Smartlocks: Self-Aware Synchronization

JONATHAN EASTEP
DAVID WINGATE
MARCO D. SANTAMBROGIO
ANANT AGARWAL

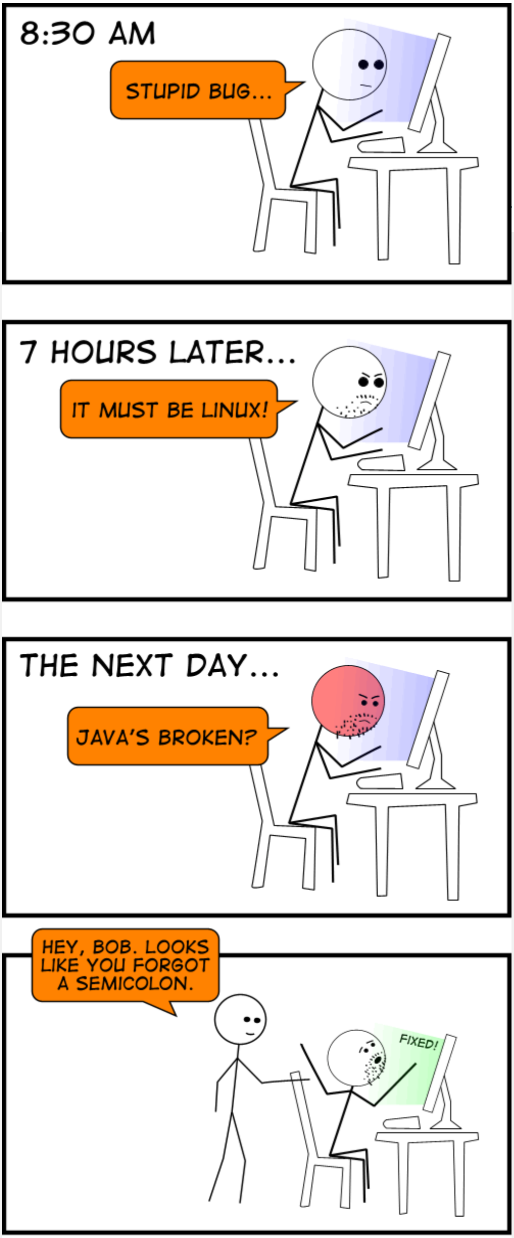
CSAIL

POLITECNICO
DI MILANO

DRESD
www.dresd.org

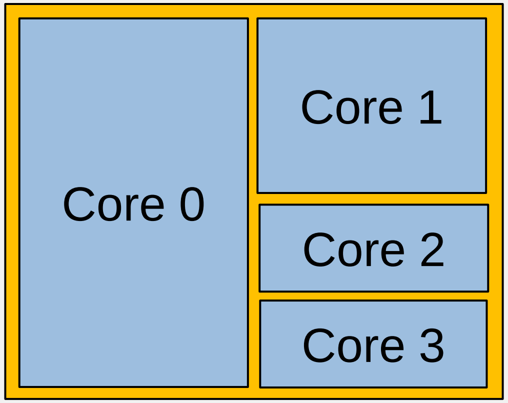
Multicores are Complex

- The good
  - Get performance scaling back on track with Moore’s Law
- The Bad
  - System complexities are skyrocketing
  - Difficult to program multicores and utilize their performance



STUFFTHATHAPPENS.COM BY ERIC BURKE

Asymmetric Multicore is Worse



Asymmetric Multicore

- The Problem
  - Different capabilities, clock speeds = new layer of complexity
  - Programmers aren't used to reasoning about asymmetry
- Why Asymmetric Multicore?
  - Improving Power / Performance
  - Increasing Manufacturing yield

