority queues up to 64 threads and beyond. 104

6-4 Concurrency Scaling of the Flat Combining Pairing Heap. The Flat Combining Pairing Heap is compared to the best existing priority queue implementations on a SPARC T2 system. It reaches maximum performance at 12 threads but outperforms all prior priority queues up to 64 threads and beyond. 105

6-5 A Comparison of the Number of Necessary CAS Successes per Data Structure Operation for Different Queues. While for most queues the necessary rate is approximately fixed as the number of threads increases, the necessary rate for the Flat Combining queue decreases. This is the primary source of its performance improvements over the other queues. 106

6-6 A Comparison of the Number of CAS Failures per Data Structure Operation for Different Queues. While for most queues the failure rate increases as the number of threads increases, the failure rate only initially increases then decreases with the Flat Combining Queue. . . 108

6-7 Concurrency Scaling: One Smart Data Structure Shared Among All Application Threads. The communication ports between optimization components, application threads, and the Flat Combining components are depicted for two different reward modes. 112

6-8 Heartbeats Scaling. The total update rate achieved by Heartbeats is compared to the ideal update rate for different requested per-thread update rates. Heartbeats sustain the ideal update rate until an inflection point at a total update rate of approximately 5.7 MHz. 116

6-9 Reward Scaling for Different Reward Monitors. The total update rate achieved by the Lazy Counter monitor and Heartbeats monitor are compared to the ideal for different requested per-thread update rates. The Lazy Counter monitor nearly achieves the ideal total update rate while Heartbeats saturate at a much lower total update rate. 117

6-10 The Practical Impact of Reward Scaling on a 16-core Intel Xeon System. The figures compare benchmark throughput using Heartbeats vs the Lazy Counter reward monitor for the Smart Queue, Skip List, and Pairing Heap. The results indicate that for the benchmark and concurrency levels in our 16-core Xeon system, both reward monitors achieve similar throughput. 118

6-15 Multi-Data-Structure Scaling. Applications scale up by adding new thread pools with each pool utilizing its own Smart Data Structure. Each pool (and corresponding Smart Data Structure) has its own reward monitor and learning engine. Learning engines all run in a single learning thread and time-multiplex its resources. 132

6-17 Multi-Optimization Scaling. Applications scale up by increasing the number of Smart Data Structures belonging to each pool of threads. We assume there is one pool of threads, with a fixed number of threads n. There are s Smart Data Structures. All s Smart Data Structures share one reward monitor and one learning engine. 138

7-1 Heartrate performance across thermal throttling events (workload changes). Smart Locks significantly outperforms reactive and TAS spin-locks, achieving near optimal. 154

7-2 Time evolution of the learned policy. Crossovers between Worker 0 and 3 reflect throttling events. 155

7-3 Speedup versus lock acquisition scheduling policy. The policy can significantly impact performance. Smart Locks learns a policy that approaches the ideal speedup. 159

7-4 Normalized execution time of SPLASH-2 applications. 6 threads with an additional thread for Smart Locks vs. 6 threads vs. 7 threads. The slowdown reflects by what factor Smart Locks must improve performance for net benefit. 163

List of Tables

2.1 Summary of Lock Algorithms 33

5.1 Summary of Performance Variation and Smart Data Structure Improvements 94

6.1 Concurrency Scaling of Flat Combining Data Structure Components. For each component, the scaling of the storage requirements, number of communication ports, and degree of internal cache line sharing are given as a function of n, the number of threads accessing the data structure. 102

6.2 Concurrency Scaling Constraints from Flat Combining Data Structures. n is the number of threads concurrently accessing a given Smart Data Structure, and the max n is defined to be the maximum number of threads before an alternative algorithm will outperform the Smart Data Structure. 110

6.3 Concurrency Scaling of Smart Data Structure Optimization Components. For each component, the scaling of storage requirements, communication ports, and the degree of cache line sharing are given as a function of n, where n is the number of threads accessing the Smart Data Structure. 113

6.4 Concurrency Scaling Constraints from Smart Data Structures Optimization Components. n is the number of threads accessing the data structure. 131

6.5 Multi-Data-Structure Scaling Constraints. 136

6.6 Application Scaling Demands. n is the number of application threads 146
6.7 Application Scaling Limits and Limit Sources. n is the total number of application threads. 147
6.8 Performance at Realistic Software Pipelines Scaling Levels 149
6.9 Performance at Maximum Scaling Levels 149

7.1 Expected Utility of Smart Locks by Scenario 160

Chapter 1

Introduction

1.1 Programming Problems

As multicores become prevalent, programming complexity is skyrocketing. Programmers expend significant effort on parallelizing problems and mapping them onto hardware in a way that keeps all threads busy and working together effectively. In many applications, the most difficult aspect of design is efficiently orchestrating collaboration among threads through shared data structures.

Unfortunately, application performance is becoming increasingly sensitive to the choice of parallel data structure algorithms and algorithm parameter settings. The best algorithm and parameter settings can depend in complicated ways on the machine's memory system architecture as well as application-specific criteria such as the load on the data structure. To make matters worse, many applications have input-dependent computation which causes the load to vary dynamically.

Writing correct software is fundamentally difficult, but writing software that is simultaneously correct and high performance across a variety of machines and inputs is a herculean task. Our view is that programmers should not be expected to code for these complexities by hand.

1.2 Self-Aware Computing

Recently, self-aware computing has been proposed as one automatic approach to freeing the programmer from this complexity. While traditional systems require the programmer to balance system constraints by hand, self-aware systems attempt to automatically monitor themselves and dynamically adapt their behavior at runtime.

Self-aware systems are an example of closed-loop optimization: they measure performance feedback and make adjustments continually as system conditions change to achieve the best performance available at any time. They are sometimes referred to as autonomic, auto-tuning, adaptive, etc., and they have been applied to a broad range of platforms including embedded / real-time [10], desktop [18], server [9, 8], and cloud computing environments [40, 38].

1.3 Smart Data Structures Introduction

One of the insights in this work is that design principles from self-aware computing can be applied to the problem of tuning data structures. This thesis introduces Smart Data Structures, a new class of self-aware parallel data structures that self-tune themselves automatically through a novel methodology based on online machine learning. Through learning and automatic tuning, Smart Data Structures relieve programmers of the burden of hand-tuning for the best performance across different machines, applications, and inputs.

Smart Data Structures are drop-in, self-optimizing replacements for standard data structures. They are implemented by layering an online learning engine on top of a standard data structure. To illustrate, we review the anatomy of a standard data structure and contrast it with the anatomy of a Smart Data Structure.

Figure 3-1 shows the components of a standard data structure. Standard data structures consist of data storage, an interface, and algorithms. The storage organizes the data, the interface specifies the operations threads may apply to the data to manipulate or query it, and the algorithms implement the interfaces while preserving correct concurrent semantics.

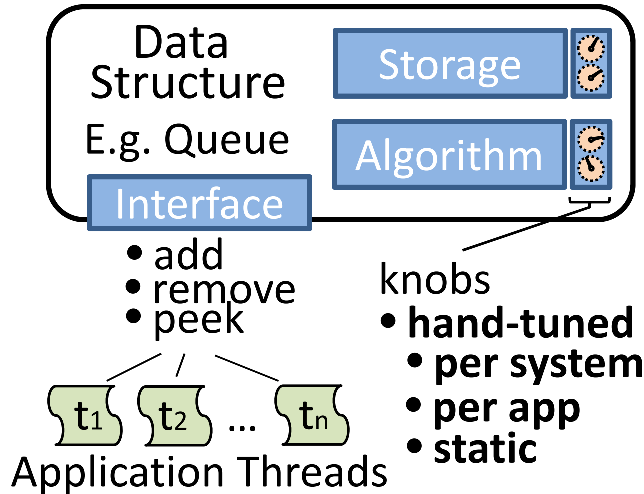


Figure 1-1: The Anatomy of Standard Parallel Data Structures. Data structures consist of storage, interfaces, and algorithms. The storage organizes the data, the interfaces specify how the data can be accessed and manipulated, and algorithms implement the interface operations while preserving correct concurrent semantics. Knobs parameterize the behavior of the storage and algorithms.

Storage and algorithms are often controlled by *knobs*: thresholds or other parameters that program implementation behaviors and heuristics. Knobs are typically configured via one-size-fits-all static defaults provided by the library programmer. When the defaults perform sub-optimally, programmers must hand-tune the knobs. This is typically done through trial and error which can increase development time and through special cases in the code which reduce readability. Though often necessary, runtime tuning is typically ignored by the programmer due to its complexity.

Figure 3-2 contrasts Smart Data Structures with standard data structures. Smart Data Structures preserve the same interfaces, storage, and algorithms. The difference is that Smart Data Structures augment standard data structures with an online learning engine that automatically and dynamically tunes the knobs to optimize storage and algorithm behaviors. Through learning, Smart Data Structures balance complex tradeoffs to find ideal knob settings and adapt to changes in the system or inputs that affect these tradeoffs.

There have been a variety of related works in adaptive data structures. Among the most well-known are several auto-tuned signal processing and linear algebra libraries:

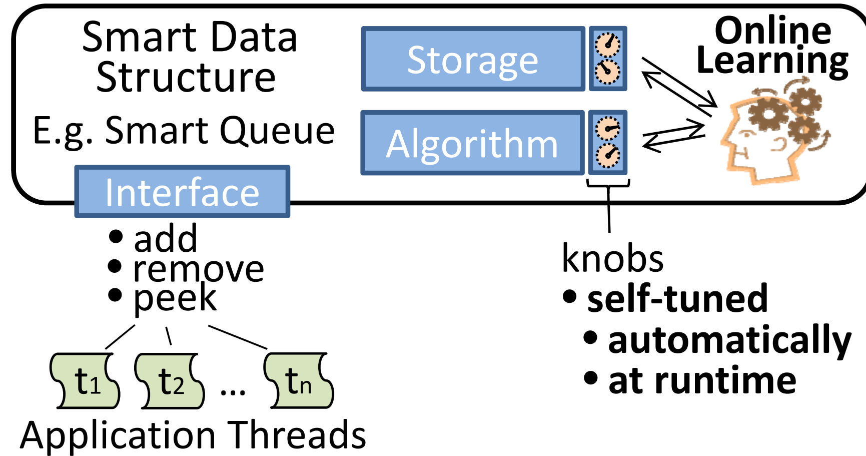


Figure 1-2: The Anatomy of Smart Data Structures. Smart Data Structures augment standard data structure interfaces, storage, and algorithms with online machine learning to internally optimize the knobs that control their behavior.

FFTW, PHiPAC, and ATLAS [11, 3, 20]. Smart Data Structures differ from these prior works in an important way. While prior works can adapt to different machine architectures and runtime conditions like input size, they typically base these decisions on thresholds computed during compile-time or install-time characterization. The problem is that such characterizations can poorly reflect realistic runtime conditions in modern systems.

Consider systems with frequency scaling, for example. Frequency scaling technologies like thermal throttling or Turboboost® from Intel® can dynamically under- or overclock some subset of the processors, altering the machine's effective performance and substantially affecting the tradeoffs that determine which algorithm and/or knob settings are best. Furthermore, vanilla multi-process environments can have complex runtime conditions as well. They can have unpredictable effective performance because applications run alongside other applications, interfering and competing in different ways for important resources like communication and memory bandwidth.

Smart Data Structures account for these complexities by taking on online approach to optimization decisions. They collect dynamic information about the system and performance tradeoffs. They balance those tradeoffs intelligently at runtime through online learning. Through learning, Smart Data Structures adapt and react to changes in the system, application, or inputs to achieve the best performance available.

1.4 Smart Data Structures Design Overview

1.4.1 Design Challenges

The overriding goal of our design is to maintain ease of use in applications while providing the highest performance available across a variety of different machines, applications, and workloads. To do so, our design must address three key challenges: 1) measuring application performance in a reliable, non-intrusive, and portable way, 2) adapting knob settings quickly so as not to miss windows of opportunity for optimization, and 3) identifying the knob settings that are best for long-term performance. The subsequent sections provide an overview of the Smart Data Structures design, with descriptions framed around how these three challenges are addressed.

1.4.2 Application Interfaces

Smart Data Structures are drop-in, self-optimizing replacements for standard, non-blocking concurrent data structures. While internally adapting their storage and algorithms, Smart Data Structures preserve fixed, standard interfaces. Smart Data Structures are implemented in C++ for shared memory C++ applications. C interfaces are provided as well for mixing with other programming languages. Thus, from the perspective of an application developer, integrating a Smart Data Structure into an application is as simple as integrating a standard data structure: the developer includes a library header file and is provided standard object-oriented interfaces.

1.4.3 Learning and Reward Architecture

Internally, a Smart Data Structure uses online machine learning to learn knob settings that best optimize its storage and algorithms. As Figure 4-1 illustrates, each Smart Data Structure attaches to an online learning engine which optimizes its knobs. That learning engine runs in a *learning thread* separate from the application threads. The number of learning threads is parameterizable. We will evaluate the case with one learning thread that multiplexes all learning engines.

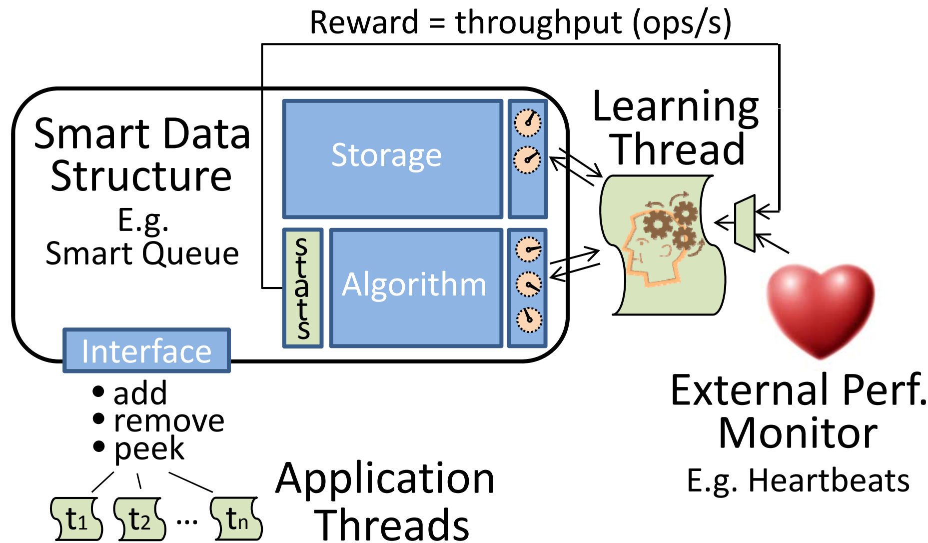


Figure 1-3: Smart Data Structures Internals. All Smart Data Structures share a learning thread which jointly optimizes the knobs that control their behavior. Performance feedback, the reward, drives the optimization.

Optimization within each learning engine is driven by performance feedback, the *reward*. The reward signal must reflect application performance accurately without perturbing it. We address Challenge 1 (ensuring that performance measurements are portable, non-intrusive, and reliable) by supplying a low-overhead, internal reward signal to the learning engine that meets these criteria for many applications: by default, Smart Data Structures measure and provide their own throughput as the reward. For generality, we also support a variety of external performance monitors that developers can use to provide application-specific reward signals.

Application Heartbeats is one external performance monitor that we recommend for its portability and ease of integration into applications [17]. Heartbeats is a framework for expressing application goals and measuring progress toward them through the abstraction of *heartbeats*. Developers insert calls to Heartbeats at significant points in the application to issue a heartbeat for each unit of progress. The learning engine uses the rate of heartbeats, the heart rate, as the reward.

We address Challenge 2 (adapting settings quickly so as not to miss windows of opportunity) in part by running learning engines in a dedicated learning thread rather than interleaving the learning computation within the application code in the

application threads. This decoupling allows learning engines to run faster, deliver optimizations sooner, and minimize disruption of the application threads.

The other way we meet this challenge is through our choice of learning algorithms. As Chapter 4.3 will elaborate, our choice of learning algorithms is also central to meeting Challenge 3 (identifying good long-term knob settings): we use a Reinforcement Learning algorithm based on the Policy Gradients method [42]. The goal of the algorithm is to find the knob settings that maximize the reward at any given time. As the name suggests, the method for improving knob settings is analogous to gradient ascent. The algorithm is online and fast, and the reward criterion that we adopt enables the algorithm to optimize for the best long-term effects.

1.5 Key Contributions

In this work, we have developed an open source prototype library of Smart Data Structures for common parallel programming needs. It is available [7] on github under a GPL license. We developed the Smart Data Structures prototype to a) demonstrate our novel methodology for optimizing data structures using online learning and to b) pose and answer research questions such as:

- Can online learning be used to optimize data structures and simplify programming?
- Is learning efficient enough to be used for fine-grained, online optimization in data structures?
- What level of performance improvements are possible using online learning?
- How well can a learning-based design scale to large concurrency levels?

The key contribution of this work is compelling answers to these questions. We show constructively, through empirical evaluations of our prototype and prior publications in this area [9, 8], that online machine learning *is* an effective strategy for automatically tuning data structures, that learning *is* efficient enough for fine-grained

online optimization of data structures, that *significant* improvements over state-of-the-art algorithms of up to 44% are possible, and that learning *is not* the scalability limiter for the data structures we have studied.

Furthermore, while this work focuses on a case study of optimizing data structures, we have taken care to design our learning engine and abstractions so that they may be applied to the online optimization of other systems in the future as well. The long-term vision for this work is the optimization of knobs in systems such as cloud resource allocators, spatially-aware OS schedulers, and adaptive hardware cache hash functions and coherence protocols.

In Figure 1-4, we give an example from our experimental results for optimizing data structures. One of the Smart Data Structures provided by our library is a Smart Pairing Heap: a concurrent heap based on the Flat Combining algorithm [12] (to be described in Chapter 3.2.3). We benchmark the Smart Pairing Heap to determine the throughput it can achieve on our machine using online learning to automatically tune a performance-critical knob in its algorithm. We compare this to the throughput of the typical approach: statically programming knob settings. Specifically, we compare against the throughput that could be achieved through painstaking hand-tuning (the ideal throughput) and the throughput that would be achieved if learning were not working well (the average throughput over the available knob settings).

The difference between the static ideal and average throughput demonstrates that finding the optimal knob value can substantially improve throughput. Further, the result shows that the Smart Pairing Heap is able to learn ideal knob values and reach the ideal static throughput. Its throughput actually slightly exceeds the ideal static throughput in some cases because, for data structures like the Pairing Heap, the ideal knob value can vary during execution and necessitate dynamic tuning. The ability to dynamically adapt knob settings is a major virtue of the Smart Data Structures approach.

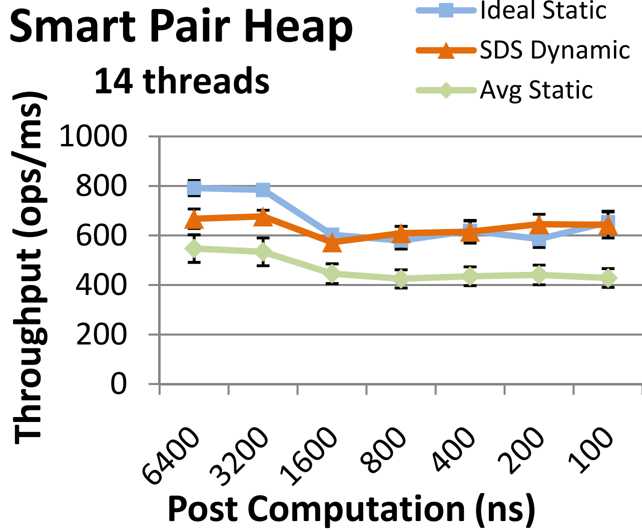


Figure 1-4: Smart Pairing Heap Throughput vs. Post Computation. Through online learning, the Smart Pairing Heap significantly improves performance over the average static bound, achieving and exceeding the ideal static bound.

1.6 Thesis Overview

The rest of this thesis is organized as follows. Chapter 2 provides background in concurrent data structures and compares Smart Data Structures to related work in auto-tuned libraries and adaptive programming frameworks. Chapter 3 presents the Smart Data Structures design. It describes the popular parallel programming data structures we provide, the base algorithms upon which Smart Data Structures layer online learning, and the knobs within them that we expose to learning for performance optimization. Then, in Chapter 4, we describe the Smart Data Structures learning design and how we address key challenges such as measuring how well a given knob setting is performing, adapting settings quickly enough to meet optimization deadlines, and planning to ensure good long-term knob settings.

Chapter 5 through Chapter 7 present our experimental results. Our results are organized in three parts: performance results for the Smart Queue, Skip List, and Pairing Heap (Chapter 5), scalability results for the Smart Queue, Skip List, and Pairing Heap (Chapter 6), and performance results for the Smart Lock (Chapter 7).

In Chapter 5, we perform five experiments that 1) evaluate various state-of-the art algorithms to determine which are highest performance and best to build our

Smart Data Structures library prototype upon, 2) study the sensitivity of Smart Data Structures to different knob settings to motivate our auto-tuning of them via machine learning, 3) determine ideal and average static performance bounds and compare the performance of Smart Data Structures with online learning against them, 4) demonstrate the adaptivity of Smart Data Structures to high frequency changes in the system and application which affect the ideal knob settings and 5) evaluate Smart Data Structures in a variety of real-world applications to demonstrate significant performance improvements and analyze different use-cases to determine when Smart Data Structures provide the most performance improvements over prior work.

In Chapter 6, we evaluate the scalability of the Smart Queue, Skip List, and Pairing Heap for various application case studies. Specifically, we determine the maximum concurrency level that each application could theoretically scale to while still achieving good performance with Smart Data Structures. We identify various scaling challenges and describe how our design addresses them.

Similar to Chapter 5, Chapter 7 presents performance results for the Smart Lock. For several application case studies, we study sensitivity to the knobs in the Smart Lock and motivate our use of learning-based auto-tuning. We determine bounds on the application performance and demonstrate that Smart Locks achieve near ideal performance. Then, we show that Smart Locks can adapt to dynamic changes in the system. Specifically, we simulate Turboboost® overclocking events and show the Smart Lock readily adapting its knobs to the changes in core clock speeds.

Next, Chapter 8 discusses future work. We describe additional areas for research, pose alternative implementation strategies, and suggest promising applications of our learning-based optimization methodology to other systems beyond data structures. Finally, in Chapter 9, we summarize our contributions and results then conclude.

Chapter 2

Background and Related Work

This work introduces a new class of data structures for parallel programming called Smart Data Structures. Building upon design principles from self-aware computing, Smart Data Structures leverage online machine learning to optimize themselves for different machines, applications, and inputs automatically. This chapter compares Smart Data Structures to related work. First, it provides historical background on concurrent data structures, leading up to the development of adaptive data structures. Then, we describe prior approaches to adaptive data structures and analyze their limitations. Next, we summarize the contributions of Smart Data Structures over prior work. Finally, we survey other examples of machine-learning-based self-aware systems as evidence of the increasing importance of machine learning in systems.

2.1 Background on Concurrent Data Structures

In this section, we provide historical perspective leading up to the development of adaptive spin-locks and other adaptive concurrent data structures. We begin with the challenges and developments that drove the creation of various spin-locks. Then, we do the same for the developments that led to the creation of the Flat Combining data structures upon which some Smart Data Structures are built. Later, in Chapter 2.2, we will describe a number of methodologies that have been developed for designing adaptive data structures.

2.1.1 Historical Evolution of Spin-Locks

In parallel programming for shared memory machines, a lock is typically used as a mechanism to limit access to a region of multi-threaded code called a *critical section*. The lock guarantees that only the thread that holds the lock at a given time can execute the critical section. Typically, the critical section, and only the critical section, references particular shared resources or shared memory locations, and the lock coordinates concurrent access to these resources or variables through *mutual exclusion*. The lock is used in a cycle of four computation phases: acquiring the lock, executing a critical section, releasing the lock, then performing the main body of computation.

Because they are important for parallel programming, variety of different lock types have been developed. Depending on the algorithms used for acquiring and releasing the lock, different types of locks have three defining properties: their protocol, their wait strategy, and their lock acquisition scheduling policy (summarized in Figure 2-1).

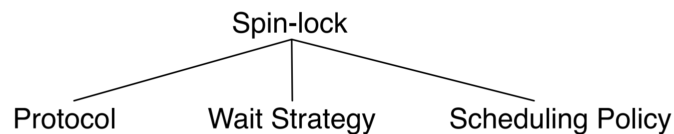


Figure 2-1: Properties of a Spin-Lock Algorithm

The protocol is the synchronization mechanism that the lock uses to guarantee mutual exclusion so that only one thread can hold a lock at a time. Typical mechanisms include global flags, counters, or distributed queues that locks manipulate using hardware-supported atomic synchronization primitives such as compare-and-swap or test-and-set. The wait strategy is the action that threads take when they fail to acquire the lock such as blocking, spinning, or spinning with backoff to reduce polling. Lastly, the lock acquisition scheduling policy determines which waiter should go next when threads are contending for the lock. For most locks, the protocol implies a particular fixed lock acquisition scheduling policy.

One popular lock type in multiprocessor and multicore applications is the spin-

lock. Spin-locks are named for their wait strategy: while threads are waiting for the lock to become free, they poll the lock variable in a tight loop called a *spin loop*. Spin-locks are well-suited for these applications because locks are generally held for short periods of time, and the spinning strategy is less costly than blocking and switching contexts.

Over the years, there have been many advances in spin-lock protocols to optimize performance and solve various challenges. One of the first spin-locks was the test-and-set lock. In the test-and-set lock, threads spin on a global shared variable which implements the lock. To acquire the lock, they must atomically test the value and successfully transition the value from 0 to non-zero. At small scales, the test-and-set lock proved remarkably efficient. However, its key deficiency is that it scales poorly under high lock contention because threads continuously execute costly synchronization primitives on the global lock variable which generate many cache invalidations and large amounts of bus or network traffic [29].

To help reduce the overhead, the test-and-test-and-set lock was invented. The basic idea was to prune execution of synchronization primitives by only performing them when the primitive was expected to succeed. Test-and-test-and-set uses less expensive non-atomic reads to determine when the lock is free. At that time, threads attempt the synchronization primitives. While these locks reduced overall bus or network traffic, they still suffered from large amounts of invalidations when the lock became free. When locks were held for short periods of time, synchronization primitives were still frequently executed and the overheads were still significant. Various other derivatives of the test-and-set lock were also proposed, including a version with backoff to reduce polling.

The scaling limitations of the test-and-set lock and its derivatives inspired the creation of scalable lock protocols based on distributed queues. In these "queue locks," waiters spin on local variables instead of global shared variables [29]. Popular queue locks include the Mellor-Crummey and Scott lock (the MCS lock) [31], the CLH variant on the MCS lock [27], and a more recent queue lock with various improvements called QOLB [23].

Unfortunately, queue locks were not without deficiencies either. First, due to the bookkeeping overhead of maintaining the queue, queue locks could not outperform test-and-set locks at small contention scales. This placed the burden of choosing the proper spin-lock algorithm on the programmer. Second, various spin-locks – but queue locks in particular – were prone to poor performance when the number of threads (or processes) exceeded the number of available cores. The problem stems from context switches when one thread spins, waiting for action from another thread that was swapped out. This problem led to various strategies for improving interactions between locks and the kernel scheduler to avoid context switches an inopportune times [24].

While much of the work in spin-locks focused on advancing spin-lock protocols, there were few efforts to advance lock acquisition scheduling policies. The few works that do exist are important predecessors to our work on Smart Locks because they are the first hints at the benefits of lock acquisition scheduling.

One example of work in lock acquisition scheduling is the write-biased readers-writer lock [30]. It enables concurrent read access and exclusive write access, prioritizing writers over readers. Another example is the priority lock [22]. Priority locks explicitly prioritize lock holders and were developed for database applications where transactions have different importance. They present challenges such as priority inversion, starvation, and deadlock, and are a rich area of research [39]. NUCA-aware locks are another example [36]. They were developed to improve performance on NUCA memory systems by releasing locks preferentially to near neighbors to improve locality. The adaptive lock acquisition scheduling policy in Smart Locks can learn to use these policies when beneficial, automatically.

Table 2.1 summarizes the various spin-lock algorithms we have discussed, detailing their protocol mechanisms, lock acquisition policies, scalability, and the contention levels for which they can be used most effectively. Reactive locks and other adaptive locks are discussed in Chapter 2.2.1. Reactive locks are closest related work to Smart Locks: they attempt to adapt protocols at runtime to use the best lock for the given contention level.

Table 2.1: Summary of Lock Algorithms

Algorithms	Protocol Mechanism	Policy	Scalability	Target Scenario
TAS	Global Flag	Pseudo-Random	Not Scalable	Low Contention
TASEB	Global Flag	Pseudo-Random	Not Scalable	Mid Contention
Ticket Lock	Two Global Counters	FIFO	Not Scalable	Mid Contention
MCS	Distributed Queue	FIFO	Scalable	High Contention
Priority Lock	Distributed Queue	Arbitrary	Scalable	Asymmetric Sharing Pattern
Reactive	Adaptive (not priority)	Adaptive (not arbitrary)	Scalable	Dynamic (not asymmetric)
Smart Locks	Adaptive (w/ priority)	Adaptive (arbitrary)	Scalable	Dynamic (w/ asymmetry)

2.1.2 Historical Evolution of Flat Combining

There are two predominant implementation strategies for concurrent data structures: lock-based designs and designs based on atomic synchronization primitives. Lock-based data structures came first and are based on a simple mechanism for ensuring correctness despite concurrency: they use a lock to restrict access to shared data within the data structure such that, at any given time, only the thread holding the lock can access the data.

The simplest lock-based designs use the lock as a coarse-lock around all shared data. One drawback of this approach is that it serializes all concurrent accesses to the data when they could potentially be completed in parallel. Amdahl famously showed that such serialization ultimately limits the scalability of parallel programs.

To reduce serialization and improve scalability, lock-based concurrency underwent a series of refinements. Programmers started building data structures with finer-grained locks, logically partitioning the data and locking each partition independently. This avoided serialization unless threads were attempting to access data from the same partition. While this helped improve scalability, lock-based data structures still had the key deficiency that, in the worst case, threads may block for indeterminate periods of time before the lock becomes available.

Data Structures based on atomic synchronization primitives were built, in part, to address these issues. Atomic synchronization primitives are hardware instructions that read, modify, and write a memory location in a shared cache line in a single, uninterruptible operation. Examples include test-and-set, fetch-and-add, and compare-and-swap instructions. These instructions temporarily lock individual cache lines, and thus permit concurrent modification of data in data structures so long as the data is on different cache lines.

Unfortunately, because of the concurrent semantics of atomic synchronization, data structures based on these primitives are incredibly complex to design and debug. Furthermore, algorithms of this type tend to require additional bookkeeping not seen in simpler lock-based designs that cause their overheads to be higher than lock-based designs at small scales. In other words, their scalability comes at the cost of increased complexity and increased overheads at small scales.

Until recently, programmers largely assumed that the scalability limitations of serialization in lock-based designs would be more significant than the overheads in data structures based on atomic primitives. Thus, the prevailing wisdom has been that data structures based on atomic primitives are higher performance. However, recent studies have shown that the overhead of atomic synchronization primitives is becoming increasingly expensive on multicores as more cores are added [12], and that shared memory systems are suffering from large amounts of cache coherence traffic as cache lines ping-pong between cores due to atomic operations.

This has led to the recent development of Flat Combining data structures by Hendler et al. [12]. Hendler et al. have shown that the Flat Combining algorithm significantly improves performance over prior data structures by eliminating the majority of synchronization overheads. Furthermore, because the Flat Combining algorithm is lock-based, it shows that what was thought to be a fundamental deficiency of lock-based data structures – serialization of accesses – can be significantly mitigated.

Flat Combining uses a lock as a coarse lock around the data structure but avoids much of serialization of prior lock-based designs by allowing threads, when they get the lock, to learn about the operations that other threads wish to perform and perform them on their behalf. The principle mechanism is a low-overhead publication list in which threads publish requests for operations. The key advantage of the design is that threads perform multiple data structure operations each time they get the lock. In contrast, conventional algorithms based on atomic primitives typically require one or more synchronization operations per data structure operation. Flat Combining's savings in synchronization overheads outweigh the serialization of the lock-based design and allows Flat Combining to significantly outperform prior art. In

Chapter 6.2.2, we show Flat Combining data structures outperform all prior art up to 64 threads.

Due to their lock-based design, however, Flat Combining data structures would still be susceptible to blocking for indeterminate periods of time. For example, if the thread that holds the lock gets swapped out, no data structure operations will be completed until the thread gets swapped back in. Luckily, in the years since data structures based on atomic synchronization primitives were first built, various efforts have been made to allow "scheduler conscious" locks to communicate with the scheduler and avoid preemption at inopportune times [24]. The Flat Combining implementation [13] actually uses a similar technology and avoids blocking for indeterminate periods of time.

Another motivation for the design of Flat Combining data structures was to optimize cache locality [12] and minimize shared memory coherence traffic. Because threads perform multiple operations within each lock, more successive operations on the data structure occur within the same thread, increasing the chances that successive operations will access memory already in the cache and already in the appropriate shared memory coherency state.

There have been several complementary efforts in data structure design to optimize utilization of memory storage, bandwidth, and access patterns. For example, the C-store relational database developed at MIT [6] organizes database information strategically so that it can be compressed and manipulated directly in the compressed format. Cache oblivious algorithms [35] are an example of an attempt to optimize memory access patterns. They are typically recursive algorithms that divide problems into smaller and smaller subproblems until subproblems become so small they fit into the cache regardless of the cache size and avoid cache capacity misses.

The Flat Combining data structures expand upon these ideas and optimize a new aspect of memory system performance: synchronization overheads. Moreover, the memory system optimizations that C-Store and cache oblivious algorithms use are complementary to optimizing synchronization overheads; they could be combined with Flat Combining. For example, an interesting extension to Flat Combining would

be to utilize compression within the data structures. Then, Flat Combining would improve memory bandwidth utilization in addition to locality and synchronization overheads. Today, Flat Combining provides state-of-the-art performance for various important multicore data structures and is a promising platform for optimization.

2.2 Adaptive Data Structures

In the ongoing quest for increased performance, various efforts have been made to design adaptive concurrent data structures. In Chapter 2.1.1, we saw that the best spin-lock algorithm depends on the contention level. Similarly, the best choice of concurrent data structures and/or the best choice of knob values within a given data structure may depend on system conditions, inputs, or other factors. The goal of adaptive data structure techniques is to tune data structure choices or knobs values. Two of the most well-known approaches to doing this are auto-tuned libraries and adaptive programming frameworks. This section describes related work in these areas.

2.2.1 Auto-Tuned Libraries

Some of the best known auto-tuned libraries are FFTW, PHiPAC, and ATLAS [11, 3, 20]. FFTW is a library for computing discrete Fourier Transforms for signal processing, PHiPAC is a fast matrix multiplication library, and ATLAS is a self-optimizing implementation of the famous BLAS (Basic Linear Algebra Subprograms) application programming interface. STAPL is another example. STAPL is an extensible library of general-purpose, parallel C++ data structures and algorithms.

These and other typical auto-tuned libraries like Olszewkski's adaptive sorting library [33] select from a repertoire of data structure and algorithm implementations at runtime based on install-time benchmarks on the machine, data sampling, and the input size. Different implementations in the repertoire are typically optimized for different cache blocking, loop unrolling, and data partitioning strategies. For signal processing, linear algebra, scientific computing, and sorting, respectively, these works have demonstrated significant improvements.

Auto-tuning has also been applied to spin-locks. To address the issue of choosing the best lock protocol for a given contention scale, Lim and Agarwal developed reactive locks [25]. Through the use of consensus objects, reactive locks adapt between test-and-set and queue locks, depending upon the lock contention level. Thresholds for determining when to switch are computed via install-time characterizations for a given machine.

The principal limitation of these prior works is that, while they may adapt to different architectures and runtime conditions like input size, they rely on thresholds computed statically at compile- or install-time. The problem is that static characterizations may poorly reflect realistic runtime conditions. One example is virtualized environments where an application may migrate from one machine to a different machine with a different architecture during the lifetime of the program. This completely shifts the tradeoffs that determine which algorithm and knob settings are best. Smart Data Structures, on the other hand, are robust to this sort of unexpected shift and subtler shifts in tradeoffs. Smart Data Structures take an online, adaptive approach to identifying and balancing tradeoffs through Reinforcement Learning.

Among the auto-tuned libraries, STAPL is an important predecessor to Smart Data Structures because it uses machine learning – albeit a very different form of learning than Smart Data Structures employ. Offline learning is used when STAPL is installed to aid in the analysis of computing static thresholds for algorithm decisions. Their approach is model-based: programming experts supply architectural and other detailed parameters upon which performance depends. Then, decision tree learners are trained, using carefully constructed training examples in attempt to avoid overfitting, to build up models for a given parameter's effect on performance. The models are then used to compute decision thresholds. Smart Data Structures, on the other hand, use online learning, and a major virtue of our online Reinforcement Learning algorithm is that it is model-free. Our view is that the skyrocketing complexity of multicore machines has made it too difficult to build accurate models.

Another important predecessor to Smart Data Structures is recent work in self-tuning reactive locks [15]. This work has shown that the dependence on static thresh-

olds in reactive locks can be eliminated by tuning the threshold online via a competitive algorithm. Unfortunately, another major deficiency of the reactive lock remains: the reactive lock makes no attempt to optimize the lock acquisition scheduling policy. Related work in write-biased readers-writer locks [30] and NUCA-aware locks [36] have hinted that the lock acquisition scheduling policy can be an important factor in performance. Indeed, our results in Chapter 7 will show that it can be used to significantly improve performance in heterogeneous multicores by optimizing access to shared resources and critical sections. To our knowledge, the Smart Lock is the first adaptive spin-lock to self-optimize its lock acquisition scheduling policy.

