Smartlocks: Lock Acquisition Scheduling for Self-Aware Synchronization

Jonathan Eastep[1], David Wingate[1], Marco D. Santambrogio[1,2], Anant Agarwal[1]

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

[2]Politecnico di Milano
Dipartimento di Elettronica e Informazione
Via Ponzio 34/5, 20133 Milano, Italy
marco.santambrogio@polimi.it

ABSTRACT

As multicore processors become increasingly prevalent, system complexity is skyrocketing. The advent of the asymmetric multicore compounds this – it is no longer practical for an average programmer to balance the system constraints associated with today's multicores and worry about new problems like asymmetric partitioning and thread interference. Adaptive, or self-aware, computing has been proposed as one method to help application and system programmers confront this complexity. These systems take some of the burden off of programmers by monitoring themselves and optimizing or adapting to meet their goals.

This paper introduces a self-aware synchronization library for multicores and asymmetric multicores called Smartlocks. Smartlocks is a spin-lock library that adapts its internal implementation during execution using heuristics and machine learning to optimize toward a user-defined goal, which may relate to performance or problem-specific criteria. Smartlocks builds upon adaptation techniques from prior work like *reactive locks* [1], but introduces a novel form of adaptation that we term *lock acquisition scheduling* designed specifically to address asymmetries in multicores. Lock acquisition scheduling is optimizing which waiter will get the lock next for the best long-term effect when multiple threads (or processes) are spinning for a lock.

This work demonstrates that lock scheduling is important for addressing asymmetries in multicores. We study scenarios where core speeds vary both dynamically and intrinsically under thermal throttling and manufacturing variability, respectively, and we show that Smartlocks significantly outperforms conventional spin-locks and reactive locks. Based on our findings, we provide guidelines for application scenarios where Smartlocks works best versus less optimally.

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, Self-Tuning, Synchronization, Performance Optimization, Asymmetric Multicore, Heterogeneous Multicore

1. INTRODUCTION

As multicore processors become increasingly prevalent, system complexity is skyrocketing. It is no longer practical for an average programmer to balance all of the system constraints and produce an application or system service that performs well on a variety of machines, in a variety of situations. The advent of the asymmetric multicore is making things worse. Addressing the challenges of applying multicore to new domains and environments like cloud computing has proven difficult enough; programmers are not accustomed to reasoning about partitioning and thread interference in the context of performance asymmetry.

One increasingly popular approach to complexity management is the use of *self-aware* hardware and software. Self-aware systems take some of the burden off of programmers by monitoring themselves and adapting to meet their goals. They have been called adaptive, self-tuning, self-optimizing, autonomic, and organic systems, and they have been applied to a broad range of systems including embedded, real-time, desktop, server, and cloud systems.

This paper introduces a self-aware synchronization library for multicores and asymmetric multicores called Smartlocks. Smartlocks is a spin-lock library that adapts its internal implementation during execution using heuristics and machine learning. Adaptations optimize toward user-defined goals programmed using Application Heartbeats [2] or other suitable application monitoring frameworks and may relate to performance or problem-specific criteria.

Smartlocks takes a different approach to adaptation than its predecessor, the *reactive lock* [1]. Reactive locks optimize

ICAC'10, June 7–11, 2010, Washington, DC, USA.
Copyright 2010 ACM 978-1-4503-0074-2/10/06 ...$10.00.

performance by adapting to scale, i.e. adjusting their algorithm to match how much lock contention there is. Smartlocks use this technique but also a novel adaptation – designed specifically for asymmetric multicores – that we call *lock acquisition scheduling*. When threads are contending for a lock, lock acquisition scheduling is granting the lock in the order expected to maximize the long-term benefit.

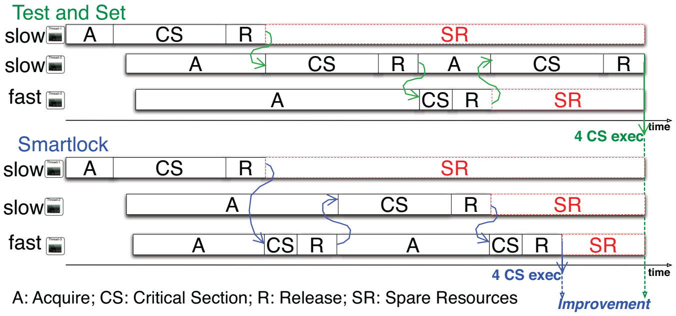

Figure 1: The Impact of Lock Acquisition Scheduling on Asymmetric Multicores. Good scheduling finishes 4 critical sections sooner and creates more spare execution resources.

One reason lock acquisition scheduling is an important performance opportunity is because asymmetries make cycles lost to spinning idly more expensive on faster cores. Figure 1 illustrates. Using spin-locks can be thought of as executing a cycle of three computation phases: acquiring the lock (A), executing the critical section (CS), then releasing the lock (R). The figure shows two scenarios where two slow threads and one fast thread contend for a spin-lock: the top uses a Test and Set lock not optimized for asymmetry; the bottom shows what happens when Smartlocks is used. We assume the memory system limits acquire and release but that the critical section executes faster on the faster core.

In the top scenario, the lock naively picks the slower thread, causing the faster thread to idle in the acquire stage. In the bottom scenario, Smartlocks prioritizes the faster thread, minimizing its spin time. The savings are put to use executing a critical section, and the threads complete 4 total critical sections sooner. That performance improvement can be utilized as a latency improvement when a task requires executing some fixed number of critical sections, or it can be utilized as a throughput improvement since more overall critical sections will get executed. Smartlocks' lock acquisition scheduler also has the advantage that it can (simultaneously or alternatively) optimize the total spare execution cycles, which can be utilized for other computations.

