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

Jonathan Eastep, David Wingate, Anant Agarwal

Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge, MA 02139
{eastep, wingated, agarwal}@mit.edu

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 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.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software Libraries*; I.2.6 [**Artificial Intelligence**]: Learning—*Connectionism and Neural Nets*

General Terms

Algorithms, Design, Performance

Keywords

Self-Aware, Autonomic, Auto-Tuning, Concurrent Data Structures, Synchronization, Performance Optimization

1. INTRODUCTION

As multicores become prevalent, programming complexity is skyrocketing. Programmers expend significant effort on parallelizing and mapping problems 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 data structure algorithms and algorithm parameter settings. The best algorithm and parameter settings depend on the machine's memory system architecture as well as application-specific criteria such as the load on the data structure. Many applications have input-dependent computation which causes the load to vary dynamically. Getting 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. Programmers should not be expected to code for these complexities by hand.

Recently, self-aware computing has been proposed as one automatic approach to managing this complexity. Self-aware systems are closed-loop systems that monitor themselves online and optimize themselves automatically. They have been called autonomic, auto-tuning, adaptive, self-optimizing, and organic systems, and they have been applied to a broad range of platforms including embedded, real-time, desktop, server, and cloud computing environments.

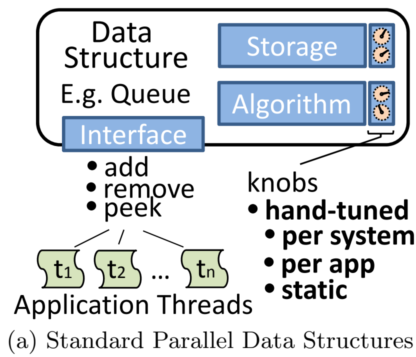
This paper introduces Smart Data Structures, a new class of parallel data structures that leverage online machine learning and self-ware computing principles to self-tune themselves automatically. As illustrated in Figure 1a, standard parallel data structures consist of data storage, an interface, and 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.

Storage and algorithms are often controlled by *knobs* which are 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 suboptimally, programmers must hand-tune the knobs; typically, they do so through trial and error and special cases in the code which increase code complexity and reduce readability. Though often necessary, runtime tuning of the knobs is typically beyond the reach of the programmer.

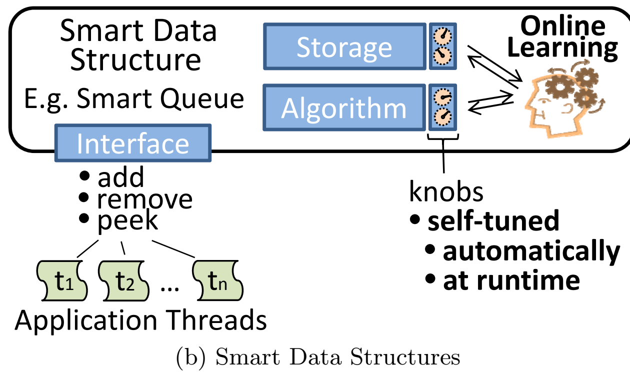
Figure 1b contrasts Smart Data Structures with standard data structures. While preserving their interfaces, Smart Data Structures augment standard data structures with an online learning engine that automatically and dynamically optimizes their knobs. Through learning, Smart Data Structures balance complex tradeoffs to find ideal knob settings and adapt to changes in the system or inputs that affect these tradeoffs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICAC'11, June 14–18, 2011, Karlsruhe, Germany.
Copyright 2011 ACM 978-1-59593-998-2/09/06 ...$10.00.

المنارة للاستشارات
www.manaraa.com



(a) Standard Parallel Data Structures



(b) Smart Data Structures

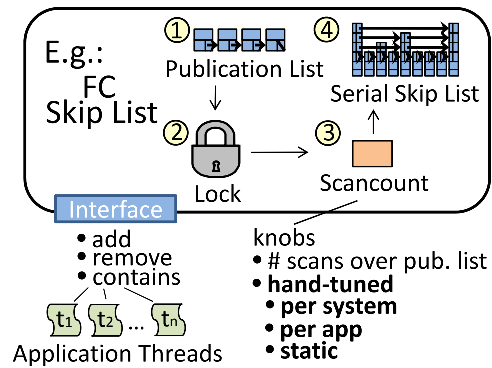
Figure 1: The Anatomy of Smart Data Structures. Smart Data Structures augment standard data structure interfaces, storage, and algorithms with online machine learning to internally optimize the knobs that control their behavior.

This work prototypes an open source (GPL) library of Smart Data Structures consisting of learning-enabled implementations of common parallel programming data structures. It is available on github [1] and includes: a Smart Lock [2] for locks, a Smart Queue for software pipelining and work queues, and a Smart Skip List and Pairing Heap for prioritized queues. A Smart Stack, Distributed Hash Table, and others are planned for future expansions.