Self-Aware Computing Can Help

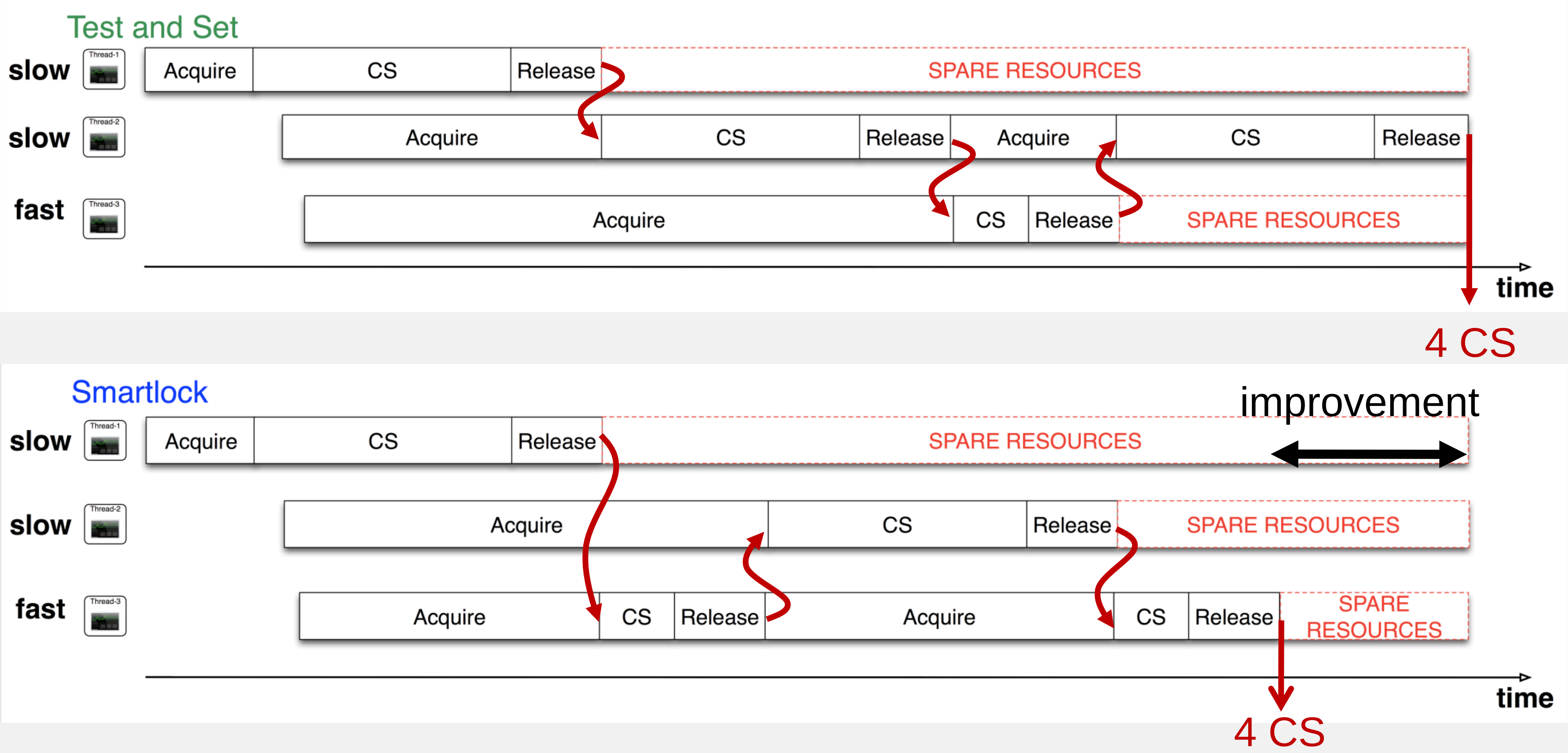
- A promising recent approach to systems complexity management
- Monitor themselves, adapting as necessary to meet their goals
- Self-aware systems
  - Goal-Oriented Computing, Ward et al., CSAIL
  - IBM K42 Operating System (OS w/ online reconfig.)
  - Oracle Automatic Workload Repository (DB tuning)
  - Intel RAS Technologies for Enterprise (hw fault tol.)

Smartlocks Overview

- Self-Aware technology applied to synchronization, resource sharing, programming models
- C/C++ spin-lock library for multicore
- Uses heuristics and machine learning to internally adapt its algorithms / behaviors
- Reward signal provided by application monitor
- Key innovation: Lock Acquisition Scheduling

Lock Acquisition Scheduling is the Key!