2.2.2 Adaptive Programming Frameworks

Adaptive Programming Frameworks are an alternative to auto-tuned libraries. Like auto-tuned libraries, they provide automatic tuning of data structures but do so at the programming language abstraction layer rather than the library abstraction layer.

An example of an adaptive programming framework is PetaBricks from MIT [1]. Petabricks is a programming language, compiler, and runtime library for building adaptive programs. Using the PetaBricks framework, programmers specify multiple implementations of each function or algorithm using a syntax that enables the compiler and runtime to make algorithm decisions and compose parallel algorithms.

One advantage of this work over prior work is that it can evaluate decision tradeoffs either offline or online. The online auto-tuner divides the cores in the system into two halves and duplicates the computation. At each step, one half of the cores use a safe configuration and race an experimental configuration in the other half. The result from the faster configuration wins and is used. If an experimental configuration wins, its improvements are gradually added to the safe configuration. Over time, the safe configuration gets faster and faster. Ansel et al. show that the benefits of running one half of the cores with a faster algorithm can sometimes be greater than 2x – enough to outweigh the cost of giving up one half of the cores for experimentation [1].

While adaptive programming frameworks are promising, our view is that they can be impractical because they require adopting new programming languages. Histori-

cally, the wide-spread adoption of new programming languages has been hampered by considerable inertia. We have implemented Smart Data Structures in C++ because, thirty years after it was invented, C++ remains one of the most predominant languages. That said, the ideas in Smart Data Structures are general and language-independent. It would be interesting to apply the ideas in Smart Data Structures to other languages and to PetaBricks. For example, the PetaBricks runtime auto-tuner might be alternatively implemented with an online learning engine.

2.3 Smart Data Structures Contributions Summary

In this section, we detail the key contributions of Smart Data Structures over previous work. To summarize:

1. To our knowledge, Smart Data Structures are the first adaptive data structure library to successfully use online machine learning.

2. To our knowledge, Smart Locks are the first adaptive spin-locks to systematically and dynamically schedule lock acquisitions to optimize access to shared resources and critical sections.

3. Smart Data Structures are designed around a more reliable closed-loop, dynamic optimization strategy than the static decision thresholds used in prior auto-tuned libraries.

4. Smart Data Structures provide a robust, model-free learning implementation which enables the programmer to extend the library to new systems (including virtualized environments) with none of the complexity of building accurate performance models.

2.3.1 Novel Online Learning Adaptation Methodology

While various auto-tuning libraries have experimented with offline machine learning to statically tune the library, to our knowledge, Smart Data Structures is the first

library to apply online machine learning successfully. Our experiments based on our prototype implementation (Chapters 5, 6, and 7) and our previous publications in this area [9, 8] show, constructively, that online machine learning is an effective optimization strategy for adaptive data structures. Further, we contribute a learning algorithm based on Policy Gradients Reinforcement Learning that is simultaneously high performance and efficient enough for online use in adaptive data structures (see Chapter 4.3).

2.3.2 Novel Lock Acquisition Scheduling Optimizations

While previous adaptive spin-locks such as reactive locks have innovated in the area of protocol adaptation, to our knowledge, Smart Locks are the first adaptive spin-lock to systematically optimize the lock acquisition scheduling policy. Our experiments in Chapter 7 and prior publications [9] show empirically that lock acquisition scheduling is an important technique for improving program performance on heterogeneous multicores because it can be used to preferentially allocate shared resources and/or critical sections to faster cores. Doing so reduces their wait times. Since fast cores can perform more work per unit time than slower cores, overall performance increases.

2.3.3 Closed-Loop, Dynamic Decision-Making

While previous auto-tuned libraries and programming frameworks may adapt to different architectures and runtime conditions, they rely on thresholds computed statically at compile- or install-time to do so. Unfortunately, static characterizations may poorly reflect realistic runtime conditions. For example, applications running in virtualized environments may migrate among different machines during the lifetime of the program, completely shifting the tradeoffs that determine the best algorithm and knob settings. Further, on a given machine, newer frequency scaling technologies like thermal throttling and Intel's Turboboost® overclocking technology, can alter the tradeoffs dynamically. Even in the absence of virtualization and frequency scaling, multi-process environments have fundamentally dynamic tradeoffs that static thresh-

olds cannot account for. In multi-process environments, different applications can run at the same time, and different combinations of applications will interfere and compete differently for hardware resources.

Smart Data Structures monitor tradeoffs online and take a closed-loop approach to optimization: through Reinforcement Learning, they are robust and reactive to dynamic changes in the system, application, or inputs that alter the best algorithm and knob settings over time.

2.3.4 Simplified Extensibility via Model-Free Learning

A major virtue of the online learning algorithm used in Smart Data Structures over prior offline learners is that our algorithm is model-free. When extending the library to new systems or data structures, this frees the programmer from having to supply specifications of the system such as architectural parameters, cache sizes and associativities, or details about the cache coherence protocol. Model-based systems are reliant on these specifications so that they can learn approximate machine performance as a function of these specifications and predict what implementations will maximize performance.

The problem is that the complexity of systems is skyrocketing and making it difficult to produce accurate specifications and models. Furthermore, many applications are moving to cloud computing environments where system specifications are a) unavailable because they are proprietary or b) unpredictable because the hardware is virtualized. The Policy Gradients Reinforcement Learning approach used by Smart Data Structures learns how to act without prior information about the environment. Furthermore, it adapts its actions when the environment changes. In our design, programmers need only specify what data structure knob to optimize, not a procedure for how to optimize it.

2.4 Additional Learning-Based Self-Aware Systems

Now that we have looked at background on concurrent data structures and compared Smart Data Structures to related work, we survey other examples of self-aware systems based on machine learning. Researchers have built self-aware systems based on learning to address a variety of important problems in multicores and clouds spanning resource allocation, scheduling and load balancing, and libraries and optimization. Our view is that these pioneering works are evidence that machine learning will play an essential role in the development of future systems.

2.4.1 Resource Allocation

Ipek et al. apply Reinforcement Learning and neural nets to multicore resource management [19, 4]. They build self-optimizing hardware agents that adapt the allocation of critical resources according to changing workloads. Hoffman et al. [18] utilize Reinforcement Learning and control theory in a software runtime framework which manages application parallelism to meet performance goals while minimizing power consumption. Tesauro et al. use Reinforcement Learning in the context of data centers to make autonomic resource allocations [38]. Wentzlaff et al. from MIT are investigating various applications of machine learning to the operating system that they are designing for multicores and clouds [40].

2.4.2 Scheduling and Load-Balancing

Fedorova et al. extend learning to heterogeneous multicores to coordinate resource allocation and scheduling [10]. Their system uses Reinforcement Learning to produce scheduling policies that balance optimal performance, core assignments, and response-time fairness. Whiteson and Stone use Q-learning to improve network routing and scheduling [41].

2.4.3 Libraries and Optimization

Work in libraries and optimization has also benefited from learning-based self-aware computing. Coons et. al apply genetic algorithms and Reinforcement Learning to compilers to improve instruction placement on distributed microarchitectures with results rivaling hand-tuning [5]. Finally, Jimenez and Lin apply the perceptron algorithm to dynamic branch prediction and propose a scheme that substantially outperforms existing techniques [21].

Chapter 3

Smart Data Structures Design

Smart Data Structures are a new class of self-aware parallel data structures that self-tune themselves automatically through a novel methodology based on online machine learning. Through learning and automatic tuning, Smart Data Structures relieve programmers of the burden of hand-tuning for the best performance across different machines, applications, and inputs.

This chapter details the design of Smart Data Structures. We begin in Chapter 3.1 by describing the architecture of Smart Data Structures, including our implementation strategy and optimization methodology. Next, in Chapter 3.2 we describe the contents and design of our open source library of Smart Data Structures. Later, in Chapter 4, we will describe the challenges of fine-grained optimization of data structures and how our learning architecture and learning algorithm address them.

3.1 Smart Data Structures Architecture

The goal of the Smart Data Structures architecture is to facilitate high-performance and robust self-optimization while simultaneously making Smart Data Structures easy to use in applications. Chapters 3.1.1 and 3.1.2 present our implementation strategy and optimization methodology. We show how Smart Data Structures achieve this goal and motivate our architectural design decisions by contrasting Smart Data Structures with standard data structures and standard optimization methodologies.

3.1.1 Implementation Strategy

For ease of use in applications, Smart Data Structures are designed to be drop-in, self-optimizing replacements for standard data structures. Smart Data Structures are implemented by layering online learning on top of standard data structures. While preserving the interfaces, Smart Data Structures internally adapt and optimize data structure components via learning.

To illustrate, we review the anatomy of a standard data structure and contrast it with the anatomy of a Smart Data Structure. Figure 3-1 shows the components of a standard data structure. Standard data structures consist of data storage, an interface, and algorithms. The storage organizes the data, the interface specifies the operations that threads may apply to the data to manipulate or query it, and the algorithms implement the interfaces while preserving correct concurrent semantics.

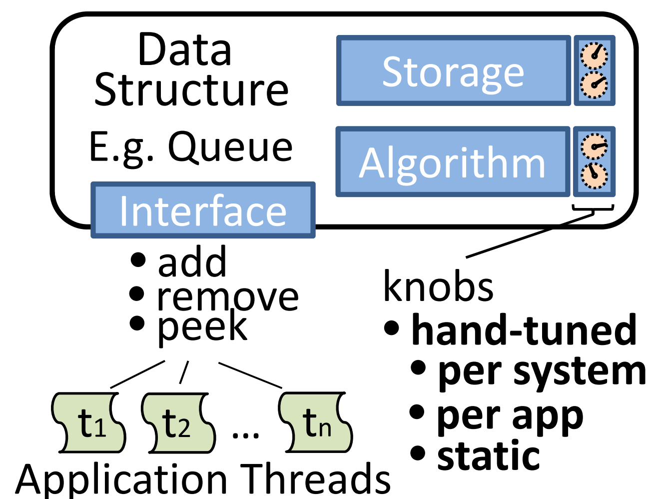


Figure 3-1: The Anatomy of Standard Parallel Data Structures. Data structures consist of storage, interfaces, and algorithms. The storage organizes the data, the interfaces specify how the data can be accessed and manipulated, and algorithms implement the interface operations while preserving correct concurrent semantics. Knobs parameterize the behavior of the storage and algorithms.

Storage and algorithms are often controlled by *knobs*: thresholds or other parameters that program implementation behaviors and heuristics. Knobs are typically configured via one-size-fits-all static defaults provided by the library programmer. When the defaults perform sub-optimally, programmers must hand-tune the knobs.

This is typically done through a) trial and error which can increase development time and/or through b) special-casing in the code which can reduce readability. Though often necessary, runtime tuning is typically ignored by the programmer due to its complexity.

Figure 3-2 contrasts Smart Data Structures with standard data structures. Smart Data Structures preserve the same interfaces, storage, and algorithms. The difference is that Smart Data Structures augment standard data structures with an online learning engine that automatically and dynamically tunes the knobs to optimize storage and algorithm behaviors. Through learning, Smart Data Structures balance complex tradeoffs to find ideal knob settings and adapt to changes in the system or inputs that affect these tradeoffs.

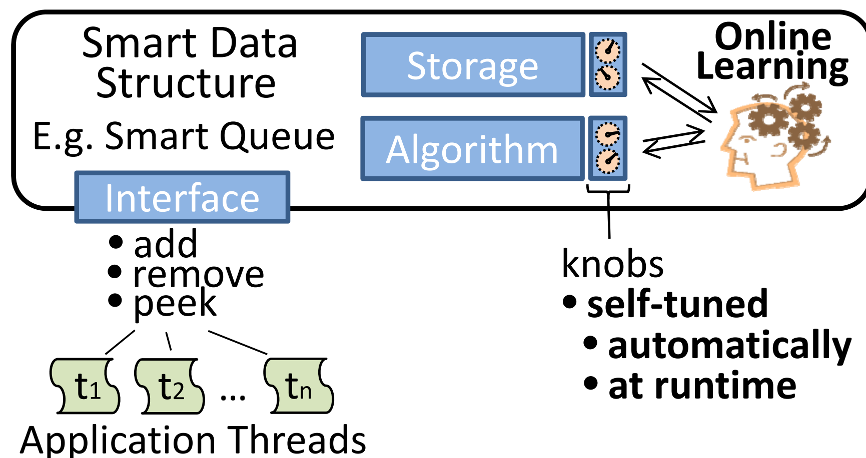


Figure 3-2: The Anatomy of Smart Data Structures. Smart Data Structures augment standard data structure interfaces, storage, and algorithms with online machine learning to internally optimize the knobs that control their behavior.

3.1.2 Optimization Methodology

The use of an online learning engine for the dynamic optimization of data structures is an important departure from the methodologies used in prior work that enables Smart Data Structures to provide a more robust framework for optimization.

There have been a variety of related works in adaptive data structures. Among the most well-known are several auto-tuned signal processing and linear algebra libraries:

FFTW, PHiPAC, and ATLAS [11, 3, 20]. While these prior works are able to adapt to different machine architectures and runtime conditions like input size, they typically base adaptation decisions on thresholds computed during compile-time or install-time characterization. The problem is that such characterizations can poorly reflect realistic runtime conditions in modern systems.

Consider systems with frequency scaling, for example. Frequency scaling technologies like thermal throttling or Turboboost® from Intel® can dynamically under- or overclock some subset of the processors, altering the machine's effective performance and substantially affecting the tradeoffs that determine which algorithm and/or knob settings are best. Furthermore, vanilla multi-process environments can have complex runtime conditions as well. They can have unpredictable effective performance because applications run alongside other applications, interfering and competing in different ways for important resources like communication and memory bandwidth.

Smart Data Structures account for these complexities by taking on online, closed-loop approach to optimization decisions. They continually collect performance feedback to infer dynamic information about the system and performance tradeoffs. It is through the use of online learning that Smart Data Structures are able to weigh complex tradeoffs at runtime. Through learning, Smart Data Structures adapt and react to changes in the system, application, or inputs to achieve the best performance available.

Another benefit of the online learning optimization methodology in Smart Data Structures is that it generalizes to the optimization of systems beyond data structures. While this work focuses on a case study of optimizing data structures, we have taken care to design our knob abstraction, our library, and our learning algorithms (as we will see in Chapter 4), so that they can be used in other systems. Our long-term vision is to apply this work toward the optimization of knobs in systems such as cloud resource managers, OS schedulers, and multicore shared memory systems as well (see Chapter 8.5 for further details).

3.2 Smart Data Structures Prototype Library

This work develops an open-source library of Smart Data Structures for popular parallel programming needs. It is available [7] on github under a GPL license. The goal of the library is to constructively demonstrate that online learning is a high performance and flexible framework for the automatic, fine-grained optimization of data structures in the face of complex, time-varying tradeoffs.

This section, describes the contents and design of the Smart Data Structures prototype library. First, we list the supported data structures and describe their significance to parallel programming. Then, we discuss the standard interfaces used by Smart Data Structures to facilitate easy integration into applications. Next, we detail the implementation of the prototype data structures, describing the significance of the knobs that we optimize within them. Finally, we discuss library facilities for easy porting and extensibility.

3.2.1 Supported Data Structures

The Smart Data Structures library includes a variety of data structures to support popular parallel programming models and orchestrate sharing of data among threads. The library provides a Smart Queue for use in global work queue, work-stealing, and pipeline-parallel programming models. It provides a Smart Skip List for concurrent sorting or sharing lists of data. A Skip List is comparable to a binary search tree in that it support insertion and search with the same asymptotic complexity but the Skip List uses a simpler implementation based on a hierarchy of increasingly sparse linked lists. The library also provides a Smart Pairing Heap for concurrent sorting. Both the Smart Skip List and Smart Pairing Heap can be used as high-performance priority queues in prioritized work queue programming models or for event-based scheduling. Finally, the library provides a Smart Lock for use in lock-based programming models with critical sections to schedule when threads get access to critical sections.

Potential applications of these data structures range from accelerating parallel graph search algorithms like Dijkstra's single-source shortest path algorithm (Smart

Queues), to prioritizing network traffic such as VOIP streams for better quality of service (Smart Skip Lists), to reducing discrete event simulation times in multicore architecture simulations (Smart Pairing Heaps), to accelerating work queue programs on heterogeneous multicores by scheduling work stealing (Smart Locks).

The Smart Data Structures prototype provides facilities for extending the library to additional data structures as well. As Chapter 3.2.4 will show, these facilities make it easy to add Smart Data Structures to the library. We have plans to expand the library to include a Smart Stack and a Smart Distributed Hash Table in the future.

3.2.2 Application Interfaces

The Smart Data Structures in the library prototype are designed to be drop-in, self-optimizing replacements for standard concurrent data structures. They are non-blocking data structures with standard interfaces. They are written in C++ for shared memory C / C++ applications. C interfaces are provided as well for mixing with other programming languages. Thus, from the perspective of an application developer, integrating a Smart Data Structure into an application is as simple as integrating a standard data structure: the developer includes a library header file and is provided standard object-oriented interfaces.

Though Smart Data Structures internally use machine learning to optimize their implementation, they preserve the same standard interfaces at all times. The Smart Queue, Skip List, and Pairing Heap provide *add*, *remove*, and *contains* methods to insert, retrieve, and search for elements. Their interfaces provide additional methods to track statistics: *size* and *empty*. The Smart Lock is a learning-enabled spin-lock with standard *lock*, *unlock*, and *trylock* interfaces with extensions for timeout and abort conditions to facilitate complex uses.

3.2.3 Data Structure Implementations

This section details the implementation of the Smart Queue, Smart Skip List, Smart Pairing Heap, and Smart Lock data structures in the Smart Data Structures proto-

type. We begin with an implementation overview. Then, we detail the Smart Queue, Skip List, and Pairing Heap together because they are based on a common algorithm. Finally, we detail the Smart Lock.

Implementation Overview

In general, different Smart Data Structures are built on top of different base algorithms and thus have different types of knobs. Different Smart Data Structures may also use different online learning algorithms to optimize their knobs. As we will see, the types of knobs in the prototype Smart Data Structures range from discrete-valued knobs to permutation orderings. So far, all Smart Data Structures in the prototype have used Reinforcement Learning as the online learning algorithm.

The Smart Queue, Skip List, and Pairing Heap build on top of the recent Flat Combining algorithm [12]. They augment Flat Combining base data structures with an online Reinforcement Learning engine. Through Reinforcement Learning, they continually optimize a performance-critical, discrete-valued knob in the Flat Combining algorithm called the *scancount*. In Chapter 5.2 we will motivate our decision to build upon Flat Combining base data structures by showing that Flat Combining outperforms the best prior algorithms over a range of scenarios.

The Smart Lock, our self-tuning spin-lock, uses Reinforcement Learning as well. However, Smart Locks use learning to optimize a permutation knob rather than a discrete-valued knob. In a Smart Lock, the permutation knob specifies the order and relative frequency with which different threads get the lock when contending for it. By continually adjusting the permutation order, the Reinforcement Learning engine schedules lock acquisitions. At any given time, the permutation order specifies the current schedule. A Smart Lock implements the schedule by building on top of a priority lock and interpreting the schedule as priorities. In other words, the Reinforcement Learning engine in the Smart Lock dynamically programs the priorities in a priority lock. Smart Locks are used in programs to protect critical sections so this optimization has the effect of scheduling access to the critical section. In Chapter 7, we will show that scheduling access to critical sections can significantly accelerate

work queue programs on heterogeneous multicores.

Smart Queues, Skip Lists, and Pairing Heaps

The Smart Queue, Skip List, and Pairing Heap augment a recently developed data structure algorithm called Flat Combining [12] with an online Reinforcement Learning Engine. In this section, we describe the Flat Combining algorithm and a performance-critical, discrete-valued knob in the algorithm called the *scancount*. We will motivate our auto-tuning of the scancount by describing how it affects data structure performance.

Figure 3-3 shows the Flat Combining data structure design. Flat Combining data structures are non-blocking, shared memory data structures that consist of a *publication list*, a test-and-test-and-set lock, a scancount, and a serial data structure. The algorithm uses the lock as a coarse-lock around the serial data structure and the publication list as a low-overhead mechanism for broadcasting the operations that the threads wish to apply to the data structure. Threads overcome the serialization of lock-based concurrency by *combining*: performing not only their operation when they have the lock but also the published operations of the other threads.

The steps of the algorithm are numbered in Figure 3-3. Each thread has a record in the publication list. In Step 1, when a thread wants to perform an operation on the data structure, it publishes a request and any necessary arguments in its record. Next, in Step 2 the thread waits for the operation to complete, spinning locally on a field in its record. While spinning, the thread will periodically attempt to acquire the lock. If successful, the thread moves to Step 3 and becomes the *combiner*; otherwise it remains in Step 2 until its operation completes. In Step 3, the combiner reads the scancount, k. Finally, in Step 4, the combiner scans the publication list k times, each time looking for operations to perform and applying them. The combiner may merge operations before applying them to improve efficiency. As soon as a thread's operation is complete, the combiner writes to that thread's record to signal it. That thread stops spinning and it returns control to the application. After its k scans, the combiner releases the lock and likewise returns control to the application.

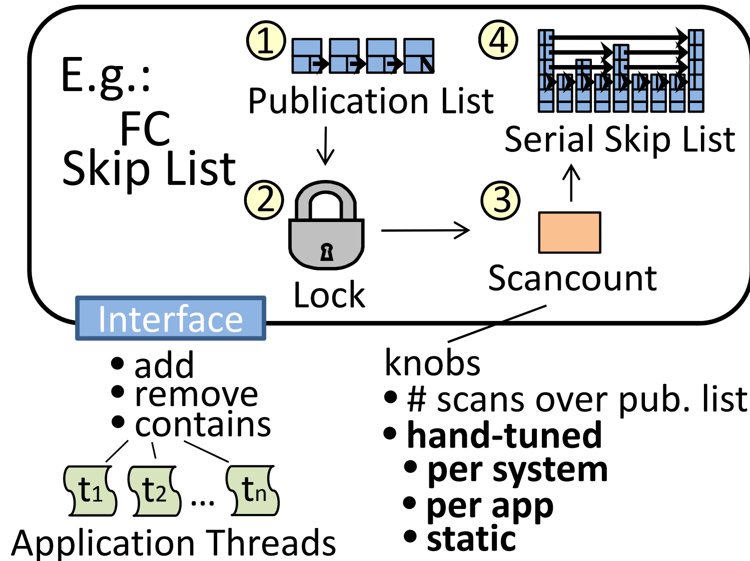


Figure 3-3: Flat Combining Data Structures. The Flat Combining Skip List is pictured. Flat Combining data structures consist of a *publication list*, a lock, a *scancount*, and a serial data structure.

Thus, the scancount dictates the number of scans combiners make over the publication list before returning control to the application. Adjusting the scancount for more scans provides opportunities to catch late-arriving requests and therefore perform more operations in each combining phase. This can improve synchronization overheads by reducing the rate at which locks must be acquired and can improve temporal locality and cache performance because more back-to-back data structure operations are performed by the same thread [12]. However, making more scans has the tradeoff that the latency of data structure operations can increase because threads remain the combiner for longer. Some applications are not affected by this latency, but we will demonstrate common application structures in Chapter 5 that are adversely affected. Increased latency can be particularly bad when the extra time spent combining is wasted because requests are not arriving quickly enough to keep the combiner busy.

In Flat Combining, the scancount is fixed to a static default value provided by the library. The Smart Queue, Skip List, and Pairing Heap significantly improve performance over Flat Combining by optimizing the scancount dynamically. As Figure

3-4 shows, they do this by augmenting Flat Combining with an online Reinforcement Learning engine which balances the tradeoffs to find the ideal scancount. Our experiments in Chapter 5 will demonstrate that, through learning, the Smart Queue, Skip List, and Pairing Heap can outperform the state-of-the-art Flat Combining data structures by up to 44% over a range of conditions. Furthermore, we will show that, through learning, these Smart Data Structures can readily adapt to rapid changes in the application workload that cause the ideal scancount to vary over time.

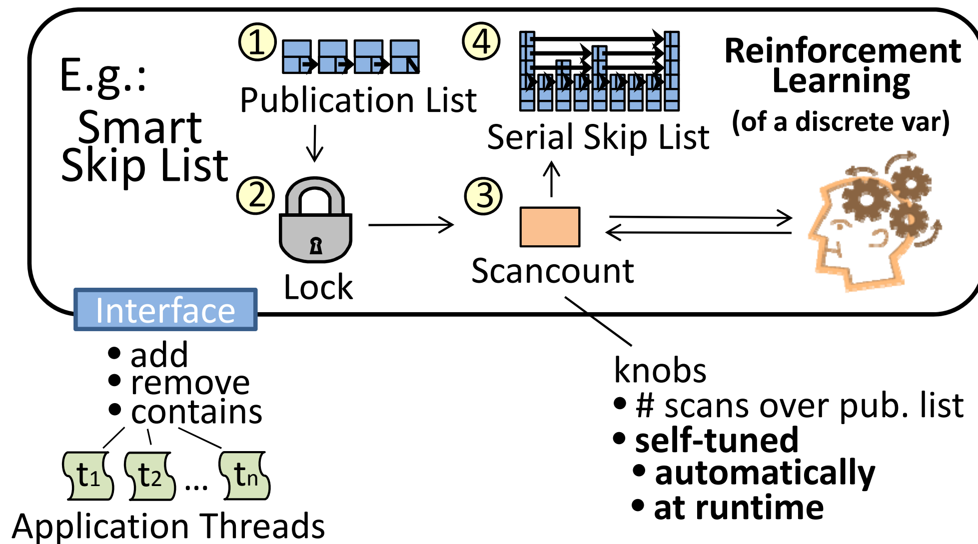


Figure 3-4: The Smart Queue, Skip List, and Pairing Heap. The Smart Skip List is pictured. These Smart Data Structures augment the Flat Combining algorithm with an online machine learning engine to optimize a performance-critical knob of the algorithm called the *scancount*.

Smart Locks

The Smart Lock is a shared memory spin-lock data structure for synchronization between threads. Applications use spin-locks to protect a region of multi-threaded code, called a *critical section*. The spin-lock guarantees that only the thread that holds the lock at a given time can execute the critical section. Typically, the critical section, and only the critical section, references particular resources or shared memory variables, and the spin-lock coordinates concurrent access to these resources or variables through mutual exclusion. Through a technique we call *Lock Acquisition Scheduling*,

the Smart Lock optimizes access to these resources or critical sections by intelligently scheduling access to the lock.

Smart Locks are based on priority locks and use the abstraction of priorities to schedule access to the lock. Each thread has a priority. Whereas thread priorities are usually statically programmed in a priority lock, Smart Locks augment a standard priority lock with an online Reinforcement Learning engine that dynamically programs thread priorities. By dynamically configuring the priorities, the learning engine controls the order and relative frequency with which contending threads will get the lock. To optimize priorities, the learning engine optimizes a permutation knob. The permutation knob specifies an ordering over the threads, and a thread's position in this ordering is its priority.

Figure 3-5 shows the steps of the priority lock algorithm. When a thread wants to get the lock, it reads its priority and adds itself to the wait queue. The wait queue is (conceptually) a priority queue, and the thread uses its priority as the key (with a minor modification) when it inserts itself. When the lock is free, the thread at the head gets it next. The others spin until they become the head of the queue.

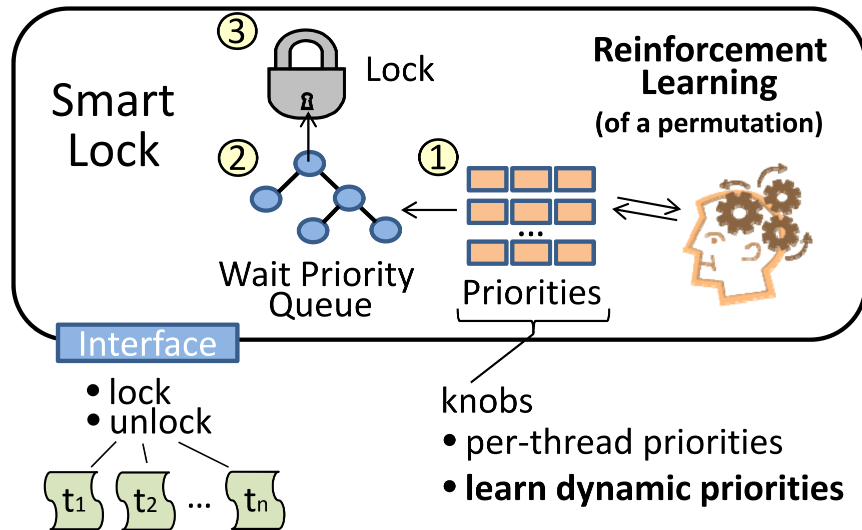


Figure 3-5

Our priority lock implementation supports a few additional complexities. First, spinning can be aborted if too much time elapses or if some other specified condition

is met. To abort, threads remove themselves from the wait priority queue. Second, a thread's priority can be updated by the learning engine while it is in the wait queue. If so, that thread will move to a new position in the queue. Third, as a performance optimization, our priority lock uses discrete priority levels – 32 or 64 levels, depending on the machine integer width. This allows us to implement the wait priority queue as a bit vector which we can atomically modify. Each bit signifies the occupancy of a different priority level, and threads can scan the bit vector efficiently to determine if they are the head of the queue. For applications with more than 64 threads, positions in the permutation ordering from the learning engine are mapped into priorities by quantizing the positions into 64 different priorities. In this case, the bits in the bit vector are multiplexed among multiple threads, with little effect on efficiency.

To summarize, the Smart Lock uses priority locks and learning to optimize the performance of an application by intelligently managing the schedule with which its threads access shared resources or variables. In general, the best lock scheduling policy varies depending on the application, and Smart Locks attempt to find the ideal policy for a given application, adapting it at runtime if necessary.

As Figure 3-6 illustrates, this is an important departure from prior spin-locks. Most spin-locks have fixed, static scheduling policies. For these spin-locks, the scheduling policy is an intrinsic property of their algorithm. For example, consider the test-and-set lock. In the test-and-set lock, when a thread wants the lock, it conceptually enters a wait set where it spins on a common shared variable until it gets the lock. All members of the set attempt to atomically test the variable for zero and transition it to non-zero value if zero. The test-and-set algorithm makes no effort to order or schedule which contending thread will get the lock next. Its scheduling policy is pseudo-random. In practice, however, non-uniformities in cache coherent shared memory systems make it so that threads running in cores near the current lock holder are more likely to win the race for the lock.

Another popular lock is the scalable Mellor-Crummey and Scott queueing lock (the MCS Lock) [29]. Unlike the test-and-set lock, the MCS Lock schedules the order in which threads will get the lock. Its scheduling policy is a fair, first-in-first-out

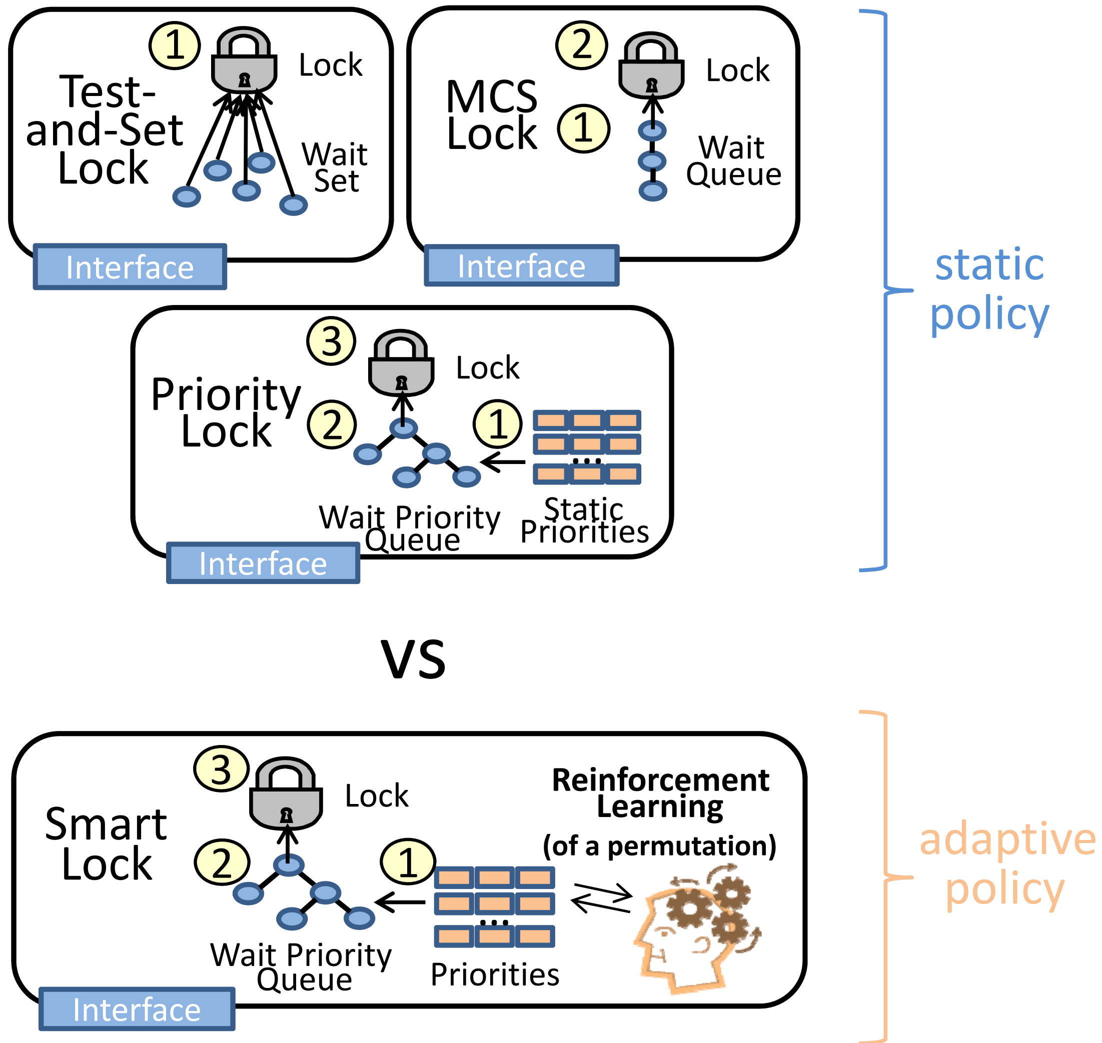


Figure 3-6

(FIFO) policy. When a thread wants the lock, it enters a wait queue. Conceptually, the head of the queue spins on the lock and will become the next lock holder when the lock is free. This is implemented by having all threads in the queue spin until signaled. When the current lock holder releases the lock, it signals the node after it in the queue. If the queue was empty when a thread inserts itself, it is the head and gets the lock.

The only prior lock we are aware of that does not rely on a fixed scheduling policy is the priority lock. Though its policy is not fixed, it is still usually statically programmed. When a thread wants to acquire the lock, it reads its priority and uses it as a key to insert itself into a wait priority queue. Only the head of the queue can

attempt the lock, and the head will become the next lock holder.

Unlike its predecessors, the Smart Lock uses a dynamic, adaptive scheduling policy. Through lock acquisition scheduling, the Smart Lock attempts to learn the best policy for the application. The major virtue of lock acquisition scheduling is that it can automatically introduce desired application-specific biases into the lock. In real-time systems, Smart Locks can learn biases to improve real-time guarantees. In clouds, they may be used to improve resource allocation fairness. In heterogeneous multicores, they can minimize lock latency for critical threads. In Chapter 7, we show that Smart Locks can accelerate work queue programs on heterogeneous multicores as well. Through lock acquisition scheduling, Smart Locks optimize access to critical sections in the work queue code concerning work stealing and, thus, intelligently schedule work stealing.

3.2.4 Library Extensibility and Other Features

This section describes features in the Smart Data Structure prototype library for portability and extensibility.

To facilitate easy porting to new architectures, the Smart Data Structures prototype abstracts architectural and memory model dependencies through a portability layer. The library has been tested most thoroughly on x86_64® systems but it should be trivial to port it to 32-bit x86® systems. Porting to SPARC® systems is expected to be relatively simple as well, though a few lingering memory barriers for accommodating its memory model differences may have been overlooked. We are told that the library has been successfully ported to Tilera® TileGX® systems as well.

To facilitate the expansion of the library to new data structures, the learning engine in our library supports the optimization of a variety of different knob types. Supported knob types range from permutations to Gaussian distributions to discrete values to binary values. Furthermore, our learning framework provides a flexible interface for joint-optimization of multiple knobs (of arbitrary type) for more advanced optimizations. Support for joint-optimization also enables optimization across multiple components. See Chapter 8.5 for promising applications of joint optimization.

In addition to the portability layer and learning framework, the library also includes a variety of performance monitoring frameworks. As we will see in Chapter 4, performance monitoring frameworks supply reward to the learning engine and drive optimization within Smart Data Structures. Chapter 6.2.5 will evaluate several of the performance monitoring frameworks included in the library.

Chapter 4

Learning Design and Challenges

4.1 Design Challenges

This work introduces a new class of data structures called Smart Data Structures which leverage online machine learning to optimize themselves at runtime. The overriding goal of our design is to maintain ease of use in applications while providing the highest performance available across a variety of different machines, applications, and workloads. The Smart Data Structures design must address three major challenges to meet this goal. Namely, Smart Data Structures must:

1. Measure performance in a reliable and non-intrusive way so that optimizations are relevant to the goals of the application
2. Adapt knob settings with fine-grained frequency and latency so as not to miss windows of opportunity for optimizations
3. Accurately identify the best long-term knob settings

The Smart Data Structures design addresses these challenges through the choice of learning architecture and the choice of online learning algorithms. This chapter describes the learning architecture in Chapter 4.2 and provides a mathematical treatment of the learning algorithm in Chapter 4.3. Throughout, descriptions are framed

around these challenges and how the Smart Data Structures design addresses them. Finally, in Chapter 4.4, we discuss some tradeoffs made by our design.