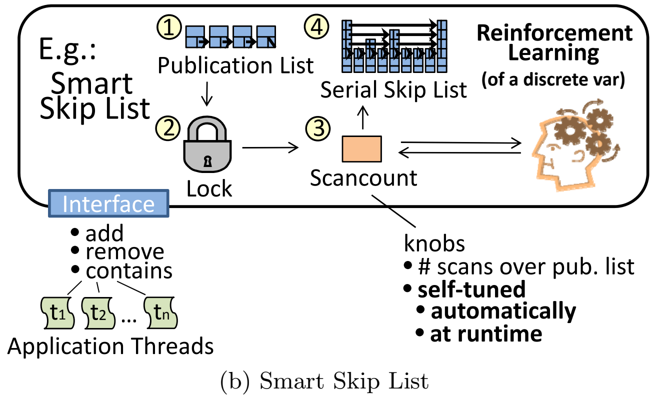
Different Smart Data Structures may use different online learning algorithms and apply learning in different ways. In previous work, we developed Smart Locks [2], self-tuning spin-locks that use Reinforcement Learning to optimize the order and relative frequency with which different threads get the lock when contending for it: an optimization called *Lock Acquisition Scheduling*. Smart Locks implement Lock Acquisition Scheduling by learning a permutation order over the contending threads. We show that Lock Acquisition Scheduling and unfair allocation of critical sections can accelerate applications on heterogeneous multicores [2].

This paper focuses on the design of the Smart Queue, Skip List, and Pairing Heap. Their implementation augments a recently developed data structure algorithm called Flat Combining [3] with an online learning engine. Like the Smart Lock, these new Smart Data Structures use Reinforcement Learning but use it to optimize an important discrete-valued knob in the Flat Combining algorithm rather than a permutation order over threads. That discrete-valued knob is called the *scancount*. In Section 4.2 we motivate our decision to build on Flat Combining by showing that it outperforms the best prior algorithms in many scenarios.

Figure 2a shows the Flat Combining data structure design. Flat Combining data structures are non-blocking, shared



(a) Flat Combining Skip List



(b) Smart Skip List

Figure 2: The Smart Queue, Skip List, and Pairing Heap. 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*.

memory data structures that consist of a *publication list*, a 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 2a. 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.

المنارة للاستشارات
www.manaraa.com

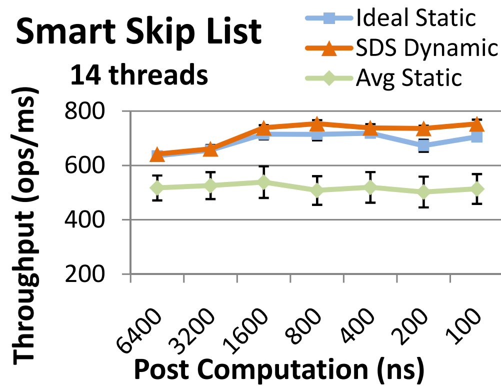


Figure 3: Smart Skip List Throughput vs. Load. Through online learning, the Smart Skip List significantly improves performance over the average static bound, achieving and exceeding the ideal static bound.

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 [3]. 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 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, hereafter referred to as simply *Smart Data Structures*, significantly improve performance over Flat Combining by optimizing the scancount dynamically. Figure 2b shows that Smart Data Structures do this by augmenting Flat Combining with an online Reinforcement Learning engine which balances the tradeoffs to find the optimal scancount. Our experiments on a 16-core Intel(r) Xeon(r) system demonstrate that, through learning, Smart Data Structures can outperform the state-of-the-art (Flat Combining) by up to 1.5x over a range of conditions and readily adapt to rapid changes in the application workload that cause the optimal scancount to vary over time.

Figure 3 gives an example of our experimental results. Using the benchmark developed by Hendler et al. [3] (described in Section 4.1), we compare the throughput of the Smart Skip List data structure which dynamically tunes the scancount to the ideal and average throughput achieved by the Flat Combining Skip List using a range of static values. The difference between the ideal and average static throughput demonstrates that finding the optimal value can substantially improve throughput. Further, the results show that the Smart Skip List is able to learn optimal scancount values and reach the ideal static throughput. Its throughput actually exceeds the ideal static throughput because, for data structures like the Skip List, the optimal scancount can vary during execution and necessitate dynamic tuning.

To summarize, the major contributions of this work are:

- a methodology based on machine learning for designing self-aware parallel data structures
- open source implementations of popular data structures that outperform the state-of-the-art by up to 1.5x

The rest of this paper is organized as follows. Section 2 compares Smart Data Structures to related works. Section 3 motivates and describes our learning-based design. Section 4 presents our experimental results. Section 5 concludes.