We empirically evaluate Smartlocks on a related scenario in Section 4. We measure throughput for a synthetic benchmark based on a work-pool programming model (without work stealing) running on an asymmetric multicore where core clocks speeds vary due to two thermal throttling events. We compare Smartlocks to conventional locks and reactive locks, and show that Smartlocks significantly outperforms them, achieving near-optimal results. A second experiment evaluates the performance impact of lock acquisition scheduling on applications from the SPLASH-2 benchmark suite [3]. We collect results running on real asymmetric multicore hardware that demonstrate Smartlocks can achieve speedup of 1.2x over other lock strategies. Based on our findings, we provide a set of guidelines for application scenarios where Smartlocks performs best versus less optimally.

The rest of this paper is organized as follows. Section 2 gives background about the historical development of various lock techniques and compares Smartlocks to related works. Section 3 describes the Smartlocks implementation. Section 4 details the benchmarks and experimental setup referenced above. Finally, Section 5 concludes and identifies several additional domains in asymmetric multicore where Smartlocks techniques may be applied.

2. BACKGROUND AND RELATED WORK

In parallel programming, a synchronization object is a mechanism that limits access to a resource to ensure its consistency while many threads (or processes) are using it. Because synchronization objects are important, there are many types of them with different access limits and properties: mutexes, barriers, condition variables, read-write locks, spin-locks, etc. Spin-locks are a type of synchronization object that are particularly efficient for multicore applications. They are commonly used and are the focus of Smartlocks.[1] This section begins with a basic description of spin-lock algorithms followed by the historical development of various algorithms and the challenges they solved. Then, this section compares Smartlocks to its most closely related works.

2.1 The Anatomy of a Lock

Using spin-locks in applications can be thought of as executing a cycle of three computation phases: acquiring the lock, executing the application's critical section, then releasing the lock. Depending on the algorithms used in its acquire and release phases, the spin-lock has three defining properties: its protocol, its wait strategy, and its scheduling policy (summarized in Figure 2).

The protocol is the synchronization mechanism the spin-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 instructions. The wait strategy is the action threads take when they fail to acquire the lock such as spinning or spinning with backoff to reduce polling.[2] Lastly, the scheduling policy determines which waiter should go next when threads are contending for a lock. Most lock algorithms have fixed scheduling policies intrinsic to their protocol mechanism, such as "Free-for-all" and "FIFO." Free-for-all refers to an unpredictable ordering while FIFO refers to a first-come first-serve fair ordering.

The best-known spin-lock algorithms include Test and Set (TAS), TAS with Exponential Backoff (TASEB), Ticket locks, MCS queue locks, and priority locks (PR Locks). Table 1 summarizes their protocol mechanisms and scheduling policies. The next section explains the historical evolution of various lock algorithms and the other table information.

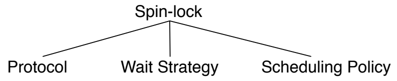

Figure 2: Properties of a Spin-Lock Algorithm

2.2 A Historical Perspective on Spin-Locks

Because spin-locks are an important part of multiprocessor and multicore programming, a wide variety of algorithms

[1] Other "Smart" synchronization objects are planned.
[2] Some non spin-locks use a blocking wait strategy.

Table 1: Summary of Lock Algorithms

Algorithms	Protocol Mechanism	Policy	Scalability	Target Scenario
TAS	Global Flag	Free-for-All	Not Scalable	Low Contention
TASEB	Global Flag	Randomizing Try-Retry	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)
Smartlocks	Adaptive (w/ priority)	Adaptive (arbitrary)	Scalable	Dynamic (w/ asymmetry)

exists. Table 1 summarizes the scalability and target scenario for various spin-lock protocols. Two of the most basic are the Test and Set lock and the Ticket lock. Both have poor scaling performance when many threads or processes are contending for the lock because they poll global lock state and degrade the shared memory system performance [4]. This led to an optimization on Test and Set called Test and Set with Exponential Backoff that limits contention by systematically introducing wait periods between polls. Other efforts to improve performance scalability were based on distributed queues. In these "queue locks," waiters spin instead on local variables [4]. Popular queue locks include the MCS lock, the CLH variant, and a recent one with various improvements called QOLB [5, 6, 7].

One key deficiency of queue locks (that Smartlocks addresses via adaptivity) is poor performance at smaller scales due to the overhead of operations on the distributed queue. Another key deficiency of the above algorithms (especially the queue locks) is that they perform poorly when the number of threads (or processes) exceeds the number of available cores, causing context switches [8]. Problems arise when a running thread spins, waiting for action from another thread that is swapped out. [9] and [8] develop lock strategies to improve such interactions between locks and the kernel scheduler and avoid descheduling threads at inconvenient times. The preemption-safe ticket locks, queue locks, and scheduler-conscious queue locks from these works could be included in Smartlocks' dynamic repertoire of lock strategies – as could future developments in scalable algorithms.

Orthogonal efforts have designed special-purpose locks for some scenarios to improve performance. These are important predecessors to Smartlocks because they are the first hints at the benefits of lock acquisition scheduling. The write-biased readers-writer lock [10] is one example that enables concurrent read access and exclusive write access, prioritizing writers over readers. Priority locks explicitly prioritize lock holders and were developed for database applications where transactions have different importance [11]. They present challenges such as priority inversion, starvation, and deadlock, and are a rich area of research [12]. NUCA-aware locks were developed to improve performance on NUCA memory systems [13]; they release locks preferentially to near neighbors to improve locality. Smartlocks' lock scheduling can emulate these policies (see Section 3.2.3).

The key advantage of Smartlocks over its predecessors is that Smartlocks is a one-size-fits-most machine learning solution that automates the discovery of good policies. Programmers can ignore the complexity of a) identifying good scheduling policies and which special-purpose lock algorithm among dozens they should use or b) programming a priority lock to make it do something useful while avoiding priority inversion, starvation, and deadlock. The next section compares Smartlocks to adaptive lock strategies.