- Thought experiment: 2 slow cores, 1 fast



Test and Set

slow Thread-1 Acquire CS Release SPARE RESOURCES

slow Thread-2 Acquire CS Release Acquire CS Release

fast Thread-3 Acquire CS Release SPARE RESOURCES

time

4 CS

Smartlock

improvement

slow Thread-1 Acquire CS Release SPARE RESOURCES

slow Thread-2 Acquire CS Release SPARE RESOURCES

fast Thread-3 Acquire CS Release Acquire CS Release SPARE RESOURCES

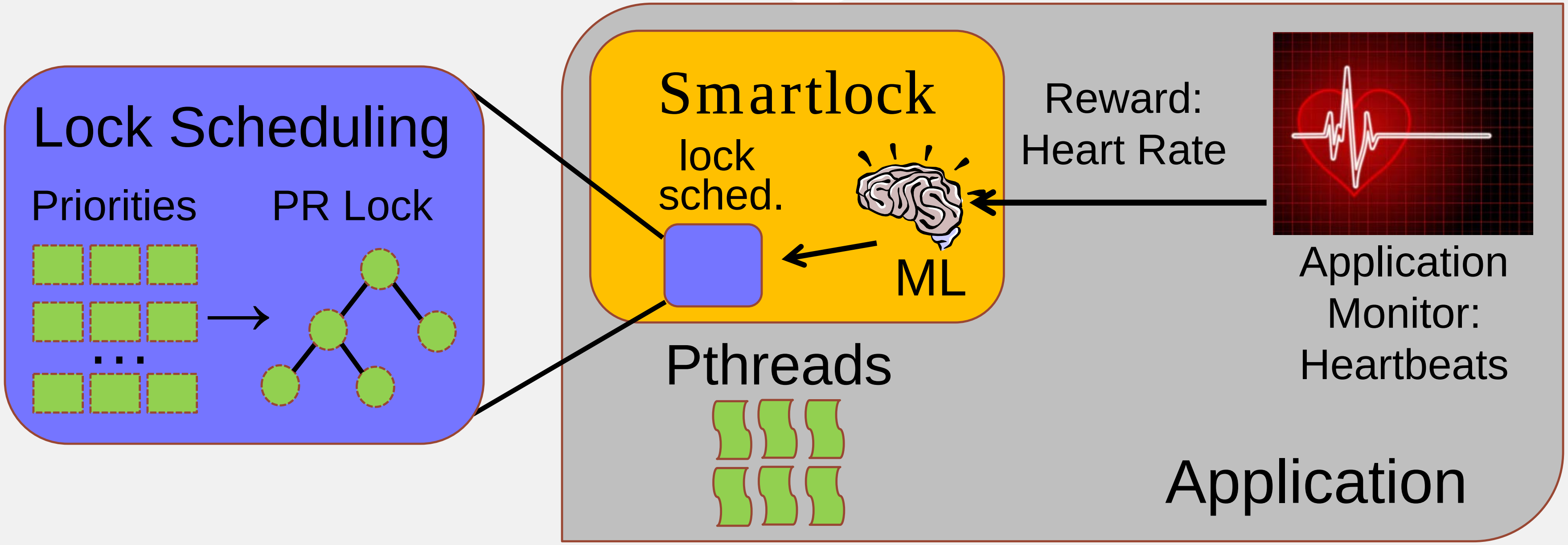
time

4 CS

Talk Outline

- Motivation
- Smartlocks Architecture
- Smartlocks Interface
- Smartlock Design
- Results
- Conclusion

Smartlocks Architecture



Lock Scheduling

Priorities

PR Lock

…

Smartlock

lock sched.

ML

Reward: Heart Rate

Application Monitor: Heartbeats

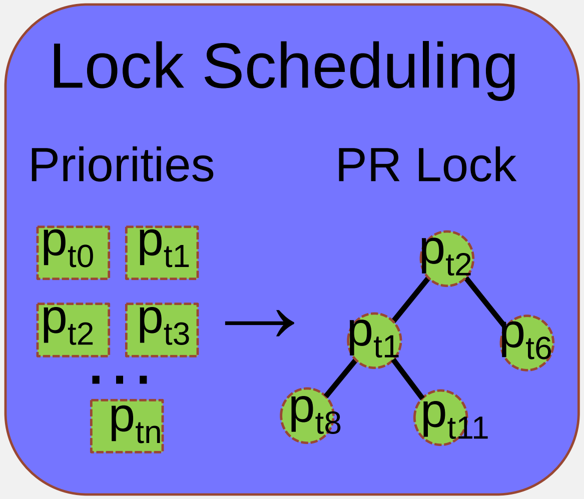
Pthreads

Application

- Each Smartlock self-optimizes as the app runs
  - Take reward from application monitoring framework
  - Reinforcement Learning adapts lock scheduling policy