2. RELATED WORK

This paper introduces a new class of data structures called Smart Data Structures. Building upon design principles from self-aware computing, Smart Data Structures leverage online machine learning to self-optimize themselves for different machines, applications, and inputs automatically. This section compares Smart Data Structures to related works. It surveys other examples of machine-learning-based self-aware systems then compares the Smart Data Structures design methodology to existing frameworks for adaptive data structures. For background on concurrent data structures, see the online appendix [4].

2.1 Learning-Based Self-Aware Systems

Self-aware computing has become a popular approach to reducing the complexity of programming and designing systems. An effective implementation strategy is to use machine learning. Researchers have built 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.

Ipek et al. apply Reinforcement Learning and neural nets to multicore resource management [5, 6]. They build self-optimizing hardware agents that adapt the allocation of critical resources according to changing workloads. Hoffman et al. [7] 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 [8]. Wentzlaff et al. from MIT are investigating various applications of machine learning to the operating system that they are designing for multicores and clouds [9].

Fedorova et al. extend learning to heterogeneous multicores to coordinate resource allocation and scheduling [10]. Their system produces scheduling policies that balance optimal performance, core assignments, and response-time fairness using Reinforcement Learning. Whiteson and Stone use Q-learning to improve network routing and scheduling [11].

Work in libraries and optimization has also benefited from learning-based self-aware computing. We developed a spinlock library called Smart Locks which uses Reinforcement Learning in a technique called *Lock Acquisition Scheduling* to optimize data structure performance in heterogeneous multicores [2]. Coons et. al apply genetic algorithms and Reinforcement Learning to compilers to improve instruction placement on distributed microarchitectures with results rivaling hand-tuning [12]. Finally, Jimenez and Lin apply the perceptron algorithm to dynamic branch prediction and propose a scheme that substantially outperforms existing techniques [13]. Our view is that these pioneering works are

المنارة للاستشارات
www.manaraa.com

evidence that machine learning will play an essential role in the development of future systems.

2.2 Adaptive Data Structures and Programs

While machine learning has been applied to a variety of self-aware systems, not much work has investigated using learning for adaptive data structures. The Smart Locks work [2] is one of the only examples. There have been a variety of related works in auto-tuned libraries and programming frameworks that use other means. Typical auto-tuned libraries select from a repertoire of implementations based on the input size, a sampling of data characteristics, and install-time benchmarks on the machine. Domain-specific libraries like PHiPAC, ATLAS, BLAS, LAPACK, FFTW, and adaptive sort have demonstrated great performance for linear algebra and sorting applications [14, 15, 16, 17].

Several programming frameworks for adaptive parallel programs exist as well. STAPL is a well-known C++ framework that provides adaptive parallel implementations of the ISO C++ data structures [18]. PetaBricks is a programming language and compiler from MIT for building adaptive programs. Using the PetaBricks framework, programmers specify multiple implementations of each algorithm using a syntax that enables the compiler to make algorithm decisions and compose parallel algorithms [19].

The limitation of these works is that, while they may adapt to different architectures and runtime conditions like input size, they typically base these decisions on thresholds computed during compile-time or install-time characterization. These characterizations may poorly reflect realistic runtime conditions. Newer technologies like dynamic frequency scaling (i.e. thermal throttling and Intel's Turboboost(r)) and even fundamental properties of multi-process environments such as variable competition for hardware resources can alter effective machine performance and substantially affect the tradeoffs that determine which algorithm and/or algorithm knob settings are best. Smart Data Structures take an online, adaptive approach to balancing tradeoffs. Through Reinforcement Learning, our design is robust to changes in effective machine performance.

3. IMPLEMENTATION

This work introduces a new class of data structures called Smart Data Structures which leverage online machine learning to optimize themselves at runtime. This section begins by describing the implementation of our open source library of Smart Data Structures. 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. Our descriptions of the implementation are framed around three key challenges that govern our use of learning and choice of learning algorithms. The major challenges are 1) measuring performance in a reliable 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. Finally, this section motivates some tradeoffs made by our design.

3.1 Application Interface

Smart Data Structures are concurrent, non-blocking data structures written in C++ for shared memory C / C++ applications. C interfaces are provided for mixing with other languages. 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.

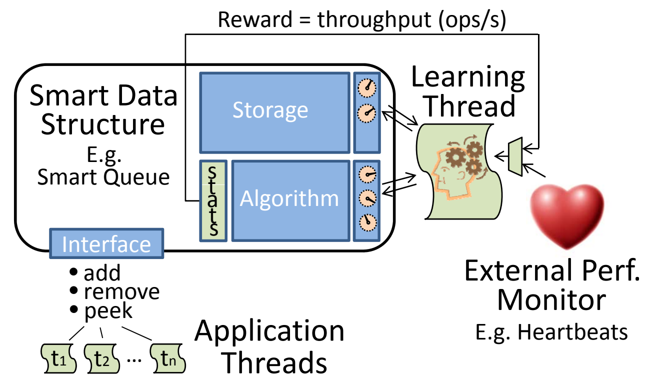


Figure 4: Smart Data Structures Internals. All Smart Data Structures share a learning thread which jointly optimizes the knobs that control their behavior. Performance feedback, the reward, drives the optimization.

