Smartlocks: Self-Aware Synchronization through Lock Acquisition Scheduling

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

[1] Massachusetts Institute of Technology
[2] Politecnico di Milano
{eastep, wingated, santambr, agarwal}@mit.edu
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 an open-source 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, power, or other problem-specific criteria. Smartlocks builds upon adaptation techniques from prior work like *reactive locks* [1], but introduces a novel form of adaptation designed for asymmetric multicores that we term lock acquisition scheduling. 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.

Our results demonstrate empirically that lock scheduling is important for asymmetric multicores and that Smartlocks significantly outperform conventional and reactive locks for asymmetries like dynamic variations in processor clock frequencies caused by thermal throttling events.

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 prevalent approach to complexity management is the use of *self-aware* hardware and software. Self-aware systems take some 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 an open-source[3] 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. Smartlocks optimizes toward a user-defined goal (programmed using Application Heartbeats [2] or other suitable application monitoring frameworks) which may relate to performance, power, problem-specific criteria, or combinations thereof.

Smartlock takes a different approach to adaptation than its closest predecessor, the *reactive lock* [1]. Reactive locks optimize performance by adapting to scale, i.e. selecting their algorithm according to how much lock contention there is. Smartlocks use this technique but also a novel adaptation – designed explicitly for asymmetric multicores – that we term *lock acquisition scheduling*. When threads or processes are spinning, lock acquisition scheduling is the procedure for determining who should get the lock next to maximize long-term benefit.

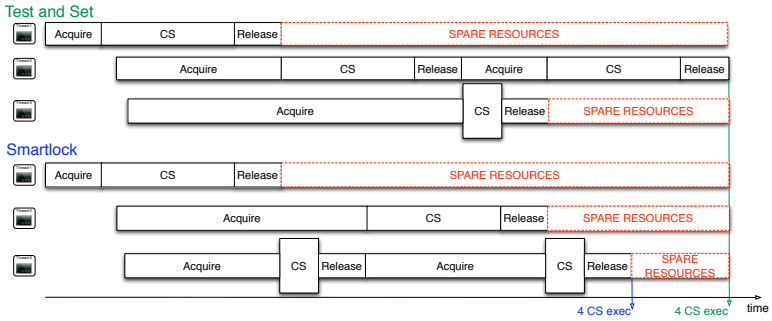

Fig. 1: The Impact of Lock Acquisition Scheduling on Asymmetric Multicores. Good scheduling finishes 4 critical sections sooner and creates more spare execution resources.

One reason lock acquisition scheduling is an important power and performance opportunity is because asymmetries make cycles lost to spinning idly more expensive on faster cores. Figure 1 illustrates the effect. Using spin-locks can be thought of as executing a cycle of three computation phases: acquiring the lock, executing the critical section (CS), then releasing the lock. 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, and the bottom shows what happens when Smartlocks is used. We assume that 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, Smartlock prioritizes the faster thread, minimizing its acquire time. The savings are put to

[3] The authors plan to release Smartlocks in January 2010.

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 the faster thread is able to execute another critical section sooner and more critical sections overall. Smartlock's 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 or for saving power by sleeping cores.

We empirically evaluate Smartlocks on a related scenario in Section 4. We measure throughput for a benchmark based on a simple 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 Smartlock significantly outperforms them, achieving near-optimal results.

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 Smartlock implementation. Section 4 details the benchmarks and experimental setup referenced above. Finally, Section 5 concludes and identifies several additional domains in asymmetric multicore where Smartlock techniques may be applied.

2 Background and Related Work

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 (CS), 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. Typical wait strategies include spinning and backoff.[4] Backoff is a spinning technique that systematically reduces the amount of polling. 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. Typical policies include "Free-for-all" and "FIFO." Free-for-all refers to an unpredictable ordering while FIFO refers to a first-come first-serve fair ordering.

[4] Lock types other than spin-locks may use different wait strategies like blocking.

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

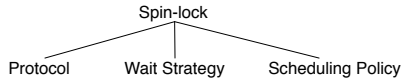

Fig. 2: Properties of a Spin-Lock Algorithm

2.2 A Historical Perspective on Lock Algorithms

Because synchronization is such an important part of parallel programming, a wide variety of algorithms exists. 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. Inefficiency typically results from degrading the performance of the shared memory system as contention increases [3].