Do Scheduling with PR Locks

- Priority Lock (PR Lock)
  - Releases lock to waiters preferentially (ordered by priority)
  - Each potential lock holder (thread) has a priority
  - To acquire, thread registers in wait priority queue
  - Usually priority settings are set statically
- Lock Acquisition Scheduling
  - Augments PR Lock w/ ML engine to dynamically control priority settings
  - Scheduling policy = the set of thread priorities



t_i = thread i; p_{ti} = priority t_i

Talk Outline

- Motivation
- Smartlocks Architecture
- Smartlocks Interface
- Smartlock Design
- Results
- Conclusion

Smartlocks Interface

Function Prototype	**Description**
smartlock::smartlock(int max_lockers, monitor *m_ptr)	Creates a Smartlock
Smartlock::~smartlock()	Destroys a Smartlock
void smartlock::acquire(int id)	Acquires the lock
void smartlock::release(int id)	Releases the lock

- Similar to pthread mutexes
- Difference is interface for external monitor
  - Smartlock queries monitor for reward signal

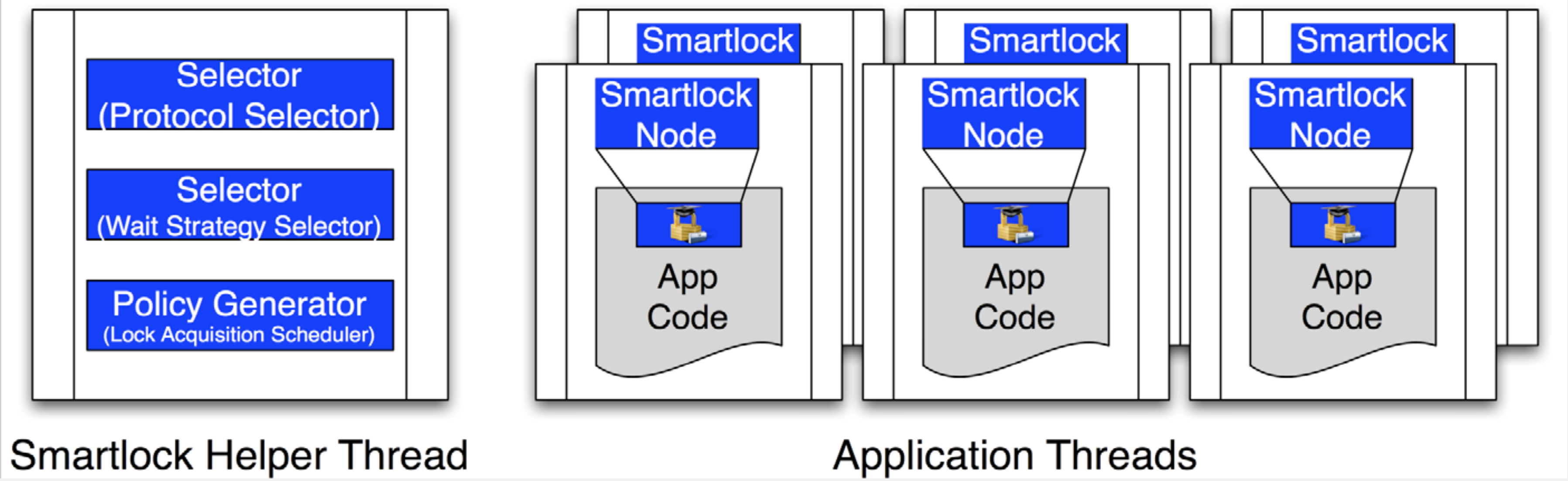
Talk Outline

- Motivation
- Smartlocks Architecture
- Smartlocks Interface
- Smartlock Design
- Results
- Conclusion

Smartlocks Design Challenges

- Major Scheduling Challenges
  - **The Timeliness Challenge**
    - Scheduling too slowly could negate benefit of scheduling
    - Where do you get compute resources to optimize?
  - **The Quality Challenge**
    - Finding policies with best long-term effects
      - No model of system to guide direct optimization methods
    - Efficiently searching an exponential policy space
    - Overcoming stochastic / partially observable dynamics

Meeting The Timeliness Challenge



- Run adaptation algorithms in decoupled helper thread
- Relax scheduling frequency to once every few locks
- For efficiency, use PR locks as scheduling mechanism
- ML engine updates priorities; PR lock runs decoupled