3.2 Smart Data Structures Internals

While preserving standard interfaces, Smart Data Structures internally use machine learning to optimize their implementation of those interfaces. Figure 4 illustrates. All Smart Data Structures share a common learning thread that runs a learning engine which jointly optimizes their knobs (their scancounts in this case). We use online machine learning as our optimization technology because it is an effective technique for balancing the complexities of joint optimization and adapting to time-varying tradeoffs.

Optimization within the learning thread is driven by a reward signal. We address Challenge 1 (ensuring that performance measurements are portable and reliable) by supplying an internal reward source that meets these criteria for many applications: by default, Smart Data Structures measure and provide their throughput as the reward. As a failsafe, we also support a variety of external performance monitors developers can use to provide application-specific reward signals. One we recommend is Application Heartbeats [20]. 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.

We address Challenge 2 (adapting settings quickly so as not to miss windows of opportunity) in part by running the learning engine in its own dedicated thread rather than interleaving the learning computation within the application code in the application threads. This decoupling allows the learning engine to run faster, deliver optimizations sooner, and minimize disruption of the application threads. As Section 3.3 will elaborate, the other way we meet this challenge is through our choice of learning algorithms.

3.3 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

المنارة للاستشارات
www.manaraa.com

long-term performance), our Smart Data Structures library employs a Reinforcement Learning (RL) algorithm [21] 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 is incredibly large and 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 online use and b) can tolerate severe partial observability, we adopt an average reward optimality criterion [22] and use policy gradients to learn a good policy [23]. In particular, we use the Natural Actor-Critic algorithm [24].

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 our case, 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 [24]. 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 each Smart Data Structure, 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

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

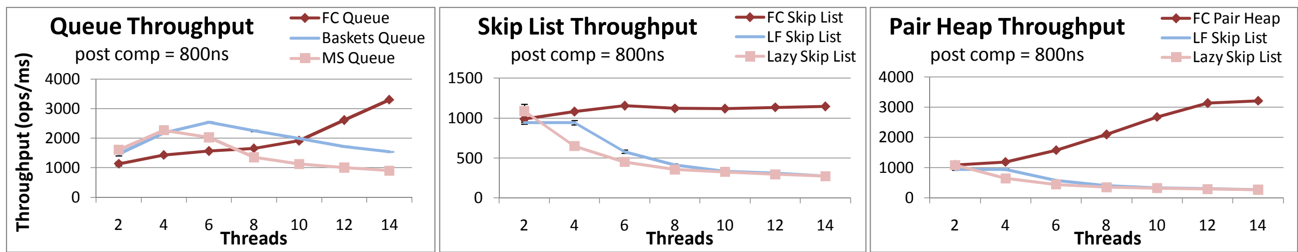
where $\delta(a_t^i)$ is a vector of zeros with a 1 in the index given by a_t^i. When enough samples are collected (or some other gradient convergence test passes), we take a step in the gradient direction: $\theta = \theta + \alpha w$, where w is computed in Alg.1 and α is a step-size parameter. Currently,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

3.4 Learning Thread Tradeoffs

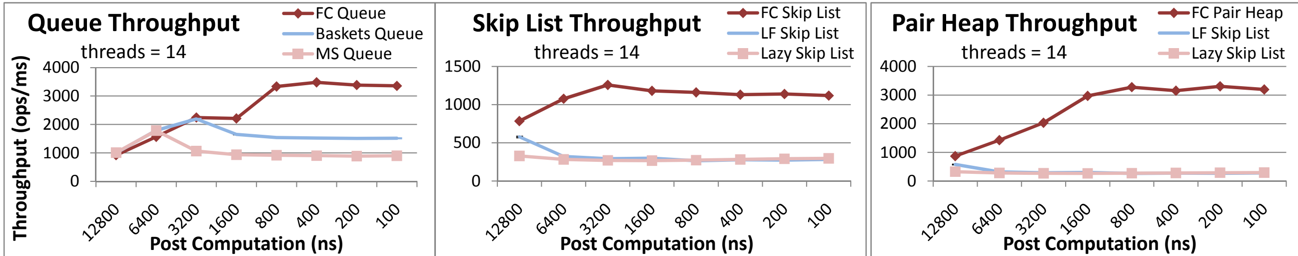
As previously described, the learning engine uses Alg. 1 to jointly optimize scancounts for all Smart Data Structures. To run the learning engine, our design adds one thread to the application. The advantage is that this minimizes application disruption and enables background optimization of the application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

المنارة للاستشارات
www.manaraa.com



(a) Throughput vs. Concurrency Level



(b) Throughput vs. Post Computation

Figure 5: 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).