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 to the lock variable. Other efforts to improve performance scalability were based on distributed queues. In these "queue locks," waiters spin on local variables to improve performance on cache-coherent shared memory systems [3]. Some popular queue locks include the MCS lock and the CLH variant [4, 5]. QOLB [6] is a recent queue lock with various improvements.

One key deficiency of queue locks is poor performance at smaller scales due to the overhead of operations on the distributed queue. Smartlock gets better performance by dynamically switching to locks with lower overhead at smaller scales. In a sense, research in scalable algorithms compliments the Smartlocks work because the Smartlock can leverage these base algorithms and their benefits in its repertoire of dynamic implementations. Table 1 summarizes the scalability and target scenario for the various lock protocols.

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)
Smartlock	Adaptive (w/ priority)	Adaptive (arbitrary)	Scalable	Dynamic (w/ asymmetry)

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 [7]. Problems arise when a running thread spins, waiting for action from another thread that is swapped out. [7] develops lock strategies to improve such interactions between locks and the OS Scheduler; its preemption-safe ticket locks / queue locks and scheduler-conscious queue locks could be integrated into Smartlock's repertoire.

Other orthogonal efforts to improve lock performance have developed special-purpose locks for some scenarios. The readers-writer lock [8] is one example that enables concurrent read access (with exclusive write access). Priority locks were developed for database applications where transactions have different importance [9]. Priority locks present many challenges, such as priority inversion, starvation, and deadlock, and are a rich area of research [10]. NUCA-aware locks were developed to improve performance on NUCA memory systems [11].

These special-purpose locks are important predecessors to Smartlocks because they are the first hints at the benefits of lock acquisition scheduling. The write-biased readers-writer scheduling policy prioritizes writing over reading. The priority lock explicitly supports prioritization of lock holders. The NUCA-aware lock prefers releasing locks to near neighbors for better locality. Smartlock lock scheduling policies can emulate these policies (see Section 3.5).

The key advantage of Smartlock over these predecessors is that Smartlock 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 the priorities in a priority lock to make it do something useful while avoiding priority inversion, starvation, and deadlock.

The next section compares Smartlocks to other 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, 12, 13]. These are Smartlock's 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 [13]. 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 also perform better than the "scalable" lock algorithms at small and medium scales by dynamically adapting their internal algorithm to match the contention scale. Table 1 compares reactive locks, Smartlocks, and the others. The key difference is that Smartlock is optimized for asymmetric multicores through a novel adaptation technology we call lock acquisition scheduling. In Section 4.2, our results demonstrate empirically that lock acquisition scheduling is an important optimization for asymmetric multicores.

The Smartlock approach differs in another important way: while prior work focused on performance optimization, Smartlock targets broader optimization goals including power, latency, application-defined criteria, and combinations thereof. 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 application-level or system-level feedback. Suitable frameworks like Application Heartbeats [2] provide a more direct measure of how well adaptations are helping an application meet its goals.[5]

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. So far, several important works have used machine learning to build a self-optimizing memory controller [14] and to coordinate management of interacting chip resources such as cache space, off-chip bandwidth, and the power budget [15]. Our insight is that machine learning can be applied to synchronization as well, and our results demonstrate that machine learning achieves near-optimal results for the benchmarks we study.

3 Smartlock

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, power, 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 Smartlock 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++ library for spin-lock synchronization and resource-sharing. The underlying implementation is dynamic, but the API abstracts the details and provides a consistent interface. The API is similar to pthread mutexes. Applications are pthreads-based but link against Smartlocks too to use its adaptive locks instead of mutexes. Applications require only trivial source modifications to use the Smartlocks API.

Table 2: Smartlock API

Function Prototype	Description
`smartlock_helper::smartlock_helper()`	Spawns Smartlock helper thread
`smartlock_helper::~smartlock_helper()`	Joins Smartlock helper thread
`smartlock::smartlock(int max_lockers, monitor* mon_ptr)`	Instantiates a Smartlock
`smartlock::~smartlock()`	Destroys a Smartlock
`void smartlock::acquire(int id)`	Acquires the lock
`void smartlock::release(int id)`	Releases the lock

The API is summarized in Table 2. The first significant difference between the Smartlock and pthread API is that Smartlocks has a function to initialize the library.[6] This function spawns a thread for use by any Smartlocks that get instantiated. The next difference is that Smartlock initialization requires a

[5] See Section 3.1 on annotating goals with the Heartbeats API.