Meeting the Quality Challenge

- Machine Learning, Reinforcement Learning
  - Need not know *how* to accomplish task just *when* you have
  - Good at learning actions that maximize long-term benefit
  - Natural for application engineers to construct reward signal
  - Addresses issues like stochastic / partially observable dynamics
- Policy Gradients
  - Computationally cheap, fast, and straightforward to implement
  - Need no model of the system (we don’t have one!)
- Stochastic Soft-Max Policy
  - Relaxes exponential discrete action space into differentiable one
  - Effective, natural way to balance exploration vs. exploitation

The RL Problem Formulation

- Goal: learn a *policy* π(action | θ)
  - Action= PR lock priority settings (exponential space)
    - k priorities levels, n threads → k^n possible priority settings
  - θ are learned parameters
  - Reward is e.g. heart rate smoothed over small window
  - Thus π is a distribution over thread prioritizations
  - At each timestep, we sample and execute a prioritization
- Optimization objective: average reward η
  - Depends on the policy, which depends on θ

$$\text{maximize } \eta(\theta) = \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \text{reward}_t$$

Use Policy Gradients Approach

- Approach: *policy gradients*
  - Idea: estimate the gradient of average reward η with respect to policy parameters θ
  - Approximate with importance sampling

$$\nabla_\theta \, \eta(\theta) \approx \frac{1}{n} \sum_{i=1}^{n} \text{reward}_i \; \nabla_\theta \log \pi(\text{action}_i | \theta)$$

  - Take a step in the gradient direction

$$\theta = \theta + \alpha \nabla_\theta \, \eta(\theta)$$

Talk Outline

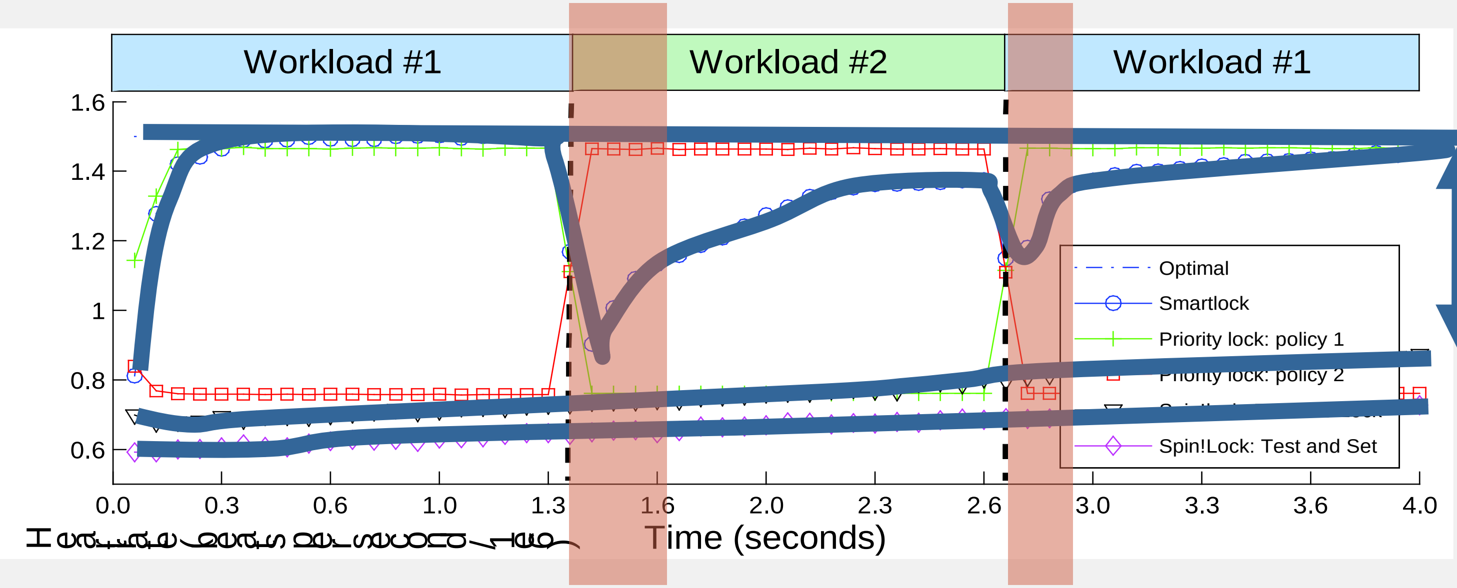
- Motivation
- Smartlocks Architecture
- Smartlocks Interface
- Smartlock Design
- Results
- Conclusion