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 $n/(n-1)$ 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 1.5x on a 16-core machine. Future work will investigate this scenario further.

4. RESULTS

This section evaluates our prototype library of Smart Data Structures. It starts with a description of our experimental setup then presents four studies. The first characterizes the performance of the best existing data structure algorithms and shows that the Flat Combining data structures [3] 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* value on data structure performance. It shows that the best value varies widely, that hand-tuning would be cumbersome, and that using the optimal 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 1.5x 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.

4.1 Experimental Setup

The experiments are performed on a 16-core (quad 4-core) Intel(r) Xeon(r) 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.

All experiments measure data structure throughput using a modified version of the synthetic benchmark developed in the Flat Combining paper [3]. 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 sec-

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

المنارة للاستشارات
www.manaraa.com

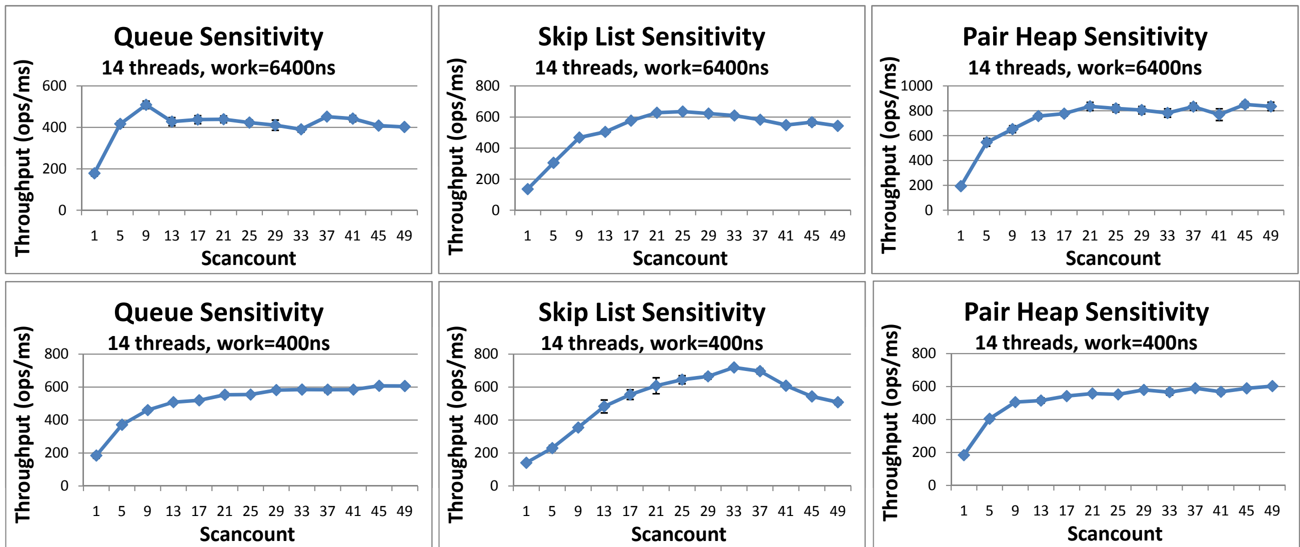


Figure 6: Data Structure Sensitivity to Scancount in Producer-Consumer Application Structures: Throughput vs Scancount Over a Range of Loads. The ideal scancount varies widely and depends on both the load and the data structure.

onds before joining threads and recording the results.[2] For Smart Data Structures, one unit of reward is credited for each operation completed. The benchmark 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.

4.2 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 Scott queue (the MS Queue) [25], the baskets queue of Hoffman et. al [26], and the Flat Combining queue [3]. The best priority queues in the literature are the Skip List based priority queue of Lotan and Shavit [27], the Flat Combining Skip List, and the Flat Combining Pairing Heap [3]. We also compare a lazy lock-based Skip List developed by Herlihy and Shavit [28]. Our benchmark studies how data structure throughput is impacted by two key variables: the number of threads operating on the data structure and load on the data structure. The load is adjusted by varying the amount of post computation between operations: decreasing post computation increases the load. The first mode of our benchmark is used (see Section 4.1).

Figure 5 shows 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. Hendler et al. analyze the sources of the improvement [3]. 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 Section 4.4 that, through machine learning, our Smart Data Structures prototype improves upon the high performance of Flat Combining by an additional factor of up to 1.5x.

4.3 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 and can be further influenced by the particular load the application places on the data structure.

