Computational Sprinting: Architecture, Dynamics, and Strategies

SEYED MAJID ZAHEDI, SONGCHUN FAN, MATTHEW FAW, ELIJAH COLE,
and BENJAMIN C. LEE, Duke University

Computational sprinting is a class of mechanisms that boost performance but dissipate additional power. We describe a sprinting architecture in which many, independent chip multiprocessors share a power supply and sprints are constrained by the chips' thermal limits and the rack's power limits. Moreover, we present the computational sprinting game, a multi-agent perspective on managing sprints. Strategic agents decide whether to sprint based on application phases and system conditions. The game produces an equilibrium that improves task throughput for data analytics workloads by 4–6× over prior greedy heuristics and performs within 90% of an upper bound on throughput from a globally optimized policy.

CCS Concepts: • **Software and its engineering** → **Power management**; • **Computing methodologies** → *Modeling methodologies*; • **Hardware** → *Temperature control;*

Additional Key Words and Phrases: Modeling, power, energy, thermal management and scheduling, resource management

ACM Reference Format:
Seyed Majid Zahedi, Songchun Fan, Matthew Faw, Elijah Cole, and Benjamin C. Lee. 2017. Computational sprinting: Architecture, dynamics, and strategies. ACM Trans. Comput. Syst. 34, 4, Article 12 (January 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/3014428

1. INTRODUCTION

Modern datacenters oversubscribe their power supplies to enhance performance and efficiency. A conservative datacenter that deploys servers according to their expected power draw will under-utilize provisioned power, operate power supplies at sub-optimal loads, and forgo opportunities for higher performance. In contrast, efficient datacenters deploy more servers than it can power fully and rely on varying computational load across servers to modulate demand for power [Fan et al. 2007]. Such a strategy requires responsive mechanisms for delivering power to the computation that needs it most.

Computational sprinting is a class of mechanisms that supplies additional power for short durations to enhance performance. In chip multiprocessors, for example, sprints activate additional cores and boost their voltage and frequency. Although originally proposed for mobile systems [Raghavan et al. 2012, 2013a], sprinting has found

This work is supported by NSF grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), SHF-1527610, and AF-1408784. This work is also supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these sponsors.
Authors' addresses: S. M. Zahedi, Computer Science Department, Duke University; email: seyedmajid.zahedi@duke.edu; S. Fan (current address), Google, California; email: songchun.fan@duke.edu; M. Faw, E. Cole, and B. C. Lee, Electrical and Computer Engineering, Pratt School of Engineering, Duke University; emails: {matthew.faw, elijah.cole, benjamin.c.lee}@duke.edu.

DOI: http://dx.doi.org/10.1145/3014428

numerous applications in datacenter systems. It can accelerate computation for complex tasks or accommodate transient activity spikes [Zheng and Wang 2015; Skach et al. 2015].

The system architecture determines sprint duration and frequency. Sprinting multiprocessors generate extra heat, absorbed by thermal packages and phase change materials [Raghavan et al. 2012; Skach et al. 2015], and require time to release this heat between sprints. At scale, uncoordinated multiprocessors that sprint simultaneously could overwhelm a rack or cluster's power supply. Uninterruptible power supplies reduce the risk of tripping circuit breakers and triggering power emergencies. But the system requires time to recharge batteries between sprints. Given these physical constraints in chip multiprocessors and the datacenter rack, sprinters require recovery time. Thus, sprinting mechanisms couple performance opportunities with management constraints.

We face fundamental management questions when servers sprint independently but share a power supply: Which processors should sprint and when should they sprint? Each processor's workload derives extra performance from sprinting that depends on its computational phase. Ideally, sprinters would be the processors that benefit most from boosted capability at any given time. Moreover, the number of sprinters would be small enough to avoid power emergencies, which constrain future sprints. Policies that achieve these goals are prerequisites for sprinting to full advantage.

We present the computational sprinting game to manage a collection of sprinters. The sprinting architecture, which defines the sprinting mechanism as well as power and cooling constraints, determines rules of the game. A strategic agent, representing a multiprocessor and its workload, independently decides whether to sprint at the beginning of an epoch. The agent anticipates her action's outcomes, knowing that the chip must cool before sprinting again. Moreover, he or she analyzes system dynamics, accounting for competitors' decisions and risk of power emergencies.