4.2 Learning Architecture

Recall that a Smart Data Structure is implemented by augmenting a base data structure with an online learning engine which continually optimizes the knobs of the base data structure. As Figure 4-1 illustrates, the learning engine performs closed-loop optimization. It collects performance feedback from an additional component called the *reward monitor*. The reward monitor provides a reward signal, and the goal of the learning engine is to learn knob settings that maximize the reward signal.

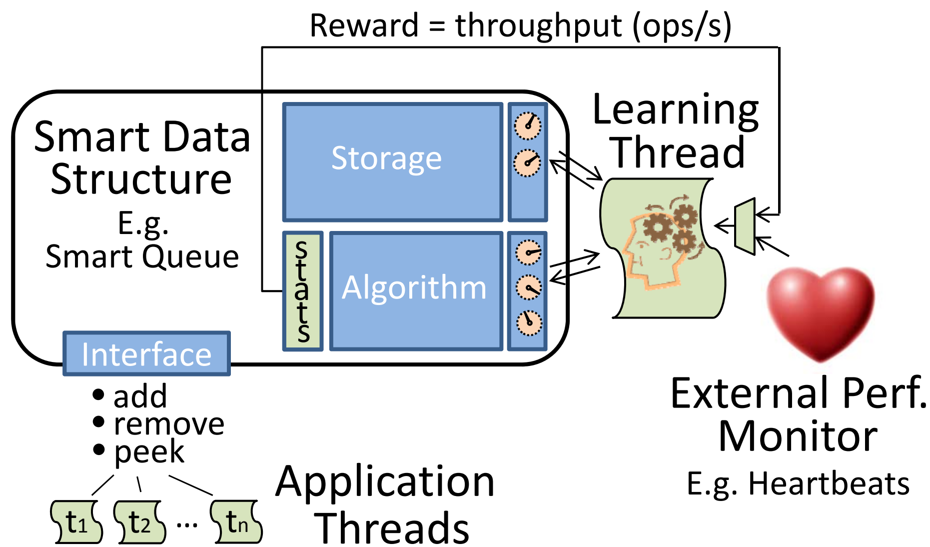


Figure 4-1: Smart Data Structures Learning Architecture. A learning engine collects performance feedback in the form of a reward signal from a reward monitor. An internal reward monitor is provided by default; for generality, external, application-specific reward monitors are also supported.

For the learning engine to improve the performance of an application, the reward signal must accurately reflect the goals of the application. Measuring the reward must also be non-intrusive; otherwise, overhead due to measurement can negate the benefits of optimization. This is Challenge 1.

The Smart Data Structures learning architecture addresses Challenge 1 in two

ways. First, Smart Data Structures provide a default, low-overhead, internal reward monitor that measures the throughput of the Smart Data Structure and provides the throughput as the reward signal. The throughput of the Smart Data Structure is often a reliable heuristic indicator of application performance. Sometimes, however, the goals of the application are not reflected in the throughput. For generality, Smart Data Structures provide an alternate solution to Challenge 1: they support external performance monitors which developers can use to provide application-specific reward signals.

Our design supports a variety of external performance monitors. One we recommend is Application Heartbeats [17]. Heartbeats is a portable framework for expressing application goals and measuring progress toward them through the abstraction of *heartbeats*. Developers insert calls to Heartbeats at significant points in the application to issue a heartbeat for each unit of progress. The learning engine uses the rate of heartbeats, the heart rate, as the reward. Application Heartbeats addresses Challenge 1 because it enables developers to provide application-specific reward signals and is simple to integrate into applications.

For the learning engine to maximize application performance, it must react quickly when changes in the system, application, or workload shuffle the tradeoffs that determine which knob settings are best at any given time; otherwise, the learning engine may invest the effort in finding an optimization but deliver that optimization when it is no longer useful. This is Challenge 2.

Smart Data Structures address Challenge 2, in part, by adopting a decoupled learning architecture. As Figure 4-1 shows, the learning engine runs in a *learning thread* that is separate (decoupled) from the application threads. This addresses Challenge 2 because, in multicores, the learning thread can run simultaneously alongside the application threads and produce optimizations with low latency after they become available.

The decoupled learning architecture also helps to minimize application disruption. It avoids the need for interleaving computation for learning within the application threads; alternatively, stealing cycles from the application threads could impede

their performance. This is especially true if the application utilizes many Smart Data Structures because there would be many learning engines to interleave in the application threads.

However, having an extra learning thread per Smart Data Structure could be disruptive to application performance if the total number of threads exceeded the available hardware thread contexts on the machine. To address this problem, our learning architecture uses a parameterizable number of learning threads and multiplexes each learning thread among multiple learning engines. By default, our design multiplexes a single learning thread among all learning engines. In Chapter 6, we describe how this is accomplished while maintaining low latency of optimizations. We also study the effect of multiplexing on performance and show that several hundred learning engines can be multiplexed within a single learning thread before Smart Data Structures are no longer able to improve performance over the base data structures upon which they are built.

The final challenge, Challenge 3, is identifying knob settings that are good for long-term performance. We want to leverage the benefits of planning rather than making near-sighted optimizations. As Chapter 4.3 will elaborate, Smart Data Structures accomplish this through our learning algorithm. In fact, our choice of learning algorithms is also the second way we address Challenge 2 (reacting and adapting knob settings quickly so as not to miss windows of opportunity). The learning algorithm we use is designed to be simultaneously reactive and good at planning.

4.3 Learning Engine Algorithm

To address both Challenge 2 (adapting settings quickly so as not to miss windows of opportunity for optimization) and Challenge 3 (identifying knob settings that are best for long-term performance), our Smart Data Structures library employs a Reinforcement Learning (RL) algorithm [37] that reads a reward signal and attempts to maximize it. Using RL in the context of Smart Data Structures presents a number of challenges: the state space can be large and is mostly unobservable, state transitions

are semi-Markov due to context switches, and the entire system is non-stationary. Because we need an algorithm that is a) fast enough for on-line use and b) can tolerate severe partial observability, we adopt an average reward optimality criterion [28] and use policy gradients to learn a good policy [42]. In particular, we use the Natural Actor-Critic algorithm [34].

The goal of policy gradients is to improve a *policy*, which is defined as a conditional distribution over "actions," given a state. At each timestep, the agent samples an action a_t from this policy and executes it. In the case of the Smart Queue, Skip List, and Pairing Heap, actions are a vector of discrete-valued scancounts, one for each Smart Data Structure; executing the action means installing each scancount in its corresponding Smart Data Structure. Throughout this section, we denote the distribution over actions (the policy) as π and parameters of the distribution as θ.

To compute the quality of any particular policy, we measure the average reward obtained by executing that policy. The average reward obtained by executing actions according to policy $\pi(a_t|\theta)$ is a function of its parameters θ. We define the average reward to be

$$\eta(\theta) \equiv \mathbb{E}\{\mathbb{R}\} = \lim_{i \to \infty} \frac{1}{i} \sum_{t=1}^{i} r_t,$$

where $\mathbb{R}$ is a random variable representing reward, and r_t is a particular reward at time t, taken either from the sum of throughputs from all Smart Data Structures or from an external monitor such as Heartbeats, and smoothed over a small window of time. The average reward is a function of the parameters because different settings induce a different distribution over actions, and different actions change the evolution of the system state over time. The average reward optimality criterion addresses Challenge 3 (finding good long-term knob settings) by attempting to maximize all future reward rather than immediate reward.

The goal of the Natural Actor-Critic algorithm is to estimate the natural gradient of the average reward of the system with respect to the policy parameters

$$\tilde{\nabla}_\theta \eta(\theta) = G^{-1}(\theta) \nabla_\theta \eta(\theta)$$

where $G(\theta)$ denotes the Fisher information matrix of the policy parameters. Once it has been computed, the policy can be improved by taking a step in the gradient direction.

Fortunately, there is a known elegant, closed-form way to compute the natural gradient which does not involve direct computation of the Fisher information matrix [34]. We address Challenge 2 (adapting knob settings quickly) through the use of this efficient algorithm. Alg. 1 shows the algorithm adapted to our case. Note that the algorithm only requires basic statistics available at each timestep: the observed reward r_t and the gradient of the log-probability of the action that is selected at each timestep $\nabla_\theta \log \pi(a_t|\theta)$. One problem is that our domain is partially observable. In a small twist on the ordinary Natural Actor-Critic algorithm, we therefore make a coarse approximation by assuming that the state is constant. Improving this by combining with a state estimation algorithm is left for future research, but the fact that this algorithm does not depend on a detailed model of the system dynamics is a major virtue of the approach.

Algorithm 1 The Natural Actor-Critic Algorithm.

1: Input: Parameterized policy $\pi(a_t|\theta)$ with initial parameters $\theta = \theta_0$ and its derivative $\nabla_\theta \log \pi(a_t|\theta)$.
2: Set parameters $\mathbf{A}_{t+1} = 0, b_{t+1} = 0, z_{t+1} = 0$.
3: For $t = 0, 1, 2, \cdots$ do
4: Sample $a_t \sim \pi(a_t|\theta_t)$ and set *scancounts* to a_t.
5: Observe r_t
6: Update basis functions:
$\tilde{\phi}_t = [1,\ \mathbf{0}]^T,\ \hat{\phi}_t = [1,\ \nabla_\theta \log \pi(a_t|\theta)^T]^T$
7: Update statistics: $z_{t+1} = \lambda z_t + \hat{\phi}_t$,
$\mathbf{A}_{t+1} = \mathbf{A}_t + z_{t+1}(\hat{\phi}_t - \gamma\tilde{\phi}_t)^T,\ b_{t+1} = b_t + z_{t+1} r_t$.
8: When desired, compute natural gradient:
$[v\ w^T]^T = \mathbf{A}_{t+1}^{-1} b_{t+1}$
9: Update policy parameters: $\theta_{t+1} = \theta_t + \alpha w$.
10: end.

So far, we have said nothing about the particular form of the policy. We must construct a stochastic policy that balances exploration and exploitation, and that can be smoothly parameterized to enable gradient-based learning. We accomplish this in the most direct way possible. For Smart Data Structures such as the Smart Queue, Skip

List, and Pairing Heap, we represent our policy as a multinomial distribution over the n different discrete values the scancount can take on. We use the exponential-family parameterization of the multinomial distribution, giving each Smart Data Structure i a set of n real-valued weights θ^i. The policy for data structure i is therefore

$$p(a_t^i = j|\theta^i) = \exp\{\theta_j^i\} / \sum_{k=1}^{n} \exp\{\theta_k^i\}.$$

from which we sample a discrete value for the scancount.

The gradient of the likelihood of an action (needed in Alg. 1) is easily computed, and is given by

$$\nabla_\theta \log \pi(a_t^i|\theta^i) = \delta(a_t^i) - \pi(a_t^i|\theta^i)$$

where $\delta(a_t^i)$ is a vector of zeros with a 1 in the index given by a_t^i. When enough samples are collected (or some other gradient convergence test passes), we take a step in the gradient direction: $\theta = \theta + \alpha w$, where w is computed in Alg.1 and α is a step-size parameter. Currently, we take 200 samples and use $\alpha = .1$.

So far, we have discussed the algorithm in the context of learning discrete-valued knobs. In addition to discrete-valued knobs, the learning algorithm can support learning and joint-optimization of a variety of other knobs. Among those supported are Gaussian distributions, binary-valued knobs, and permutation orderings (used in Smart Locks).

For all knob types, the algorithm for improving the policy is identical. The only difference is the interpretation of the policy and how many parameters (real-valued weights) make up the policy. For example, for the binary knob, we learn a distribution over two weights from which we sample a knob setting of true or false.

4.4 Learning Thread Tradeoffs

As previously described, the learning engine uses Alg. 1 to jointly optimize scancounts for all Smart Data Structures. To run the learning engine, our design adds one thread to the application. The advantage is that this minimizes application disruption and

enables background optimization of the application as it is running. The use of an extra thread also represents a tradeoff because an application could potentially have utilized the extra thread for parallelism. The extra thread is only justified if it provides a net gain in performance. Fortunately, net gains are easy to achieve in common scenarios which this section will describe.

First, by Amdahl's Law, typical applications reach a saturation point where serial bottlenecks limit scalability and adding parallelism no longer benefits performance. Here, adding an optimizing thread is not only justified, it is one of the only ways to continue improving performance. Most applications are expected to reach their limit before they can fully utilize future manycore machines, and many reach those limits today.

Second, for memory-intensive applications, it is well-known that multicore shared memory systems are becoming a scalability bottleneck: adding threads can increase sharing in the memory system until it saturates and limits performance. Smart Data Structures can help scalability by reducing memory synchronization operations and cache miss rates through better locality and reduced shared memory invalidations.

Finally, for the remaining applications, if we assume n hardware thread contexts, our design must improve performance by a factor of roughly $n/(n-1)$ to outweigh the performance lost to utilizing one thread for optimization instead of application parallelism. The required improvements diminish as the number of cores increase: on today's 16-core and 8-core machines, a factor of just 1.07x and 1.14x are needed. Our results achieve gains up to 1.44x on a 16-core machine. Future work will investigate this scenario further.

Chapter 5

Performance Results

This chapter evaluates the Smart Queue, Smart Skip List, and Smart Pairing Heap in our prototype library of Smart Data Structures. It starts with a description of our experimental setup then presents five studies. The first characterizes the performance of the best existing data structure algorithms and shows that the Flat Combining data structures [12] are the best choice to build our Smart Data Structures prototype upon because they achieve the best performance on our system. The second study quantifies the impact of the *scancount* knob setting on data structure performance. It shows that the best value varies widely, that hand-tuning would be cumbersome, and that using the ideal scancount can substantially improve performance. The third study evaluates the performance of Smart Data Structures. It derives performance bounds from the second study then shows that Smart Data Structures achieve near-ideal performance under a variety of conditions in many cases. We show that Smart Data Structures improve performance over the state-of-the-art by as much as 1.44x in our benchmarks. The fourth study demonstrates the advantage of the learning approach to auto-tuning in Smart Data Structures: the ability to adapt the scancount to changing application needs. Since it is common for the load on a data structure to be variable in producer-consumer application structures,[1] we dynamically vary the load on the Smart Data Structures and show that they achieve near-ideal performance even under high variation frequencies. The fifth study evaluates the Smart Queue,

[1] E.g. work queues where the complexity of individual work items may vary

Skip List, and Pairing Heap in a variety of real-world applications to demonstrate significant performance improvements and analyze different use-cases to determine when Smart Data Structures provide the most benefit.

Later in Chapter 6, we will study the scalability of Smart Data Structures. Then, in Chapter 7, we will present experimental results for the Smart Lock.

5.1 Experimental Setup

The experiments are performed on a 16-core (quad 4-core) Intel® Xeon® E7340 system with 2.4 GHz cores, 16 GB of DRAM, and a 1066 MHz bus. Each core runs 1 thread at a time. Benchmarks use up to 15 threads at once (on 15 cores), reserving one core for system processes. Where applicable, one of the 15 available cores is utilized for machine learning. Threads are not affinitized to particular cores and can move around during execution. Benchmarks are compiled for Debian Linux (kernel version 2.6.26) using gcc 4.3.2 and O3 optimizations.

The experiments in Chapter 5.2-5.5 measure data structure throughput using a modified version of the synthetic benchmark developed in the Flat Combining paper [12]. Modifications are limited to adding support for benchmarking Smart Data Structures, adding a second operating mode for evaluating producer-consumer application structures, and varying the scancount parameter used by the Flat Combining algorithm.

The original operating mode instantiates a data structure and spawns n threads that request enqueue and dequeue operations at random with equal likelihood. Between operations, each thread performs post computation. Post computation is modeled as a delay loop of integer arithmetic instructions. For a given n, decreasing the post computation increases the load on the data structure. The benchmark runs for 10 seconds before joining threads and recording the results.[2] For Smart Data Structures, one unit of reward is credited for each operation completed. The benchmark

[2]The benchmark supports instantiation of multiple data structures, other distributions of enqueue and dequeue operations, and different durations as well.

takes the number of threads, the amount of work between operations, and a static scancount setting (where applicable) as parameters.

In the operating mode we added, threads can be configured as producers or consumers. Producers perform only enqueue operations and skip the post computation between operations. Consumers perform only dequeue operations and do perform the post computation. In our experiments, we use one producer and $n-1$ consumers to model a work queue application structure where a master enumerates work to be performed by the workers. The Flat Combining data structures are non-blocking; thus, to model a producer-consumer application structure, consumers spin until they successfully dequeue valid data. For Smart Data Structures, one unit of reward is credited for each valid item that is dequeued.

In all experiments, we average 10 trials per configuration. We calculate error bars where appropriate using standard error: $\frac{s}{\sqrt{10}}$, where s is the sample standard deviation.

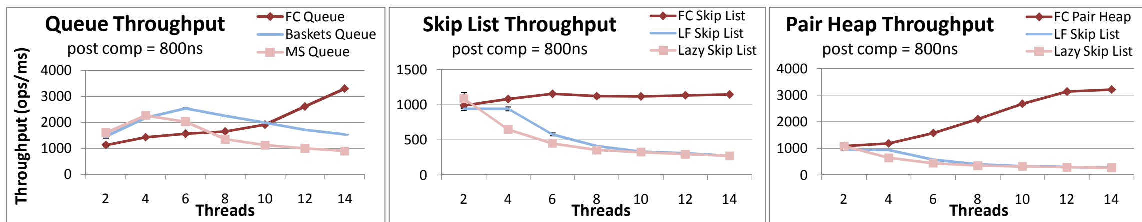
5.2 Performance of Existing Alternatives

This experiment characterizes the performance of the best existing concurrent queue and priority queue implementations to determine which to build Smart Data Structures upon. The best queues are the Michael and Scott queue (the MS Queue) [32], the baskets queue of Hoffman et. al [16], and the Flat Combining queue [12]. The best priority queues in the literature are the Skip List based priority queue of Lotan and Shavit [26], the priority queue based on the Flat Combining Skip List, and the priority queue based on the Flat Combining Pairing Heap [12]. We also compare a priority queue based on the lazy lock-based Skip List developed by Herlihy and Shavit [14].

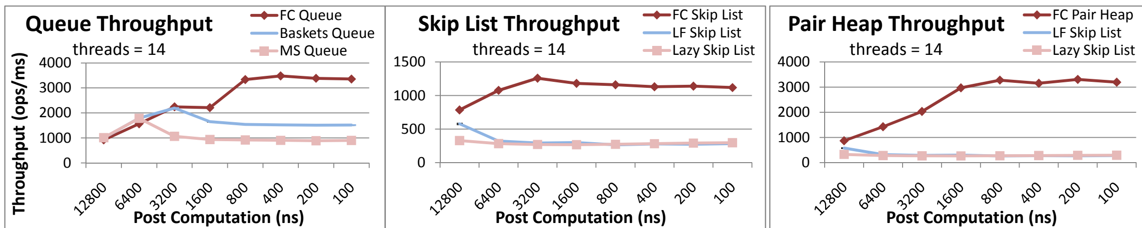
Our benchmark studies how data structure throughput is impacted by two key variables: the number of threads operating on the data structure and the load on the data structure. The load is adjusted by varying the amount of post computation between operations: decreasing the post computation increases the load. The first

mode of our benchmark is used (see Chapter 5.1 for a description).

Figure 5-1 shows the results. In the first series of graphs, we fix the amount of post computation and vary the number of threads. In the second series, we fix the number of threads and vary the amount of post computation. We find that the Flat Combining data structures significantly outperform the others over a wide range of concurrency levels and loads. The Flat Combining Queue, Skip List, and Pairing Heap achieve up to 2x, 4x, and 10x improvements, respectively, over the best prior algorithms.



(a) Throughput vs. Concurrency Level



(b) Throughput vs. Post Computation

Figure 5-1: Performance Characterization of the Best Existing Algorithms. The Flat Combining Queue, Skip List, and Pairing Heap substantially outperform the others at higher concurrency levels and heavier loads (lower post computation).

Hendler et al. analyze the sources of the improvement [12]. They show that Flat Combining a) significantly reduces synchronization overheads and b) improves cache performance because centralizing the operations via combining improves locality and reduces shared memory invalidations. We demonstrate in Chapter 5.4 that, through machine learning, our Smart Data Structures prototype improves upon the high performance of Flat Combining by an additional factor of up to 1.44x.

It is interesting to note, however, that the Flat Combining data structures are not always highest performance at small concurrency levels and low load (high post

computation). In these cases, in prior data structures, synchronization overheads no longer significantly degrade performance. If a prior data structure requires less bookkeeping to complete an operation, it might outperform Flat Combining. Flat Combining works best at higher loads and higher concurrency levels. Nevertheless, the Flat Combining Skip List and Pairing Heap outperform prior works at the lowest levels as well. Only in the case of the queues do prior works outperform the Flat Combining Queue at low loads and concurrency levels.

Overall, we find that the Flat Combining data structures offer the best performance on today's machines, and the trends suggest that they will significantly outperform prior data structures as the number of cores in tomorrow's machines increases.

5.3 Scancount Sensitivity

This study quantifies the impact of the *scancount* value on Flat Combining data structure performance to motivate our auto-tuning of this knob via machine learning. Recall that the scancount determines how many scans of the publication list that the combiner makes when combining. We expect that increasing the scancount will improve performance because it provides more opportunities to catch late-arriving requests and increases the number of operations performed on average in each combining phase. This reduces synchronization overheads and improves cache performance. However, making more scans has the tradeoff that the average latency of data structure operations can increase. Some applications are not sensitive to added latency, but some are. The best scancount value balances these tradeoffs. We will show that the best scancount depends on the particular load the application places on the data structure as well.

For two common application structures, this study evaluates different static values for the scancount and examines the impact on data structure throughput for different loads. We use the two operating modes of the benchmark described in Chapter 5.1 to benchmark the two application structures. In Application Structure 1, threads have no data dependency: they run autonomously, requesting enqueue and dequeue

operations at random with equal likelihood. In Structure 2, threads follow a producer-consumer pattern analogous to a work queue program with a master that enumerates work for workers to perform.

For Structure 1, we find that the data structures generally benefit from the highest static scancount assignment (graphs are omitted for space). This is expected since threads have no data inter-dependency and thus are not impacted by latency. For Structure 2, however, we find that throughput can be adversely affected by high latency. When the producer thread becomes the combiner, work is not being enumerated and inserted into the queue; the consumers can run out of work and spin idly. Thus, there is an ideal scancount after which throughput begins to degrade.

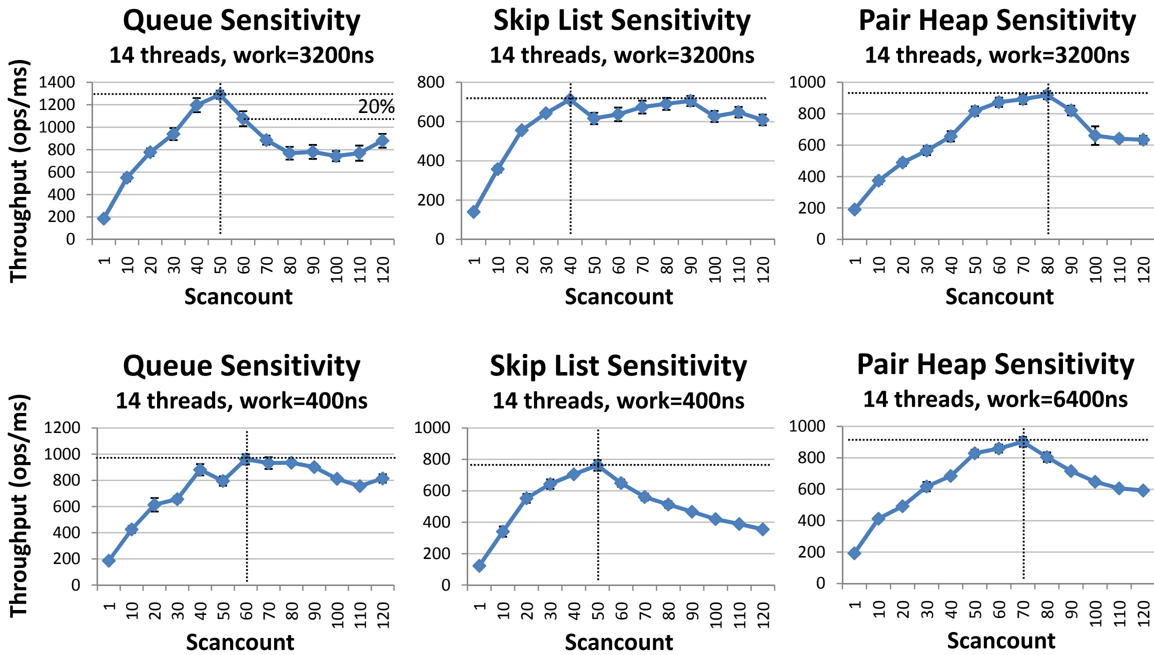


Figure 5-2: Sensitivity to the Scancount in Producer-Consumer Application Structures: Throughput vs Scancount Over a Range of Loads. The ideal scancount varies widely and depends on both the load and the data structure.

Figure 5-2 shows characteristic excerpts of our results for Structure 2. The figure shows throughput as a function of the static scancount value for different loads. The crosshairs in each graph denote the maximum throughput and scancount value for which it occurs. We find that throughput can be greatly improved by using ideal scancount values. Further, if we look at the graphs for a particular data structure,

we find that no single scancount value works best for all loads. A near-miss of the ideal scancount value can result in throughput that is significantly (20%) below the maximum. Third, we find that the precise dependence of the ideal scancount on the load differs from one data structure to the next.

Together, these findings indicate that the ideal scancount would be complex to predict by hand. Smart Data Structures provide an automatic approach that relieves programmers of this burden. Chapter 5.4 will show that Smart Data Structures readily discover the ideal scancounts and significantly improve performance by using them.

5.4 Performance of Smart Data Structures

This study evaluates the performance of Smart Data Structures. We will quantify how well Smart Data Structures are able to optimize the scancount knob by comparing Smart Data Structure performance against ideal and baseline performance bounds. The benchmark is the producer-consumer benchmark described in Chapter 5.1 which measures throughput for different loads on the data structure. We will show that Smart Data Structures achieve near-ideal performance in many cases across the different loads.

The ideal throughput bound represents the best performance a programmer could achieve if they swept over the space of scancount values to determine the highest performance value for their application. It is a static throughput bound because we assume the scancount value does not change during program execution. We derive the ideal static throughput bound from the results in Chapter 5.3 which give throughput as a function of the scancount for different loads. We are interested in the ideal throughput for each different load; together, these throughputs make up the ideal bound. Thus, for a given load, we look up the corresponding graph in Chapter 5.3 and use the maximum throughput.

The baseline throughput is taken to be the throughput that the learning engine would achieve if it were unable to learn the scancount. For each load, it is an average

over the throughput of all studied scancount values. In other words, it is an expectation of throughput when scancount values are randomly selected. It is computed using the results in Chapter 5.3 as well.

Figures 5-3, 5-4, and 5-5 show the results. We find that Smart Data Structures improve performance over the baseline in all cases studied. Indeed, they achieve near-ideal throughput for most loads, rivaling the performance of thorough hand-tuning while relieving programmers of the burden and complexity of hand-tuning. Overall, the Smart Queue, Smart Skip List, and Smart Pairing Heap each substantially improve throughput over the average static bound by up to 1.44x, 1.39x, and 1.39x, respectively.

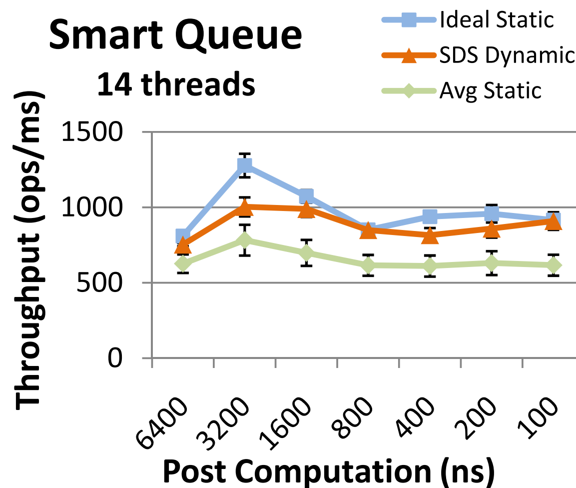


Figure 5-3: Smart Queue Throughput vs Post Computation: A Comparison Against Ideal and Average Static Throughput Bounds. The Smart Queue achieves near ideal static throughput for most data structure loads.

This shows that the overheads of layering optimization on top of the base Flat Combining data structures is low enough that the advantages of optimizing the scancount knob far outweigh the costs. The costs are not, however, zero. This is why performance usually approaches the static ideal bound but does not exceed it.

There are a few data points where Smart Data Structure performance matches or somewhat exceeds the ideal static bound. Take Figure 5-5, for example. At post computation of 200ns, Smart Pairing Heap throughput exceeds the static ideal bound. The error bars for the Smart Pairing Heap and static ideal bound somewhat overlap,

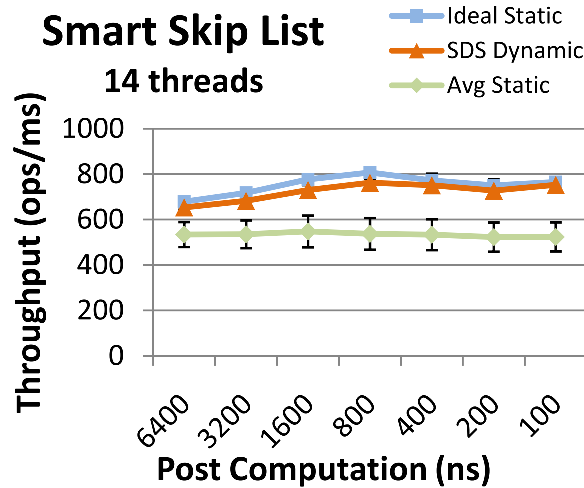


Figure 5-4: Smart Skip List Throughput vs Post Computation: A Comparison Against Ideal and Average Static Throughput Bounds. The Smart Skip List achieves near ideal static throughput for most data structure loads.

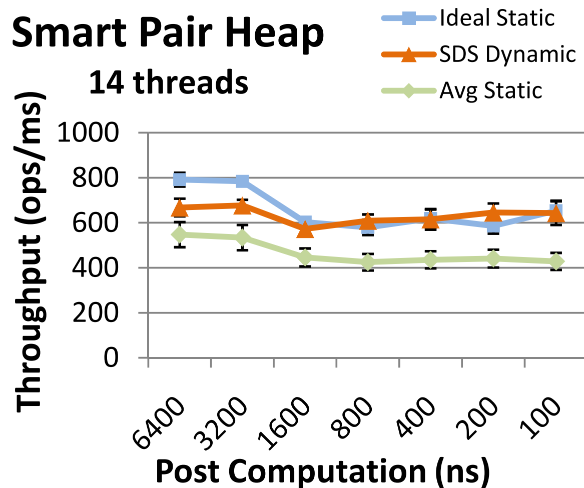


Figure 5-5: Smart Pairing Heap Throughput vs Post Computation: A Comparison Against Ideal and Average Static Throughput Bounds. The Smart Pairing Heap achieves near ideal static throughput for most data structure loads.

but additional experiments suggest that the improvement is likely due to the Smart Pairing Heap's ability to dynamically tune the scancount rather than be limited to using static values.

For data structures like the Flat Combining Pairing Heap, even for a given load, the ideal knob settings may vary during execution. A major virtue of the online learning approach in Smart Data Structures is that it capitalizes on the performance

potential of dynamically adjusting knob settings. Dynamic tuning is often necessary but impractical to achieve by hand, and Smart Data Structures provide this facility automatically. Chapter 5.5 will investigate how well Smart Data Structures adapt dynamically. Future work will attempt to identify the dynamic throughput bound (the optimal bound) in addition to the ideal static bound.

Overall, the results demonstrate that Smart Data Structures are able to learn ideal scancount values for our benchmarks and that the overhead of learning is low enough that net throughput improvements are high. Further, the improvements provided by Smart Data Structures multiply the already high performance benefits of Flat Combining that we have shown in Chapter 5.2.

5.5 Adaptivity of Smart Data Structures

This study will demonstrate a key advantage of the learning approach to auto-tuning taken by Smart Data Structures: the ability to balance dynamic tradeoffs and adapt the scancount to changing application needs. We have shown in Chapters 5.3 and 5.4 that data structure throughput depends critically on optimizing the scancount for different loads. Therefore, we will quantify how well Smart Data Structures adapt to dynamic variation in the load. We will reuse the benchmark in Chapter 5.4 which compared Smart Data Structure throughput to ideal and average static bounds. This time, we will vary the load dynamically during benchmark execution and compare against dynamic ideal and average bounds. We will show that Smart Data Structures readily adapt and achieve near-ideal throughput even when the load changes as rapidly as once every $10\mu s$.

It is important to adapt knob settings for different loads because it is common for the load on data structures in applications to vary depending upon the input to the application. One popular class of producer-consumer applications with input-dependent loads are video processing applications. They may use a work queue model to coordinate the parallel processing of each video frame. The processing complexity can vary significantly over the timescale of a just a few frames (as little as 15ms)

because scene complexity varies from one scene to the next. The complexity of the input scene determines processing time for work items and therefore the rate at which work items are removed from the data structure and the rate at which new work items are generated and inserted in the data structure.

Our experiment looks at a related producer-consumer application structure and studies the adaptivity of Smart Data Structures under variable load. To vary the load, we break the benchmark up into equal intervals where each interval places a different load on the Smart Data Structure. We vary the frequency with which intervals change. The sequence of loads from one interval to the next is given by a schedule. We look at three schedules, selected at random. Each is a cycle of 10 different loads, repeated throughout the 10 second duration of the benchmark. The schedules are:

Schedule 1: $800 \rightsquigarrow 6400 \rightsquigarrow 200 \rightsquigarrow 3200 \rightsquigarrow 1600 \rightsquigarrow 100 \rightsquigarrow 400 \rightsquigarrow 100 \rightsquigarrow 400 \rightsquigarrow 800$

Schedule 2: $1600 \rightsquigarrow 200 \rightsquigarrow 400 \rightsquigarrow 1600 \rightsquigarrow 200 \rightsquigarrow 1600 \rightsquigarrow 3200 \rightsquigarrow 100 \rightsquigarrow 200 \rightsquigarrow 800$

Schedule 3: $800 \rightsquigarrow 100 \rightsquigarrow 6400 \rightsquigarrow 200 \rightsquigarrow 200 \rightsquigarrow 100 \rightsquigarrow 400 \rightsquigarrow 800 \rightsquigarrow 3200 \rightsquigarrow 400$

This time we compare against dynamic not static throughput bounds. We are still able to derive the dynamic bounds from the graphs in Figures 5-3, 5-4, and 5-5. To do so, we need to derive the expected ideal and average throughput for executing a given schedule of loads. In fact, we only need to need to know the expected throughput for one cycle through the schedule because the benchmark executes the same schedule over and over again and overall throughput will be approximately the same as the throughput for one cycle.

To get the ideal dynamic throughput for one cycle of a schedule, we look up the maximum throughput for each load in the schedule. Since each interval is of equal length, we average these maximum values and the result is the expected ideal dynamic throughput. We use a similar procedure to get the average dynamic throughput. For each load in the schedule, we determine the average throughput over all studied scancount values. Then, since each interval is of equal length, we average these to get the expected average dynamic throughput.

The ideal dynamic throughput will be hard for Smart Data Structures to achieve because it assumes that the scancount is switched instantaneously when the load changes to reflect the ideal scancount for the new load. The learning engine within Smart Data Structures will require a short time to react and find the new ideal scancount value.

Figure 5-6 shows the results. We vary the frequency with which intervals change. Each cluster compares ideal dynamic, average dynamic, and Smart Data Structure throughput for a given variation frequency.

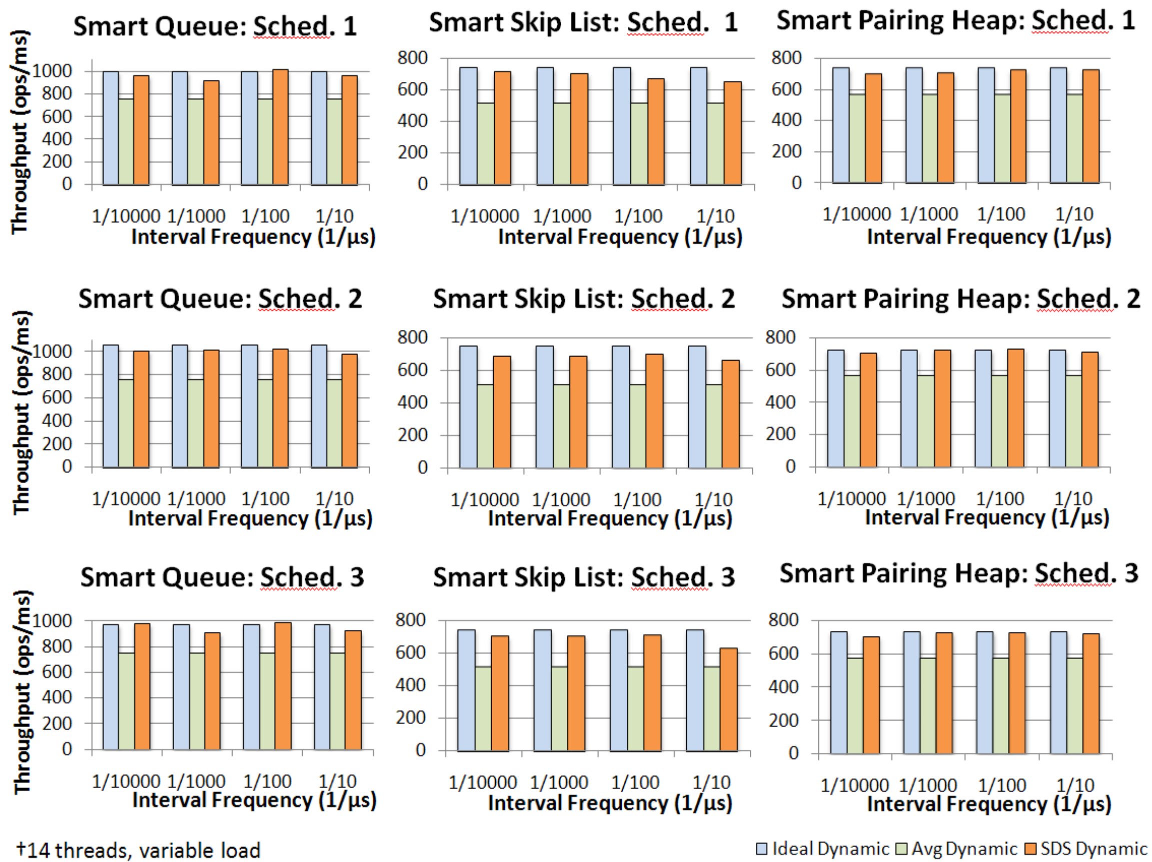


Figure 5-6: Smart Data Structures Throughput Under Variable Load: A Comparison Against Ideal Dynamic and Average Dynamic Throughput for Different Variation Frequencies. In many cases, Smart Data Structures achieve near-ideal throughput. Throughput slowly decreases as changes in the load become more frequent.