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 Section 4.1 to benchmark the two application

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

المنارة للاستشارات
www.manaraa.com

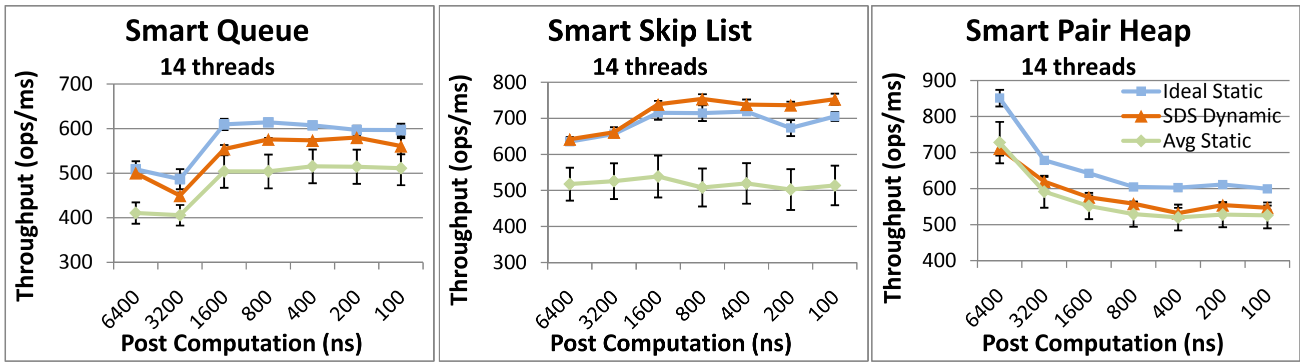


Figure 7: Smart Data Structures Throughput vs Load: A Comparison Against Ideal and Average Static Throughput Bounds. Smart Data Structures improve throughput in most cases, achieving or exceeding the ideal static throughput in many cases.

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 consistently benefit from the highest static scancount assignment (graphs are omitted for space). This is expected since threads have no data interdependency 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. Figure 6 shows throughput for Structure 2 as a function of the static scancount value for different loads. These graphs are characteristic excerpts of our results. We find that throughput can be greatly improved by using optimal scancount values, that the optimal values depend significantly on the load and type of data structure, and that the optimal values are hard to predict. Achieving optimal static settings by hand would be cumbersome, potentially requiring trial and error or special cases in the code which increase complexity and reduce readability. Smart Data Structures provide an automatic approach, and Section 4.4 will show that they are able to find these optima.

4.4 Performance of Smart Data Structures

This study evaluates the performance of Smart Data Structures. It builds on Section 4.3 to derive ideal and average static throughput bounds that, taken together, show the variation in throughput of Smart Data Structures in response to the scancount value. We quantify the performance benefit of learning by comparing the throughput of Smart Data Structures against the bounds. The benchmark is the producer-consumer benchmark described in Section 4.1. The ideal and average static throughput bounds for a given load are computed by varying the scancount, recording the throughput over all scancount values, and taking the maximum and average, respectively.

Figure 7 shows Smart Data Structures throughput over a range of loads. We find that Smart Data Structures improve performance in almost all cases. Further, the Smart Queue and Smart Skip List substantially improve throughput over the average static bound. The Smart Queue achieves near-ideal results, the Smart Skip List exceeds the static ideal throughput, and the Smart Pairing Heap sees moderate improvement. Respectively, they improve throughput over the average static bound by up to 1.2x, 1.5x, and 1.05x.

The improvements of the Smart Pairing Heap are lowest because, of the three data structures, the Pairing Heap provides the least opportunity for improvement via optimizing the scancount. This is evident in Figure 7 by looking at the distance between bounds. In addition, while our design attempts to minimize the overhead of integrating machine learning into the Flat Combining data structures, small overheads do persist which somewhat reduce the throughput achieved. This is evident in Figure 7 by looking at the distance from Smart Queue throughput to the ideal static bound. The Smart Queue finds the optimal scancount but pays a small overhead to do so.

The other interesting result is that the throughput of the Smart Skip List actually exceeds the static ideal bound. Experiments suggest that the additional improvement derives from dynamically tuning the scancount rather than using static values. For data structures like the Flat Combining Skip List, even for a given load, the optimal knob settings 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. Section 4.5 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.

Over all, the results demonstrate that Smart Data Structures are able to learn optimal 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 Section 4.2.