Experimental Setup 1

- Simulated 6-core single-ISA asymmetric multicore w/ dynamic clock speeds
- Throughput benchmark
  - Work-pile programming model (no stealing)
  - 1 producer, 4 workers
  - Record how long to perform n total work items
  - Fast cores finish work faster; if they spin it’s bad
- Two thermal Throttling Events

Performance as a Function of Time

- Workload 1: Worker 0 @ 3.16GHz, others @ 2.11GHz
- Workload 2: Worker 3 @ 3.16GHz, others @ 2.11GHz
- Workload 3: Same as Workload 1



best w/ pri.
Smartlock

gap

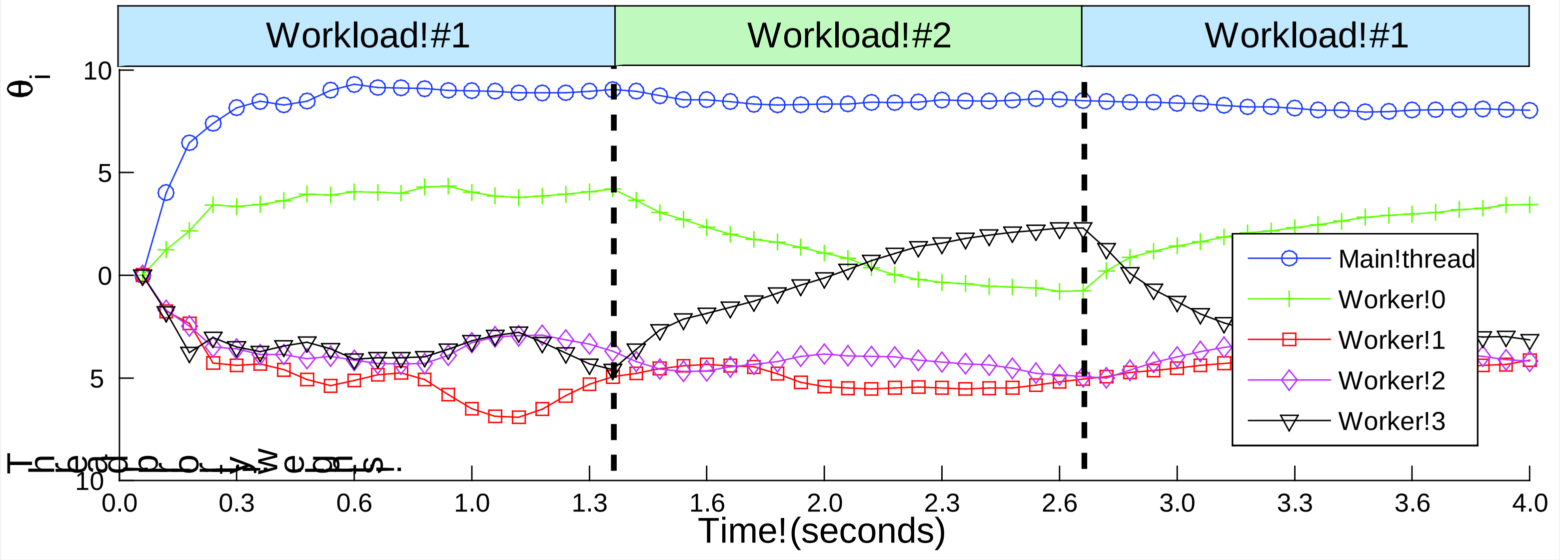
best w/o pri.
TAS

adaptation time-scale

Policy as a Learned Function of Time

- Workload 1: Worker 0 @ 3.16GHz, others @ 2.11GHz
- Workload 2: Worker 3 @ 3.16GHz, others @ 2.11GHz
- Workload 3: Same as Workload 1