2.3 Adaptive Locks

Various adaptive techniques have been proposed to design synchronization strategies that scale well across different systems, under a variety of dynamic conditions [1, 14, 15]. These are Smartlocks' closest related works. A recent patch for Real-Time Linux modifies kernel support for user-level locks so that locks switch wait strategies from spinning to blocking if locks spin too long. This is an adaptation that Smartlocks could use. Other works are orthogonal compiler-based techniques that build multiple versions of the code using different synchronization algorithms then sample throughout execution, switching to the best code as necessary [15]. Other techniques such as *reactive locks* [1] are library-based like Smartlocks and have the advantage that they can be improved over time, having those improvements reflected in apps that dynamically link against them.

Like Smartlocks, reactive locks perform better than the *scalable* lock algorithms at smaller scales by dynamically adapting their internal protocol to match the contention scale. Table 1 compares reactive locks, Smartlocks, and the other locks. The key difference is that Smartlocks is optimized for asymmetric multicores through a novel adaptation technology we call lock acquisition scheduling. Section 4.1.2 demonstrates empirically that lock acquisition scheduling is an important optimization for asymmetric multicores.

The Smartlocks approach also differs in another important way: while prior work focused on performance optimization, Smartlocks targets broader optimization goals including locality, latency, application-specific criteria, or combinations thereof. Furthermore, whereas existing approaches attempt to infer performance from indirect statistics internal to the lock library (such as the lock contention level), Smartlocks can use monitoring frameworks for end-to-end application- or system-level feedback. Suitable frameworks like Application Heartbeats [2] provide a direct measure of how well adaptations are helping an application meet its goals.[3]

2.4 Machine Learning in Multicore

Recently, researchers have begun to realize that machine learning is a powerful tool for managing the complexity of multicore systems. Several important recent works have used machine learning to build a self-optimizing memory controller [16] and to coordinate management of interacting chip resources such as cache space, off-chip bandwidth, and the power budget [17]. Our insight is that machine learning can be applied to synchronization as well, and our results demonstrate that machine learning significantly improves performance for the benchmarks we study.

[3] See Section 3.1 on annotating goals with Heartbeats.

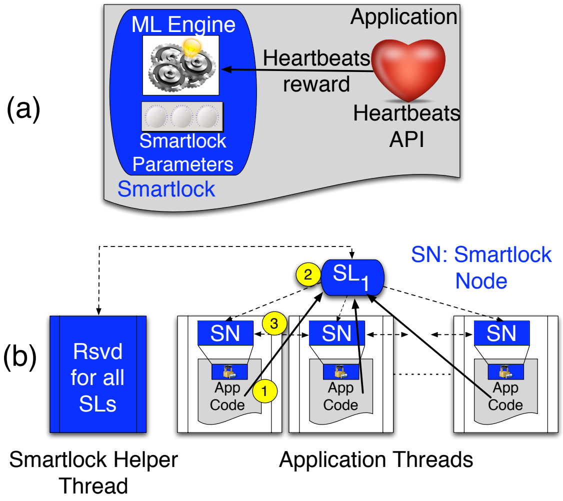

Figure 3: a) Application-Smartlocks Interaction. Smartlock ML engine tunes Smartlock to maximize monitor's reward signal. b) Implementation Architecture. Interface abstracts underlying distributed implementation and helper thread.

3. SMARTLOCKS

Smartlocks is a spin-lock library that adapts its internal implementation during execution using heuristics and machine learning. Smartlocks optimizes toward a user-defined goal (programmed using Application Heartbeats [2] or other suitable application monitoring frameworks) which may relate to performance, problem-specific criteria, or combinations thereof. This section describes the Smartlocks API and how Smartlocks are integrated into applications, followed by an overview of the Smartlocks design and details about each component. We conclude by describing the machine learning engine and its justification.

3.1 Programming Interface

Smartlocks is an adaptive C / C++ library for spin-lock synchronization and resource-sharing. The underlying implementation is parallel and dynamic, but the interfaces abstract the details and are essentially identical to pthreads mutexes. Smartlocks applications are pthreads applications that use pthreads thread spawning etc. but Smartlocks instead of mutexes. The key interface difference is that Smartlocks creation takes a pointer to an application monitoring object that will provide a reward signal for optimization.

One suitable application monitoring framework is the Application Heartbeats framework. Heartbeats is a generic, portable programming interface developed in [2] that applications use to indicate high-level goals and measure their performance or progress toward meeting them. The framework also enables external components such as system software or hardware to query an application's heartbeat performance, also called the *heart rate*. Goals may include throughput, power, output quality, or combinations.

Figure 3 part a) shows the interaction between Smartlocks, the external application monitor (in this case Heartbeats), and the application. The application instantiates a Heartbeat object and one or more Smartlocks. Each Smartlock is connected to the Heartbeat object which provides the reward signal that drives the lock's optimization process. The signal feeds into each lock's machine learning engine and heuristics which tune the lock's protocol, wait strategy, and lock scheduling policy to maximize the reward.

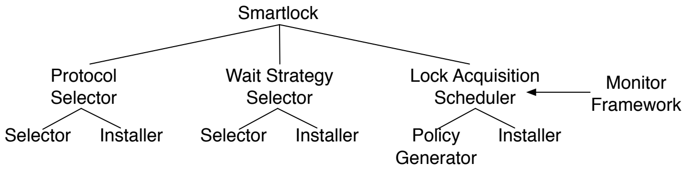

Figure 4: Smartlock Functional Design. A component to optimize each aspect of a lock.

As illustrated in Figure 3 part b), Smartlocks are shared memory objects. All application threads acquire and release a Smartlock by going through a top-level wrapper that abstracts the internal distributed implementation of the Smartlock. Each application thread actually contains an internal Smartlock node that coordinates with other nodes for locking (and waiting and scheduling). Each Smartlock object has adaptation engines as well that run alongside application threads in a separate helper thread. Adaptation engines for each Smartlock get executed in a round-robin fashion. The helper thread may run in a reserved core or share a core with an application thread or another application via SMT technologies (e.g. Intel's hyperthreading). On future many-cores, spare cores are expected to be available.