[6] This could be made unnecessary by auto-initializing upon the first Smartlock instantiation.

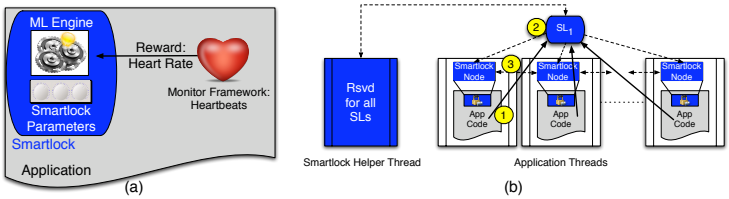

Fig. 3: a) Application-Smartlocks Interaction. Smartlock ML engine tunes Smartlock to maximize monitor's reward signal. b) Smartlock Implementation Architecture. Smartlock interface abstracts underlying distributed implementation and helper thread.

pointer to an application monitoring object that will provide a reward signal to the Smartlock for optimization. The last difference is that Smartlock's acquire and release functions require a unique thread or process id (zero-indexed).[7]

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, latency, output quality, others, and combinations thereof.

Figure 3 part a) shows the interaction between Smartlocks, the external application monitor (in this case the Heartbeats framework), 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.

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 in a *smartlock_helper* object. Adaptation engines for each Smartlock get executed in a round-robin fashion.

3.2 Design Overview

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

[7] This could be made unnecessary through the use of thread local storage or other techniques.

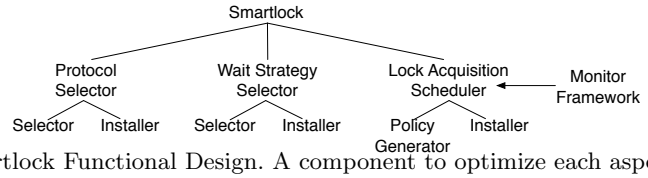

Fig. 4: Smartlock Functional Design. A component to optimize each aspect of a lock.

3.3 Protocol Selector

The Protocol Selector is responsible for protocol adaptation within the Smartlock. Supported protocols include {TAS, TASEB, Ticket Lock, MCS, PR Lock} which each have different performance scaling characteristics. The performance of a given protocol depends upon its implementation and how much contention there is for the lock. The Protocol Selector tries to identify what scale of contention the Smartlock is experiencing and match the best protocol.

There are two major challenges to adapting protocols dynamically. The first is determining an algorithm for when to switch and what protocol to use. The second is ensuring correctness and good performance during protocol transitions.

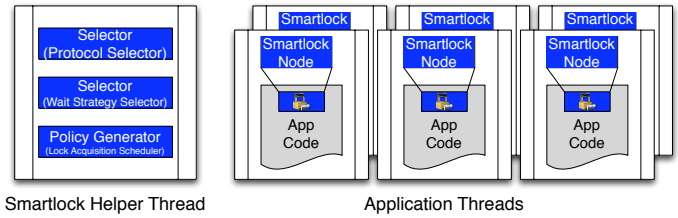

Fig. 5: Smartlock Helper Thread Architecture. Each Smartlock's adaptation components run decoupled from application threads in a separate helper thread.

As previously shown in Figure 4, the Protocol Selector has a component to address each problem: a Selector and an Installer. The Selector heuristically identifies what scale the Smartlock is experiencing and matches the best protocol, similar to 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 the current region, the Selector initiates installation of a new protocol. Alternatively, the self-tuning approach in [16] 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 simple, standard technique called consensus objects [1]. An explanation of consensus objects can be found in Section 3.6. Protocol transitions have some built-in hysteresis to prevent thrashing.

3.4 Wait Strategy Selector

The Wait Strategy Selector adapts wait strategies. Supported strategies include {spinning, backoff}, and could be extended to include blocking or hybrid spinning-blocking. Designing the Wait Strategy Selector poses two challenges. The first is designing an algorithm to determine when to switch strategies. The

second is 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 implementation of the Selector is presently trivial since none of the protocols in Smartlock's current 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 Smartlock's helper thread as illustrated in Figure 5. Careful handling of transitions in the wait strategy are again achieved through consensus objects (see Section 3.6).