The most important result is that, even under the highest variation frequency of $\frac{1}{10\mu s}$, Smart Data Structures achieve near-ideal throughput in many cases. The Smart Queue and Pairing Heap still achieve 85% of the performance difference between the

average dynamic and ideal dynamic bounds. The Smart Skip List achieves about 60%. This shows that Smart Data Structures are reacting quickly and effectively to changes in the load.

An interesting secondary feature of the results is that the throughput slowly decreases as the interval frequency increases. This is because the learning engine takes non-zero time to react to changes in the load and converge on the ideal scancount for the new load; as the interval period becomes smaller and smaller, the reaction time and the impact of using a sub-optimal scancount slowly becomes non-negligible.

As the interval frequency increases, the degree to which performance degrades depends upon a) how sub-optimal the scancount is when it is used in the new interval before the learning engine finds the ideal scancount and b) how long it takes the learning engine to find the ideal scancount. In Chapter 5.3, we showed that even narrowly missing the ideal scancount can result in up to 20% deviations from ideal throughput. We see small degradations in our results because the learning engine is able to quickly converge on new scancount values in tens of microseconds.

Currently, the finest interval granularity that our benchmark supports is $10\mu s$ due to timing overheads. If we could further decrease the period, we would expect throughput to continue decreasing because the learning engine would not be able to keep up with the frequency of changes. It would react by making a best effort and optimizing the scancount for groups of intervals rather than individual intervals. It would find the scancount that worked best for several successive intervals.

Overall, we find that Smart Data Structures react rapidly to changes in the the ideal knob settings and nearly achieve the ideal dynamic throughput in many cases.

5.6 Application Case Studies

This study evaluates the performance of Smart Data Structures in popular, real-world applications. Our goal will be to gain an understanding of when applications will benefit most from using Smart Data Structures.

We begin in Chapter 5.6.1 by describing the applications we study and explain-

ing how Smart Data Structures are used within them. Then, in Chapter 5.6.2, we compare the performance of Smart Data Structures to prior data structures, for each application. We will demonstrate scenarios in which Smart Data Structures substantially improve performance over prior data structures and scenarios in which they are unable to offer improvement. Next, in Chapter 5.6.3, we will compare Smart Data Structure performance to performance bounds to determine how well learning is able to dynamically tune knobs in these applications. We will use a methodology similar to the experiment in Chapter 5.4 where we compare application performance with learning and dynamic knob tuning versus performance using static knob values. Finally, in Chapter 5.6.4, we will provide usage guidelines for Smart Data Structures based on what we have learned in these experiments.

5.6.1 Application Descriptions

This section describes the case-study applications we will evaluate. We have selected four popular benchmarks drawn from the well-known Parsec [2] benchmark suite and canonical parallel programming kernels. We will describe their importance, the parallelism models they use, and the data structures they rely upon.

Parsec Dedup The first application is Dedup from the Parsec benchmark suite. Dedup is short for de-duplication. It is a server application that compresses a data stream via a combination of global and local compression techniques and is widely used in backup storage systems. The application uses the pipeline parallelism model, expressing the compression algorithm as a stream computation with multiple stages that execute in parallel. There are 5 stages in the pipeline. The first and last stage use a single thread, while the middle 3 stages use a parameterizable number of threads. Successive stages are connected via a concurrent queue which we will replace with a variety of different queue implementations to compare performance. We use the Parsec "simlarge" input for this benchmark.

Parsec Ferret The next application is Ferret from the Parsec benchmark suite. Ferret is a server application that implements content-based similarity search. It is an emerging next-generation search engine for non-text data types. In the Parsec library, ferret is configured for image search. The application uses the pipeline parallelism model with a 6-stage pipeline. The first and last stage use a single thread. The middle 4 use a parameterizable number of threads. As in Dedup, successive stages are connected through a concurrent queue which we will replace to study the performance of different queues. We use the Parsec "simlarge" input for this benchmark also.

Parallel Sort The next application is a popular parallel programming kernel called Parallel Sort. It is often used for parallel discrete event simulation in system modeling and also has important applications in network routing for traffic prioritization. Sorting is implemented using a concurrent priority queue. In general, each data item has a key, and the data are sorted by key. The parallelism model utilizes a single global priority queue shared by all threads. We will replace the priority queue to study the performance of different priority queue implementations. We will measure maximum sorting throughput for random integer keys.

Traveling Salesman The final application is colloquially known as the Traveling Salesman Problem. It is a canonical NP-hard problem in combinatorial optimization with applications in planning, logistics, and the manufacture of microchips. Slight modifications of the problem also have applications in DNA sequencing (cities represent DNA fragments and distances measure similarity between DNA fragments). The goal is to find a tour which passes through each city in the input once before returning to the start city, while minimizing the total distance traveled. We will solve this problem exactly using the branch-and-bound algorithm. The parallelism model uses a global work queue which all threads share and access. We will replace this global work queue to study the performance of different queue implementations. We will measure execution time using the largest cities in the United States as the input.

5.6.2 Smart Data Structures Versus Previous Work

In this section, we will compare the performance of Smart Data Structures to prior data structures in the case-study applications. The first two applications that we look at will demonstrate typical scenarios where Smart Data Structures significantly improve performance over prior data structures. The other two applications have been selected to highlight two scenarios in which Smart Data Structure are unable to provide improvements. We will show that the reason that Smart Data Structures are unable to provide improvements in the first one is that the choice of queue implementations has little effect on performance because the overhead of accessing the queue is outweighed by the amount of computation between queue accesses. In the other application, a custom data structure exploits knowledge about the application and subsequently outperforms the Smart Data Structure. Ultimately, we will use these results in Chapter 5.6.4 to form usage guidelines for when Smart Data Structures can be expected to benefit application performance the most. We will also show in the next section, Chapter 5.6.3, that Smart Data Structures are able to optimize knob settings effectively in these applications.

In this experiment, we will evaluate application performance for a variety of different data structure implementations including Smart Data Structures, state-of-the-art data structures, the original data structure used in each application (where applicable), and a baseline lock-based data structure built on top of a pthreads mutex and a serial data structure. In each experiment, we vary the number of threads used in the application. Our experimental setup utilizes the same system, compiler, and operating system infrastructure as our experiments in Chapter 5.1. Namely, we use a 16-core Intel® Xeon® E7340 system with 2.4 GHz cores, 16 GB of DRAM, and a 1066 MHz bus. Each core runs 1 thread at a time. Benchmarks are compiled for Debian Linux (kernel version 2.6.26) using gcc 4.3.2 and O3 optimizations. Benchmarks use up to 16 threads at once (on 16 cores). Where applicable, one of the 16 available cores is utilized for machine learning. Threads are not affinitized to particular cores and can move around during execution. We average results from 10 runs of each application

configuration for each performance data point.

We begin with the results for Parallel Sort in Figure 5-7. We replace the priority queue in the application with different data structures that implement priority queue functionality. We compare the Smart Pairing Heap to a) the state-of-the-art Flat Combining Pairing Heap, b) the popular priority queue developed by Lotan and Shavit [26] which is based on a lock-free Skip List (LF Skip List) and c) a baseline Mutex-Based Heap which uses a pthreads mutex to protect accesses to a serial heap data structure. The benchmark measures the throughput of each data structure in units of the number of random integer keys that can be sorted per millisecond.

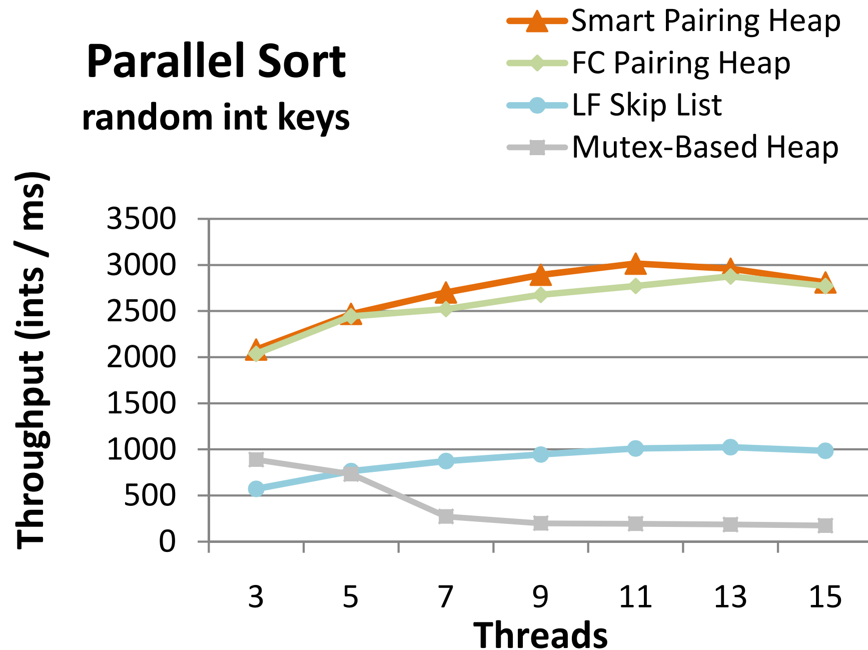


Figure 5-7

The results show that the Smart Pairing Heap significantly outperforms previous work, improving throughput by up to 9% over the Flat Combining Pairing Heap and by up to 300-400% over the more popularly used LF Skip List.

Interestingly, when used as a priority queue, none of the data structures exhibits ideal throughput scaling as the number of threads increases. Throughput of the Mutex-Based Heap actually decreases and reaches a minimum after 7 threads. The degradation comes from increased contention in the system bus as more threads spin

on the mutex attempting to acquire it and interfere with bus operations from the thread that has the mutex and is doing useful work. The throughput of the LF Skip List saturates after 9 threads because overheads from synchronization eventually limit the rate at which threads can complete operations. Throughput of the Flat Combining Pairing Heap (and thus the Smart Pairing Heap as well) eventually saturates between 11 and 13 threads.

What these scaling trends tell us is that, at larger concurrency levels, we can expect the benefits of the Smart Pairing Heap to be comparable to the results at the concurrency levels studied in this experiment. To summarize, the Smart Pairing Heap significantly improves upon previous work by up to 9% over the Flat Combining Pairing Heap and by up to 300-400% over the popular LF Skip List. These performance improvements come for free by merely dropping in the Smart Data Structure in place of prior data structures.

Next, we study the performance of different data structures in the Traveling Salesman application. Figure 5-8 gives the results. We compare the Smart Queue to a) the state-of-the-art concurrent queue from Michael and Scott (the MS Queue) [32], b) the Flat Combining Queue, and c) a baseline Mutex-Based Queue which uses a pthreads mutex to protect accesses to a serial queue data structure. The benchmark measures throughput normalized to the throughput of the Mutex-Based Queue at 3 threads.

The results show that the Smart Queue outperforms the other queues for most concurrency levels. The performance improvement of the Smart Queue over the MS Queue is highest at 15 threads, at 37%. Improvement over the MS Queue ranges from -45% to 37% with 9-15% improvements being common. The MS Queue slightly outperforms the Flat Combining Queue in this application for most concurrency levels. Thus, the improvements of Smart Data Structures over Flat Combining are slightly higher.

Data Structure scaling is better in this application but still not ideal for any of the data structures. As before, we see performance decrease in the Mutex-Based Queue after 5 threads as well due to bus contention. We expect that the Flat Combining

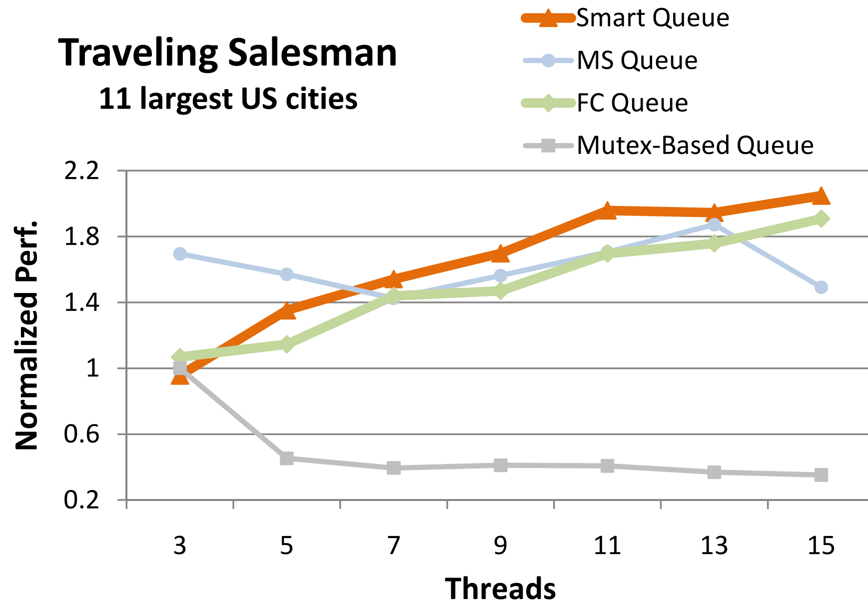


Figure 5-8

Queue (and thus the Smart Queue) have not yet reached maximum performance and will continue to improve performance at higher concurrency levels than the 16 threads available on our machine. Based on our findings in Chapter 5.2, we expect that the MS Queue, however, has reached its scaling limit and its performance will decrease as more threads are added.

One interesting thing that these scaling trends indicate is that we can expect the maximum performance (at any number of threads) of the Smart Queue to outperform the maximum performance of the popular MS Queue by an increasing margin as more threads are added.

Next we study the Parsec Dedup application. Dedup is an example of an application where queue performance is not a significant contributer to overall execution time. Thus, optimizations to queue performance will be unable to noticeably improve application performance and using Smart Data Structures will neither help nor hurt performance. Figure 5-9 gives the results. We compare the Smart Queue to a) the Flat Combining Queue, b) the original queue in Dedup which is based on pthreads condition variables and c) a baseline Mutex-Based Queue. The x-axis shows the number of threads used in each of the 5 pipeline stages in the application. The y-axis

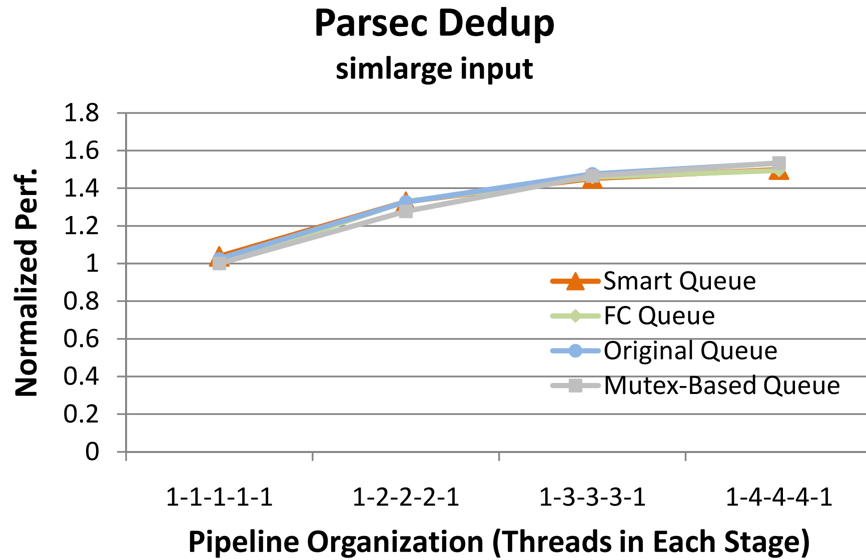


Figure 5-9

gives performance normalized to the performance of the Mutex-Based Queue at 3 threads.

The results show that, indeed, all queue implementations achieve nearly the same performance. The reason that the queue implementation has little effect on overall performance is because the time between successive queue accesses in each stage is large relative to the time to access the queue; improvements to the average queue access time are dominated by the larger computation time between accesses.

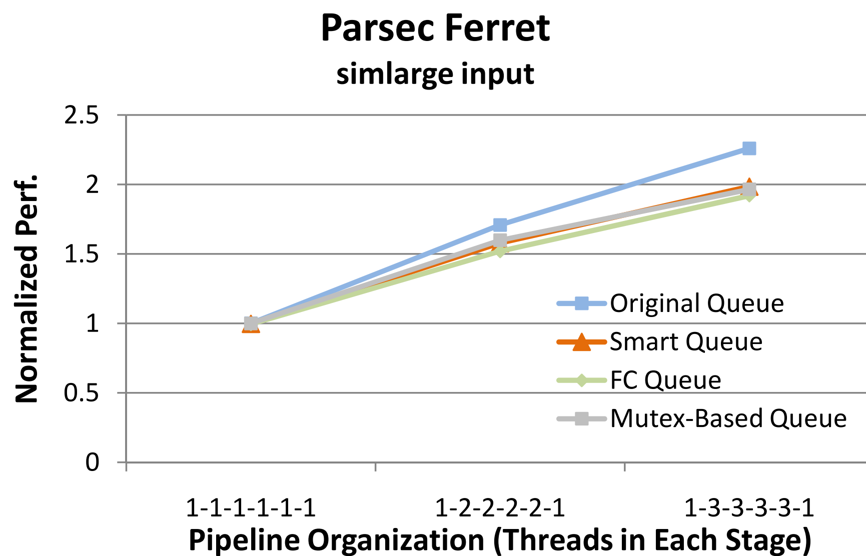


Figure 5-10

Figure 5-10 presents results for the final application, the Parsec Ferret application. Ferret is an example of an application where knowledge about the behavior of the application can be exploited in a data structure which can outperform a Smart Data Structure. This application illustrates the point that Smart Data Structures are not a silver bullet for getting the best performance theoretically possible; rather, a major virtue of Smart Data Structures is that they typically get good performance without requiring the development of application-specific data structures.

In this experiment, we compare the Smart Queue to a) the original custom queue in Ferret which is based on pthreads condition variables, b) the Flat Combining Queue, and c) a baseline Mutex-Based Queue. The x-axis shows the number of threads used in each of the pipeline stages in the application. The y-axis gives performance normalized to the performance of the Mutex-Based Queue at 3 threads.

The results show that the Smart Queue, Flat Combining Queue, and Mutex-Based Queue have similar performance and are outperformed by the original, custom queue. The reason that the custom queue performs better is that its design avoids producer starvation when a pipeline queue becomes empty. Producer starvation can occur when pipeline stages are imbalanced: if one stage processes work more quickly than its predecessor, it can overwhelm the queue between them with fruitless dequeue requests and cause producer enqueue requests to be starved. The Smart Queue is usually able to avoid producer starvation because it processes enqueue operations ahead of dequeue operations, but in this application, it is not able to match the performance of the custom queue. We expect that a queue based on a Smart Lock, however, may learn to schedule producers ahead of consumers more effectively, and thereby improve performance; we will investigate this possibility in future work.

To summarize our findings, we have shown that Smart Data Structures significantly improve performance over prior data structures for the Parallel Sort and Traveling Salesman applications. We showed that the Smart Pairing Heap improved performance in Parallel Sort by up to 9% over the highest-performing prior data structure and that the Smart Queue in the Traveling Salesman application improved performance by up to 37% over the overall highest performance data structure. With

the Dedup application, we highlighted a scenario in which Smart Data Structures are unable to offer performance improvements because queue performance is not a significant contributor to end-to-end application performance. Finally, with the Ferret application, we highlighted an assumption about when Smart Data Structures will be used: when developing application-specific data structures is too complex, too much work, or undesired.

We note that, in addition to the four benchmarks we have analyzed in this section, we have also considered the Parsec Bodytrack and x264 applications as other examples of applications that use concurrent queues for pipeline parallelism. We found that Bodytrack actually uses pipeline parallelism in a superficial way so we did not analyze it. Namely, Bodytrack uses a pipeline to make I/O asynchronous with respect to the main, serial computation. In this case, queue access overheads will be dominated by long I/O times and the queue implementation will have little effect on performance. x264 was more interesting. x264 is a video encoding application that builds a complex pipeline at runtime to model dependencies between frames. Each stage corresponds to a frame, and the pipeline has the form of a Directed Acyclic Graph with multiple root nodes formed by stages corresponding to the intra-code "index" frames (I frames). Unfortunately, the pipeline is implemented with a custom data structure interface that we were not able to replace with standard queue implementations to evaluate different implementations.

5.6.3 Smart Data Structures Versus Performance Bounds

In this section, we will evaluate how well the learning engine in the Smart Queue, Skip List, and Pairing Heap is able to optimize scancount values in our case-study applications. We will establish application performance bounds and compare the performance of using Smart Data Structures against them. We will show that application performance is sensitive to the choice of scancount values, with performance varying by as much as 32% for different values. Further, we will show that in all but one of our applications, Smart Data Structures are able to achieve at least $\frac{1}{3}$ of their potential performance improvements.

The performance bounds we are interested in are the static ideal and average performance bounds as in Chapter 5.4. To measure these bounds, we use a similar procedure as we did in Chapter 5.4: we manually vary the scancount for different runs of the application and determine the ideal scancount and average scancount over all studied scancount values. The static ideal bound on Smart Data Structure performance is the performance that would be achieved if the learning engine is able to learn the ideal scancount value and the overhead of layering optimization on top of the Flat Combining data structures is zero. Because the overhead of optimization is small but not zero, we will not be able to achieve ideal performance in some cases. The static average bound is the performance we would expect if the learning engine is unable to learn scancount values and picks scancount values at random. The potential performance improvement of Smart Data Structures is given by the difference between the static ideal and average bounds.

Figures 5-11, 5-12, 5-13, and 5-14 show the results for the Parallel Sort, Traveling Salesman, Parsec Dedup, and Parsec Ferret applications, respectively. On the x-axis, we vary the number of threads in the application. The y-axis measures the performance of the application – sometimes normalized (where indicated) relative to the average bound.

First we will discuss variation in application performance. The Parallel Sort and Traveling Salesman applications see high variation in application performance. This is reflected in the difference between the ideal and average bounds. For Parallel Sort, the percent variation ranges from 7-21% with an average of 15% over all concurrency levels studied. For the Traveling Salesman application, the percent variation ranges from 19-53% with an average of 32% over all concurrency levels. Consistent with the results from Chapter 5.6.2, the Parsec Dedup and Parsec Ferret applications see little performance variation due to different scancount values. The percent variation for Dedup ranges from 1.5-3% with an average of 2%. The percent variation for Ferret ranges from 0-8% with an average of 5%.

Now we will discuss the performance of Smart Data Structures relative to the performance bounds. We measure the percent of potential improvement that the Smart

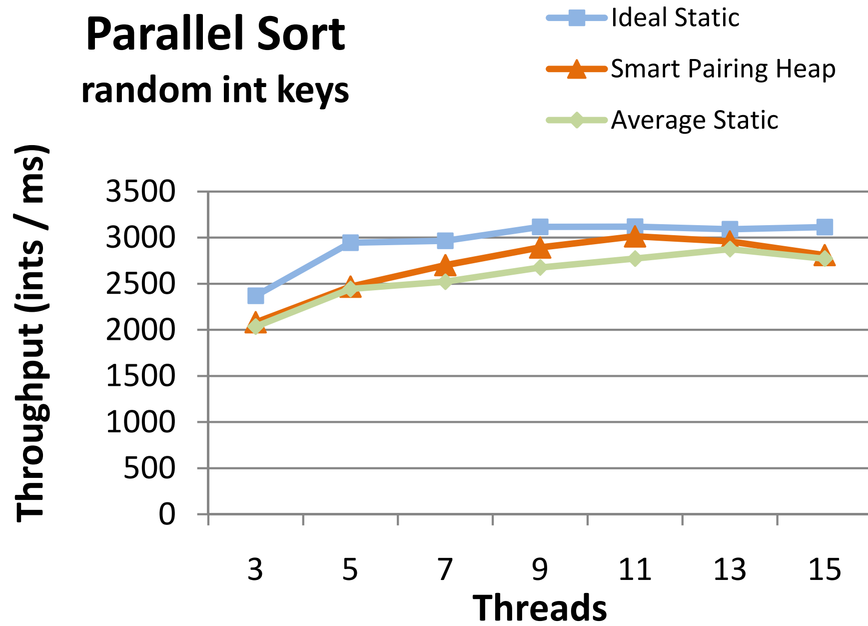


Figure 5-11

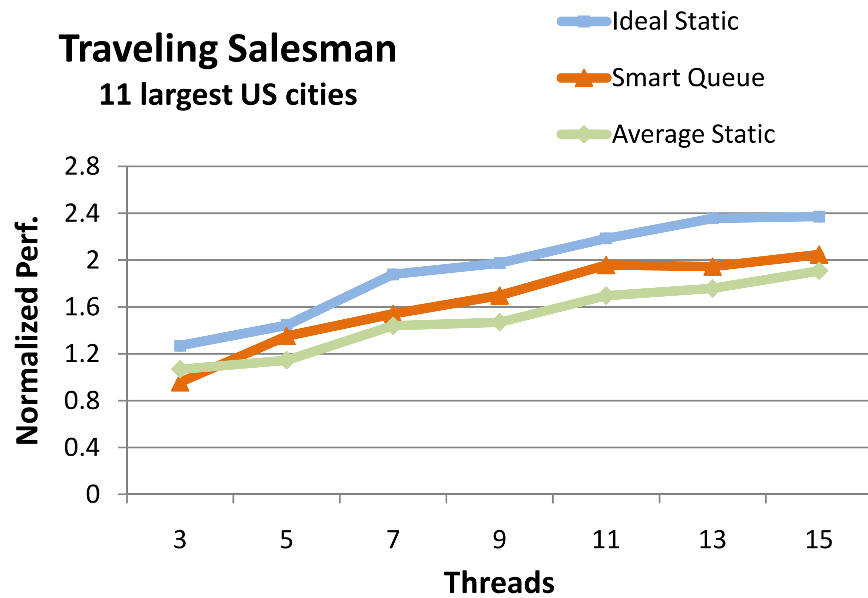


Figure 5-12

Data Structure achieves: the percent of the difference between the ideal and average bound that the Smart Data Structure achieves. For all except the Traveling Salesman application, the Smart Data Structure achieves at least $\frac{1}{3}$ of the potential improvement. For the Parallel Sort application, the percent improvement ranges from 5-70% with an average of 33%. For the Traveling Salesman application, the percent improve-

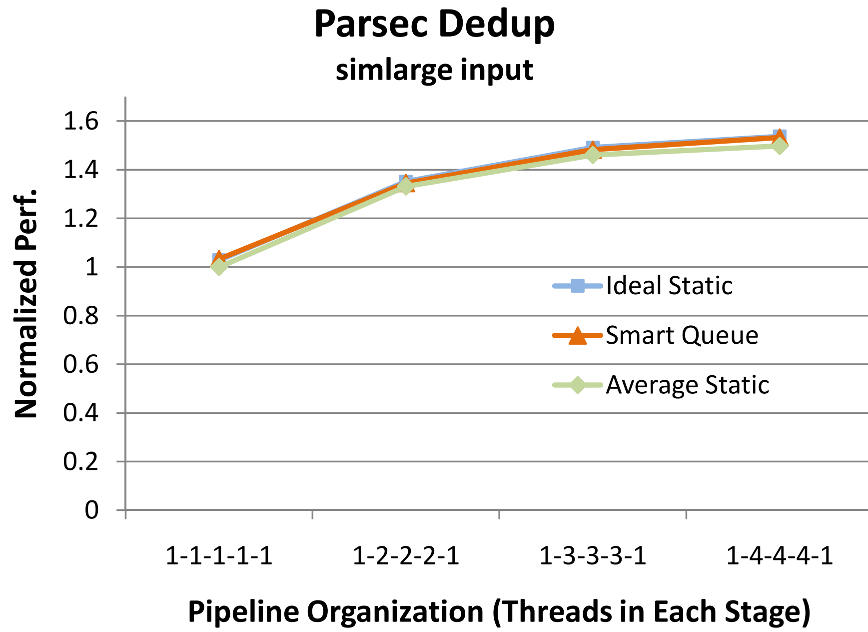


Figure 5-13

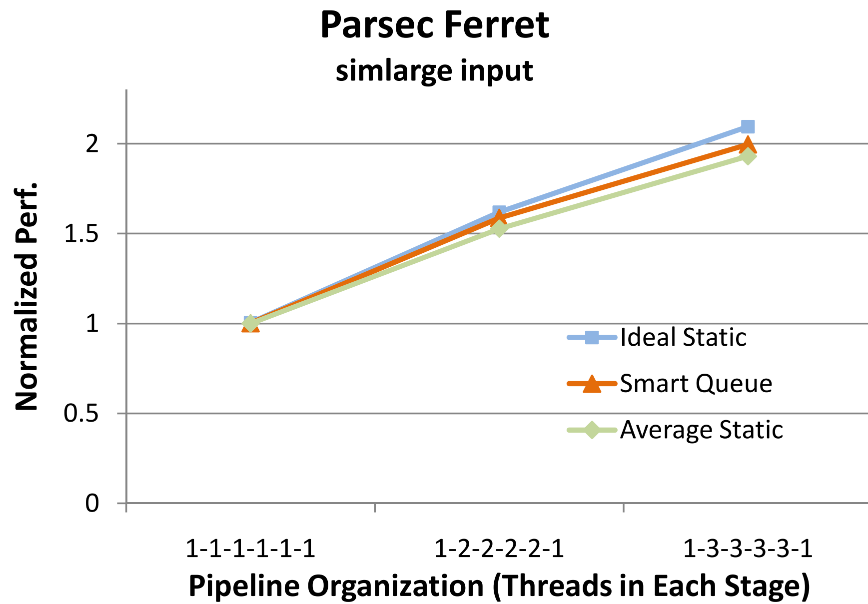


Figure 5-14

ment ranges significantly from -73-74% with an average of 31%. For Parsec Dedup and Parsec Ferret, despite the low variation in performance from different scancount values, the Smart Data Structure is still able to capitalize on the small margin of performance improvement that is available. For Dedup, the percent improvement ranges from 68-115% with an average of 86%. For Ferret, the percent improvement

ranges from 40-66% with an average of 50%. Table 5.1 summarizes these performance variation and performance improvement statistics for easy comparison.

Application	% Performance Variation	% Improvement Achieved
Parallel Sort	7-21%, 15% avg	5-70%, 33% avg
Traveling Salesman	19-53%, 32% avg	-73-74%, 31% avg
Parsec Dedup	1.5-3%, 2% avg	68-115%, 86% avg
Parsec Ferret	0-8%, 5% avg	40-66%, 50% avg

Table 5.1: Summary of Performance Variation and Smart Data Structure Improvements

The next section will build on what we have learned to provide a set of usage guidelines for when Smart Data Structures will be able to improve application performance.

5.6.4 Usage Guidelines

This section will provide a set of usage guidelines for the Smart Queue, Skip List, and Pairing Heap based on our findings in Chapters 5.6.2 and 5.6.3. We begin by summarizing the results then discuss what they teach us about when the Smart Queue, Skip List, and Pairing Heap will benefit applications most.

The main result of Chapter 5.6.2 was a demonstration that the Smart Queue, Skip List, and Pairing Heap can significantly improve performance for applications with fine-grained parallelism and high queue access rates like Parallel Sort and the Traveling Salesman application. Specifically, the Smart Pairing Heap improved performance over the state-of-the-art by up to 9% while the Smart Queue improved Traveling Salesman performance over the state-of-the-art by up to 37% – for free, just by replacing standard data structures with Smart Data Structures.

The second important conclusion we reached in Chapter 5.6.2 was that applications with coarse-grained parallelism and low queue access rates were not sensitive to queue performance. Indeed, no matter how much a novel queue algorithm improves queue performance, the improvement has little effect on overall application performance because the time spent between queue accesses is much larger than the queue access time on average. Neither Smart Data Structures nor any other novel queue

design can help in this scenario.

The third finding of Chapter 5.6.2 which we corroborated in Chapter 5.6.3 was that Smart Data Structures can achieve a significant portion (50%) of their potential improvement over the base Flat Combining data structures upon which they are built but still not improve application performance because the Flat Combining data structures may not necessarily outperform custom data structures. For the Parsec Ferret application, we found that the custom queue provided by the benchmark performed better because it was designed to address a deficiency in the application design: namely, the application pipeline stages could become imbalanced and lead to producer starvation, and the custom queue achieves higher performance in this scenario than the Smart Queue.

While the Smart Queue is designed to minimize producer starvation by processing enqueue operations ahead of dequeue operations, it is a general-purpose queue and must balance tradeoffs for the best performance across a wide variety of applications; it therefore handles producer-starvation less effectively than the custom queue. This type of custom queue (based on pthreads condition variables) is known to perform sub-optimally when producer-starvation is infrequent; thus, the Smart Queue will outperform it in these scenarios. It would be interesting, however, to extend the functionality of the Smart Queue (using the existing machine learning infrastructure) so that it can adapt at runtime between different queue algorithms. Perhaps one of the algorithms could be the custom queue from Ferret. We will investigate this possibility in future work.

Based on these findings, the Smart Queue, Skip List, and Pairing Heap can be expected to provide the most application performance improvement when:

- the application's parallelism and data structure accesses are fine-grained

- application performance is sensitive to queue performance

- Flat Combining is higher performance than other queue algorithms

Chapter 6

Scalability Results

6.1 Introduction

In this chapter, we study how well the Smart Queue, Smart Skip List, and Smart Pairing Heap perform when applications are scaled to larger numbers of threads. For different application case-studies, we will estimate the maximum number of threads to which the application can be scaled before Smart Data Structures stop maintaining a certain level of performance improvement.

Different applications can be scaled to different numbers of threads because they use Smart Data Structures differently and place different demands on them. The demands depend on the type of scaling the application uses. Some applications scale using a single Smart Data Structure and sharing it among more and more threads. An example of this type of application is a work queue application, and we call this type of scaling *concurrency scaling*.

Or, one application may utilize multiple Smart Data Structures. This can introduce two different types of scaling demands, depending on whether or not the knobs in different Smart Data Structures need to be jointly optimized.

In the simple case, the Smart Data Structures are logically independent and can be independently optimized with the same performance results as if they were jointly optimized. We refer to this type of scaling as *multi-data-structure scaling*. An example of this type of application is a distributed work queue work-stealing application. Each

thread owns a Smart Data Structure. Threads access their own Smart Data Structure most of the time but occasionally, when they run out of work, they steal work from Smart Data Structures owned by other threads.

When applications use multiple Smart Data Structures that need to be jointly optimized, we call this *multi-optimization scaling*. An example of this type of application is a multi-stage software-pipelined application: different threads belong to different pipeline stages, and successive pairs of stages communicate with one another through a Smart Data Structure. Joint optimization of the Smart Data Structures in each stage is beneficial in this case to help ensure that optimizations preserve balance in the throughputs of the pipeline stages, for the best overall pipeline throughput.

Applications can use combinations of these types of scaling as well and therefore place multiple types of demands on Smart Data Structures. For example, superscalar software-pipelined applications have multiple copies of a pipeline which they execute in parallel. The data structures in each pipeline are jointly optimized (multi-optimization scaling), and multiple pipelines run independently (multi-data-structure scaling).

Thus, an increase in application threads by a factor x does not simply imply that each Smart Data Structure will be accessed by a factor x as many threads. Nor does the storage required for Smart Data Structures necessarily increase by a factor x. In this chapter, we will evaluate a variety of typical applications to determine what combinations of scaling types they use and how the demands implied by each type of scaling affect a Smart Data Structure's ability to improve performance in these applications.

The chapter begins with three sections that analyze the different demands that concurrency scaling (Chapter 6.2), multi-data-structure scaling (Chapter 6.3), and multi-optimization scaling (Chapter 6.4) place on Smart Data Structures. We identify potential bottlenecks in the Smart Data Structures design for each type of scaling. We will show that most potential bottlenecks have been eliminated in our design. We will show that the remaining bottlenecks derive from two sources: the primary source is 1) the base data structures upon which Smart Data Structures are built

and the secondary source is 2) the scaling of the learning overheads in Smart Data Structures. We will build up an analytical model to estimate constraints on each of the three types of application scaling due to these bottlenecks.

In Chapter 6.5.2, we evaluate the application case-studies using this analytical model and estimate their scalability. We define the *scalability* of an application to be the maximum number of threads the application can use before Smart Data Structures can no longer improve performance by at least $\frac{1}{3}$ of their potential. The potential improvement is taken to be the difference between two static performance bounds: the performance achieved using the ideal knob settings and the average performance over all available knob settings.