3.2 Design Overview

Figure 4 shows the components of the Smartlock design: the Protocol Selector, the Wait Strategy Selector, and the Lock Acquisition Scheduler. Each corresponds to one of the three general features of lock algorithms (see Section 2.1) and is responsible for the runtime adaptation of that feature.

3.2.1 *Protocol Selector*

The Protocol Selector is responsible for protocol adaptation within the Smartlock. Supported protocols include {TAS, TASEB, Ticket Lock, MCS, PR Lock}, each with different performance scaling depending on the amount of lock contention. The Protocol Selector identifies what contention scale the Smartlock is experiencing and dynamically matches the best protocol. There are two major challenges to doing this: 1) determining an algorithm for when to switch and what protocol to use and 2) ensuring correctness and good performance during protocol transitions.

As shown in Figure 4, the Protocol Selector has Selector and Installer components to address each problem. The Selector uses the method in [1]: it measures lock contention and compares it against threshold regions empirically derived for the given host architecture and configuration. When contention deviates from one region, the Selector initiates installation of a new protocol. Alternatively, the self-tuning approach in [18] could be used. As illustrated in Figure 5, the Selector executes in the Smartlock helper thread (described previously in Section 3.1). The Installer installs a new protocol using a standard technique called consensus objects [1]. See Section 3.2.4 for details. Protocol transitions have some built-in hysteresis to prevent thrashing.

3.2.2 *Wait Strategy Selector*

The Wait Strategy Selector adapts wait strategies. Supported strategies include {spinning, backoff}, and could be extended to include blocking or hybrids. Designing the Wait Strategy Selector poses two challenges: 1) designing an algo-

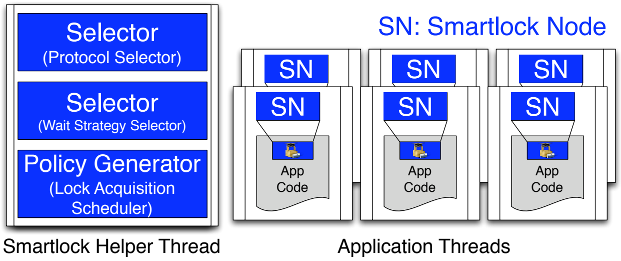

Figure 5: Smartlock Helper Thread Architecture. Each Smartlock's adaptation components run decoupled from application threads in a separate helper thread.

rithm to determine when to switch strategies and 2) ensuring correctness and reasonable performance during transitions. The Wait Strategy Selector has a component to address each of these problems: a Selector and an Installer.

The current implementation of the Selector is trivial since none of the protocols in Smartlocks' repertoire supports dynamic wait strategies. I.e. TASEB is fixed at backoff and MCS and PR Lock are fixed at spinning. Eventually, the Wait Strategy Selector adaptation algorithm will run in the Smartlocks' helper thread as illustrated in Figure 5. Careful handling of transitions in the wait strategy are again achieved through consensus objects (see Section 3.2.4).

3.2.3 Lock Acquisition Scheduler

The Lock Acquisition Scheduler is responsible for adapting the scheduling policy, which determines which thread should acquire a contended lock. As illustrated in Figure 4, the Lock Acquisition Scheduler consists of the Policy Generator and the Installer which use machine learning to optimize the scheduling policy and smoothly coordinate policy transitions, respectively. Like the Protocol Selector and Wait Strategy Selector, the Lock Acquisition Scheduler implementation poses challenges: a) generating timely scheduling decisions and b) generating good scheduling decisions.

To address the timeliness challenge, since locks are typically held for less time than it takes to compute which waiter should go next, the scheduler adopts a decoupled architecture (shown in Figure 5) that does not pick every next lock holder but works at a more relaxed granularity. The scheduler enforces scheduling decisions through priority locks and per-thread priority settings (the scheduling policy) which it updates every few lock acquisitions. Updates happen without blocking any Smartlock acquire and release operations. Smoothly handling decoupled policy updates requires no special efforts in the Installer since a) only one of Smartlocks' protocols (the PR Lock) supports non-fixed policies (see Table 1) and b) our PR Lock implementation allows partial or incremental policy updates (or even modification while a thread is in the wait pool) with no ill effects.

To address the quality challenge of finding good policies, the Policy Generator uses an efficient machine learning engine (described in Section 3.2.5). The use of machine learning is a deliberate design decision that enables Smartlocks to address a wide variety of asymmetries. The authors believe that multicores will become increasingly complex and dynamic and that developing heuristics that are a) general and b) capable of co-optimizing for several interacting and competing goals is too hard. If such heuristics can even be developed, machine learning is a robust, general, and simpler (though perhaps less familiar) alternative. Important recent work in coordinating management of interacting CMP chip resources has come to a similar conclusion [17].

3.2.4 Consensus Objects

Consensus objects is an approach from [1] for ensuring correctness in the face of dynamic protocol changes. Formally, consensus objects require that a) that each protocol has a unique consensus object (logically valid or invalid) and that only one can be valid at a time, b) that a process must able to atomically access the consensus object while completing the protocol even though other processes may attempt to access the consensus object, and c) that the state of the lock protocol can only be modified by the process holding the consensus object [1]. To satisfy these requirements, Smartlocks uses the locks themselves as the consensus objects and executes transitions only in the release operation. The consensus objects are used in conjunction with a mode variable that indicates what protocol should be active. The Protocol Selector sets the mode variable to initiate a change. Next time a lock holder executes a release operation, it releases only the new lock protocol (resetting the state) while not releasing the old lock. Any threads waiting to acquire the old lock will fail and retry the acquire using the new protocol.

3.2.5 Policy Generator Learning Engine

To adaptively prioritize contending threads, Smartlocks use a Reinforcement Learning (RL) [19] algorithm which reads a reward signal from the application monitor and attempts to maximize it. From the RL perspective, this presents a number of challenges: the state space is mostly unobservable, state transitions are semi-Markov due to context switches, and the action space is exponentially large. 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 [20] and use policy gradients to learn a good policy [21].