3.5 Lock Acquisition Scheduler

The Lock Acquisition Scheduler is responsible for adapting the scheduling policy, which determines who should acquire a contended lock. As illustrated in Figure 4, the Lock Acquisition Scheduler consists of two components: the Policy Generator and the Installer. The Policy Generator uses machine learning to optimize the scheduling policy, and the Installer smoothly coordinates policy transitions.

Like the Protocol Selector and Wait Strategy Selector, designing a scheduler for lock acquisitions presents challenges. The first challenge is generating timely scheduling decisions, and the second is generating good scheduling decisions.

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. Rather, the scheduler works at a coarser granularity and enforces scheduling decisions through prioritization and priority locks. The scheduler updates the policy (the lock holder priority settings) every few lock acquisitions, independent of any Smartlock acquire and release operations driven by the application.

Smoothly handling decoupled policy updates requires no special efforts in the Installer. Transitions between policies are easy because a) only one of Smartlock's protocols (the PR Lock) supports non-fixed policies (see Table 1) and because 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.

The Policy Generator uses machine learning to address the second challenge: coming up with algorithms that yield good scheduling decisions. While the optimizations this paper explores in Section 4 (adapting to dynamic changes in clock frequencies) could be achieved heuristically by reading clock frequencies and ranking processor priorities accordingly, Smartlock's machine learning approach has some advantages. Heuristics are brittle and special-purpose whereas machine learning is a generic optimization strategy; good heuristics can be too difficult to develop when problems get really complex or dynamic; and finally, real-world problems often require co-optimizing for several problems simultaneously, and it is too difficult to come up with effective heuristics for multi-layered problems. Important recent work in coordinating management of interacting CMP chip resources has come to a similar conclusion [15].

3.6 Consensus Objects

Consensus objects is an approach from [1] for ensuring correctness in the face of dynamic protocol changes. The formal requirements for consensus objects are a) that each protocol has a unique consensus object (logically valid or invalid) and 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 (see Section 3.3) 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 then retry the acquire operation using the new protocol.

3.7 Policy Generator Learning Engine

To adaptively prioritize contending threads, Smartlocks use a Reinforcement Learning (RL) [17] 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 exponential. Because we need an algorithm that is a) fast enough for on-line use and b) can tolerate severe partial observability, we adopt an average reward optimality criterion [18] and use policy gradients to learn a good policy [19].

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 policy by π, and we will denote any parameters of the policy by θ. The goal of policy gradients is to estimate the gradient of the average reward of the system with respect to the policy parameters, and 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.

The average reward obtained by executing 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 expectation is approximated with importance sampling, as follows (we omit the full derivation due to space):

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

where the sequence of rewards r_t is obtained by executing the sequence of actions a_t sampled from $\pi(\cdot|\theta)$.

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 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. Let $\mathcal{S}$ be the set of all threads. We first sample the highest-priority thread by sampling from $p(a_1) = \exp\{\theta_{a_1}\}/\sum_{j\in\mathcal{S}} \exp\{\theta_j\}$ where $p(a_1)$ is a simple multinomial distribution over threads. We then sample the next-highest priority thread by removing the first thread and renormalizing the priorities. The distribution over the second thread is then defined to be: $p(a_2|a_1) = \exp\{\theta_{a_2}\}/\sum_{j\in\mathcal{S}-\{a_1\}} \exp\{\theta_j\}$. We repeat this process $|\mathcal{S}| - 1$ times, until we have sampled a complete set of priorities. The overall policy distribution π is therefore:

$$\pi(a_t|\theta) = p(a_1)p(a_2|a_1)\cdots p(a_{|\mathcal{S}|}|a_1,\cdots,a_{|\mathcal{S}|-1}) = \prod_{i\in\mathcal{S}} \frac{\exp\{\theta_{a_i}\}}{\sum_{j\in\mathcal{S}-\{a_1,\cdots,a_{i-1}\}} \exp\{\theta_j\}}.$$

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. Higher-order policy optimization methods, such as stochastic conjugate gradients [20] or Natural Actor Critic [21], would also be possible. Future work could explore these and other agents.

4 Experimental Results

This section demonstrates how Smartlocks helps address asymmetry in multicores by applying Smartlocks to the problem of thermal throttling and dynamic clock speed variations. It describes our experimental setup then presents results.

4.1 Experimental Setup

Our setup emulates an asymmetric multicore with six cores where core frequencies are drawn from the set {3.16 GHz, 2.11 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 Smartlock. 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.