4.5 Adaptivity of Smart Data Structures

This 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. Further, it assesses how well Smart Data Structures are able to adapt dynamically. One common factor that can lead to changing needs in producer-consumer and other application structures is input-dependent behavior. For example, consider a work queue producer-consumer scenario: suppose a graphics application uses a work queue to coordinate the

المنارة للاستشارات
www.manaraa.com

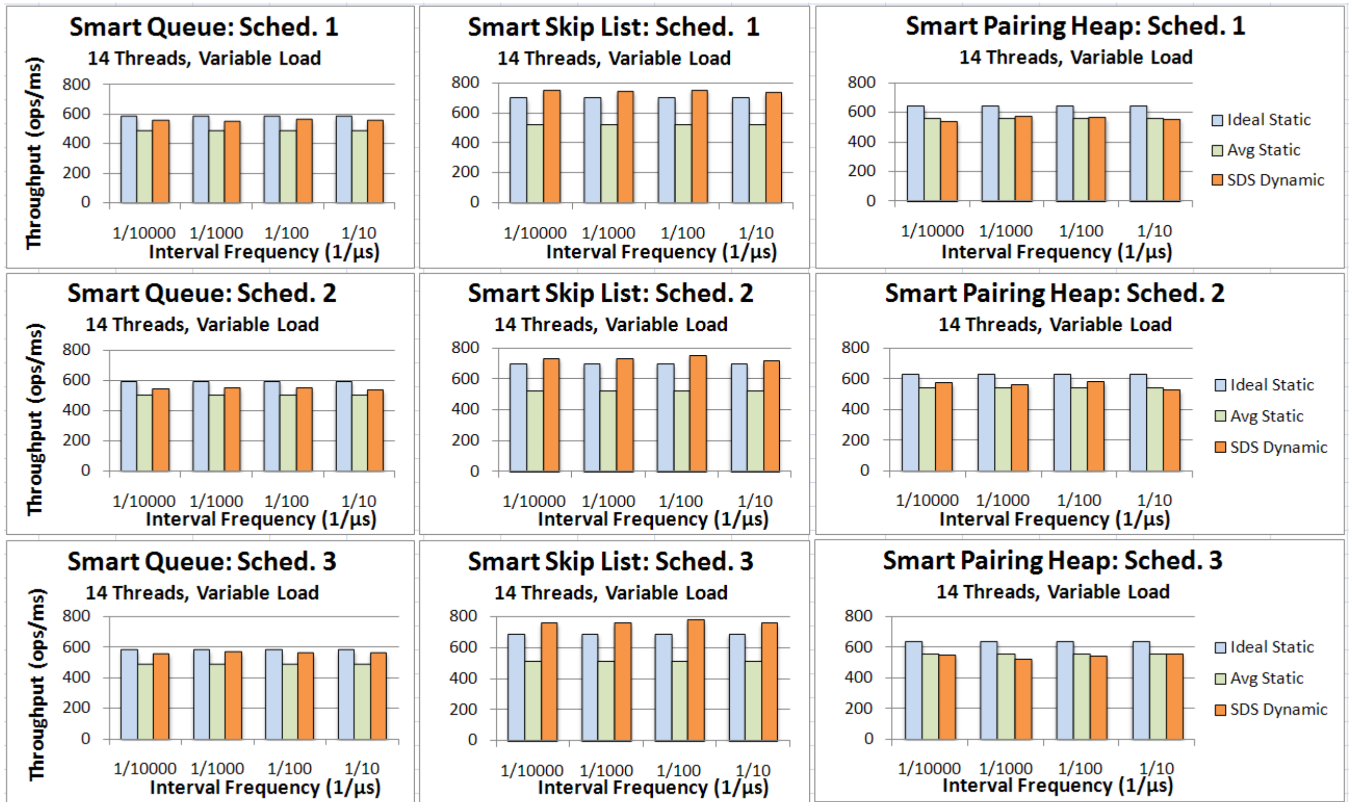


Figure 8: Smart Data Structures Throughput Under Variable Load: A Comparison Against Ideal and Average Static Throughput for Different Variation Frequencies. In many cases, Smart Data Structures achieve near-ideal throughput (or better).

parallel rendering of each input video frame. The scene complexity can vary significantly over the timescale of a just a few frames (as little as 15ms). Scene complexity affects the computational complexity of the work items involved in rendering the scene. The computational complexity of work items determines the rate at which new work is generated and thus the load placed on the work queue data structure.

We have shown in Sections 4.3 and 4.4 that data structure throughput depends critically on optimizing the scancount for different loads and data structures. Using the benchmark described in Section 4.1, this experiment varies the load on Smart Data Structures over a range of different variation frequencies and compares throughput to computed ideal and average bounds. For a given variation frequency, the benchmark is broken into $\frac{1}{f}$ periods, or intervals. In each interval, the benchmark puts a different load on the data structure. We look at three dynamic schedules of loads, selected at random. Each is a cycle of 10 different loads, repeated throughout the 10 second duration of the benchmark. They are: $800 \leadsto 6400 \leadsto 200 \leadsto 3200 \leadsto 1600 \leadsto 100 \leadsto 400 \leadsto 100 \leadsto 400 \leadsto 800$ and $1600 \leadsto 200 \leadsto 400 \leadsto 1600 \leadsto 200 \leadsto 1600 \leadsto 3200 \leadsto 100 \leadsto 200 \leadsto 800$ and $800 \leadsto 100 \leadsto 6400 \leadsto 200 \leadsto 200 \leadsto 100 \leadsto 400 \leadsto 800 \leadsto 3200 \leadsto 400$.