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 from this policy and executes it. In our case, actions are the priority ordering of the threads (since there are n threads and k priority levels, there are k^n possible actions). Throughout this section, we will denote the distribution over actions as π, and we will denote parameters of the distribution by 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 $\pi(\cdot|\theta)$ is a function of its parameters θ. We define the average reward to be $\eta(\theta) \equiv \mathbb{E}\{\mathbb{R}\} = \lim_{t\to\infty} \frac{1}{t}\sum_{i=1}^{t} r_i$, where $\mathbb{R}$ is a random variable representing reward, and r_i is a particular reward at time i. 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 goal of policy gradients is to estimate the gradient of the average reward of the system with respect to the policy parameters then optimize the policy by moving in the direction of expected improvement. The reward is the signal from the application monitor (e.g. the heart rate), smoothed over a small window of time. Since it is intractable to compute the expectations in $\eta(\theta)$ exactly, policy gradients approximates with importance sampling [21], as follows:

$$\nabla_\theta \eta(\theta) = \nabla_\theta \mathbb{E}\{\mathbb{R}\} \approx \frac{1}{N}\sum_{t=1}^{N} r_t \nabla_\theta \log \pi(a_t|\theta) \qquad (1)$$

where the sequence of rewards r_t is obtained by executing the sequence of actions a_t sampled from $\pi(\cdot|\theta)$. At its core, therefore, our learning algorithm only requires a sequence of rewards r_t and the ability to compute the gradient of the log probability of the action selected at time t. The fact that this algorithm does not depend on a detailed model of the system dynamics is a major virtue of the approach.

So far, we have said nothing about the particular form of the policy. We must address the exponential size of the naive action space and construct a stochastic policy that balances exploration and exploitation, and that can be smoothly parameterized to enable gradient-based learning.

We address all of these issues with a stochastic soft-max policy. We parameterize each thread i with a real valued weight θ_i, and then sample a complete priority ordering over threads. This relaxes an exponentially large discrete action space into a continuous policy space.

To explain our sampling algorithm, let $\mathcal{T} = \{1, \cdots, n\}$ be the set of all thread indices. We will define the random variable t_i^l as an indicator stating that thread i has priority level l; let t^l be the thread id that was assigned priority level l, and let t_i be the assigned priority level for thread i. We first sample the highest-priority thread by sampling from $p(t_i^1) = \exp\{\theta_i\}/\sum_{j\in\mathcal{S}} \exp\{\theta_j\}$. This is a simple multinomial over threads. We then sample the next-highest priority thread by removing the first thread from the pool of available threads and renormalizing the priorities. The distribution over the second thread is then defined to be

$$p(t_i^2|t^1) = \exp\{\theta_i\}/\sum_{k\in\mathcal{T}-\{t^1\}} \exp\{\theta_k\}.$$

We repeat this process $|\mathcal{T}|-1$ times, until we have sampled a complete set of priorities. The overall policy distribution π is therefore:

$$\pi(a_t|\theta) = p(t_i^1)p(t_i^2|t^1)\cdots p(t_i^{|\mathcal{T}|}|t^1, t^2, \cdots, t^{|\mathcal{T}|-1}).$$

The gradient needed in Eq. 1 is easily computed. Let p_{ij} be the probability that thread i was selected to have priority j, with the convention that $p_{ij} = 0$ if the thread has already been selected to have a priority higher than j. Then the gradient for parameter i is simply

$$\nabla_{\theta_i} \log \pi(a_t|\theta) = 1 - \sum_j p_{ij}.$$

When enough samples are collected (or some other gradient convergence test passes), we take a step in the gradient direction: $\theta = \theta + \alpha\nabla_\theta \eta(\theta)$, where α is a step-size parameter.

4. EXPERIMENTAL RESULTS

This section presents two experiments that a) demonstrate that Smartlocks can address a variety of asymmetries in multicores and b) identify various application scenarios where Smartlocks works well and a scenario where it cannot help much. Then, this section presents a set of usage guidelines for Smartlocks based on the experimental findings.

The first experiment evaluates a synthetic benchmark that applies Smartlocks to dynamic asymmetries. It demonstrates the performance and adaptivity of Smartlocks under dynamic and unexpected variation in core frequencies caused by thermal throttling. The second experiment focuses on intrinsic asymmetries in core performance resulting from manufacturing variability. It is a study of what impact a spin-lock's scheduling policy can have on end-to-end application performance on asymmetric multicores; SPLASH-2 and synthetic benchmarks are evaluated.

The results show a) that the lock acquisition scheduling policy can have a significant impact on application performance, b) that Smartlocks is able to learn good policies automatically, and c) that Smartlocks learns good policies fast enough to execute them for the majority of the application and significantly improve overall performance.

4.1 Thermal Throttling Experiment

This experiment applies Smartlocks to dynamic system asymmetries. It evaluates the performance and adaptivity of Smartlocks versus standard lock strategies in a thermal throttling scenario where core clock frequencies vary dynamically and unexpectedly. The section starts with a description of the experimental setup then presents results.

4.1.1 Experimental Setup

The experimental setup emulates an asymmetric multicore with six cores 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 app uses pthreads for thread spawning and Smartlocks within the work-pile data structure. The app is compiled using `gcc` v.4.3.2. The benchmark uses 6 threads: one for the main thread, four for workers, and one reserved for Smartlocks. The main thread generates work while the workers pull work items from the queue and perform the work; each work item requires a constant number of cycles to complete. On the asymmetric 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, the reserved thread runs on a reserved core but it could alternatively share a core with an application thread at the cost of some adaptation latency.

This experiment models an asymmetric 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 asymmetry using a simpler yet powerful software method: adjusting the virtual performance of threads by manipulating the reward signal supplied by the application monitor. The experiment uses Application Heartbeats [2] as the monitor and manipulates the number of heartbeats such that at each point where threads would ordinarily issue 1 beat, they instead issue 2 or 3, depending on whether they are emulating a 2 GHz or 3 GHz core.