Since asymmetric multicores are not widely available yet, the 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 3.16 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 2.11 GHz or 3.16 GHz core.

The experiment simulates a throttling runtime environment and two thermal-throttling events that change core speeds.[8] No thread migration is assumed. Instead, the virtual performance of each thread is adjusted by adjusting heartbeats. The main thread always runs at 3.16 GHz. At any given time, 1 worker runs at 3.16 GHz and the others run at 2.11 GHz. The thermal throttling events change which worker is running at 3.16 GHz. The first event occurs at time 1.4s. The second occurs at time 2.7s and reverses the first event.

Having threads issue heartbeats with different weights creates the illusion in the reward signal of their having different physical throughput. This method artificially inflates the number of total heartbeats issued by the application but preserves the pertinent signature of the asymmetry: different threads issue different numbers of total heartbeats but in an appropriate ratio. Another point is that this method results in less contention in the work queue than there really would be because faster threads get credit for completing more than one work item but only acquire the lock once to do so. With non-prioritized spin-locks, more contention could make fast cores less likely to get the lock and more likely to idle; thus, we expect this method underestimates the benefit of Smartlock.

4.2 Results

Figure 6 shows several things. First, it shows the performance of the Smartlock against existing reactive lock techniques. Smartlock performance is the curve labeled "Smartlock" and reactive lock performance is labeled "Spin-lock: reactive lock." 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.[9] The graph also compares

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

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

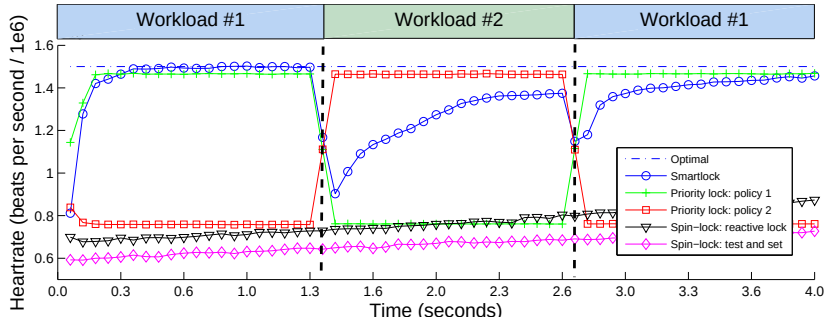

Fig. 6: Performance results on thermal throttling experiment. Smartlocks adapt to different workloads; no single static policy is optimal for all of the different conditions.

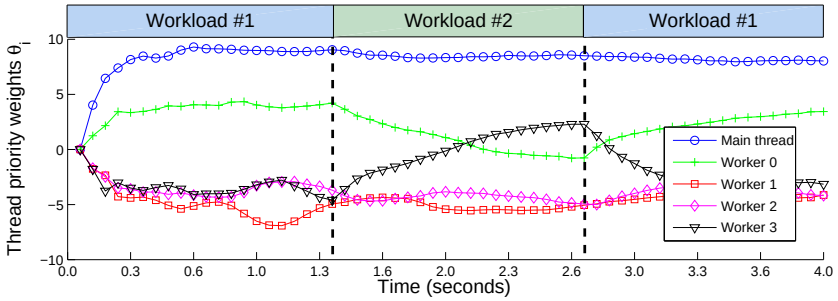

Fig. 7: Time evolution of weights in the lock acquisition scheduling policy.

Smartlock against a baseline Test and Set spin-lock labeled "Spin-lock: test and set" for reference. 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 Smartlock). As the figure shows, Smartlock outperforms the reactive lock and the baseline, implying that reactive locks are sub-optimal for this and similar benchmark scenarios.

The second thing that 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 "Priority lock: policy 1" and "Priority lock: policy 2"). Policy 1 is optimal for two regions of the graph: from the beginning to the first throttling event and from the second throttling event to the end. 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). Policy 2 is optimal for the region of the graph between the two throttling events; 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 Smartlock approaches optimal performance and readily adapts to the two thermal throttling events. Within each region of the graph, Smartlock approaches the performance of the two hand-coded priority lock policies. Performance dips after the throttling events (time=1.4s and time=2.7s) but improves quickly.