The ideal and average throughputs are computed from Figure 7 by a) looking up the maximum and average throughput achieved for the load in a given interval and b) assuming the scancount can be switched instantaneously when the load changes to achieve that throughput for the whole interval. Figure 8 shows the results. Even under the highest variation frequency of $\frac{1}{10\mu s}$, Smart Data Structures achieve near-ideal throughput in many cases. As expected, throughput does slowly decrease as the interval frequency increases. Currently, the finest period 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 approach the average bound.

5. CONCLUSION

This paper demonstrates a methodology for using machine learning to design self-aware data structures. We introduce a new class of data structures called Smart Data Structures that leverage online learning to optimize themselves automatically and help eliminate the complexity of hand-tuning data structures for different systems, applications, and workloads. We prototype an open source library of Smart Data Structures [1] for common parallel programming needs and demonstrate that it substantially improves upon the best existing data structures by up to 1.5x in our experiments.

The Smart Data Structures design methodology is not a silver bullet for mitigating all of the complexities of multicore programming, but it does provide a foundation for researchers to further investigate possible synergies between programming and the power of adaptation through machine learning. Our view is that machine learning is a robust and high performance framework for making complicated tradeoffs, and that machine learning will play an essential role in the development of future systems.

المنارة للاستشارات
www.manaraa.com

6. REFERENCES

[1] J. Eastep, D. Wingate, and A. Agarwal, "Smart Data Structures Project." [Online]. Available: github.com/mit-carbon/Smart-Data-Structures

[2] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal, "Smartlocks: Lock Acquisition Scheduling for Self-Aware Synchronization," in *ICAC 2010 Proceedings*, June 2010.

[3] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat Combining and the Synchronization-Parallelism Tradeoff," in *SPAA '10: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2010, pp. 355–364.

[4] J. Eastep, D. Wingate, and A. Agarwal, "Appendix: Data Structures Background." [Online]. Available: github.com/mit-carbon/Smart-Data-Structures/wiki/Data-Structures-Background

[5] 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*, 2008, pp. 39–50.

[6] R. Bitirgen, E. Ipek, and J. 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*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 318–329.

[7] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: A Framework for Self-aware Management of Multicore Resources," CSAIL, MIT, Tech. Rep. MIT-CSAIL-TR-2011-016, March 2011.

[8] G. Tesauro, "Online Resource Allocation Using Decompositional Reinforcement Learning," in *Proc. AAAI-05*, 2005, pp. 9–13.

[9] D. Wentzlaff, C. Gruenwald, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. E. Miller, and A. Agarwal, "An Operating System for Multicore and Clouds: Mechanisms and Implementation," in *SoCC*, 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] S. Whiteson and P. Stone, "Adaptive Job Routing and Scheduling," *Engineering Applications of Artificial Intelligence*, vol. 17, pp. 855–869, 2004.

[12] K. E. Coons, B. Robatmili, M. E. Taylor, B. A. Maher, D. Burger, and K. 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 Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2008, pp. 32–42.

[13] D. A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2001, p. 197.

[14] J. Bilmes, K. Asanovic, C.-W. Chin, and J. 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*. New York, NY, USA: ACM, 1997, pp. 340–347.

[15] R. C. J.Dongarra, "Automatically Tuned Linear Algebra Software," Knoxville, TN, USA, Tech. Rep., 1997.

[16] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT." IEEE, 1998, pp. 1381–1384.

[17] M. Olszewski and M. 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*, 2004, pp. 17–23.

[18] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. Amato, and L. Rauchwerger, "STAPL: Standard Template Adaptive Parallel Library," in *SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. New York, NY, USA: ACM, 2010, pp. 1–10.

[19] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," *SIGPLAN Not.*, vol. 44, no. 6, pp. 38–49, 2009.

[20] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal, "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *ICAC 2010 Proceedings*, 2010.

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.

[22] S. Mahadevan, "Average reward reinforcement learning: Foundations, algorithms, and empirical results," *Machine Learning*, vol. 22, pp. 159–196, 1996.

[23] R. J. Williams, "Toward a theory of reinforcement-learning connectionist systems," Northeastern University, Tech. Rep. NU-CCS-88-3, 1988.

[24] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *European Conference on Machine Learning (ECML)*, 2005, pp. 280–291.

[25] M. M. Michael and M. 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*. New York, NY, USA: ACM, 1996, pp. 267–275.

[26] M. Hoffman, O. Shalev, and N. Shavit, "The Baskets Queue," in *OPODIS'07: Proceedings of the 11th International Conference on Principles of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 401–414.

[27] I. Lotan and N. Shavit, "Skiplist-Based Concurrent Priority Queues," in *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2000, p. 263.

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

المنارة للاستشارات

www.manaraa.com