6.2 Concurrency Demands

This section analyzes constraints on application scalability due to concurrency scaling and the demands it places on a Smart Data Structure. Concurrency scaling refers to the situation where increasing the number of threads in the application leads to more and more threads accessing a given Smart Data Structure. The largest concurrency level a Smart Data Structure can support is ultimately constrained by two sources of bottlenecks: the primary source is bottlenecks intrinsic to the base data structures upon which Smart Data Structures are built and the secondary source is bottlenecks deriving from optimization in Smart Data Structures. We refer to these as *Data Structure Bottlenecks* and *Smart Data Structure Bottlenecks*, respectively.

Data structure bottlenecks can be subdivided into bottlenecks from increased communication between components in the data structure and bottlenecks due to the data structure algorithm itself. In Chapters 6.2.1 and 6.2.2, we analyze bottlenecks of each type. We will show that the Flat Combining data structures upon which the Smart Data Structures prototype is built have a communication bottleneck which does not limit performance in practice. Rather, we will show that their main bottleneck is an algorithmic bottleneck. Chapter 6.2.3 quantifies the constraints on concurrency scaling due to the algorithmic bottleneck.

Smart Data Structure bottlenecks can be subdivided into bottlenecks from communication among optimization and data structure components, overheads from reward monitoring, and the runtime of the learning engine. Chapters 6.2.4, 6.2.5, and 6.2.6 will analyze bottlenecks of each type. We will show that there are no communication bottlenecks unless external reward monitors introduce them. Then, we will study the effect of different external reward monitors on concurrency scaling limits and demonstrate a scalable external reward monitor that avoids a bottleneck. Finally, we will show that learning overheads are currently well within the necessary range to enable concurrency scaling to hundreds of threads, and we will develop an analytical model to estimate the impact of learning overheads in existing and future Smart Data Structures. Chapter 6.2.7 will use this analytical model to quantify constraints on concurrency scaling due to learning overheads.

6.2.1 Data Structure Communication Bottlenecks

This section evaluates communication bottlenecks in the Flat Combining data structures. We will analyze the communication among application threads and the components of Flat Combining data structures. We will show that accessing the test-and-test-and-set lock component of Flat Combining data structures is a potential bottleneck in applications. However, we will show later in Chapter 6.2.2 that accessing the lock is not a scalability bottleneck in practice.

Figure 6-1 shows the scaling of a) the storage of Flat Combining components and b) the number of communication ports between application threads and the Flat Combining components as a function of the number of threads accessing the data structure. Here, we define a *communication port* to be an abstract connection between a thread and a component (or between two components). Through a communication port, the thread accesses the resources within the component. Unless otherwise stated, communication ports are assumed to be bi-directional. The number of communication ports leading to any component is informative because it indicates the maximum number of threads that can access that component at a given time. If the number of communication ports is large, there may be shared memory

bottlenecks, as we will see.

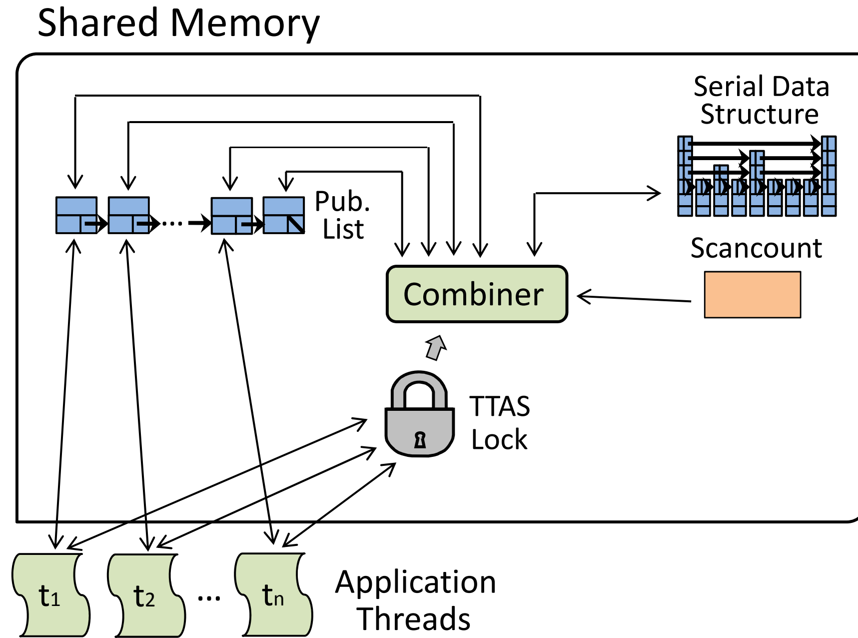


Figure 6-1: Concurrency Scaling: One Flat Combining Data Structure Shared Among n Application Threads. The communication ports between the application threads and the Flat Combining components are depicted.

As indicated in the figure and summarized in Table 6.1, the size of the publication list increases linearly with the number of application threads. Each record in the publication list is shared between the application thread that owns it and the combiner. At any given time, there is only one thread doing combining, so each record has two communication ports. The storage required for the lock is fixed for increasing numbers of threads, but its communication ports increase linearly with the number of threads. The storage and communication ports are fixed for the remaining components.

For the foreseeable future, linear scaling of storage requirements should not impact performance since cache and DRAM sizes are much larger than the number of threads. However, in the worst case, linear scaling in the number of communication ports can point to the possibility of linear scaling in the number of threads that access certain cache lines. Linear scaling in the sharing of cache lines can bottleneck typical shared memory systems, but it depends on the types of access. If all accesses are reads,

Object Type	Storage	Communication Ports	Degree of Cache Line Sharing
TTAS Lock	O(1)	n	n
Pub. List	O(n)	n/a	n/a
Pub. List Record	O(1)	2	2
Scancount	O(1)	1	1
Serial Data Structure	n/a	1	1

Table 6.1: Concurrency Scaling of Flat Combining Data Structure Components. For each component, the scaling of the storage requirements, number of communication ports, and degree of internal cache line sharing are given as a function of n, the number of threads accessing the data structure.

linear scaling is no problem. If all are writes, linear sharing can lead to large amounts of cache coherence traffic that degrade performance. The number of communication ports connected to a component can be thought of as an upper bound on the degree of sharing of any cache line within it.

Table 6.1 also indicates the degree of cache line sharing implied by the communication ports connected to each Flat Combining component. Only the lock component's communication ports scale linearly. It is a test-and-test-and-set lock. In this type of lock, all threads go through their port to spin on a single shared memory variable (the lock). The degree of sharing of the lock cache line therefore grows linearly with the number of threads. Fortunately, the test-and-test-and-set lock mostly reads the lock memory variable and avoids writes where possible: it polls the lock memory variable until the lock is free and only then attempts to modify the lock variable (via atomic test-and-set) to acquire the lock. We will see in Chapter 6.2.2 that it is also used infrequently in the Flat Combining algorithm; therefore, the test-and-test-and-set lock is not a communication bottleneck in the Flat Combining design. Chapter 6.2.2 will show that the main bottleneck in Flat Combining is an algorithmic bottleneck.

6.2.2 Data Structure Algorithm Bottlenecks

This section will evaluate the algorithmic bottlenecks of the Flat Combining data structures. The scalability bottlenecks of the Flat Combining algorithm have been previously established: Hendler et al. study them in the Flat Combining paper [12], and we review their results here for reference.

Hendler et al. benchmark the Flat Combining Queue, Skip List, and Pairing Heap and other state-of-the-art algorithms (described in Chapter 5.2) using the first mode of the benchmark we describe in Chapter 5.1. In the benchmark, one Flat Combining data structure is instantiated and accessed by all threads. Threads run independently, issuing add and remove operations on the data structure at random with equal likelihood. In this case, threads request accesses as fast as possible with no post computation between accesses. The benchmark is run with 64 threads on a 128-way Oracle® Enterprise SPARC T5140® server (Maramba) machine running Solaris 10. It is a 2-chip Niagara system, with each chip having a shared L2 cache and 8 cores that multiplex 8 hardware threads each.

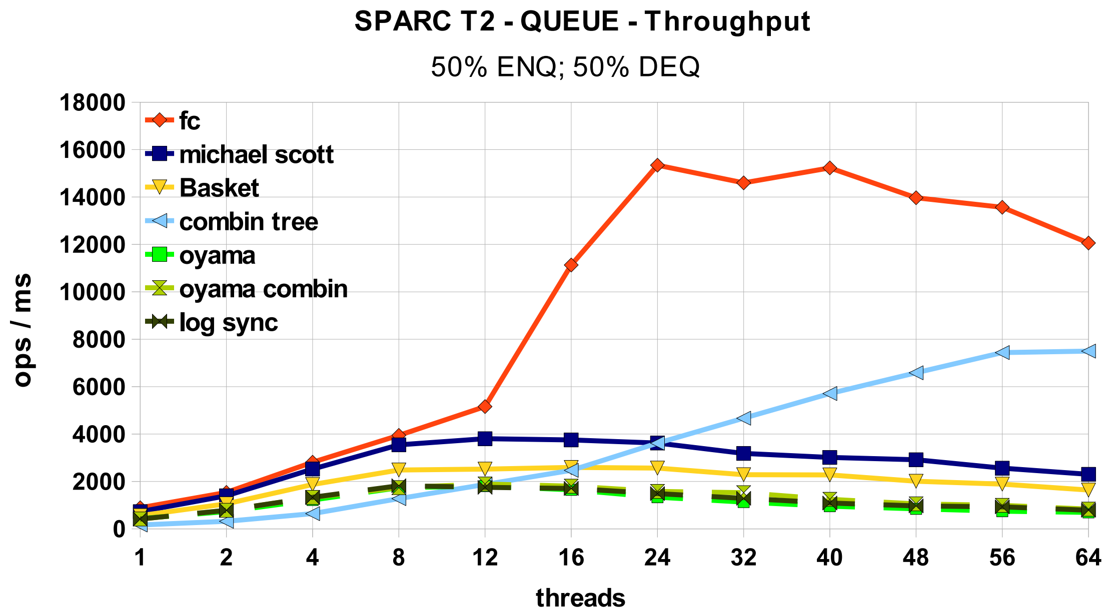


Figure 6-2: Concurrency Scaling of the Flat Combining Queue. The Flat Combining Queue is compared to the best existing queue algorithms on a SPARC T2 system. It reaches maximum performance at 24 threads but outperforms all prior queues up to 64 threads. Some time after 64 threads, the Combining Tree Queue is expected to overtake it as the highest performance queue.

Figures 6-2, 6-3, and 6-4 show the benchmark throughput results as the number of threads is varied from 1 to 64 threads. For up to 64 threads, The Flat Combining Queue, Skip List, and Pairing Heap substantially outperform the best existing queue and priority queue algorithms.

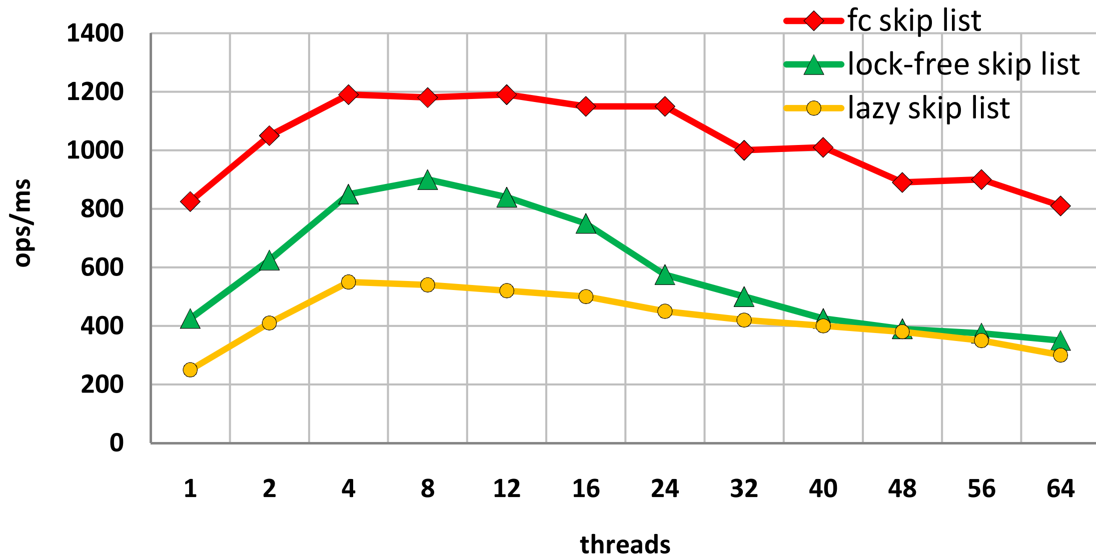


Figure 6-3: Concurrency Scaling of the Flat Combining Skip List. The Flat Combining Skip List is compared to the best existing priority queue implementations on a SPARC T2 system. It reaches maximum performance at 4 threads but outperforms all prior priority queues up to 64 threads and beyond.

Nevertheless, none of the state-of-the-art algorithms (except for one) exhibit ideal scaling. Only the Combining Tree Queue scales linearly, but, as we will see, it uses more synchronization operations per data structure operation than the Flat Combining Queue, and thus incurs more overhead and has lower absolute performance [12]. The maximum throughput for the Flat Combining Queue, Skip List, and Pairing Heap occurs at 24, 4, and 12 threads, respectively. Note that while the Flat Combining data structures have reached their maximum throughput, their throughput continues to outperform all other prior art up to 64 threads. After reaching the max throughput, throughput somewhat degrades due to the overhead of traversing a longer publication list when combining [12].

Hendler et al. analyze the source of the Flat Combining algorithm's performance improvements over the best existing queue algorithms. They show that Flat Combining's improvements derive from reduced synchronization overheads, and we review their findings here. We also use these results to show that the linear scaling in the

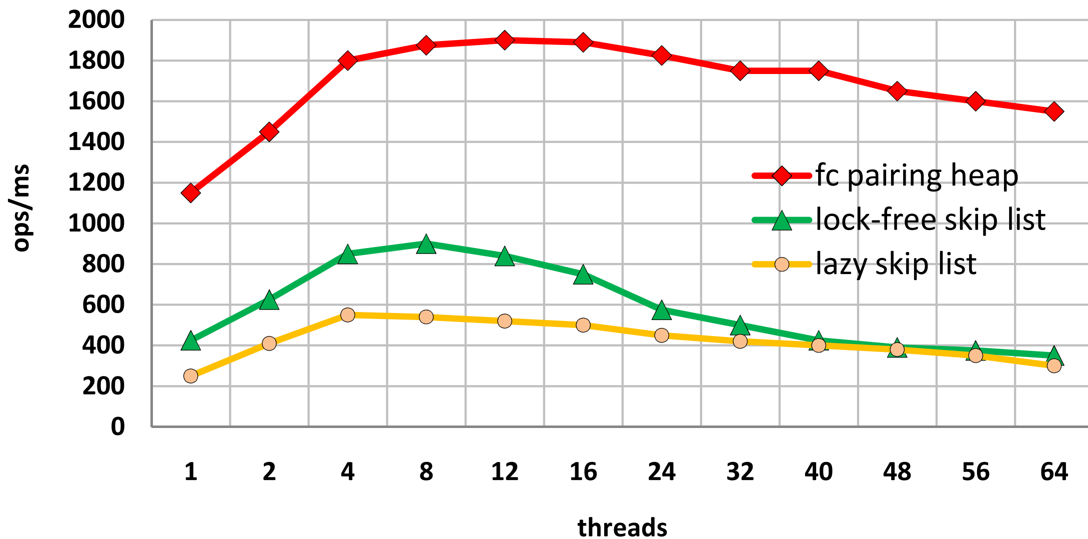


Figure 6-4: Concurrency Scaling of the Flat Combining Pairing Heap. The Flat Combining Pairing Heap is compared to the best existing priority queue implementations on a SPARC T2 system. It reaches maximum performance at 12 threads but outperforms all prior priority queues up to 64 threads and beyond.

number of communication ports leading to the test-and-test-and-set lock do not lead to the bottleneck in Flat Combining performance. Rather, the bottleneck is due to the Flat Combining algorithm and its use of the lock to serialize data structure operations.

Hendler et al. measure the rate of synchronization operations issued by each of the state-of-the-art queue data structures. They collect these statistics for the same benchmark used to get Figure 6-2. Recall that the experiments run on a SPARC machine. Synchronization operations are hardware-supported primitives. On the SPARC, the Flat Combining data structures use the SPARC compare-and-swap (CAS) synchronization instruction. Thus, the rate of synchronization operations is given in terms of the number of CAS instructions issued per data structure operation. The CAS instruction compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This guarantees that the new value is calculated based on up-to-date

information.

The rate of synchronization operations is divided into two parts: the rate of successful and unsuccessful CAS operations per data structure operation. Figure 6-5 shows the rate of successful CAS operations per data structure operation for the best existing concurrent queues. In all queues except for the Flat Combining and Combining Tree Queues, the rate of successful CAS operations is approximately constant as the number of threads increases. In the Combining Tree Queue it increases logarithmically with the number of threads because the depth of the synchronization tree in the queue grows logarithmically. In Flat Combining, the necessary rate of CAS successes actually decreases. This is the primary source of Flat Combining's performance improvements.

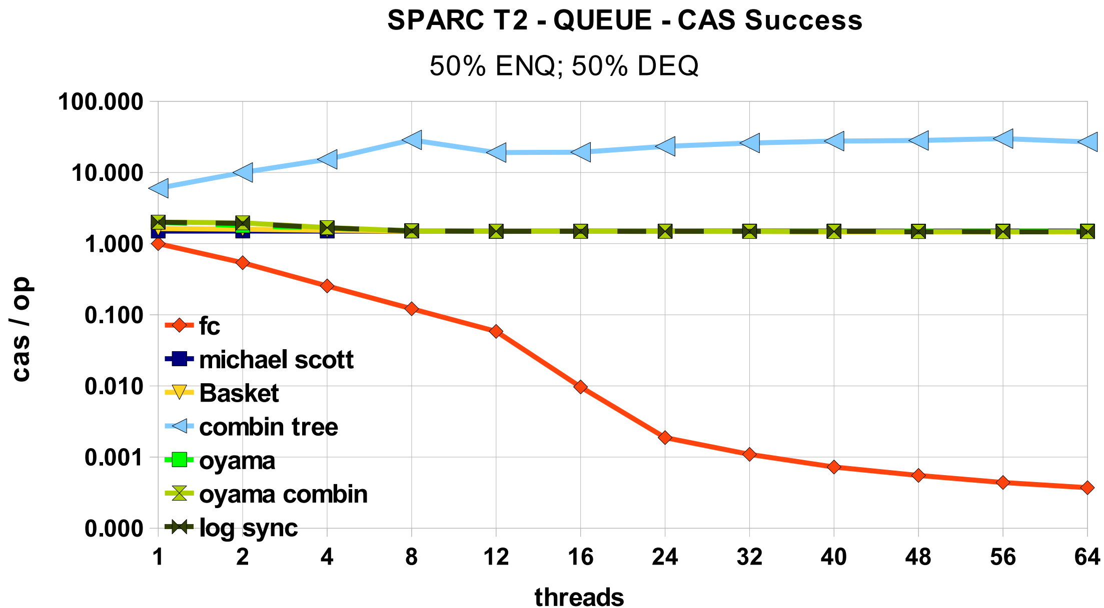


Figure 6-5: A Comparison of the Number of Necessary CAS Successes per Data Structure Operation for Different Queues. While for most queues the necessary rate is approximately fixed as the number of threads increases, the necessary rate for the Flat Combining queue decreases. This is the primary source of its performance improvements over the other queues.

The reason for the reduction in the necessary successful CAS operations is that the Flat Combining algorithm only issues CAS operations when one combiner has finished and another thread wishes to become the combiner by acquiring the lock.

While a thread has the lock and is the combiner, it performs many data structure operations for the cost of that one CAS operation. In contrast, the other data structure algorithms require at least one successful CAS operation per individual data structure operation.

The reason that the CAS success rate decreases as the number of threads increases in Flat Combining, is that adding threads increases the efficiency of combining and enables the same rate of data structure operations to be performed for fewer CAS operations. Adding threads increases the global rate of requests which results in more requests for a thread to fulfill while it is the combiner. Those extra operations take more time and reduce the rate at which the combiner changes. Since CAS operations are only required when the combiner changes, the result is a reduced rate of CAS successes.

Flat Combining also improves performance over the other queues by reducing overheads due to CAS failure rates. Figure 6-6 shows the rate of unsuccessful CAS operations per data structure operation for the different queues. For all except the Flat Combining Queue, the rate of unsuccessful CAS operations per data structure operation increases as the number of threads increases. In these queues, the rate increases because a) their algorithms loop, attempting CAS operations until one succeeds and b) the likelihood of success decreases as the number of threads increases because more threads compete to complete the CAS operation on the same memory locations.

In the Flat Combining queue, the rate of CAS failures per data structure operation is much lower because a) fewer successful CAS operations are needed to complete each data structure operation in the first place and b) the algorithm uses a test-and-test-and-set lock to prune most CAS attempts. The test-and-test-and-set lock only attempts CAS operations when a non-atomic read of the lock variable indicates the lock is free and the CAS attempt is likely to succeed.

It is also important to note that, in Flat Combining, the total rate of CAS operations per data structure operation does not increase but actually decreases as the number of threads increases. This shows that the lock variable is less and less fre-

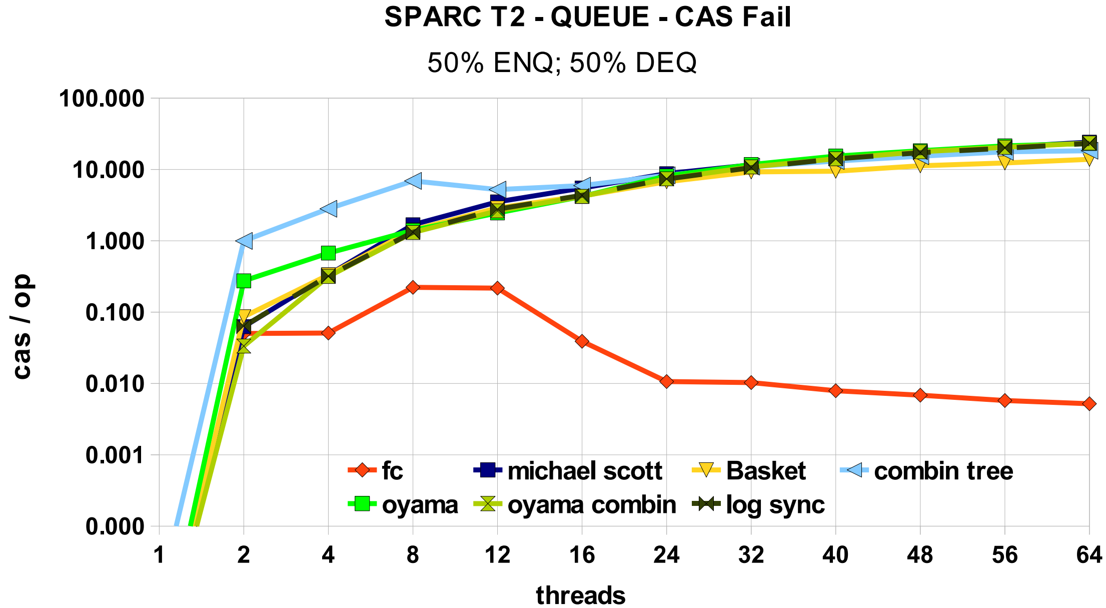


Figure 6-6: A Comparison of the Number of CAS Failures per Data Structure Operation for Different Queues. While for most queues the failure rate increases as the number of threads increases, the failure rate only initially increases then decreases with the Flat Combining Queue.

quently atomically modified even though the number of threads that may access the lock is increasing. In Chapter 6.2.1, we identified that the lock could become a communication shared memory bottleneck if it were frequently atomically modified since the cache line containing the lock variable must be shared by more and more threads as the number of threads increases. The fact that the total rate of CAS operations decreases shows that the lock variable is infrequently modified and that the lock is not, therefore, a communication bottleneck in Flat Combining. Lock communication bottlenecks are not responsible for the concurrency scaling limits in Flat Combining data structures.

Rather, we conclude that the bottleneck is an algorithmic bottleneck, deriving from the way the lock is used: the Flat Combining algorithm uses the lock to coarse-lock a serial data structure and serialize all operations upon it. Flat Combining overcomes that serialization up to a certain number of threads through combining and allowing the lock holder to perform multiple data structure operations instead of just its own while it holds the lock. Eventually, however, serialization of data

structure operations causes performance to plateau. In Figure 6-2, for example, we see that the Flat Combining Queue reaches maximum performance at 24 threads.

At concurrency levels beyond 64 threads, scalable techniques like the Combining Tree Queue will eventually overtake the performance of the Flat Combining Queue. Nevertheless, up to 64 threads, the Flat Combining queue still substantially outperforms the Combining Tree Queue. Because we are interested in knowing how many threads Smart Data Structures can scale to via concurrency scaling before they no longer improve upon prior data structures, we will estimate that the Flat Combining algorithm limits concurrency scaling of the Smart Queue to 64 threads. No scalable algorithms for the priority queue are known (to our knowledge), so the Flat Combining algorithm does not limit scalability for the Smart Skip List and Smart Pairing Heap.

There is, however, ongoing research in hierarchical Flat Combining algorithms which use multiple combiners and may greatly extend the scalability of the Flat Combining algorithm so that it continues to outperform scalable algorithms like the Combining Tree even at large parallelism scales. Our techniques for optimizing the scancount in Flat Combining will still apply to hierarchical Flat Combining algorithms, and there will be new opportunities to learn the best number of parallel combiners and partitioning of threads among them for the best performance.

6.2.3 Summary of Data Structure Concurrency Constraints

In Chapters 6.2.1 and 6.2.2 we studied *Data Structure Bottlenecks*. We analyzed communication and algorithmic bottlenecks in the Flat Combining data structures upon which our Smart Data Structures prototype builds. In this section, we will summarize these findings and quantify the concurrency scaling limits that these bottlenecks place on applications.

The main bottlenecks in concurrency scaling are as follows. In Chapter 6.2.1, we showed that the Flat Combining data structure exhibits only one potential communication bottleneck: its test-and-test-and-set lock. Chapter 6.2.2 showed that the communication between application threads and the lock is not the limiter in scaling

the Flat Combining data structures up to 64 threads. Rather, the algorithm is; the limit derives from the way the algorithm uses the lock to serialize accesses to the serial data structure. Flat Combining overcomes the cost of serialization up to a point by reducing synchronization overheads. For some data structures, however, scalable algorithms are known that will eventually outperform Flat Combining.

The main conclusion of Chapter 6.2.2 is that the Flat Combining Queue outperforms prior art up to 64 threads and that the Flat Combining Skip List and Pairing Heap will continue to outperform prior art up to arbitrary numbers of threads because no scalable algorithms are known.

These data structure scaling limits imply limits on overall application scalability. There are three ways an application can scale (concurrency scaling, multi-data-structure scaling, and multi-optimization scaling), but if the application relies entirely on concurrency scaling as the number of application threads is increased, the concurrency scaling of the Smart Data Structure determines application scalability. In other words, such an application can scale to 64 threads if it uses Smart Queues before the Smart Queue will no longer be the highest performance data structure. Since scalable algorithms comparable to the Skip-List-based and Pairing-Heap-based priority queue are not known, the application can scale to unbounded number of threads if it uses these data structures. Table 6.2 lists these constraints for convenience.

Type	max n
Smart Queue	64
Smart Skip List	unbounded
Smart Pairing Heap	unbounded

Table 6.2: Concurrency Scaling Constraints from Flat Combining Data Structures. n is the number of threads concurrently accessing a given Smart Data Structure, and the max n is defined to be the maximum number of threads before an alternative algorithm will outperform the Smart Data Structure.

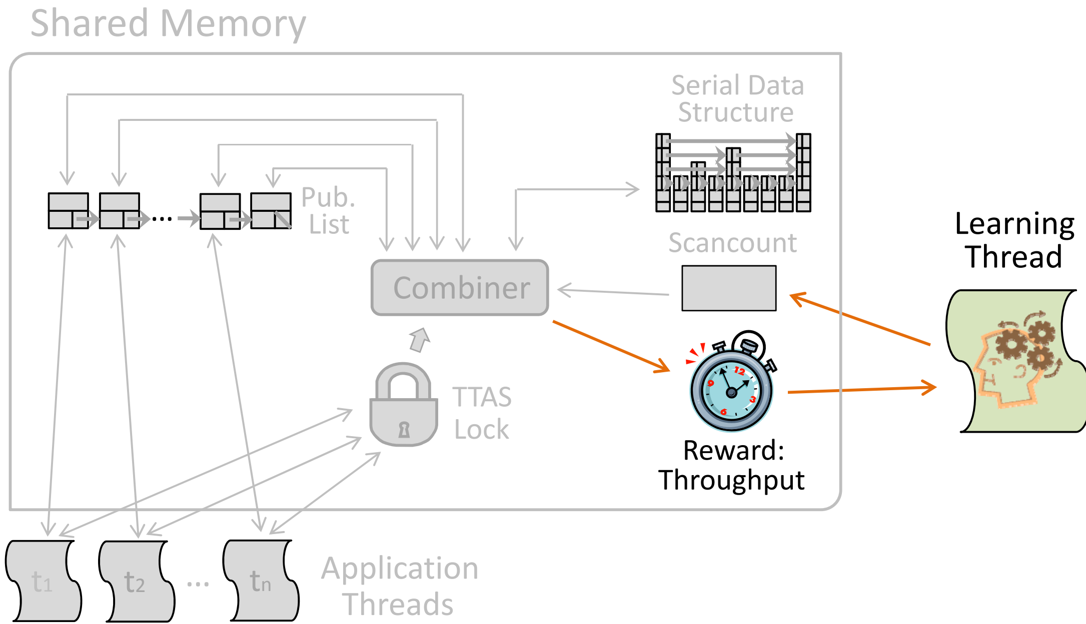
6.2.4 Smart Data Structures Communication Bottlenecks

This section begins the analysis of Smart Data Structure Bottlenecks: the incremental bottlenecks that are introduced by the learning components in a Smart Data Struc-

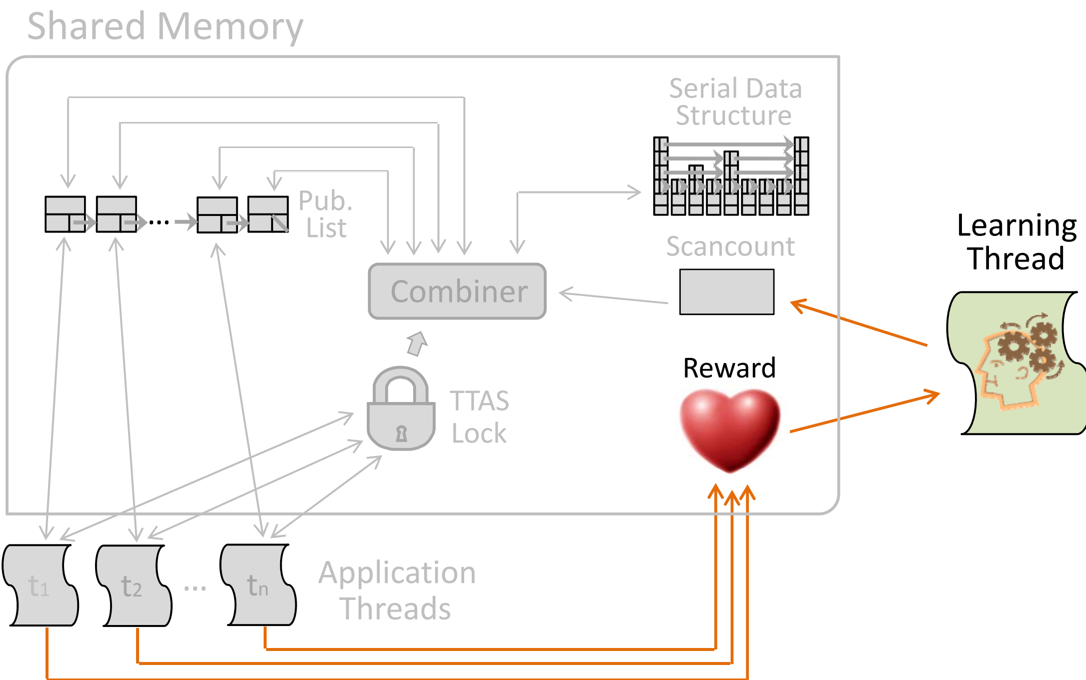
ture to optimize the knobs in the base data structure. There are three sources of potential bottlenecks which may limit concurrency scaling: 1) communication bottlenecks among application threads, Flat Combining components, and learning components, 2) bottlenecks introduced by collecting and supplying reward for learning, and 3) bottlenecks from increasing learning overheads. This section focuses on the first source: communication bottlenecks. We will show that optimization does not introduce incremental communication bottlenecks unless external reward monitors contribute them. The other two sources of potential concurrency bottlenecks will be covered in Chapters 6.2.5 and 6.2.6, respectively.

Figure 6-7 shows a Smart Data Structure shared among all application threads. It depicts the optimization components of Smart Data Structures and their communication with each other, the application threads, and the base Flat Combining components. The optimization components of Smart Data Structures are the reward monitor and the learning thread. Two reward configurations are depicted: Figure 6-7a shows the reward configuration where the Smart Data Structure provides its own internal throughput as the reward while Figure 6-7b shows the case for external performance monitors (see Chapter 4.2). As before, the number of communication ports between components indicates the maximum number of threads that can be accessing a component at any time.

Augmenting Flat Combining with optimization components requires adding one port to the scancount to enable the learning thread to optimize and modify it. The internal reward monitor requires one write and one read port for the combiner and learning thread, respectively. Since each thread may update the reward in the case of external monitors, external monitors require one write port per application thread and one read port for the learning thread. As the application scales to larger numbers of threads, the storage for the scancount and learning thread is invariant. The storage for the reward depends on the reward monitor implementation: if the reward is stored in one central variable, storage is fixed as the number of threads increases; if it is stored in distributed variables, there is one distributed variable per thread, so storage grows linearly. Thus, reward storage grows at most linearly. As before, cache and



(a) Internal Reward Mode



(b) External Reward Mode

Figure 6-7: Concurrency Scaling: One Smart Data Structure Shared Among All Application Threads. The communication ports between optimization components, application threads, and the Flat Combining components are depicted for two different reward modes.

DRAM sizes should easily accommodate linear scaling since the number of threads is comparatively small. Table 6.3 summarizes the storage and communication scaling related to Smart Data Structures optimization components.

Object Type		Storage	Communication Ports	Degree of Cacheline Sharing
Scancount		O(1)	2	2
Reward Monitor	Internal	O(1)	2	2
	External	O(n)	n+1	2 (n+1 naïve)
Learning Thread		O(1)	2	1

Table 6.3: Concurrency Scaling of Smart Data Structure Optimization Components. For each component, the scaling of storage requirements, communication ports, and the degree of cache line sharing are given as a function of n, where n is the number of threads accessing the Smart Data Structure.

Only the reward monitor's communication ports may scale linearly, and the ports only scale linearly if the external reward mode is used. In naïvely implemented external reward monitors, this can lead to shared memory variables inside the reward monitor being shared by $n+1$ threads. Unfortunately, n of these sharers are writers, and if writes are high frequency, this can potentially cause cache lines to ping-pong between threads (thus cores) and degrade performance. Chapter 6.2.5 will study different reward monitor implementations and the effect of their overhead on Smart Data Structure performance. In addition, it will demonstrate a scalable external reward monitor that removes the bottleneck by reducing cache line sharing to 2-ways.

It is interesting to question the need for support for external monitors given that they may introduce communication or other bottlenecks into the Smart Data Structures design. We acknowledge that the internal reward mode is safer because it is free from communication bottlenecks. However, one of the design goals of Smart Data Structures is to provide a learning framework that will work well in a variety of systems and applications, and we acknowledge that the internal reward mode may not always provide the best indication of application performance. The internal reward mode measures application performance by measuring Smart Data Structure throughput. The problem is that different applications may have other goals than maximizing throughput: e.g. minimizing latency. Our Smart Data Structures design supports external monitors to enable application developers to measure progress

toward application goals in application-specific ways. The scalable reward monitor evaluated in the next section gives this freedom while avoiding communication and other bottlenecks.

6.2.5 Smart Data Structures Reward Bottlenecks

This section looks at the concurrency scaling bottlenecks in Smart Data Structures due to reward overheads. Different application use-cases will imply different numbers of threads using the reward monitor and different reward update frequencies. Our goal in this section is to determine typical demands as well as upper bounds on the demands that applications can place on the reward monitor before its overheads limit the ability of Smart Data Structures to improve performance. We evaluate two external monitors – Application Heartbeats [17] and a scalable monitor we developed based on a concurrent counter algorithm we call *lazy counters* – and study the effect of update rates on reward overheads.

In the case of Heartbeats, n application threads atomically increment a shared counter variable, making the degree of cache line sharing increase linearly as more threads are added. The lazy counter avoids *n-way* sharing by assigning a separate counter to each thread and spacing counters so they map to different cache lines. To increment the reward, a thread increments its own counter. When the learning thread reads the reward, it sums over the counters from all threads (see Appendix A for additional details). Thus, each counter is shared no more than 2 ways. The reason this works is that the learning thread reads the reward infrequently compared to the rate at which the application threads update the reward. As for storage scaling: the storage of Heartbeats is invariant to increasing numbers of threads, but the storage for the lazy counter monitor grows linearly. Like before, we assume that the number of threads is small compared to cache and DRAM sizes, so this should have negligible impact on performance.

Since application threads may desire high frequency access to Smart Data Structures and update the reward once per access, it is imperative that the external reward monitor can sustain reward updates at comparable frequencies, without degrading

performance. Hendler et al. [12] demonstrate that Flat Combining data structure throughput can exceed 16000 ops/ms in large-scale machines which equates to reward update rates in excess of 16 MHz total across all threads. Experiments on our 16-core Xeon® system achieve throughput up to 3500 ops/ms which equates to sustaining a 3.5 MHz total reward update rate. To summarize, an external reward monitor should scale beyond a 3.5 MHz total reward update rate for the best performance on our system, and ideally, beyond 16 MHz for the best performance on large machines with greater parallelism.