Policy as a Learned Function of Time



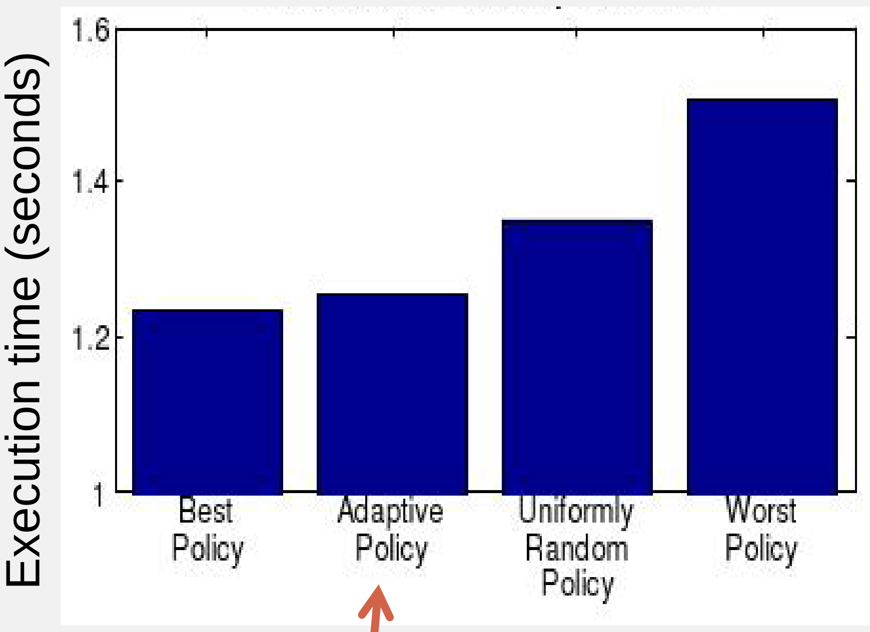
Experimental Setup 2

- Hardware asymmetry using *cpufrequtils*
  - 8-core Intel Xeon machine
  - {2.11,2.11,2.11,2.11,2.11,2.11,3.16,3.16} GHz
  - 1 core reserved for Machine Learning (not required: helper thread could share a core)
- Splash2
  - First results: Radiosity
  - Computes equilibrium dist. of light in scene
  - Parallelism via work queues with stealing
  - Work items imbalanced (function of input scene)
  - Heartbeat for every work item completed

Radiosity Performance vs. Policy

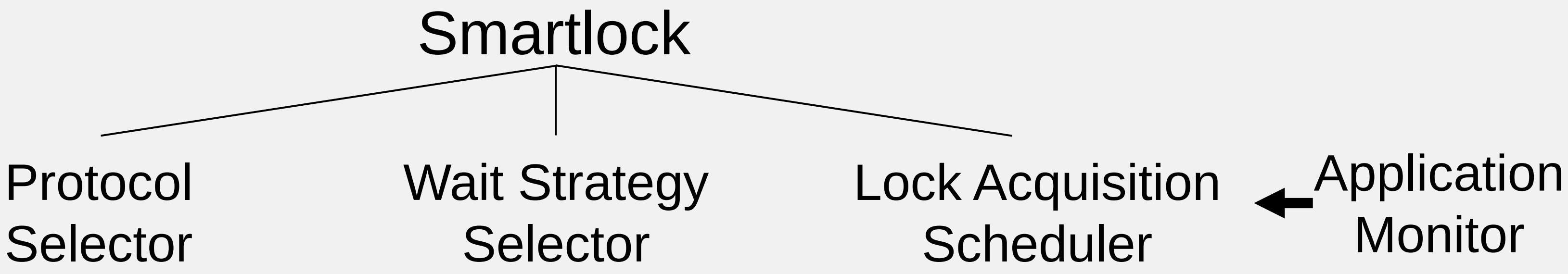
- Benchmark
  - Study how lock scheduling affects performance
  - ~20% difference between best and worst policy
  - TAS (uniformly random) is in the middle
  - Smartlock within 3% of best policy

Radiosity (lower is better)



Smartlocks

Smartlocks is Bigger Than This



Smartlock

Protocol Selector

Wait Strategy Selector

Lock Acquisition Scheduler

Application Monitor

- Smartlock adapts each aspect of a lock
  - Protocol: picks from {TAS,TASEB,Ticket,MCS,PR Lock}
  - Wait Strategy: picks from {spin, spin with backoff}
  - Scheduling Policy: arbitrary, optimized by RL engine
- Smartlocks has an adaptation component for each
- This talk focuses on Lock Acquisition Scheduler

Conclusion

- Smartlocks is a self-aware software library for synchronization / resource-sharing
- Ideal for multicores / applications with dynamic asymmetry
- Lock Acquisition Scheduling is the key innovation
- Smartlocks is open source (COMING SOON!)
  - Code: http://github.com/Smartlocks/Smartlocks
  - Project web-page: https://groups.csail.mit.edu/carbon/smartlocks