The experiment simulates a throttling runtime environment and two thermal-throttling events that change core speeds.[4] No thread migration is assumed. Instead, the vir-

[4] We inject throttling events as opposed to recording natural events so we can determine a priori some illustrative scheduling policies to compare Smartlocks against.

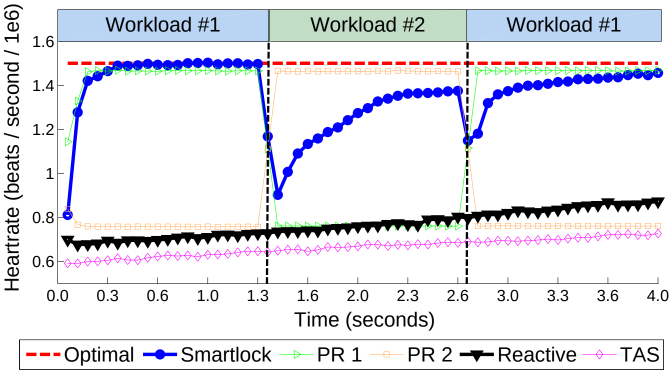

Figure 6: Heartrate performance across thermal throttling events (workload changes). Smartlocks significantly outperforms reactive and TAS spin-locks, achieving near optimal.

tual performance of each thread is adjusted by adjusting heartbeats. The main thread always runs at 3 GHz. At any given time, 1 worker runs at 3 GHz and the others run at 2 GHz. The thermal throttling events change which worker is running fast. The events occur at time 1.4s and 2.7s, the second event reversing the effect of the first.

4.1.2 Results

Figure 6 shows several things. First, it shows the performance of the Smartlock against existing reactive spin-lock techniques. The performance of any reactive lock implementation is upper-bounded by its best-performing internal algorithm at any given time. The best algorithm for this experiment is the write-biased readers-writer lock (described in Section 2.2) so the reactive lock is implemented as that.[5] The graph also compares Smartlocks against a baseline Test and Set spin-lock. The number of cycles required to perform each unit of work has been chosen so that the difference in acquire and release overheads between lock algorithms is not distracting but so that lock contention is high; what *is* important is the policy intrinsic to the lock algorithm (and the adaptivity of the policy in the case of the Smartlocks). As the figure shows, Smartlocks outperforms the reactive lock and the baseline, implying that reactive locks are suboptimal for this and similar benchmark scenarios.

The second thing Figure 6 shows is the gap between reactive lock performance and optimal performance. One lock algorithm / policy that can outperform standard techniques is the priority lock and prioritized access. The graph compares reactive locks against two priority locks / hand-coded priority settings (the curves labeled “PR 1” and “PR 2”). Dividing the graph into three regions surrounding the throttling events, PR 1 is optimal for the first and last region. Its policy sets the main 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). PR 2 is optimal for the middle region of the graph; its policy sets the main thread and worker 3 to a high priority value and all other threads to a low priority value. In each region, a priority lock outperforms the reactive lock, clearly demonstrating the gap between reactive lock performance and optimal performance.

The final thing that Figure 6 illustrates is that Smartlocks approaches optimal performance and adapts to the two thermal throttling events. Within each region of the graph, Smartlocks approaches the performance of the two hand-coded priority lock policies. Performance dips after the throttling events but improves quickly. During the performance dips, Smartlocks’ policy is suboptimal but still better than the reactive lock and Test and Set policies. The adaptation time-scale is on the order of a few hundred milliseconds and is expected to improve as we optimize Smartlocks.

Figure 7 shows the time-evolution of the Smartlock’s internal weights $\wp_i$. Initially, threads all have the same weight, implying equal probability of being selected as high-priority threads. Between time 0 and the first event, Smartlocks learns that the main thread and worker 0 should have higher priority, and uses a policy similar to the hand-coded one. After the first event, the Smartlock learns that the priority of worker 0 should be decreased and the priority of worker 3 increased, similar to the second hand-coded one. After the second event, Smartlock relearns the first workload policy.

4.2 Scheduling Policy Experiment

This experiment applies Smartlocks to intrinsic system asymmetries in core performance: specifically, variation in maximum core frequencies as a result of manufacturing variability. This experiment evaluates what impact a spin-lock’s scheduling policy has on end-to-end application performance on such a system. It benchmarks *radiosity* and *raytrace* from SPLASH-2 and a synthetic benchmark called *queuetp*, replacing key locks within their concurrent data structures with Smartlocks whose policies are varied as part of the experiment. The results show that the lock scheduling policy can significantly impact application performance even on moderately asymmetric multicores like the system we study.[6] The results additionally demonstrate that Smartlocks can learn good policies quickly and significantly improve overall application performance. The next sections detail the experimental setup and present the results.

4.2.1 Experimental Setup

The experiment uses pthreads versions of all applications, replacing key pthreads mutexes with Smartlocks. The lock scheduling policies within the Smartlocks are varied to a) compare the performance of common policies, two custom policies, and Smartlocks’ default adaptive policy, and to b) estimate upper and lower bounds on benchmark execution time. 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 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 Smartlocks’ optimizing helper thread which runs on a spare core as it would in future many-core computers. It could alternatively share a core with an application thread via hyperthreading or other SMT support (see Section 4.3.2). Large inputs are used for all applications. The threads are fixed to particular cores by setting 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 an asymmetry multicore with heterogeneous, fixed

[5] This is the highest performing algorithm for this problem known to the authors to be used in a reactive lock.

[6] We expect larger asymmetries will result in larger effects.

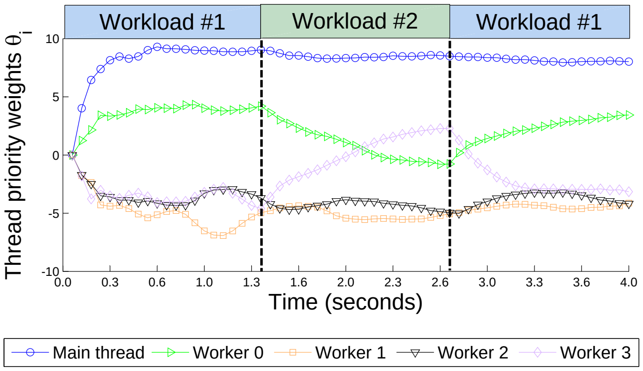

Figure 7: Time evolution of the learned policy. Crossovers between Worker 0 and 3 reflect throttling events.