To measure the maximum update frequencies Heartbeats and lazy counters can sustain, we make a slight modification to our throughput benchmark described in Chapter 5.1. We instantiate a Smart Data Structure and use the first mode of the benchmark (the mode in which application threads run independently), but, instead of looping through a cycle of issuing an operation on the Smart Data Structure, updating the reward, and executing a post computation delay loop, the threads skip the operation on the Smart Data Structure in each cycle. Threads still update the reward each cycle as normal. The learning engine is enabled, as normal, and reads the reward periodically to optimize the scancount. The scancount is merely ignored since no operations on the Smart Data Structure are being performed. We fix the number of threads at the maximum we use in our experiments on our machine (14), and we vary the length of the post computation delay loop between a thread's successive reward updates.

The "throughput" in this benchmark is therefore the total reward update rate across all threads that is achieved. It is the product of the number of threads and the average reward update rate achieved by each thread. By setting different post computation delays between updates, we request different per-thread update rates which the reward monitors may or may not be able to sustain. Ideally, a given reward monitor sustains the requested per-thread rate, and the ideal total update rate is the product of the number of threads and the reciprocal of the post computation delay (ignoring loop overheads between post computation periods). When the reward monitor cannot sustain the requested per-thread rates, the total update rate falls

below the ideal.

Figure 6-8 shows benchmark results for the Heartbeats reward monitor. It shows the achieved total reward update rate and the ideal rate as a function of the requested per-thread update rate. We find that Heartbeats sustain ideal update rates until an inflection point at approximately a 5.7 MHz total reward update rate, beyond which the reward update rate saturates. The maximum reward update rate is between 6.6 and 7.6 MHz. Before the inflection point, the 5.7 Mhz that Heartbeats achieves does exceed the necessary 3.5 MHz for good performance on our 16-core Xeon® system. To an approximation, 5700 data structure operations per ms is therefore the Smart Data Structures throughput beyond which using Heartbeats as the reward monitor will introduce performance bottlenecks.

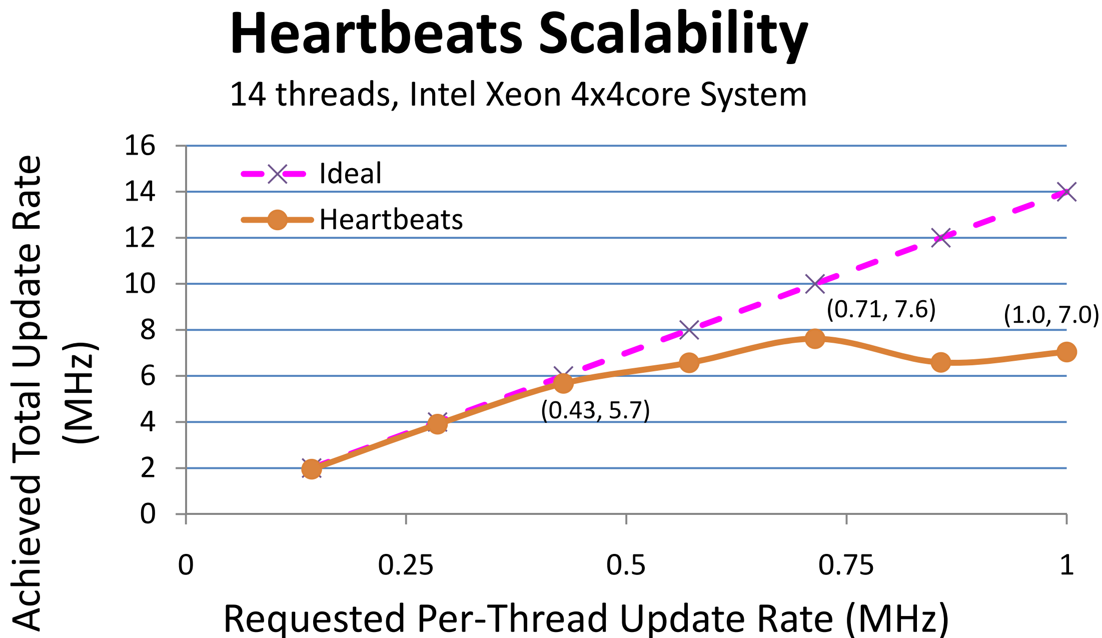


Figure 6-8: Heartbeats Scaling. The total update rate achieved by Heartbeats is compared to the ideal update rate for different requested per-thread update rates. Heartbeats sustain the ideal update rate until an inflection point at a total update rate of approximately 5.7 MHz.

Figure 6-9 shows the scaling of the reward monitor built upon lazy counters. We run the same benchmark on a 48-core (quad 12-core) AMD® Opteron® 6168 system with 1.9GHz cores and 32GB of DRAM. We run under Debian Linux (kernel version 2.6.31.13) and compile with the same version of gcc and optimization levels: gcc 4.3.2

and O3. We run the experiment with 46 threads and show the ideal and achieved total reward update rate as a function of the requested per-thread reward update rate. We also show the Heartbeats total reward update rate for reference. Again, the ideal curve ignores loop overheads, but loop overheads are significant in this case since higher update frequencies are studied. The results demonstrate that the lazy counter scales very well up to and beyond 46 cores and 360 MHz total reward update rates, making it a suitable reward monitor for high-concurrency applications of Smart Data Structures. As the figure shows, Heartbeats fail to sustain these rates.

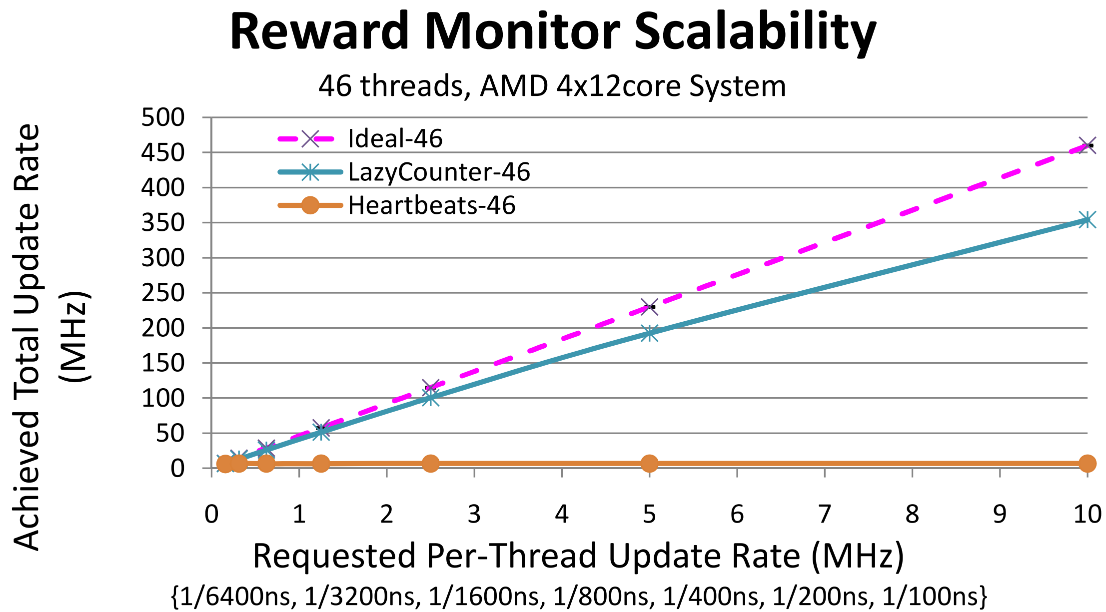


Figure 6-9: Reward Scaling for Different Reward Monitors. The total update rate achieved by the Lazy Counter monitor and Heartbeats monitor are compared to the ideal for different requested per-thread update rates. The Lazy Counter monitor nearly achieves the ideal total update rate while Heartbeats saturate at a much lower total update rate.

While the overhead of the Lazy Counter monitor is much better, the Heartbeats reward monitor is nevertheless adequate for our machine. In Figure 6-10, we demonstrate this fact by comparing end-to-end Smart Data Structure performance on our machine using both monitors. The benchmark is the second mode of the benchmark we describe in Chapter 5.1 where one master thread produces work for worker threads. We fix the number of threads at 14 and vary the post computation between Smart Data Structure accesses. The results indicate similar performance between

Heartbeats and lazy counters. This illustrates two interesting points: a) Heartbeats provide sufficient scalability for our benchmarks on our machine and b) lazy counters do not sacrifice performance at small scales to achieve good scalability at large scales. This suggests that the monitor based on lazy counters can be used at any application scale and achieve the best performance.

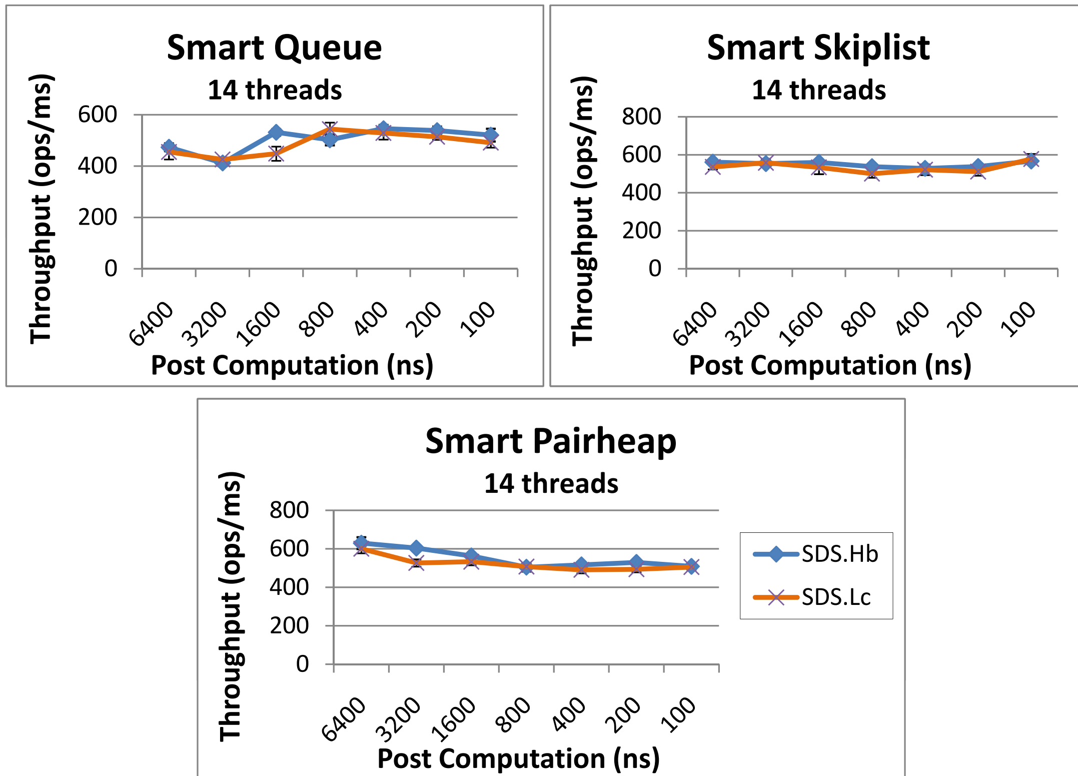


Figure 6-10: The Practical Impact of Reward Scaling on a 16-core Intel Xeon System. The figures compare benchmark throughput using Heartbeats vs the Lazy Counter reward monitor for the Smart Queue, Skip List, and Pairing Heap. The results indicate that for the benchmark and concurrency levels in our 16-core Xeon system, both reward monitors achieve similar throughput.

6.2.6 Smart Data Structures Learning Bottlenecks

Recall that we wish to determine the maximum number of threads that an application can utilize before Smart Data Structures in the application can no longer achieve at least $\frac{1}{3}$ of their potential performance improvement. Increases in the learning runtime due to concurrency scaling may eventually become a bottleneck that limits

this number of threads.

The focus of this section is the impact of concurrency scaling on the learning runtime. We are interested in determining a) how the learning runtime varies with the number of threads accessing a Smart Data Structure and b) how much the learning runtime can increase before the Smart Data Structure optimizes knobs too slowly to meet the requested performance improvement target.

First, Chapter 6.2.6 will develop an analytical model to estimate the scaling of the learning runtime. We will show that the learning runtime may or may not depend on the number of threads, depending upon the type of Smart Data Structure used. The learning runtime actually depends on the number of parameters being learned not the number of threads. For some Smart Data Structures, the number of parameters is fixed so the learning runtime does not increase. For others, the number of parameters does increase with the number of threads.

Then, Chapter 6.2.6 will artificially slow down the learning runtime to identify the maximum runtime that can be tolerated before missing the performance improvement target. We define this maximum runtime as the *tolerable runtime*, T_t. We will measure the tolerable runtime and show that Smart Data Structures can learn many parameters before they are unable to meet their performance improvement targets. For Smart Data Structures where the number of parameters scales with the number of threads, this will enable concurrency scaling to large numbers of threads.

In later sections (Chapter 6.3.1 and 6.4.1), we will expand upon the analytical model developed here to estimate the learning runtime due to multi-data-structure and multi-optimization scaling in addition to concurrency scaling. Those sections will use the model to estimate constraints on multi-data-structure and multi-optimization scaling. Ultimately, Chapter 6.5.2 will combine all of the constraints to estimate the scalability of a variety of case-study applications.

Learning Runtime

This section develops an analytical model to estimate the constraints on concurrency scaling due to the scaling of learning overheads in Smart Data Structures. We will

begin by developing a formula to relate the learning runtime to the number of threads accessing a given Smart Data Structure. Then, we will develop an inequality that expresses a constraint on the maximum number of threads that can access a Smart Data Structure before it fails to deliver at least $\frac{1}{3}$ of its potential performance improvements.

We start with the formula to relate the learning runtime to the number of threads accessing a Smart Data Structure. The learning runtime may or may not increase as the number of threads increases. It depends on which Smart Data Structure is being used. The learning runtime actually depends on the number of parameters that must be learned. Different Smart Data Structures have different types of knobs, and different types of knobs require learning different numbers of parameters. Furthermore, as the number of threads increases, the number of parameters grows differently depending on the type of knob.

In the case of the Smart Queue, Skip List, and Pairing Heap, the knob is the scancount, and the scancount is a discrete-valued variable. The number of parameters required for learning a discrete-valued knob is equal to the number of different discrete values that the variable can take on. In our design, we fix the possibilities for the scancount to 13 different discrete values. Thus, the number of discrete values does not change as more and more threads access a Smart Queue, Skip List, and Pairing Heap, and subsequently, the learning runtime does not change.

In the case of the Smart Lock, however, the number of parameters does grow as the number of threads increases, and the learning runtime subsequently increases as well. The Smart Lock uses a permutation-order knob. This knob supplies each thread with a priority to use in the priority lock. The permutation-order requires learning one parameter for each thread that accesses the Smart Lock. Thus the number of parameters is equal to the number of threads.

Thus, we wish to express the learning runtime as a function of the number of parameters, p, not the number of threads. To understand the precise dependence of the learning runtime on p, we need a precise definition of the learning runtime. Recall the learning algorithm in Alg. 1 in Chapter 4.3. The algorithm executes a

cycle of three computations. In the first stage of the cycle, the algorithm samples knob settings from the latest learned parameters to try them out and measure the resulting reward. In the second stage, it computes the gradient. In the third stage, its tests for convergence of the gradient and, if converged, steps the learned parameters in the direction of the gradient. We denote the runtime of these computations as T_s, T_g, and T_c, respectively, to represent the sampling, gradient, and convergence test runtimes. We define the *learning runtime*, T_l, as the period between successive gradient computations. This is equal to the runtime of one cycle:

$$T_l \ = \ T_s \ + \ T_g \ + \ T_c$$

The sampling runtime, T_s, is independent of the number of threads in the application and the number of parameters being learned. It is set by our algorithm and approximately the same for all applications. The gradient is determined by solving the least-squares problem using QR factorization. Its runtime in our implementation is $T_g \ = \ 2{\cdot}p^3 \ + \ 4{\cdot}p^2$ floating point operations. The convergence test runtime is $T_c \ = \ p^2 \ + \ p$ floating point operations. Thus, in terms of p, T_l is:

$$T_l \ = \ T_s \ + \ 2{\cdot}p^3 \ + \ 5{\cdot}p^2 \ + \ p$$

For $p \geq 13$, we approximate T_l as:

$$T_l \approx T_s \ + \ 2{\cdot}p^3$$

We need not make any assumptions about how large T_s is relative to the other components because we will cancel T_s later. We also note that, to determine scaling limits, we are interested in determining the maximum p (and subsequent learning runtime) at which Smart Data Structures can still maintain $\frac{1}{3}$ of their potential performance improvement, as requested. For large p, (e.g. $p \ = \ 50$), the error in the approximation we made is less than 5%.

Now that we have T_l in terms of the number of parameters, we would like to

develop an inequality bounding the values of p to those where a given Smart Data Structure can sustain the requested performance improvement. After the learning runtime increases beyond some threshold, the Smart Data Structure will no longer be able to sustain the requested performance improvement. We call this threshold, the *tolerable runtime*, T_t. Thus we constrain T_l such that

$$T_l \leq T_t$$

We substitute our approximation of T_l to give this inequality in terms of the number of parameters:

$$T_s \; + \; 2{\cdot}p^3 \leq T_t$$

If we determine the tolerable runtime for a given application and Smart Data Structure, we can determine the range of p for which the constraint is satisfied. To measure T_t, we will artificially slow down the learning runtime in the application's Smart Data Structure. We will empirically measure when the learning runtime causes the Smart Data Structure to miss its requested performance improvement target.

We must introduce the slowdown consistently with how the learning runtime would slow down in reality if the number of parameters were actually increased rather than artificially modeled. To do this, we choose some fixed number of parameters, p_t, and model an increase in this number of parameters by introducing a slowdown factor of α_t in the terms of the learning runtime that depend on the number of parameters:

$$T_t \approx T_s \; + \; \alpha_t{\cdot}2{\cdot}{p_t}^3$$

We refer to α_t as the *tolerable slowdown* of the variable component of the learning runtime. It is important to note that this is the slowdown of only the portions of the learning time that depend on the number of parameters.

In general, the tolerable runtime, T_t will be different for different Smart Data Structures. We will need to determine T_t for each Smart Data Structure. To determine T_t for a given Smart Data Structure, we need only rerun our benchmarks from Chapter

5.4 and Chapter 5.5 and artificially slow down the gradient and convergence runtimes by different slowdown factors using delay loops. Any number of threads can be used. We measure performance and determine the maximum learning runtime for which the Smart Data Structure is still able to improve performance by at least $\frac{1}{3}$ of its potential. This maximum learning runtime is the tolerable runtime. If we record the number of parameters that were learned (for whatever number of threads were used), we can determine the tolerable slowdown α_t.

Once we have T_t, the constraint on the maximum number of parameters an application can scale to is approximately given by $T_s + 2{\cdot}p^3 \leq T_t$. Substituting the approximation for T_t, this can be rewritten as:

$$T_s + 2{\cdot}p^3 \leq T_s + \alpha_t{\cdot}2{\cdot}p_t^3$$

The T_s terms cancel. Then, the constant factor of 2 cancels. We solve for p and get a final inequality bounding the number of parameters that can be learned:

$$p \leq p_t{\cdot}\sqrt[3]{\alpha_t}$$

The constraint on p is a function of the tolerable slowdown α_t (the slowdown factor at which the tolerable runtime occurred) and p_t the number of parameters we noted that were learned in the tolerable runtime benchmark.

In the next section, we will measure the tolerable runtime of the Smart Queue, Skip List, and Pairing Heap so that we can get a numerical answer for the bound on the number of parameters that these Smart Data Structures can scale to.

Runtime Margins

In this section, we perform benchmarks to determine the tolerable slowdown, α_t, for the Smart Queue, Skip List, and Pairing Heap data structures. Recall that this will allow us to compute a numerical bound on the number of parameters that these Smart Data Structures can learn while still meeting their target of improving performance

by at least $\frac{1}{3}$ of their potential improvement. The potential improvement is taken to be the difference between two performance bounds: the ideal performance and the average performance over all possible knob settings. We will show that α_t is greater than 256x for these Smart Data Structures.

In general, the tolerable slowdown, α_t will be different for different Smart Data Structures. To determine α_t for a given Smart Data Structure, we expand the two experiments that we used in Chapter 5.4 and Chapter 5.5.

Recall that the first experiment evaluates the performance of Smart Data Structures against static ideal and average throughput bounds for different loads on the data structure (configured via the post computation delay). Now, we additionally slow down the gradient and convergence stages of the learning algorithm by different factors and study the effects. The slowdown models increased learning runtime due to scaling to larger numbers of parameters. Recall that the second experiment evaluates the adaptivity of Smart Data Structures when the tradeoffs determining ideal knob settings are time-varying. We vary the load on the data structure over different variation frequencies and compare throughput against dynamic ideal and dynamic average bounds. Just as in the first experiment, we now additionally inject slow down and study the effects.

This setup allows us to sweep over the space of different loads, variation frequencies, and slowdowns. Additionally, it allows us to study both applications with stationary ideal knob settings and time-varying ideal knob settings. Our goal will be to use this experimental setup to conservatively estimate values for the tolerable runtime for arbitrary applications.

In our two benchmarks, α_t is all that we need to measure because we already know the number of parameters being learned in the benchmark (p_t) and we have an equation in terms of α_t and p_t bounding the maximum number of parameters that the Smart Queue, Skip List, and Pairing Heap can scale to using concurrency scaling:

$$p \leq p_t \cdot \sqrt[3]{\alpha_t}$$

Recall that $p_t = 13$ for the Smart Queue, Skip List,and Pairing Heap because the number of parameters that must be learned is always fixed. These Smart Data Structures use a single scancount knob and that scancount knob is a discrete-variable with 13 possible discrete values. The bound thus reduces to:

$$p \leq 13 \cdot \sqrt[3]{\alpha_t}$$

We now present the results of our two benchmarks and determine the tolerable slowdown, α_t. Figure 6-11, 6-12, and 6-13 show representative samples of our results for the first benchmark.

Each graph shows throughput versus the learning slowdown factor for a given load (configured via the post computation delay). First, the results indicate that slowing down the learning engine never causes throughput to dip below the static average bound and harm throughput. Second, we observe that the throughput generally decreases slightly as we increase the learning slowdown but that slowdown does not significantly impact performance in this benchmark. In other words, in the first benchmark, the learning runtime can be slowed by more than 512x and Smart Data Structures will still achieve the requested performance improvement.

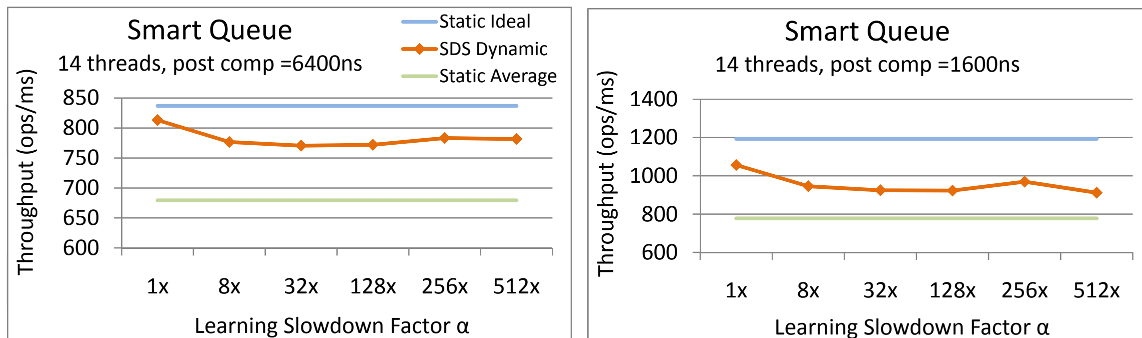


Figure 6-11

The reason that the learning slowdown does not significantly impact performance in this benchmark is that the ideal knob settings are approximately fixed, or stationary, for the duration of the benchmark. The effect of the learning slowdown is that learning engine initially converges more slowly on the ideal knob settings but still

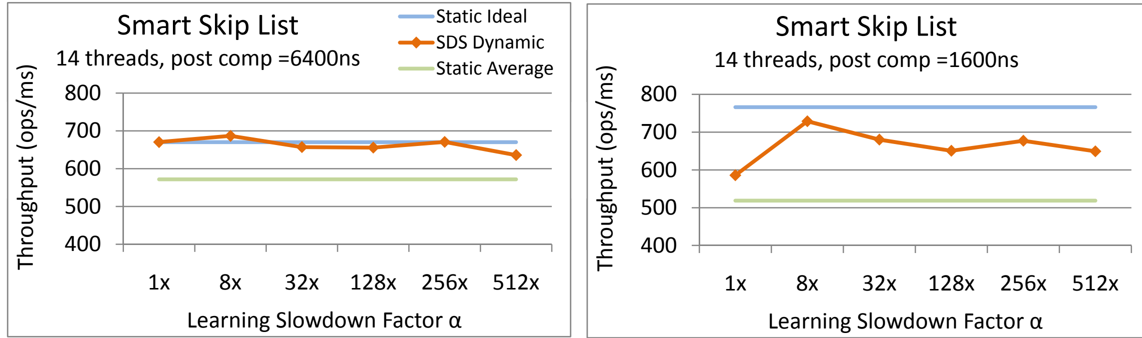


Figure 6-12

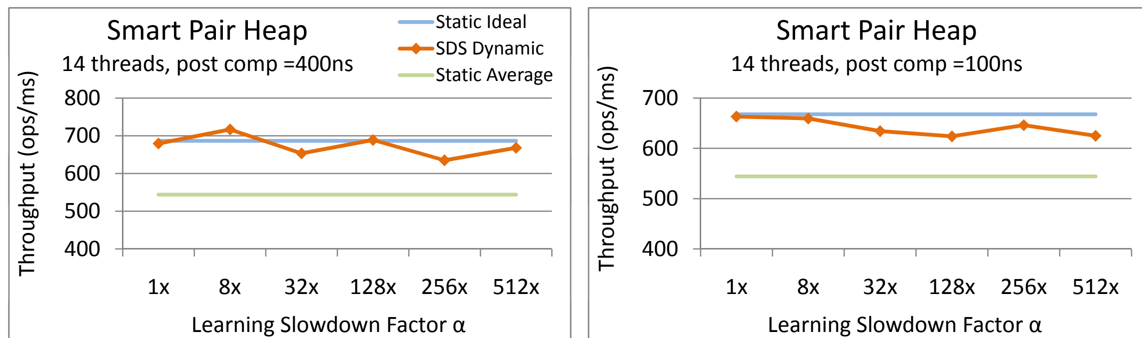


Figure 6-13

uses the ideal settings for the majority of the benchmark. The benchmark runs for 10 seconds while the learning engine updates knob settings several orders of magnitude faster than that even when significantly slowed.

What this tells us is that the learning algorithm runs much faster than minimally necessary in long-running applications with stationary ideal knob settings. Other applications may have time-varying ideal knob settings. Our second benchmark studies the effect of the learning slowdown in applications with rapidly time-varying ideal knob settings to determine the tolerable runtime.

Figure 6-14 shows the results. There are three clustered bar graphs: one for the Smart Queue, one for the Smart Skip List, and one for the Smart Pairing Heap. In each graph, we show throughput versus the variation frequency. Within each cluster, we vary the learning slowdown. The lines in each graph represent the dynamic ideal throughput and dynamic average throughput bounds.

First, the results show that slowing down the learning engine never causes through-

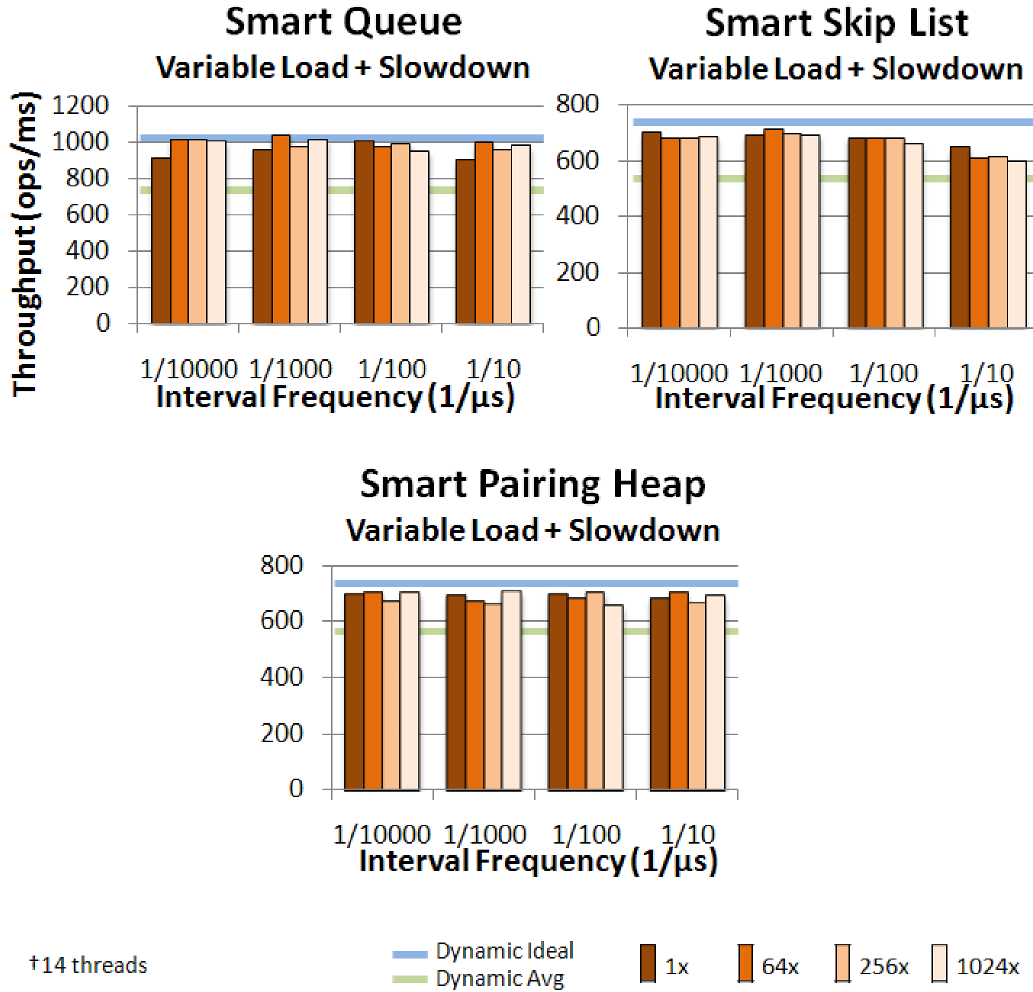


Figure 6-14

put to dip below the dynamic average bound. Second, we see that the throughput improvement is always above the $\frac{1}{3}$ target for slowdown up to 256x even at the highest variation frequency of $\frac{1}{10\mu s}$. For the Smart Queue and Smart Pairing Heap, throughput improvement is actually above the required $\frac{1}{3}$ at 1024x slowdown as well. For simplicity and to be conservative, we say that the tolerable slowdown is $\alpha_t = 256$ for all three of these Smart Data Structures in the second benchmark. Over both experiments, the tolerable slowdown is thus $\alpha_t = 256$.

The most interesting feature of the results for the second benchmark is the trend in how the throughput degrades as we slow down the learning engine. We do not see much degradation at the lower interval frequencies. At the higher interval frequencies,

however, the learning engine begins to have difficulty adapting the scancount at the same rate that the ideal scancount is changing. In this situation, the learning engine makes a best effort by adapting the scancount as frequently as possible and finding the best setting for use over several successive intervals rather than finding the ideal setting for each interval. The degradation in performance is because the best-effort scancount is sub-optimal for the intervals it must cover.

The amount of throughput degradation depends upon a) how sub-optimal the best-effort scancount is in each of the intervals it must cover and b) how many intervals must be covered by a single best-effort scancount. In Chapter 5.3, we showed that it is not uncommon for a near-miss of the ideal scancount to result in throughput 20% below the ideal. We also showed that, without slowdown, the learning engine updates scancounts in tens of microseconds. With slowdown, this time increases and more and more intervals must be covered by a single best-effort scancount. With slowdown, the best-effort scancount becomes a compromise among more intervals, and since the ideal scancount varies significantly from one load to the next, the best-effort scancount tends to become more and more sub-optimal.

To summarize, we have conservatively estimated that the tolerable slowdown for the Smart Queue, Skip List, and Pairing Heap is 256x. Because we took care to study the tolerable slowdown over a range of loads and variation frequencies for both stationary and time-varying ideal knob settings, we can extrapolate this result to arbitrary applications. We estimate that $\alpha_t = 256$ (at least) for any applications using the Smart Queue, Skip List, and Pairing Heap.

It is likely that these tolerable slowdowns can be increased by optimizing the efficiency of our learning algorithm implementation. We have not made attempts to do so here. Further, we have made no effort to tune knobs in the learning algorithm itself to improve performance. Tuning the performance of the learning algorithm would probably improve the tolerable slowdown though and is an interesting area for future research. It is also important to emphasize that the tolerable slowdown is 256x when a single learning thread is used for learning engines. It is possible to parallelize the learning engine to use additional threads; this will speed up the learning runtime

and enable additional tolerable slowdown if desired.

6.2.7 Summary of Smart Data Structures Concurrency Constraints

To summarize the results of Chapter 6.2.5 and Chapter 6.2.6, the optimization components of Smart Data Structures exhibit two potential bottlenecks as applications scale to more threads and more threads access Smart Data Structures: overheads introduced into Smart Data Structures by naïvely-designed external monitors and increased learning runtimes in the learning thread.

In Chapter 6.2.5, we demonstrated a scalable reward monitor design that removes the reward bottleneck and scales to 360 MHz reward update rates on a 48-core system. We showed that its scalability does not introduce overheads at smaller scales, so it can be used in all scenarios. We also demonstrated that the less scalable Heartbeats reward monitor sustains sufficient reward update rates for our system and benchmarks.

In Chapter 6.2.6, we showed that the runtime of learning does not necessarily depend on the number of threads accessing a Smart Data Structure. The learning runtime actually depends on the number of parameters that must be learned. Some Smart Data Structures like the Smart Lock require learning one parameter for each thread, while others like the Smart Queue, Skip List, and Pairing Heap have fixed requirements. We also showed that, for a single Smart Data Structure, the learning engine runs much faster than necessary to maintain good performance improvements.

From Chapter 6.2.6 and 6.2.6, we have all of the information necessary to calculate the constraint on the maximum number of parameters that Smart Data Structures can learn due to optimization overheads before they are unable to improve performance by at least $\frac{1}{3}$ of their potential. Recall that the potential improvement is defined as the difference between two performance bounds: the performance achieved using the ideal scancount and the average performance over all possible scancount settings.

The bound on the number of parameters is given by:

$$p \leq p_t \cdot \sqrt[3]{\alpha_t}$$

Filling in known values for the Smart Queue and Skip List, we have:

$$p \leq 13 \cdot \sqrt[3]{256}$$

$$p \leq 82$$

We can use this result to approximate a constraint on the concurrency scaling of applications due to learning runtimes. As summarized in Table 6.4, the Smart Queue, Skip List, and Pairing Heap use a fixed number of parameters, so they do not constrain application concurrency scaling. We can also use our results to speculate on the constraints for Smart Locks as well as future Smart Data Structures we have not developed yet. Since the Smart Lock has one parameter for each thread that accesses it, we may expect an application with one Smart Lock to scale to 82 threads before the Smart Lock is no longer able to achieve the desired level of performance improvement. We would need to determine the tolerable slowdown (α_t) for Smart Locks as in Chapter 6.2.6 for a more accurate estimate, but it is interesting and informative to speculate based on the results we do have.

For a new Smart Data Structure, if we know the dependence of the number of parameters on the number of application threads, we can apply these formulas to speculate on concurrency constraints as well. Just as we did with the Smart Lock, we would be assuming that the same tolerable slowdown held.

If further concurrency scaling is desired, it is possible to increase the tolerable slowdown by making the learning engine run faster. For every constant factor that the learning engine becomes faster, the number of parameters that can be sustained increases by the same factor. We have made no attempts so far to optimize the learning engine code. It is also possible to parallelize the learning engine to make it run faster.

Type	#parameters	max n
Smart Queue	13	unbounded
Smart Skip List	13	unbounded
Smart Pairing Heap	13	unbounded
Smart Lock	up to 82	82

Table 6.4: Concurrency Scaling Constraints from Smart Data Structures Optimization Components. n is the number of threads accessing the data structure.

6.3 Multi-Data-Structure Demands

In Chapter 6.2, we looked at constraints on concurrency scaling due to concurrency bottlenecks. This section estimates constraints on multi-data-structure scaling. Recall that multi-data-structure scaling refers to instantiating more and more Smart Data Structures in an application, where these data structures are logically independent and need not be jointly optimized.

We begin in Chapter 6.3.1 by analyzing multi-data-structure scaling and evaluating the incremental communication, reward, and learning bottlenecks it implies. We will show that only the learning overhead increases due to multi-data-structure scaling. The learning overhead increases because our design chooses to use a single learning thread to run learning engines for all Smart Data Structures. Because a given learning engine only gets a portion of the cycles in the learning thread, it effectively runs slower.

We will extend our analytical model so that it can estimate constraints due to this increased learning overhead in multi-data-structure scaling. We will show that the constraints are not placed directly on the number of application threads, but rather on how many learning engines the application can use. In Chapter 6.3.2, we use the model to calculate limits on scalability and show that the limits are not likely to be reached in practice. Thus, learning overheads do not constrain multi-data-structure scaling in practice.