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, Smartlock learns that the main thread and worker 0 should have higher priority, and uses a policy very similar to optimal 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, again learning a policy similar to the optimal hand-coded one. After the second event, Smartlock learns a third optimal policy.

5 Conclusion

Smartlocks is a novel open-source synchronization library designed to remove some of the complexity of writing applications for multicores and asymmetric multicores. Smartlocks is a self-aware computing technology that adapts itself at runtime to help applications and system software meet their goals. This paper introduces a novel adaptation strategy ideal for asymmetric multicores that we term lock acquisition scheduling and demonstrates empirically that it can be applied to dynamic frequency variation. The results show that Smartlocks, owing to lock acquisition scheduling, can significantly outperform conventional locks and reactive locks, Smartlock's closest predecessor, on asymmetric multicores.

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, and managed runtimes. Smartlocks can be applied to load-balancing and mitigating thread interference. For example, placing a Smartlock between applications and each DRAM port could learn an optimal partitioning of DRAM bandwidth; Smartlocks guarding disks could learn adaptive read/write policies. In general, Smartlocks could mitigate many interference problems by giving applications a share of each resource and intelligently managing access.

Smartlocks may also be a key component in developing asymmetry-optimized parallel programming models. One strong candidate is a modification of the work-pool with work-stealing programming model that maintains the self scheduling property of work-stealing but optimizes by adapting the work-stealing heuristic; this can be accomplished by replacing the lock around the work-pool data structure and prioritizing access to each thread's work pool.

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.

Bibliography

[1] Lim, B.H., Agarwal, A.: Reactive synchronization algorithms for multiprocessors. SIGOPS Oper. Syst. Rev. **28**(5) (1994) 25–35

[2] Hoffmann, H., Eastep, J., Santambrogio, M., Miller, J., Agarwal, A.: Application heartbeats for software performance and health. Technical Report MIT-CSAIL-TR-2009-035, MIT (Aug 2009)

[3] Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1) (1991) 21–65

[4] Mellor-Crummey, J.M., Scott, M.L.: Synchronization without contention. SIGARCH Comput. Archit. News **19**(2) (1991) 269–278

[5] Magnusson, P., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. (Apr 1994) 165–171

[6] Kägi, A., Burger, D., Goodman, J.R.: Efficient synchronization: let them eat qolb. In: Proceedings of the 24th annual international symposium on Computer architecture, New York, NY, USA, ACM (1997) 170–180

[7] Kontothanassis, L.I., Wisniewski, R.W., Scott, M.L.: Scheduler-conscious synchronization. ACM Trans. Comput. Syst. **15**(1) (1997) 3–40

[8] Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: Proceedings of the 3rd ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (1991) 106–113

[9] Johnson, T., Harathi, K.: A prioritized multiprocessor spin lock. IEEE Trans. Parallel Distrib. Syst. **8**(9) (1997) 926–933

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

[11] Radović, Z., Hagersten, E.: Efficient synchronization for nonuniform communication architectures. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (2002) 1–13

[12] Karlin, A.R., Li, K., Manasse, M.S., Owicki, S.: Empirical studies of competitve spinning for a shared-memory multiprocessor. In: Proceedings of the thirteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM (1991) 41–55

[13] Diniz, P.C., Rinard, M.C.: Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback. ACM Trans. Comput. Syst. **17**(2) (1999) 89–132

[14] Ipek, E., Mutlu, O., Martínez, J.F., Caruana, R.: Self-optimizing memory controllers: A reinforcement learning approach. In: Proc of the 35th International Symposium on Computer Architecture. (2008) 39–50

[15] Bitirgen, R., Ipek, E., Martinez, J.F.: 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) 318–329

[16] Hoai Ha, P., Papatriantafilou, M., Tsigas, P.: Reactive spin-locks: A self-tuning approach. In: Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks, Washington, DC, USA, IEEE Computer Society (2005) 33–39

[17] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA (1998)

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

[19] Williams, R.J.: Toward a theory of reinforcement-learning connectionist systems. Technical Report NU-CCS-88-3, Northeastern University (1988)

[20] Schraudolph, N.N., Graepel, T.: Towards stochastic conjugate gradient methods. In: Proc. 9th Intl. Conf. Neural Information Processing. (2002) 853–856

[21] Peters, J., Vijayakumar, S., Schaal, S.: Natural actor-critic. In: European Conference on Machine Learning (ECML). (2005) 280–291