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 Section 3.1) is integrated into the application to drive the Smartlocks 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 Smartlocks works well: in work queues where Smartlocks can be used as the locking mechanism for work stealing. In this context, varying the lock 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 Smartlock 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 Smartlocks 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 Smartlock. The same custom policies are used. Heartbeats is used, again, with reward credited for each work item completed.

Queuetp.

The *queuetp* synthetic benchmark (designed for this paper) measures the throughput of a 2-stage software pipeline that uses a concurrent pipeline queue built around a Smartlock. It is interesting because it suggests that Smartlocks can be used in self-optimizing data structures for asymmetric multicores to achieve good performance with minimal programming and design effort. The first pipeline stage is 1 producer thread; the second is 5 consumers. The stages are balanced on a homogeneous multicore, but on an asymmetric multicore, the benchmark demonstrates a) that asymmetry-unaware software-pipelines can become imbalanced and b) that Smartlocks is a natural mechanism for addressing such imbalances. The producer runs at 3.17 GHz and all consumers run at 2 GHz. No custom policies are used. The application monitor is a custom monitor that credits reward proportional to how close to balanced (half-full) the queue is. The pipe queue implementation guards all operations with a global Smartlock and is optimized for the moderate parallelism typical of software pipelines.

4.2.2 Results

Figure 8 shows end-to-end execution time of *radiosity*, *raytrace*, and *queuetp* across the different lock scheduling policies. Execution time is normalized against the best performing policy (the lower bound), and the policies are ordered from worst to best in each graph. Together, the upper and lower bound policies capture the variation the application experiences in execution time as a function of the policy.

In *radiosity*, the upper and lower bound 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 Smartlocks performs within 2% of the lower bound, potentially improving performance by 1.2x. In *raytrace*, the custom policies yield the upper and lower bound on execution time again. Smartlocks nearly achieves the lower abound, but *raytrace* does not see much benefit from lock scheduling. In *queuetp*, the upper bound policy is the Random policy, and the lower bound policy is the FIFO policy. For *queuetp*, a substantial 3.3x improvement in application performance is achievable, and Smartlocks gets within about 3% of it.

The results demonstrate that the Smartlocks machine learning approach to learning 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 asymmetry. Section 4.3 further analyzes these benchmarks and the custom policies used in them to provide guidelines for when Smartlocks works best versus less optimally.

4.3 Smartlocks Usage Guidelines

This section defines a set of usage guidelines for Smartlocks based on our findings. We describe a) various use-cases of locks in applications where we have experimentally shown that Smartlocks significantly improves performance and b) some expected limitations of Smartlocks. Then, we study the Smartlocks helper thread architecture, 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 scheduling.

4.3.1 Self-Optimizing Data Structures

The results in Section 4.2.2 suggest that spin-lock scheduling policies may have some unobvious implications for locality and load balancing in multicore applications. They demonstrate a scenario where Smartlocks' adaptive policy

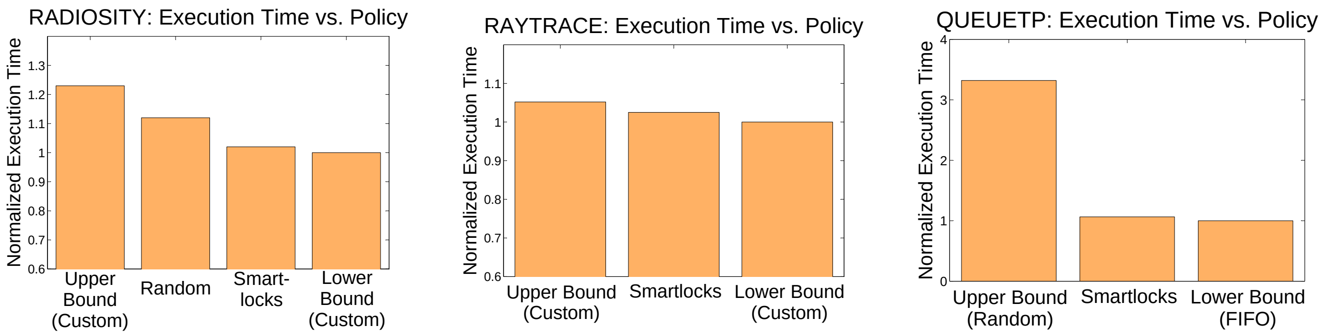

Figure 8: Execution time (lower is better) versus lock scheduling policy. The policy can significantly impact execution time. Smartlocks learns a policy that approaches the lower bound on execution time.

Table 2: Expected Utility of Smartlocks by Scenario

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

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 and the FIFO policy in *queuetp* that improves queue throughput by preventing the unexpectedly fast producer from filling up the queue and starving consumers). Additionally, the *raytrace* results show a scenario where Smartlocks is not able to improve performance much: when lock contention is low. Together, these results help us to understand when Smartlocks works well vs. less optimally.

Table 2 summarizes our findings from Section 4.2.2 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 Smartlocks can significantly improve software pipeline queues and work queues. We expect negligible gains on graph / grid and good gains on memory heaps.

In graph / grid data structures, there are often thousands of nodes, each with a lock. We expect that the current optimization engine architecture within Smartlock may not scale to thousands of Smartlocks if asymmetries change rapidly. The problem is that the optimization engine for each instantiated lock executes in the same shared helper 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. Scalability can be mitigated by spawning multiple helper 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 Smartlocks to make improvements.