Later, in Chapter 6.5.2, we will ultimately use the analytical model and these constraints to estimate the scalability of a variety of case-study applications.

6.3.1 Multi-Data-Structure Incremental Bottlenecks

Applications use multi-data-structure scaling when they scale up by adding new thread pools with each pool utilizing a different Smart Data Structure. Figure 6-15 illustrates. The application instantiates l Smart Data Structures. We assume that each Smart Data Structure is accessed by a fixed number of threads, n. Thus, the application has $l{\cdot}n$ total threads, divided into l thread pools, with each pool accessing a different Smart Data Structure. Each Smart Data Structure has its own reward monitor and learning engine. All l learning engines run in a single learning thread, time-multiplexing its resources.

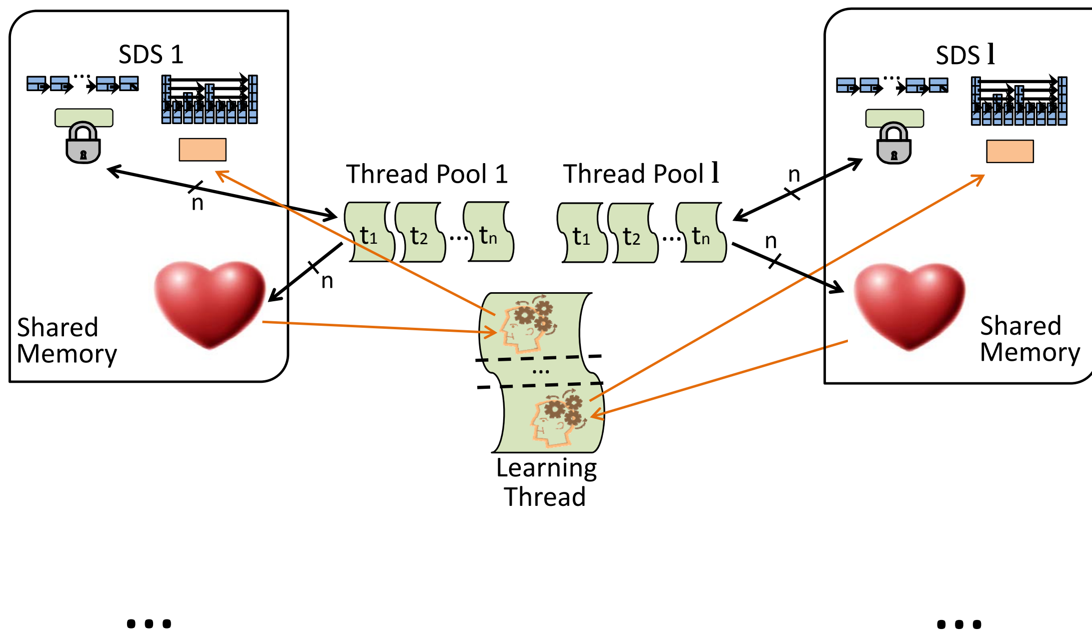


Figure 6-15: Multi-Data-Structure Scaling. Applications scale up by adding new thread pools with each pool utilizing its own Smart Data Structure. Each pool (and corresponding Smart Data Structure) has its own reward monitor and learning engine. Learning engines all run in a single learning thread and time-multiplex its resources.

Because the number of threads accessing a given Smart Data Structure does not increase as the number of total threads increases in multi-data-structure scaling, multi-data-structure scaling places no new incremental demands on communication within the base data structure components in a Smart Data Structure. The communication between a reward monitor, learning engine, and base data structure components uses

no more ports than concurrency scaling, so neither does multi-data-structure scaling place incremental demands on communication between optimization components and the base data structure components.

The storage requirements do increase proportionally with increases in the number of Smart Data Structures instantiated. However, because the number of application threads (and thus the number of instantiated Smart Data Structures) is small relative to cache and DRAM sizes, we neglect the effect of the increased storage requirements.

Since the number of application threads accessing each reward monitor is not increasing, bottlenecks from reward updates will be no worse than they were with pure concurrency scaling in Chapter 6.2.5.

The only new demands that multi-data-structure scaling introduces are increases in the learning overheads. Our design chooses to run learning engines from all Smart Data Structures in a common learning thread. Each learning engine gets a portion of the cycles in the learning thread and, therefore, effectively runs slower.

We would like to determine the constraint that multi-data-structure scaling places on overall application scalability. To determine the maximum number of threads that applications can scale to using multi-data-structure scaling, we expand our analytical model of the learning runtime from Chapter 6.2.6. We will show that we only need to introduce an additional factor in the model to account for the time-multiplexing of the learning thread.

We need to understand what effect time-multiplexing the learning thread has on the learning time, T_l. Recall the algorithm used by each learning engine (see Alg. 1 in Chapter 4.3). The algorithm executes a cycle of three stages. In the first stage of the cycle, the algorithm samples knob settings from the latest learned parameters to try them out and measure the resulting reward. In the second stage, it computes the gradient. In the third stage, it tests for convergence of the gradient and, if converged, adjusts the learned parameters in the direction of the gradient.

The sampling stage runs for a fixed amount of time regardless of how many parameters are learned or how many learning engines are sharing the learning thread. After a fixed amount of time has passed, all of the samples that could be collected

are used, and the learning algorithm moves to the next stage, the gradient stage.

In other words, as illustrated in Figure 6-16, each learning engine is implemented as a state machine with three states: a sampling state, a gradient state, and a convergence test state. The sampling state get visited over and over until the fixed amount of time elapses. Each visit results in sampling a new knob setting and measuring the resulting reward. The other states get visited once before transitioning to the next state in the cycle.

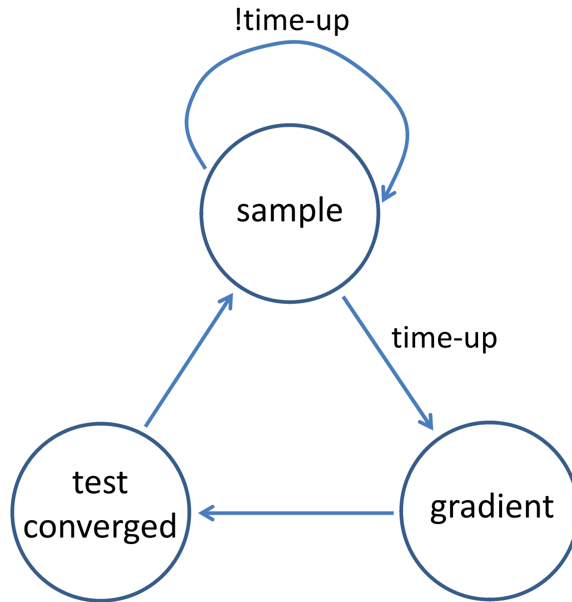


Figure 6-16

Our design implements time-multiplexing of the learning thread by stepping the state machine of each learning engine in round-robin order, one state transition at a time. Since the time allocated to sampling is fixed, only the time spent in the gradient and convergence states increases as more learning engines share the learning thread. The gradient and convergence state runtimes are effectively increased by a factor l, where l is the number of learning engines in the application. Thus the learning time T_l is:

$$T_l \approx T_s + l \cdot 2 \cdot p^3$$

We have already studied the effect that an increase in learning runtimes will have on Smart Data Structure performance. At some point, the learning runtime will

increase enough that a Smart Data Structure will no longer able to achieve at least $\frac{1}{3}$ of its potential performance improvement. In Chapter 6.2.6, we empirically measured this threshold which we call the tolerable runtime. We gave an inequality bounding the learning time to less than or equal to the tolerable runtime:

$$T_l \leq T_t$$

We found it more convenient to express the tolerable runtime as a function of two factors: the tolerable slowdown of the gradient and convergence test runtimes (α_t) and the number of parameters that were learned in the Smart Data Structure (p_t). The tolerable runtime is given by:

$$T_t \approx T_s \ + \ \alpha_t \cdot 2 \cdot {p_t}^3$$

Substituting for T_l and T_t in the inequality, we have:

$$T_s \ + \ l \cdot 2 \cdot p^3 \leq T_s \ + \ \alpha_t \cdot 2 \cdot {p_t}^3$$

Canceling terms and reducing constant factors, we have a final inequality expressing the constraint on the number of parameters and learning engines that Smart Data Structures can scale to using multi-data-structure and concurrency scaling:

$$l \cdot p^3 \leq \alpha_t \cdot {p_t}^3$$

For Smart Data Structures like the Smart Queue, Skip List, and Pairing Heap, we measured the tolerable slowdown, α_t. We found that $\alpha_t = 256$ and p_t is fixed in these data structures at $p_t = 13$ regardless of how many threads access them. Thus, for these data structures, the inequality reduces to:

$$l \cdot p^3 \leq 256 \cdot 13^3$$

6.3.2 Multi-Data-Structure Scaling Constraints

We would like to determine the maximum number of threads we can scale an application to via multi-data-structure scaling before Smart Data Structures can no longer maintain at least $\frac{1}{3}$ of their potential performance improvements. We have shown that the limits of multi-data-structure scaling do not directly depend on the number of threads. Rather, the limits depend on the number of learning engines needed.

For Smart Data Structures like the Smart Queue, Skip List, and Pairing Heap, we derived a formula that will give us a numerical bound on the number of learning engines and the number of parameters that can be learned:

$$l \cdot p^3 \leq 256 \cdot 13^3$$

For the purposes of multi-data-structure scaling, we are not interested in scaling up the number of parameters p, so we will use a fixed value $p = 13$. 13 is the number of parameters that the Smart Queue, Skip List, and Pairing Heap require learning. The inequality reduces to:

$$l \leq 256$$

Thus, applications can use up to 256 learning engines before the learning runtime impedes the Smart Data Structure's ability to improve performance. The constraint applies only to the number of learning engines. In other words, the number of thread pools in the application is constrained, but the number of threads (n) in each thread pool is not. For different size thread pools, we give constraints on the maximum number of application threads in Table 6.5.

# threads per pool, n	# learning engines, l	max application threads, $l \cdot n$
1	up to 256	256
2	up to 256	512
64	up to 256	16384

Table 6.5: Multi-Data-Structure Scaling Constraints.

In practice, since the maximum number of learning engines is large, we do not see learning overheads as being a bottleneck to multi-data-structure scaling. Further-

more, scalability can be extended by enabling the use of multiple learning threads.

6.4 Multi-Optimization Demands

In Chapter 6.3, we looked at constraints on multi-data-structure scaling due to multi-data-structure bottlenecks. This section estimates constraints on multi-optimization scaling. Recall that multi-optimization scaling is similar to multi-data-structure scaling in that it refers to instantiating more and more Smart Data Structures in an application. The difference is that, in multi-optimization scaling, these data structures need to be jointly optimized for the best performance.

This section is organized similarly to the section on multi-data-structure scaling. We begin in Chapter 6.4.1 by analyzing multi-optimization scaling and evaluating the incremental communication, reward, and learning bottlenecks it implies over multi-data-structure scaling. We will show that only the learning overhead increases. However, it increases more substantially in multi-optimization scaling. This is because a learning engine must now jointly optimize multiple knobs and thus more parameters, and the learning runtime grows cubically in the number of parameters being learned. We will extend our analytical model so that it can estimate constraints due to this increased learning overhead. Then, in Chapter 6.4.2, we use the analytical model to estimate constraints on application scalability due to multi-optimization scaling.

Later, in Chapter 6.5.2, we will ultimately use the analytical model and these constraints to estimate the scalability of a variety of case-study applications.

6.4.1 Multi-Optimization Incremental Bottlenecks

Applications use multi-optimization scaling when they increase the number of Smart Data Structures being jointly optimized by a given learning engine and reward monitor. Figure 6-17 illustrates. For simplicity, the figure shows an application using only multi-optimization scaling. There is a single pool of a fixed number of threads, n. The application scales by increasing the number of Smart Data Structures, s, that are accessed by the pool. All s Smart Data Structures share one reward monitor and

one learning engine. The learning thread is not time-multiplexed.

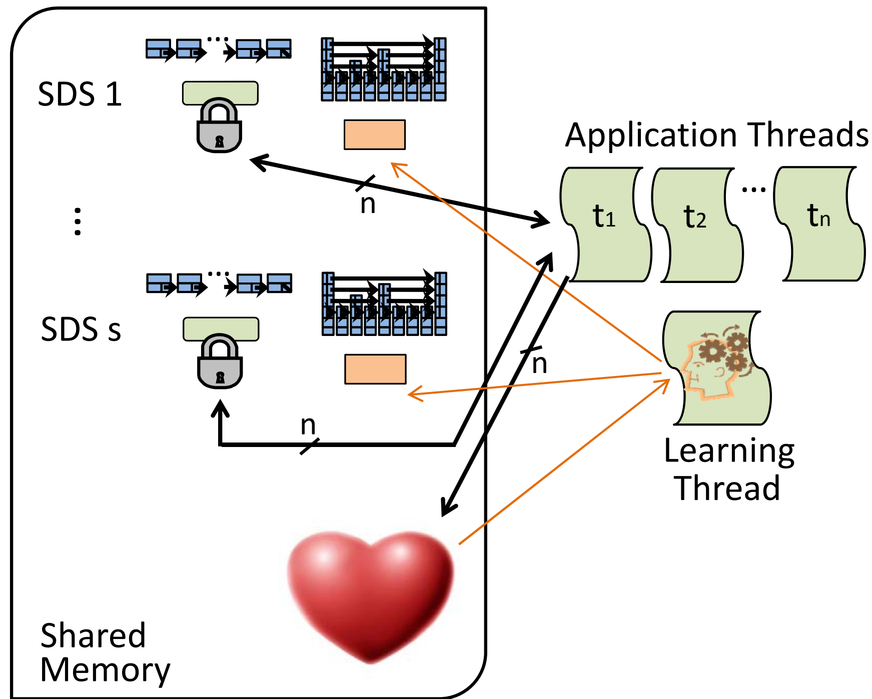


Figure 6-17: Multi-Optimization Scaling. Applications scale up by increasing the number of Smart Data Structures belonging to each pool of threads. We assume there is one pool of threads, with a fixed number of threads n. There are s Smart Data Structures. All s Smart Data Structures share one reward monitor and one learning engine.

As was the case with multi-data-structure scaling, the number of threads accessing a given Smart Data Structure is not increasing. Therefore, multi-optimization scaling places no new incremental demands on communication within the base data structure components in a Smart Data Structure. Neither does multi-optimization place incremental demands on communication between optimization components and the base data structure components because the number of ports between them is no worse than the case with concurrency scaling. The learning engine must now connect to more knobs, but this does not increase the number of sharers of cache lines internal to the knobs and therefore does not represent new demands on communication.

The storage requirements do increase proportionally with increases in the number of Smart Data Structures instantiated. However, the number of instantiated Smart Data Structures is small relative to cache and DRAM sizes, so we can again neglect

the effect of the increased storage requirements.

The only potential incremental demands over concurrency scaling and multi-data-structure scaling are from reward and learning overheads. As more Smart Data Structures are added, one learning engine and one reward monitor are shared among all Smart Data Structures. The demands on the reward monitor may increase because there are more data structures to supply reward updates for. However, we have demonstrated a scalable reward monitor based on lazy counters (see Chapter 6.2.5) that can tolerate sufficiently high update rates that reward overheads are not a bottleneck.

The only incremental demands that multi-optimization scaling introduces over concurrency scaling are learning overheads, and the learning overheads introduced by multi-optimization scaling are more significant than the learning overheads introduced by multi-data-structure scaling. In the case of multi-optimization scaling, the overhead derives from the fact that all Smart Data Structures share a common learning engine and the learning engine must jointly optimize knobs for all Smart Data Structures. This implies that it must learn more parameters.

We would like to determine the constraint on the number of Smart Data Structures that can be jointly optimized before learning runtimes cause Smart Data Structures to miss their performance improvement targets. Fortunately, we have already developed an inequality that provides a bound on the number of parameters that can be learned (p) without degrading performance improvements. We will extend this inequality to accommodate multi-optimization scaling and jointly optimizing parameters for s Smart Data Structures at once in a learning engine.

To review, when concurrency and multi-data-structure scaling are being used, the bounds on the number of learning engines l and number of parameters that can be learned p are given by:

$$l \cdot p^3 \leq \alpha_t \cdot {p_t}^3$$

To extend the inequality so that it accommodates multi-optimization scaling in addition to concurrency and multi-data-structure scaling, we need only adjust the

left-hand side. We will think of p not as the number of parameters corresponding to the knobs in a single Smart Data Structure but as the average number of parameters required for each Smart Data Structure. We will think of s not as the total number of Smart Data Structures but as the number of Smart Data Structures jointly optimized by each learning engine. Thus the inequality is given by:

$$l \cdot (s \cdot p)^3 \leq \alpha_t \cdot {p_t}^3$$

For Smart Data Structures like the Smart Queue, Skip List, and Pairing Heap, we found that $\alpha_t = 256$ and p_t is fixed in these data structures at $p_t = 13$ regardless of how many threads access them. Thus, for these data structures, the inequality reduces to:

$$l \cdot (s \cdot p)^3 \leq 256 \cdot 13^3$$

6.4.2 Multi-Optimization Scaling Constraints

We would like to determine the maximum number of threads we can scale an application to via multi-optimization scaling before Smart Data Structures can no longer maintain at least $\frac{1}{3}$ of their potential performance improvements. We have shown that the limits of multi-optimization scaling do not directly depend on the number of threads. Rather, the limits depend on how many total parameters a learning engine must learn to jointly optimize all of the Smart Data Structures connected to it.

For Smart Data Structures like the Smart Queue, Skip List, and Pairing Heap, we derived a formula that will give us a numerical bound on the number of learning engines (l), number of Smart Data Structures per learning engine (s), and number of parameters (p), that an application can scale to:

$$l \cdot (s \cdot p)^3 \leq 256 \cdot 13^3$$

For the purposes of pure multi-optimization scaling, we are not interested in scaling up the average number of parameters p, so we will use a fixed value $p = 13$. 13 is

the number of parameters that the Smart Queue, Skip List, and Pairing Heap require learning. We are also not interested in scaling the number of learning engines, so we will use $l = 1$. The inequality reduces to:

$$s \leq \sqrt[3]{256}$$

$$s \leq 6.35$$

Thus, applications can use up to 6 Smart Data Structures via multi-optimization scaling before the learning runtime impedes the Smart Data Structure's ability to improve performance by the requested amount. If an application needs to jointly optimize more than 6 Smart Data Structures, it is possible to parallelize the learning engine so that it can run faster. We leave this to future work.

In practice, we have found that multi-optimization is not often needed because Smart Data Structures can typically be optimized independently and approximate the performance of joint optimization. In other words, less expensive multi-data-structure scaling can be used in place of multi-optimization scaling in many cases. The main applications that we think will benefit from multi-optimization are applications that use pipeline parallelism. When Smart Data Structures connect stages of a pipeline, jointly optimizing them should help maintain the balance of each pipeline stage and maximize overall pipeline throughput.

Fortunately, this type of application seldom utilizes pipelines deeper than 5 or 6 stages, so the maximum number of Smart Data Structures to jointly optimize would be 5. These applications typically scale by using multiple pipelines. Multiple pipelines can be independently optimized via multi-data-structure scaling. We have shown in Chapter 6.3.2 that the limits of multi-data-structure scaling occur at such large scales that they are not likely to limit application scaling in practice.

6.5 Case Studies

In the previous sections, we determined constraints on application scalability when concurrency, multi-data-structure, or multi-optimization scaling were used in isolation. In this section, we combine these constraints to estimate the scalability of applications that use combinations of these types of scaling. Further, we have said nothing about the degree to which real applications rely on each of these types of scaling. In this section, we will study typical applications to identify specifically how much concurrency, multi-data-structure, and multi-optimization scaling they require as they scale to larger numbers of threads. We will use the combined scaling limits to estimate the maximum number of threads each of these applications can scale to before Smart Data Structures are no longer able to sustain at least $\frac{1}{3}$ of their performance benefits.

Chapter 6.5.1 begins by summarizing the bottlenecks that eventually constrain application scalability and the analytical model we developed to estimate their combined constraints on application scalability. Then, Chapter 6.5.2 describes our application case studies and identifies their specific utilization of concurrency, multi-data-structure, and multi-optimization scaling. Finally, Chapter 6.5.3 uses the analytical model to estimate overall limits on the maximum number of threads our case-study applications can scale to.

We will show that, for all but one of the case study applications, Smart Data Structures scale up to 64 threads or beyond before another data structure would be higher performance or before Smart Data Structures would be unable to sustain at least $\frac{1}{3}$ of their potential performance improvements. For the other application, we will show that the scaling limit due to Smart Data Structures is not a scaling limit in practice because the application itself does not scale this far. We will also show that the ultimate constraints on application scalability usually derive from the base data structures upon which Smart Data Structures layer optimization rather than from scaling constraints due to optimization.

6.5.1 Combined Constraints

In Chapter 6.2, Chapter 6.3, and Chapter 6.4, we have identified a variety of potential bottlenecks in our design. We have shown how our design eliminates many of them. For a few that remain, we have quantified the point at which they begin to limit the ability of Smart Data Structures to maintain at least $\frac{1}{3}$ of their potential performance improvements. This section summarizes these bottlenecks and the analytical model we developed to estimate their constraints on application scalability.

To summarize our results, we have investigated many potential bottlenecks and have found two sources that can eventually limit application scalability:

1. the concurrency scaling of the base data structures upon which Smart Data Structures are built
2. the scaling of the Smart Data Structures learning components due to:
   - more learning engines multiplexing the learning thread via multi-data-structure scaling
   - increased runtimes of individual learning engines via multi-optimization scaling and jointly optimizing multiple knobs

For the Smart Queue, Skip List, and Pairing Heap, we have determined the concurrency limits that derive from their base data structures (the Flat Combining data structures). The Flat Combining Queue, Skip List, and Pairing Heap outperform prior art up to 64 threads concurrently accessing the data structure. Shortly after 64 threads, the Flat Combining queue is overtaken by the scalable Combining Tree Queue. The Flat Combining Skip List and Pairing Heap remain superior to prior art as concurrency is scaled beyond 64 threads. Thus, we conclude (conservatively) that the concurrency limit for Smart Data Structures due to base data structure limits is 64 threads accessing a given Smart Data Structure.

For the Smart Queue, Skip List, and Pairing Heap, we have also determined the limits due to learning overheads. We found that the learning engine runs much faster than necessary in basic Smart Data Structure configurations. We showed that this

headroom enables multi-data-structure scaling beyond 256 Smart Data Structures and multi-optimization scaling up to 82 jointly-optimized parameters in a single learning engine.

We would now like to map these constraints into scalability limits on real applications. Real applications may use a combination of concurrency, multi-data-structure, and multi-optimization scaling, and we have derived an inequality in Chapter 6.3 to bound the maximum number of learning engines (l), maximum number of Smart Data Structures jointly optimized by each learning engine (s), and the maximum number of parameters on average in each Smart Data Structure (p):

$$l \cdot (s \cdot p)^3 \leq \alpha_t \cdot {p_t}^3$$

Here, α_t and p_t relate to the tolerable runtime, and we have empirically measured them for the Smart Queue, Skip List, and Pairing Heap. For these data structures, the inequality reduces to:

$$l \cdot (s \cdot p)^3 \leq 256 \cdot 13^3$$

In the next two sections, we look at typical application structures and determine what combinations of concurrency, multi-data-structure, and multi-optimization scaling they use.

6.5.2 Case Study Applications

We consider five application structures / data structure use-cases: a global work queue, work-stealing, discrete event simulation, network traffic prioritization, and software pipelining. We begin with descriptions of these applications then provide Table 6.6 to relate the number of application threads they use to their concurrency, multi-data-structure, and multi-optimization demands.

Global Work Queue A global work queue program uses a parallelism model that divides computation into units of work. All threads share a common Smart Queue which they populate with work and extract work from. In general, performing a unit

of work leads to the generation of new units of work. Scaling this application requires pure concurrency scaling.

Work Stealing Like global work queues, work stealing is a parallelism model that divides computation into units of work. The difference is that each thread owns a Smart Queue which it populates with work and extracts work from. When a thread's queue is empty, it has run out of work and may access other Smart Queues to steal work from other threads. Scaling this application requires a combination of concurrency and multi-data-structure scaling. However, we assume concurrency scaling is limited to 64 threads accessing a given Smart Data Structure. In other words, we limit the stealing so that threads may steal work from no more than 63 other threads. For each thread, those 63 are chosen to guarantee that no data structure is accessed by more than 64 total threads.

Parallel Discrete Event Simulation Parallel discrete event simulation is a simulation method which keeps a simulation clock and processes simulation events when their time matches the simulation clock. Events may be generated out of order, so applications use priority queues (in this case a Smart Pairing Heap), to process them in chronological order. When a single global priority queue is used, scaling this application requires pure concurrency scaling. Hierarchical discrete event simulation is also possible. Hierarchical discrete event simulation is an implementation technique to improve performance using a hierarchy of priority queues instead of a centralized priority queue. Scaling the hierarchical version requires a combination of concurrency and multi-data-structure scaling.

Network Traffic Prioritization Network traffic prioritization is an Internet routing strategy for prioritizing different types of traffic. For example, it can be used to improve the latency and quality of VOIP conversations by routing VOIP streams with high priority. It can be implemented with a global Smart Skip List. Scaling this application requires pure concurrency scaling.

Software Pipelining Software pipelining is a method for parallelizing stream computations into multiple stages such that the output of one stage is the input to the next and all stages run in parallel for increased processing throughput. Scaling this application requires a combination of concurrency and multi-optimization scaling. Super-scalar software pipelining is also possible. Super-scalar software pipelining replicates a simple pipeline and runs all pipelines in parallel. Scaling this type of application requires a combination of concurrency, multi-data-structure, and multi-optimization scaling.

Table 6.6 relates the number of application threads used in these applications, n, to their concurrency, multi-data-structure, and multi-optimization demands. As the table indicates, application structures that globally access data structures have concurrency requirements that grow linearly in the number of threads. Application structures that linearly increase the number of data structures as more threads are added either linearly increase the number of learning engines needed or linearly increase the number of Smart Data Structures that must be jointly optimized by a given learning engine.

Application Scenario	# SDS	concurrency	l engines	$s{\cdot}p$ params per engine
Global Work Queue	1	n	1	13
Work Stealing (64-way Stealing)	$n \mid n{\leq}4096$	64	n	13
Discrete Event Sim. Simple	1	n	1	13
Discrete Event Sim. Hierarchical	$n+1$	n	$n+1$	13
Net Traffic Prioritization	1	n	1	13
Software Pipeline Simple	$n-1$	2	1	$13{\cdot}(n-1)$
Software Pipeline Superscalar (5-stage pipes)	$4{\cdot}n/5$	2	$n/5$	$13{\cdot}4$

Table 6.6: Application Scaling Demands. n is the number of application threads

6.5.3 Overall Scaling Results

To determine scalability limits for a given application, we divide the limits into two parts: limits from Flat Combining and limits due to learning overheads. From Chapter 6.5.1, we know that the maximum concurrency that Flat Combining data structures can sustain is:

$$concurrency = 64 \; threads \; accessing \; any \; data \; structure$$

We also know an inequality that gives the constraints of learning overheads on the maximum values of l, s, and p that Smart Data Structures can sustain:

$$l{\cdot}(s{\cdot}p)^3 \leq \alpha_t{\cdot}{p_t}^3$$

For the Smart Queue, Skip List, and Pairing Heap, we have empirically measured α_t and p_t, and the inequality reduces to:

$$l{\cdot}(s{\cdot}p)^3 \leq 256{\cdot}13^3$$

In Table 6.7, we use these limits to give a numerical value for the constraint on application scalability deriving from each source. To calculate the constraint on the maximum number of threads due to learning, we need to express the inequality in terms of the number of application threads, n. We do this by substituting for l and $s{\cdot}p$ using the values in Table 6.6.

Application Scenario	Max n due to FC	Max n due to SDS	Max n
Global Work Queue	**64**	unbounded	64
Work Stealing (64-way Stealing)	4096	**256**	256
Discrete Event Sim. Simple	**64**	unbounded	64
Discrete Event Sim. Hierarchical	**64**	255	64
Net Traffic Prioritization	**64**	unbounded	64
Software Pipeline Simple	unbounded	**7 (7-stage)**	7
Software Pipeline Superscalar (5-stage pipes)	unbounded	**20 (4x 5-stage)**	20

Table 6.7: Application Scaling Limits and Limit Sources. n is the total number of application threads.

As the table illustrates, the base data structures are the scalability limiter for most of these applications. Furthermore, for the work-stealing application, while the learning overheads are technically the scalability limiter, the limit is 256 threads which will not be a limitation in practice – especially since more learning threads can be added to increase scalability. Only the software pipelines are limited by the scaling of the learning overheads in Smart Data Structures. We will show, however, that these theoretical limits will also not limit scalability in practice.

The scalability limits on software pipelines derive from the use of a single learning engine to jointly optimize multiple Smart Data Structures. The problem is that

runtime of the learning algorithm scales cubically in the number of parameters that must be learned, and each Smart Data Structure is contributing new parameters to learn. Luckily, typical software pipelined applications (such as those in the Parsec benchmark suite [2]), do not use more than 5- or 6-stage pipelines. This only requires jointly optimizing up to 5 Smart Data Structures with a single learning engine – well within the limit. Software pipelined applications typically avoid deep pipelines because deep pipelines introduce programming complexity and overwhelm on-chip communication bandwidth. Replicating pipelines and running multiple pipelines in parallel is more common in our experience.

As Table 6.7 shows, Smart Data Structures can sustain 4 parallel 5-stage software pipelines. More parallel pipelines can be supported if we enable the use of multiple learning threads in our design. Further, deeper pipelines than 7-stage pipelines can be supported if we parallelize the learning algorithm to make it run faster.

Since it is interesting to consider more realistic pipelines, we will study a 5-stage simple pipeline. We would like to estimate how much of the potential performance improvement the Smart Queue is able to achieve for this application. To determine this, we must first calculate how much the learning has slowed down. Let us denote this slowdown by α. Then, we can estimate the percent of potential performance achieved by looking at the results in Figure 6-14 in Chapter 6.2.6.

Recall that the graph for the Smart Queue shows Smart Queue performance as a function of the learning slowdown for different variation frequencies. To determine what performance improvement we should expect, we need to know how rapidly the idea knob settings change in the 5-stage pipeline application. For simplicity, we will assume the worst case: that ideal settings vary rapidly with a frequency of $\frac{1}{10\mu s}$. Once we have calculated the learning slowdown (α), we can focus on the bar in the $\frac{1}{10\mu s}$ cluster corresponding to α.

Thus, a simple procedure for estimating performance improvement is to assume ideal knob settings change every $10\mu s$ and look at the bar corresponding to the calculated slowdown α.

To get α, we have an equation relating α to l, s, and p, which we will solve for α:

$$\alpha \;=\; l\cdot\left(\frac{s\cdot p}{13}\right)^3$$

We know l and $s\cdot p$ for this application: $l = 1$ since one learning engine is used and $s\cdot p = 4\cdot 13$ since there are 4 Smart Queues each requiring learning 13 parameters. The equation reduces to:

$$\alpha \;=\; 1\cdot\left(\frac{4\cdot 13}{13}\right)^3 \;=\; 64$$

Figure 6-14 shows that, at $\alpha = 64$, the Smart Queue achieves about 90% of its potential performance improvement. Recall that the potential improvement is taken to be the difference between the performance achieved using the ideal dynamic scancount and the average dynamic performance achieved over all scancount values. Table 6.8 summarizes this result.

Application Scenario	Threads	SDS Type	Learning Slowdown	Expected Perf.
Software Pipeline Simple (5-stage)	5	Smart Queue	64x	90%

Table 6.8: Performance at Realistic Software Pipelines Scaling Levels

We will also use this methodology to estimate how much of the potential performance improvement Smart Data Structures achieve in the case-study applications. We calculate the learning slowdown and look up the corresponding performance improvement in the graphs in Figure 6-14 for the Smart Queue, Skip List, and Pairing Heap. Table 6.9 gives the results. For all applications, at the maximum scalability level, the Smart Data Structure can be expected to achieve at least $\frac{1}{3}$ of its potential performance improvement. This is what we requested. Actually, the achieved percent performance improvement is about $\frac{2}{3}$ or more for these applications.

Application Scenario	Limiter	Max Threads	SDS Type	Learning Slowdown	Performance
Global Work Queue	FC	64	Smart Queue	1	65%
Work Stealing (64-way Stealing)	Learning	256	Smart Queue	256	80%
Discrete Event Sim. Simple	FC	64	Smart Pair Heap	1	70%
Discrete Event Sim. Hierarchical	FC	64	Smart Pair Heap	65	80%
Net Traffic Prioritization	FC	64	Smart Skip List	1	60%
Software Pipeline Simple (7-stage)	Learning	7	Smart Queue	216	80%
Software Pipeline Superscalar (4x 5-stage)	Learning	20	Smart Queue	256	80%

Table 6.9: Performance at Maximum Scaling Levels

Overall, we find that the most significant limiter to scalability in Smart Data

Structures is the base data structures upon which they are built. We look forward to the development of more scalable data structures – possibly parallel Flat Combining data structures – which will allow us to scale Smart Data Structures further in the future.

Chapter 7

Smart Locks Performance Results

This chapter evaluates the performance of Smart Locks. In it, we perform experiments on systems with performance heterogeneities between threads and study the effect of the lock acquisition scheduling policies implied by different locks on application performance. We show that Smart Locks are able to learn intelligent lock acquisition scheduling policies automatically and optimize access to shared resources and/or critical sections protected by the lock. Through online Reinforcement Learning, Smart Locks learn application-specific schedules that maximize long-term performance and adapt the schedule dynamically as necessary. Then, based on our findings in these experiments, we provide a set of usage guidelines for Smart Locks.

7.1 Experiment Overview

The first experiment studies dynamic heterogeneities from frequency scaling. In particular, we look at overclocking technologies like Intel's® Turboboost® which overclock cores individually to improve performance if their power and thermal headroom allows. We simulate Turboboost® overclocking events in a synthetic benchmark and show that the Smart Lock's lock acquisition scheduling can significantly improve the performance of a work-pile data structure in such performance heterogeneity scenarios. Further, we show that Smart Locks can readily adapt to dynamic changes in the system that affect the ideal scheduling policy.

The second experiment studies static manufacturing heterogeneities. We look at application performance when applications run on machines whose cores operate at different clock speeds due to circuit imperfections. We study the performance of popular SPLASH-2 benchmarks and show a) that the scheduling policy of a lock in these applications significantly effects performance and b) that Smart Locks are able to learn near-optimal scheduling policies.

7.2 Dynamic Overclocking Experiment

This experiment applies Smart Locks to dynamic performance heterogeneities in core performance. It evaluates the performance and adaptivity of Smart Locks versus standard lock strategies in an overclocking scenario analogous to Turboboost® where core clock frequencies vary dynamically and unexpectedly. The section starts with a description of the experimental setup then presents results.

7.2.1 Experimental Setup

The experimental setup emulates a heterogeneous multicore with six cores (which each run one thread) where core frequencies are drawn from the set {3 GHz, 2 GHz}. The benchmark is synthetic, and represents a simple work-pile programming model (without work-stealing). The application uses the pthreads library for thread spawning and Smart Locks within the work-pile data structure. The application is compiled using `gcc` v.4.3.2. The benchmark uses 6 threads on 6 cores: one for the master thread, four for workers, and one reserved for Smart Locks. The master thread generates work while the workers pull work items from the pile and perform the work.Application Heartbeats [17] are used to supply the reward: heartbeats are credited for each work item completed.

We assume, for simplicity, that each work item requires a constant number of cycles to complete. On a heterogeneous multicore, workers will, in general, execute on cores running at different speeds; thus, x cycles on one core may take more wall-clock time to complete than on another core. In this experiment, work items are small so

that workers must frequently retrieve work from the pile. It is a well-known deficiency of work-piles built using locks that the producer (the master) can be starved because it must compete with $n - 1$ consumers (workers) for the lock. We will show that a major virtue of the Smart Lock is that it eliminates this starvation by scheduling the producer with high precedence.

This experiment models a heterogeneous multicore but runs on a homogeneous 8-core (dual quad core) Intel Xeon(r) X5460 CPU with 8 GB of DRAM running Debian Linux kernel version 2.6.26. In hardware, each core runs at its native 3.17 GHz frequency. Linux system tools like *cpufrequtils* could be used to dynamically manipulate hardware core frequencies, but our experiment instead models clock frequency heterogeneity using a software method: when a thread completes a work item, it credits either 2 or 3 heartbeats where it would normally issue 1, proportional to the modeled clock speed of the core it runs on (2 GHz or 3 GHz).

The experiment simulates two overclocking events that change core speeds. For the cores whose speeds change, we reflect the change by adjusting the number of heartbeats they issue for each completed work item. We note that, in this experiment, we choose to simulate events (while suppressing events in hardware) as opposed to recording real events in the hardware to simplify the illustration of the benefit of Smart Locks; it allows us to determine a priori illustrative scheduling policies to compare Smart Locks against and greatly simplifies the derivation of performance bounds.

7.2.2 Results

In this experiment, we will compare benchmark performance across overclocking events for different spin-locks and their lock acquisition scheduling policies. A test-and-set lock, a write-biased lock, two hand-programmed priority locks, and a Smart Lock are compared. The test-and-set represents baseline lock performance. The write-biased lock preferentially schedules writers when determining which contending thread will get the lock next. This improves performance by helping prevent producer starvation. The two hand-programmed priority locks, together, give a bound

on ideal performance. Since we know a priori what speeds different cores will be, we hand-program a priority lock to perform well in each case. The goal of the Smart Lock will be to learn these hand-optimized policies and adapt when the overclocking events occur to change the policy.

Dividing the graph into three regions surrounding the overclocking events, Hand-Opt 1 is optimal for the first and last region. Its policy sets the master thread and worker 0 to a high priority value and all other threads to a low priority value (e.g. high = 2.0, low = 1.0). Hand-Opt 2 is optimal for the middle region of the graph; its policy sets the master thread and worker 3 to a high priority value and all other threads to a low priority value.

Figure 7-1 shows the benchmark results. It exhibits several interesting features. First, it shows that the ideal performance from the priority locks is substantially higher than that achieved by the test-and-set and write-biased standard locks (hereafter referred to as simply standard locks). This demonstrates that lock acquisition scheduling and application-specific biases in the scheduling can provide significant benefits over the fixed policies of standard locks on heterogeneous multicores.

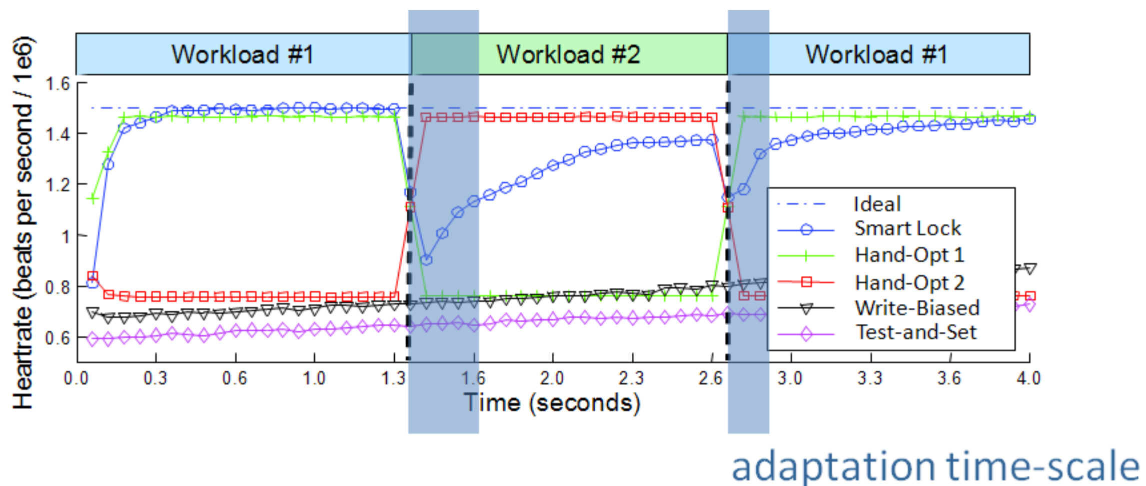


Figure 7-1: Heartrate performance across thermal throttling events (workload changes). Smart Locks significantly outperforms reactive and TAS spin-locks, achieving near optimal.

Second, the results show that the Smart Lock achieves near-ideal performance in the first region before the first overclocking event. This shows that Smart Locks are

effective at learning the need for static biases in scheduling, and that the overhead of learning in Smart Locks relative to the overhead of priority locks is low; Smart Locks are able to optimize priority locks without degrading performance from the ideal.

Third, the results show that Smart Locks readily adapt the lock acquisition scheduling policy after the overclocking events. After each event, Smart Lock performance approaches the performance of the corresponding hand-programmed priority lock. Performance does dip temporarily after the each event but improves quickly. Interestingly, during the performance dips, Smart Locks' policy is suboptimal but still better than the standard locks. At the lowest point, the performance approaches the performance of the write-biased lock because, while the policy is suboptimal for the worker threads, it still correctly identifies the need for prioritizing the master. Within an adaptation time-scale of a few hundred milliseconds after each event, the Smart Lock adapts the policy and achieves near-ideal performance.

Figure 7-2 elucidates both the source and duration of the performance dips. It shows the time-evolution of the Smart Lock's internal learned parameters θ_i. To a certain approximation, these weights can be interpreted as the relative priority between different threads. Ranking threads by their weight gives the priority order, and the distance between two thread weights gives how strongly that relative order is preferred. For their priority to switch, thread weights must cross over.

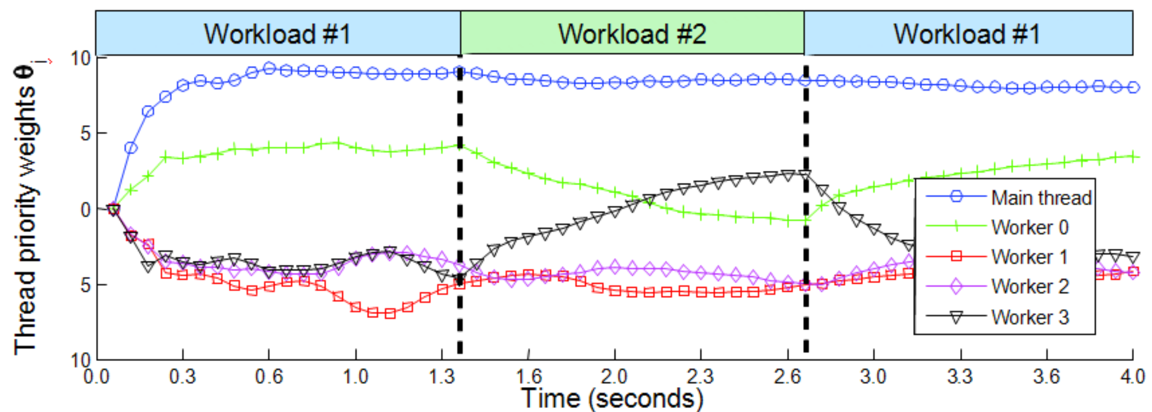


Figure 7-2: Time evolution of the learned policy. Crossovers between Worker 0 and 3 reflect throttling events.

In the figure, threads all have the same weight initially, implying equal probability

of being selected as high-priority threads. Between time 0 and the first event, Smart Locks learns that the master thread and worker 0 should have higher priority, and uses a policy similar to the hand-programmed one. After the first event, the Smart Lock learns that the priority of worker 0 should be decreased and the priority of worker 3 increased, similar to the second hand-programmed one. After the second event, Smart Lock relearns the first workload policy.

The performance dip after the first overclocking event lasts longer than the dip after the second event because it takes the thread weights of workers 0 and 3 longer to cross over after the first event than it does after the second event. This is because, just before the second event, the weights are closer together than they were before the first event, so the weights have less bias to overcome before they cross over to the ideal policy.

7.3 SPLASH-2 Static Heterogeneity Experiment

This experiment applies Smart Locks to intrinsic system heterogeneities in core performance from manufacturing imperfections which affect the maximum speeds of different cores. We envision systems in which faster cores are not limited to the lowest common denominator in core speeds and are allowed to run heterogeneously rather than preserving homogeneity.

This experiment evaluates what impact a spin-lock's scheduling policy has on end-to-end application performance on such a system. It benchmarks the *radiosity* and *raytrace* applications from SPLASH-2, replacing key locks within their concurrent data structures with Smart Locks whose policies are varied as part of the experiment. The results show that the lock acquisition scheduling policy can significantly impact application performance even on moderately heterogeneous multicores like the system we study. We expect larger heterogeneities will result in larger effects. The results additionally demonstrate that Smart Locks can learn good policies quickly and significantly improve overall application performance. The next sections detail the experimental setup and present the results.

7.3.1 Experimental Setup

The experiment uses pthreads implementations of all applications, replacing key pthreads mutexes with Smart Locks. The lock acquisition scheduling policies within the Smart Locks are varied to a) compare the performance of common policies, two custom policies, and Smart Locks' default dynamically adaptive policy, and to b) estimate bounds in performance. The common policies are taken from the policies intrinsic to two popular spin-locks: test-and-set (Random) and ticket locks (FIFO). The Random policy grants the lock to a waiter at pseudo-random while the FIFO policy grants locks to waiters fairly in the order they arrive. The custom policies are application-specific policies introduced where the common policies are not expected to be upper and lower bounds.

All benchmarks are run with 6 application threads (excluding the startup thread) and one thread reserved for the Smart Data Structures learning thread which runs on a spare core. Large inputs are used for all applications. Each thread is fixed to a core by setting thread affinity. The experiment uses the Linux system tool *cpufrequtils* to configure an 8-core Intel Xeon(r) X5460 system with 8GB of DRAM to emulate a heterogeneous multicore with heterogeneous, fixed clocks speeds of {3.17, 3.17, 2, 2, 2, 2, 2, 2} GHz. Debian Linux kernel version 2.6.26 and `gcc` v.4.3.2 are used.

The following paragraphs describe the benchmarks, custom policies unique to them, and application-specific details such as how a monitor (see Chapter 4.2) is integrated into the application to drive the Smart Locks adaptive policy.

Radiosity The *radiosity* benchmark is a graphics application that computes the equilibrium distribution of light in a scene. Its parallelism employs distributed work queues with work stealing. Work items are imbalanced because the amount of work per item depends on the input scene. *radiosity* was chosen to demonstrate a general scenario where Smart Locks works well: in work queues where Smart Locks can be used as the locking mechanism for work stealing. In this context, varying the lock acquisition scheduling policy allows us to vary the work-stealing heuristic. We have hand-coded good and bad custom policies. The good policy a) optimizes cache locality

by programming the Smart Lock in each queue to prefer the thread that owns the queue most highly then b) minimizes spin idling on the fast cores by ordering the remaining threads by how fast their cores are. The bad policy essentially inverts the good policy. We run the benchmark with 6 worker threads: 2 on the fast cores, 4 on the slow cores. Application Heartbeats is used as the application monitor, and a unit of reward is credited for each work item completed.

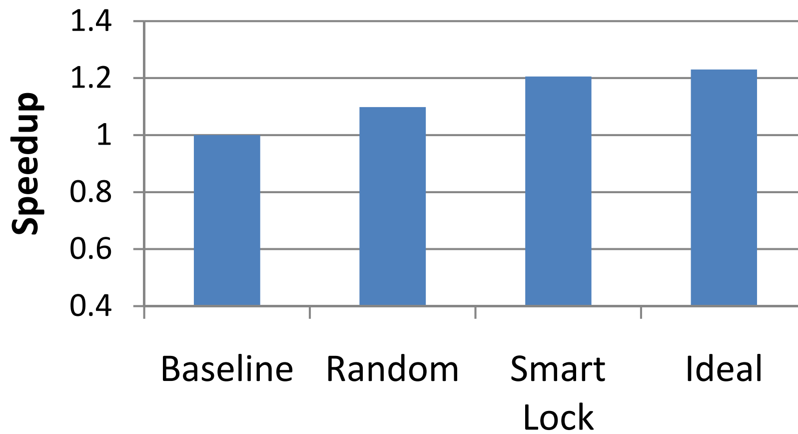
Raytrace The *raytrace* benchmark is a graphics application that renders a 3-d scene using the raytracing algorithm. It was selected to illustrate a general scenario where Smart Locks has little benefit. *raytrace* uses a distributed work queue but differs from *radiosity* in that it has little work stealing. The queues are preloaded with work, so for most of the execution, a worker does not need to steal work and lock contention is negligible. We run this benchmark with 6 worker threads (just like in *radiosity*) and replace the existing locking mechanism in each queue with a Smart Lock. The same custom policies are used. Heartbeats is used, again, with reward credited for each work item completed.

7.3.2 Results

Figure 7-3 shows the performance of the different lock acquisition scheduling policies in *radiosity* and *raytrace*. Speedups are relative to the bad policy. Together, the speedup bounds from the ideal and bad policies capture the variation the application experiences in performance as a function of the policy.

In *radiosity*, the ideal and bad policies are the custom policies. Together, they show that a good scheduling policy can improve performance by a significant 1.23x. As expected, the Random policy performs about half-way between the bounds. The results show that Smart Locks performs within 2% of the ideal speedup, potentially improving performance by 1.2x. In *raytrace*, the custom policies yield the ideal and lowest speedup again. The Smart Lock policy nearly achieves the ideal speedup, but *raytrace* does not see much benefit from lock acquisition scheduling.

Overall, the results demonstrate that the Smart Locks machine learning approach



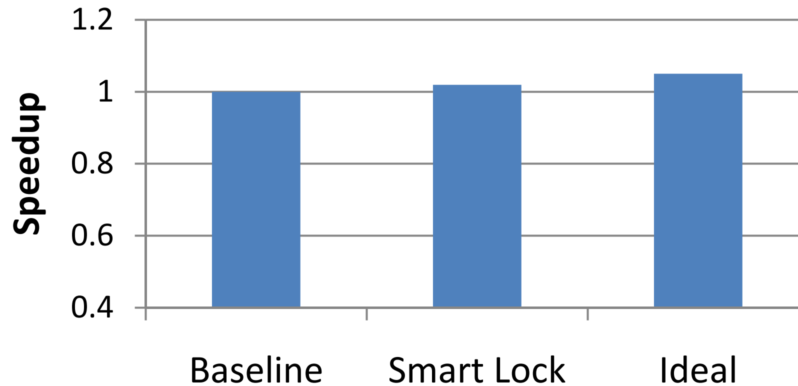


Figure 7-3: Speedup versus lock acquisition scheduling policy. The policy can significantly impact performance. Smart Locks learns a policy that approaches the ideal speedup.

to optimizing and adapting policies is able to a) learn good policies and b) learn them quickly enough that a good policy is used for the majority of execution for the applications studied. We expect performance improvements to be greater on future machines with greater degrees of heterogeneity. Chapter 7.4 further analyzes these benchmarks and the custom policies used in them to provide guidelines for when Smart Locks works best versus less optimally.

7.4 Smart Locks Usage Guidelines

This section defines a set of usage guidelines for Smart Locks based on our findings. We describe a) various use-cases of locks in applications where we have experimentally shown that Smart Locks significantly improves performance and b) some expected limitations of Smart Locks. Then, we study the implications of the Smart Data Structures learning thread architecture on Smart Locks, demonstrating common scenarios where either a) running an extra thread for optimization does not introduce appreciable performance overhead or b) the overhead can be outweighed by the performance gains of lock acquisition scheduling.

7.4.1 Self-Optimizing Data Structures

The results in Chapter 7.3.2 suggest that spin-lock scheduling policies may have some nonobvious implications for locality and load balancing in multicore applications. They demonstrate a scenario where Smart Locks' adaptive policy significantly improves performance by automatically learning policies (i.e. the custom policy in *radiosity* that optimizes for locality and minimizes the spin times of fast cores). Additionally, the *raytrace* results show a scenario where Smart Locks is not able to improve performance much: when lock contention is low. Together, these results help us to understand when Smart Locks works well vs. less optimally.

Table 7.1: Expected Utility of Smart Locks by Scenario

Application Scenario	Expected Utility
Work queues	Good
Pipeline queues	Great
Graph / grid	Neutral
Heap locking	Good or Neutral

Table 7.1 summarizes our findings from Chapter 7.3.2 and additional expectations. We studied the SPLASH-2 and PARSEC benchmark suites to see how locks were used and found they are most often used in the concurrent data structures that coordinate the parallelism in the applications – specifically, in those listed in the table. We have already shown that Smart Locks can significantly improve work queues.

For multi-stage software pipelines, we expect strong gains because Smart Locks can prevent producer starvation and/or help balance pipeline stages to maximize overall pipeline throughput. For graph /grid applications, we expect small to negligible gains. Finally, for memory heaps, we expect good to neutral gains, depending on application conditions.

In graph / grid data structures, there are often thousands of nodes, each with a lock. We expect that the current learning thread architecture within Smart Lock may not scale to thousands of Smart Locks if heterogeneities change rapidly. The problem is that the learning engine for each instantiated lock executes in the same shared learning thread and may not execute frequently enough to be responsive to rapid changes. Because these applications have data-dependent behavior, we do expect rapid changes. Future scalability can be mitigated by spawning multiple learning threads threads.

As for memory allocator heap locking, the impact of lock acquisition scheduling will depend on whether or not the application uses dynamic allocation or allocates upon initialization then reuses memory. The problem with the latter is that reusing memory avoids the allocator and thus makes lock contention in the memory allocator heap low, providing no opportunity for Smart Locks to make improvements.

7.4.2 Learning Thread Sensitivity Analysis

As explained in Chapter 4.2, the optimization components of Smart Locks run decoupled in a shared learning thread. This section addresses the question of what performance tradeoffs there are for running that learning thread alongside applications. We discuss the overhead for each of three common multicore scenarios: many-cores, multicores with SMT, and multicores without SMT.

Many-Cores In many-core machines, hundreds of cores are available for applications. Except for embarrassingly parallel applications, applications will eventually reach scalability limits on these machines where further parallelization (adding cores) no longer improves performance. This is a well-known consequence of Amdahl's Law

and/or the increasing overheads of communication vs computation as parallelism becomes more fine-grained. One way to continue to improve performance is by utilizing spare cores to run optimization threads. The Smart Data Structures learning thread is one example of this class. Some many-core computers are available today: i.e. the Tilera Tile-Gx® with up to 100 cores. Many-cores chips from Intel® and AMD® are coming in the next few years.

Multicores With SMT In a multicore SMT machine, the Smart Data Structures learning thread can run in the same core as an application thread and share execution resources. The learning thread is the ideal candidate for SMT because it is computation-heavy and light on other resources. Applications should see nearly the full performance gains of lock acquisition scheduling while hiding the overhead of the learning thread. SMT multicores such as Intel's® current x86® multicore are widely available today.

Multicores Without SMT Large-scale multicores with or without SMT are many-cores and will thus benefit from Smart Locks. On small-scale multicores without SMT, using Smart Locks can improve performance if the performance gains of lock acquisition scheduling outweigh the performance tradeoffs of taking a thread away from the application for parallelism; otherwise, Smart Locks should not be used. The exact tradeoff to overcome is different for each application and depends on its scalability.

In Figure 7-4, we quantify the tradeoff of taking away a core for SPLASH-2 applications. We compare 7 application threads vs. 6 application threads plus 1 learning thread. For reference, we also compare against 6 application threads. We use the standard large inputs to the applications and run on an 8-core Intel Xeon(r) x5460, each core at 3.17 GHz. Our system runs Debian Linux kernel version 2.6.26, has 8GB of DRAM, and all code is compiled with `gcc` v.4.3.2.[1]

For this scenario and our system, we find that lock acquisition scheduling would need to lower execution time by {1.1x, 1.16x, 1x, .93x, 1.28x} to benefit *barnes*, *fmm*,

[1] *ocean* requires 2^n threads and *volrend* fails on our system.

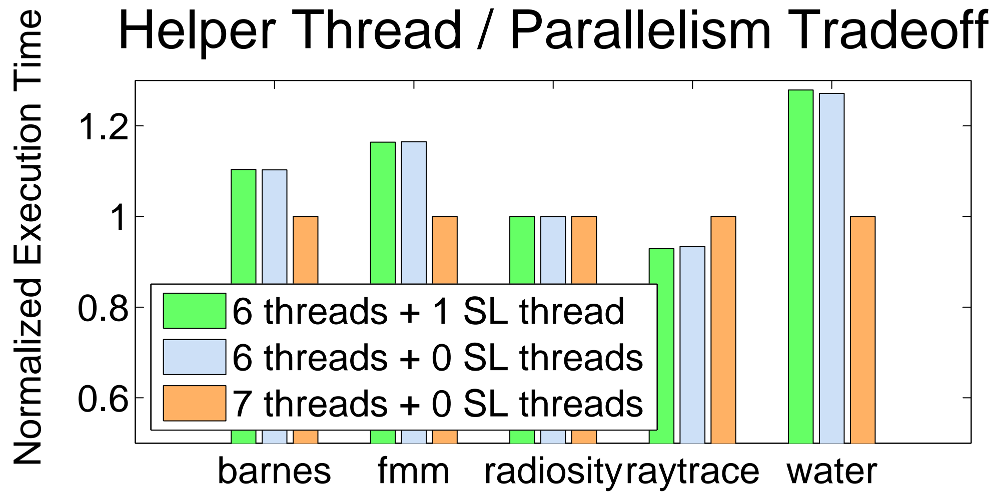


Figure 7-4: Normalized execution time of SPLASH-2 applications. 6 threads with an additional thread for Smart Locks vs. 6 threads vs. 7 threads. The slowdown reflects by what factor Smart Locks must improve performance for net benefit.

radiosity, *raytrace*, and *water*, respectively.[2] In Chapter 7.3.2, we demonstrated that lock acquisition scheduling does indeed improve *radiosity* by up to 1.2x. Thus, the max net improvement is nearly 1.2x even after accounting for the extra thread. A future study will determine if Smart Locks can yield net improvement for the other applications. Regardless, Smart Locks is expected to do well on a) many-cores and b) multicores with SMT support.

[2] *radiosity*, *raytrace* don't benefit from the extra core; they are known to scale poorly relative to other SPLASH-2 apps.

Chapter 8

Future Work

Smart Data Structures take a novel, online learning approach to optimizing the performance of parallel data structures. Our hope is that the results of this work and our open source library of Smart Data Structures will help lay the foundation for future research in the area of optimizing systems via machine learning.

This chapter identifies several interesting ideas for further research. First, we discuss opportunities for extending the scalability of Smart Data Structures in the coming decade as larger and larger machines become available. Next, we discuss several additional data structures to which we believe our design methodology can be applied for substantial improvements. Then, we describe the possibility for an additional adaptation in Smart Data Structures on top of adapting data structure knob settings. Then, we describe an alternative integration strategy for online learning that, while more complex, may potentially increase performance. Finally, we discuss applications of the novel online learning optimization methodology developed in this work to systems beyond data structures.

8.1 Scalability Enhancements

Chapter 6 demonstrates that our design is expected to achieve good performance improvements up to 256 threads in typical application scenarios. Nevertheless, it is possible to extend the scalability of our design further. This section describes a few

approaches to extending scalability.

Flat Combining Scalability We have shown that the Flat Combining algorithm upon which the Smart Data Structures library is built is the most significant scalability limiter. Flat Combining scalability can be significantly improved by developing hierarchical versions of the Flat Combining algorithm with multiple parallel combiners. There is ongoing work in parallel combining. We leave to future work the task of monitoring progress on these algorithms and incorporating them when they become available.

Learning Algorithm Complexity The algorithmic complexity of the learning engine can also limit scalability for applications that need joint optimization of Smart Data Structures. So far, we have not run into such scaling limits in our application studies. Nevertheless, the complexity of our learning algorithm can be improved to provide further scalability. The learning algorithm must solve a dense system of linear equations. To do so, we currently use a somewhat naïve method of QR factorization and solving the least-squares problem. More efficient algorithms are known. Furthermore, the learning algorithm can be parallelized. Parallel algorithms exist which, using n^2 processors, would reduce the asymptotic complexity of the learning algorithm to $O(n)$ from the current $O\left(n^3\right)$. Further investigation is left to future work.

Learning Thread Time-Multiplexing Some of the case-study applications in Chapter 6.5.2 on scalability are limited (to a lesser degree) by the time-multiplexing of a single learning thread in our design. Future work on the library will enable support for multiple learning threads. A linear increase in the number of learning threads reduces time-multiplexing linearly and thus increases application scalability by the same factor in many cases. Since most applications only reach scaling limits due to time-multiplexing when they use 256 threads or more, dedicating a few additional threads to learning will be comparatively inexpensive.

8.2 Additional Smart Data Structures

In this work, we have demonstrated "Smart" versions of queues, skip lists, pairing heaps, and locks – several important data structures for parallel programs and programming models – which substantially improve performance through the use of online learning. We are also investigating the design of other Smart Data Structures. Additional implementations based on Flat Combining are possible: a Smart Stack is one example. However, the Smart Data Structures design methodology is applicable to a wide range of other data structures.

For example, we are investigating ways to use learning to improve load-balancing in Smart Distributed Hash Tables. We suspect that learning can significantly optimize the organization of nodes and data. We are also designing a Smart Counter that uses learning to adapt its implementation at runtime to optimize for different concurrency levels. For example, at large concurrency levels, lazy counters would be used because they scale better. The tradeoff is that reads may incur extra overhead if a global total is needed because counter data is distributed across multiple counters. At low concurrency levels, an implementation based on a global shared counter would be used. Updates and reads would rely on atomic operations which scale less well, but reads of the global total would incur less overhead.

8.3 Additional Axes of Adaptation

We have demonstrated that online machine learning is an effective technique for optimizing data structure knob settings and that Smart Data Structures can significantly improve application performance by optimizing knob settings. There is another adaptation that we can layer on top of knob optimization: adapting data structure algorithms at runtime using online learning. The previous discussion of the Smart Counter alluded to algorithm adaptation. Since our online learning framework supports joint-optimization of multiple parameters, it can learn to adapt algorithms while optimizing knob settings.

One interesting challenge to solve in adapting algorithms is creating a consensus protocol to ensure correct operation in the face of algorithm switches: threads with pending requests based on the previous algorithm should gracefully transition to using the new algorithm, and the interaction between threads temporarily using different algorithms should not lead to race conditions or other errors. Another interesting challenge involves gracefully migrating data when the algorithm switches, so that data stored for the previous algorithm is available for use by the next algorithm. We believe that it will be possible to identify a certain set of algorithms that can all utilize the same underlying storage to avoid the need for migration.

8.4 Alternative Learning Integration Strategies

The current Smart Data Structures design integrates online learning through the use of a learning thread. As described in Chapter 4.2, the learning thread is an extra dedicated thread which runs decoupled from the threads of the application. This design is motivated by producing knob optimizations as quickly as possible and reacting to rapid changes in the system or application which change the optimal knob settings over time. Our experiments in Chapter 6.2.6, however, show that our learning engine may run much faster than necessary in some cases. Moreover, the use of the learning thread is not entirely free of tradeoffs. In most cases they are easily justified (see Chapter 4.4), but they may perhaps be removed.

Future work will investigate adding a new integration method for our online learning engine. This method will interleave learning computation into the application threads. One approach is to step the learning engine at the end of every i^{th} combining phase; in other words, rather than the learning thread, the combiner steps the learning engine, and it steps it periodically rather than as often as possible. The tradeoff of this approach is that the latency of data structure operations may increase since combining takes longer. Some applications will be negatively affected.

Another approach is to attempt to utilize spare cycles in the threads that are waiting for their operation to complete and not combining. While waiting, threads

otherwise spin idly. Instead, they could work together to carry out a distributed learning algorithm. This algorithm may be complex to implement, however, because a) it is a parallel learning algorithm and b) threads will join in and leave the parallel algorithm dynamically. Despite the implementation complexity, this approach may provide comparable or better reactivity to changing system conditions than the decoupled learning thread architecture we currently adopt while simultaneously eliminating the reliance on an extra thread.

8.5 Applications to Other Systems

While this thesis focuses on a case study of optimizing data structures using online machine learning, we have taken care to design our abstractions and learning engine so that they may be applied to the online optimization of other systems as well.

The abstraction of the knob is one that is natural for use in many different systems. For example, in cloud managers, a promising knob to optimize may be the policy used for resource allocation. A promising knob in OS scheduling is the spatial location of different application threads on the chip: strategic co-location of threads can minimize communication costs. Promising knobs in the hardware include the cache hash function and cache coherence protocol: a strategic hash function or adaptive coherence protocol may help minimize overheads due to hotspots.

Like the abstraction of knobs, our learning engine is designed for generality as well. Our learning engine supports a variety of different types of knobs from permutation orders suited to allocation policies and scheduling to discrete-valued knobs suited to protocol selection. Our learning engine also supports Gaussian distributions and boolean-valued knobs, and it can be easily extended to learn other types of knobs. Further, the learning engine is designed for efficient joint optimization of multiple knobs. This will enable it to compose knob optimizations from multiple components or different layers of a system. For example, in future work, we will explore a framework for jointly optimizing knobs within the OS, application (e.g. knobs in Smart Data Structures), and the hardware at the same time.

Chapter 9

Conclusion

This thesis demonstrated a novel methodology for using online machine learning to design self-aware data structures. We developed a new class of data structures based on this methodology called Smart Data Structures which optimize themselves continuously and automatically to help eliminate the complexity of hand-tuning data structures for different systems, applications, and workloads. We developed an open source library of Smart Data Structures with which we evaluated our learning-based optimization methodology. Through empirical evaluations of our prototype, we showed constructively that:

- Online machine learning *is* an effective strategy for automatically tuning data structures
- Learning *is* efficient enough for fine-grained online optimization of data structures
- Online learning can enable *significant* improvements of up to 44% over state-of-the-art algorithms
- Learning *is not* the scalability limiter for the data structures we have studied

The results that we have demonstrated with data structures suggest that online learning is a promising approach to optimizing other systems as well. While this

work focused on data structures, we designed our learning engine and abstractions to be general, flexible, and composable so that they may be applied effectively in other systems. Our learning-based methodology will not be a silver bullet for mitigating all of the complexities of optimizing systems, but we hope it provides a foundation for researchers to further investigate synergies between systems and learning. Our view is that online learning is a robust and high performance framework for weighing complicated dynamic tradeoffs, and that online learning will play an essential role in the development of future systems.

Appendix A

Lazy Counter Algorithm

Chapter 6 evaluated a scalable reward monitor that we designed around a concurrent counter algorithm that we call *lazy counters*. This appendix describes lazy counters and how we implement a reward monitor based on them.

Lazy counters are so called because, by design, they do not guarantee up-to-date values when read by a different thread than the thread that owns them. Lazy counters have one owner in the sense that they can be written by only one thread; any other thread may read them only. Lazy counters rely on a property of shared memory systems which guarantees that, after some machine-specific delay for coherency messages to propagate through the system, the other threads will eventually read the latest value written by the owner.

In the context of Smart Data Structures, we want to build a reward monitor using lazy counters. We observe that all application threads may potentially increment the reward by issuing writes to it and that only the learning thread needs to read it. We also observe that writes are much more frequent than reads because a) there are many application threads and one learning thread and b) the learning thread typically performs much more computation between reads than the application threads do between writes.

Thus, as indicated in Figure A-1, we dedicate a separate lazy counter for each of the n application threads and require the learning thread to read the n counters individually and sum them to get the total reward. Because of the semantics of

coherent shared memory, the learning thread may read a slightly out-of-date reward.

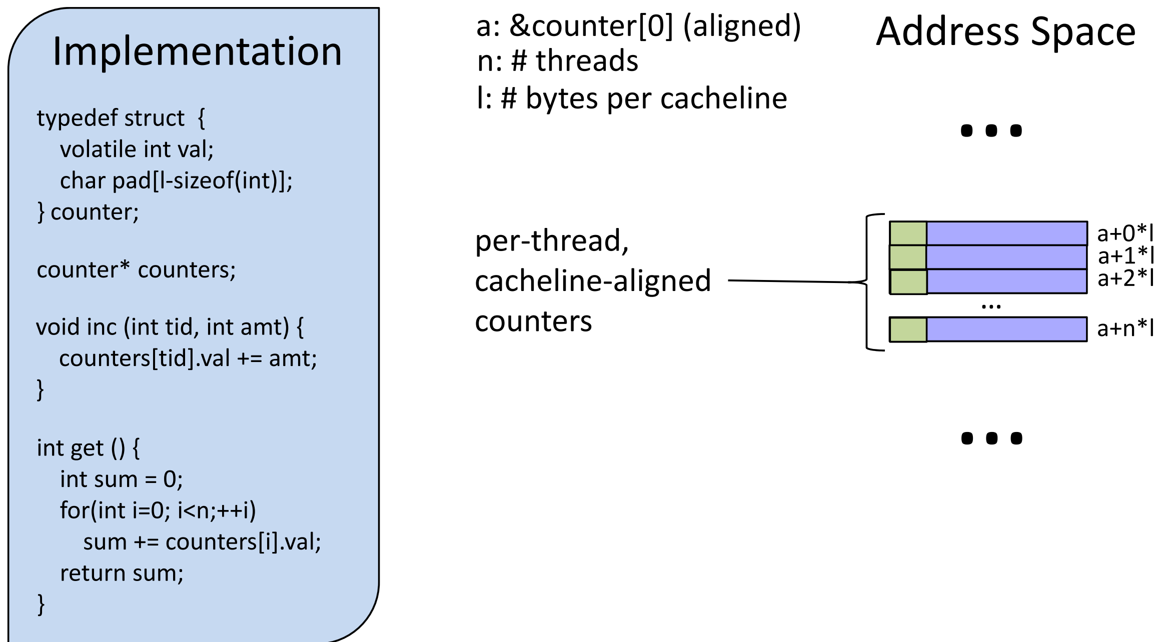


Figure A-1

This strategy is high performance because a) reads are much less frequent than writes and b) lazy counters are spaced in memory such that they map to separate cache lines and avoid shared memory bottlenecks. One tradeoff, however, is that storage grows linearly in the number of application threads. A simple modification can limit storage requirements: multiplex a fixed number of counters among the n threads. Threads that share a counter will now have to atomically increment it. The read by the learning thread, however, remains non-atomic. Atomic operations introduce synchronization overhead, but if the number of threads sharing a counter is small, the overhead is small. Since modern machines have multiple megabytes of cache, in many cases, making this tradeoff to minimize storage will be unnecessary.

Bibliography

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. *SIGPLAN Not.*, 44(6):38–49, 2009.

[2] 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.

[3] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.

[4] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[5] Katherine E. Coons, Behnam Robatmili, Matthew E. Taylor, Bertrand A. Maher, Doug Burger, and Kathryn S. McKinley. Feature Selection and Policy Optimization for Distributed Instruction Placement Using Reinforcement Learning. In *PACT '08: Proceedings of the 17th International Conference on Parallel Archi-*

tectures and Compilation Techniques, pages 32–42, New York, NY, USA, 2008. ACM.

[6] Mike Stonebraker Daniel, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Alex Rasin, Nga Tran, Stan Zdonik, and Waltham Ma. C-Store: A Column-Oriented DBMS, 2005.

[7] Jonathan Eastep, David Wingate, and Anant Agarwal. Smart Data Structures Project. github.com/mit-carbon/Smart-Data-Structures.

[8] Jonathan Eastep, David Wingate, and Anant Agarwal. Smart Data Structures: An Online Machine Learning Approach to Multicore Data Structures. In *ICAC 2011 Proceedings*, June 2011.

[9] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: Lock Acquisition Scheduling for Self-Aware Synchronization. In *ICAC 2010 Proceedings*, June 2010.

[10] A. Fedorova, D. Vengerov, and D. Doucette. Operating System Scheduling on Heterogeneous Core Systems. In *Proceedings of the Workshop on Operating System Support for Heterogeneous Multicore Architectures*, 2007.

[11] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of ICASSP*, pages 1381–1384. IEEE, 1998.

[12] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *SPAA '10: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, New York, NY, USA, 2010. ACM.

[13] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining C++ Framework Version 2.0. sites.google.com/site/cconcurrencypackage/flat-combining-and-the-synchronization-parallelism-tradeoff, 2010.

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

[15] Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Reactive Spinlocks: A Self-tuning Approach. In *Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks*, pages 33–39, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Moshe Hoffman, Ori Shalev, and Nir Shavit. The Baskets Queue. In *OPODIS'07: Proceedings of the 11th International Conference on Principles of Distributed Systems*, pages 401–414, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] Henry Hoffmann, Jonathan Eastep, Marco Santambrogio, Jason Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *ICAC 2010 Proceedings*, 2010.

[18] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A Framework for Self-aware Management of Multicore Resources. Technical Report MIT-CSAIL-TR-2011-016, CSAIL, MIT, March 2011.

[19] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 39–50, 2008.

[20] R. C J.Dongarra. Automatically Tuned Linear Algebra Software. Technical report, Knoxville, TN, USA, 1997.

[21] Daniel A. Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 197, Washington, DC, USA, 2001. IEEE Computer Society.

[22] Theordore Johnson and Krishna Harathi. A Prioritized Multiprocessor Spin Lock. *IEEE Trans. Parallel Distrib. Syst.*, 8(9):926–933, 1997.

[23] Alain Kägi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 170–180, New York, NY, USA, 1997. ACM.

[24] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious Synchronization. *ACM Trans. Comput. Syst.*, 15(1):3–40, 1997.

[25] B. H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 28(5):25–35, 1994.

[26] Itay Lotan and Nir Shavit. Skiplist-Based Concurrent Priority Queues. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 263, Washington, DC, USA, 2000. IEEE Computer Society.

[27] P. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 165–171, Apr 1994.

[28] Sridhar Mahadevan. Average Reward Reinforcement Learning: Foundations, Algorithms, and Empirical Results. *Machine Learning*, 22:159–196, 1996.

[29] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[30] John M. Mellor-Crummey and Michael L. Scott. Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[31] John M. Mellor-Crummey and Michael L. Scott. Synchronization Without Contention. *SIGARCH Comput. Archit. News*, 19(2):269–278, 1991.

[32] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC '96: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, New York, NY, USA, 1996. ACM.

[33] Marek Olszewski and Michael Voss. Install-Time System for Automatic Generation of Optimized Parallel Sorting Algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 17–23, 2004.

[34] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural Actor-Critic. In *European Conference on Machine Learning (ECML)*, pages 280–291, 2005.

[35] Harald Prokop. Cache-Oblivious Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[36] Zoran Radović and Erik Hagersten. Efficient Synchronization for Nonuniform Communication Architectures. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–13, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[37] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[38] Gerald Tesauro. Online Resource Allocation Using Decompositional Reinforcement Learning. In *Proceedings of AAAI-05*, pages 9–13, 2005.

[39] Cai-Dong Wang, Hiroaki Takada, and Ken Sakamura. Priority Inheritance Spin Locks for Multiprocessor Real-Time Systems. *International Symposium on Parallel Architectures, Algorithms, and Networks*, 0:70, 1996.

[40] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason E. Miller, and Anant Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *SoCC*, 2010.

[41] Shimon Whiteson and Peter Stone. Adaptive Job Routing and Scheduling. *Engineering Applications of Artificial Intelligence*, 17:855–869, 2004.

[42] R. J. Williams. Toward a Theory of Reinforcement-Learning Connectionist Systems. Technical Report NU-CCS-88-3, Northeastern University, 1988.