4.3.2 Helper Thread Sensitivity Analysis

As explained in Section 3.2, the adaptation components of each instantiated Smartlock run decoupled in a shared helper thread. This section addresses the question of what performance tradeoffs there are for running that helper 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-cores, 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.[7] One way to continue to improve performance is by utilizing spare cores to run optimization threads. The Smartlocks helper thread is one example of this class. Some many-core computers are available today: i.e. the Tilera Tile-Gx(r) 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 Smartlocks helper thread can run in the same core as an application thread and share execution resources. The Smartlocks helper 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 scheduling while hiding the overhead of the helper 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 Smartlocks. On small-scale multicores without SMT, using Smartlocks can improve performance if the performance gains of lock scheduling outweigh the performance tradeoffs of taking a thread away from the application for parallelism; otherwise, Smartlocks should not be used. The exact tradeoff to overcome is different for each application and depends on its scalability.

In Figure 9, we quantify the tradeoff of taking away a core for SPLASH-2 applications. We compare 7 application threads vs. 6 application threads and 1 Smartlock 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.[8]

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

[7] 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.

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

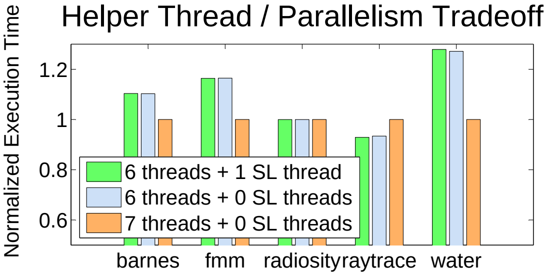

Figure 9: Normalized execution time of SPLASH-2 applications. 6 threads with an additional thread for Smartlocks vs. 6 threads vs. 7 threads. The slowdown reflects by what factor Smartlocks must improve performance for net benefit.

water, respectively.[9] In Section 4.2.2, we demonstrated that lock 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 Smartlocks can yield net improvement for the other applications. Regardless, Smartlocks is expected to do well on a) many-cores and b) multicores with SMT support.

5. CONCLUSION

Smartlocks is a novel self-aware spin-lock library designed to remove some of the complexity of programming multicores and asymmetric multicores. Smartlocks automatically adapts itself at runtime to help applications meet their goals.

This paper introduces a novel adaptation strategy called lock acquisition scheduling for asymmetric multicores and demonstrates empirically that a) Smartlocks can be applied to a variety of asymmetries in multicores, b) that Smartlocks and lock acquisition scheduling can significantly outperform existing lock strategies, and c) that the machine learning engine at the heart of Smartlocks can learn good policies and install them quickly enough to improve end-to-end performance for the applications studied.

In the same way that atomic instructions act as building blocks to construct higher-level synchronization objects, Smartlocks can serve as an *adaptive* building block in many contexts such as operating systems, libraries, system software, DB / webservers, managed runtimes, and programming models. We have demonstrated that Smartlocks can improve performance in the work queue programming model by adapting the work-stealing heuristic. We expect that Smartlocks can also be applied to thread interference and load-balancing issues on multicores by giving applications a share of resources and intelligently managing access. Smartlocks between applications and DRAM ports could learn optimal partitionings of DRAM bandwidth, and Smartlocks guarding disks could learn adaptive read/write policies.

Smartlocks are, of course, not a silver bullet, but they do provide a foundation for researchers to further investigate possible synergies between multicore programming and the power of adaptation through machine learning.

6. REFERENCES

[1] B. H. Lim and A. Agarwal, "Reactive synchronization algorithms for multiprocessors," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, pp. 25–35, 1994.

[2] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal, "Application heartbeats: A generic interface for expressing performance goals and progress in self-tuning systems," SMART Workshop 2010. Online document, http://ctuning.org/dissemination/smart10-02.pdf.

[3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *ISCA 1995 Proceedings*, June 1995, pp. 24–36.

[4] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[5] ——, "Synchronization without contention," *SIGARCH Comput. Archit. News*, vol. 19, no. 2, pp. 269–278, 1991.

[6] P. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," Apr 1994, pp. 165–171.

[7] A. Kägi, D. Burger, and J. R. Goodman, "Efficient synchronization: let them eat qolb," in *ISCA 1997 Proceedings*. New York, NY: ACM, 1997, pp. 170–180.

[8] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott, "Scheduler-conscious synchronization," *ACM Trans. Comput. Syst.*, vol. 15, no. 1, pp. 3–40, 1997.

[9] B. Mukherjee and K. Schwan, "Implementation of scalable blocking locks using an adaptive thread scheduler," in *IPPS 1996 Proceedings*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 339–343.

[10] J. M. Mellor-Crummey and M. L. Scott, "Scalable reader-writer synchronization for shared-memory multiprocessors," in *PPoPP 1991 Proceedings*. New York, NY, USA: ACM, 1991, pp. 106–113.

[11] T. Johnson and K. Harathi, "A prioritized multiprocessor spin lock," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 9, pp. 926–933, 1997.

[12] C.-D. Wang, H. Takada, and K. Sakamura, "Priority inheritance spin locks for multiprocessor real-time systems," *Parallel Architectures, Algorithms, and Networks, International Symposium on*, vol. 0, p. 70, 1996.

[13] Z. Radović and E. Hagersten, "Efficient synchronization for nonuniform communication architectures," in *SC02 Proceedings*. Los Alamitos, CA: IEEE Computer Society Press, 2002, pp. 1–13.

[14] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "Empirical studies of competitve spinning for a shared-memory multiprocessor," in *SOSP 1991 Proceedings*. New York, NY.: ACM, 1991, pp. 41–55.

[15] P. C. Diniz and M. C. Rinard, "Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 89–132, 1999.

[16] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA 2008 Proceedings*, 2008, pp. 39–50.

[17] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO 2008 Proceedings*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 318–329.

[18] P. Hoai Ha, M. Papatriantafilou, and P. Tsigas, "Reactive spin-locks: A self-tuning approach," in *ISPAN 2005 Proceedings*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 33–39.

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

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

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

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