We find the equilibrium in the computational sprinting game, which permits distributed management. In an equilibrium, no agent can benefit by deviating from his or her optimal strategy. The datacenter relies on agents' incentives to decentralize management as each agent self-enforces his or her part of the sprinting policy. Decentralized equilibria allow datacenters to avoid high communication costs and unwieldy enforcement mechanisms in centralized management. Moreover, equilibria outperform prior heuristics. In summary, we present the following contributions:

—**Sprinting Architecture (Section 2).** We present a system of independent sprinters that share power—a rack of chip multiprocessors. Sprinting multiprocessors activate additional cores and increase clock rates. Sprints are constrained by chips' thermal limits and rack power limits.
—**Sprinting Game (Section 3).** We define a repeated game in which strategic agents sprint based on application phases and system conditions. The game divides time into epochs and agents play repeatedly. Actions in the present affect performance and the ability to sprint in the future.
—**Dynamics and Strategies (Section 4).** We design agents who sprint when the expected utility from doing so exceeds a threshold. We devise an algorithm that optimizes each agent's threshold strategy. The strategies produce an equilibrium in which no agent benefits by deviating from her optimal threshold.
—**Performance (Sections 5 and 6).** We evaluate the game for Spark-based datacenter applications, which exhibit diversity in phase behavior and utility from sprinting. The game increases task throughput by 4–6× when compared to prior heuristics in which agents sprint greedily.

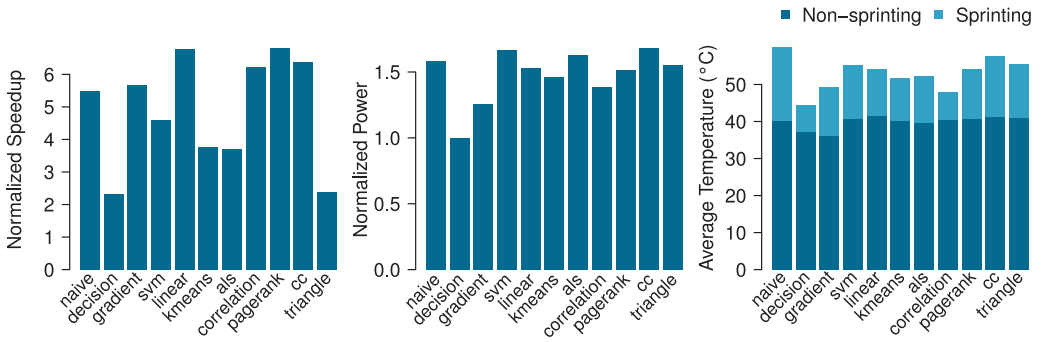

Fig. 1. Normalized speedup, power, and temperature for varied Spark benchmarks when sprinting. Nominal operation supplies three cores at 1.2GHz. Sprint supplies 12 cores at 2.7GHz.

2. THE SPRINTING ARCHITECTURE

We present a sprinting architecture for chip multiprocessors in datacenters. Multiprocessors sprint by activating additional cores and increasing their voltage and frequency. Datacenter applications, with their abundant task parallelism, scale across additional cores as they become available. We focus on applications built atop the Spark framework, which extends Hadoop for memory caching [Zaharia et al. 2010]. In Figure 1, Spark benchmarks perform 2–7× better on a sprinting multiprocessor but dissipate 1.8× the power. Power produces heat.

Sprinters require infrastructure to manage heat and power. First, the chip multiprocessor's thermal package and heat sink must absorb surplus heat during a sprint [Raghavan et al. 2012; Shao et al. 2014]. Second, the datacenter rack must employ batteries to guard against power emergencies caused by a surplus of sprinters on a shared power supply. Third, the system must implement management policies that determine which chips sprint.

2.1. Chip Multiprocessor Support

A chip multiprocessor's maximum power level depends on its thermal package and heat sink. Given conventional heat sinks, thermal constraints are the primary determinant of multiprocessor performance, throttling throughput and overriding constraints from power delivery and off-chip bandwidth [Li et al. 2006]. More expensive heat sinks employ phase change materials (PCMs), which increase thermal capacitance, to absorb and dissipate excess heat [Raghavan et al. 2013a; Shao et al. 2014]. The quality of the thermal package, as measured by its thermal capacitance and conductance, determines parameters of the sprinting game.

The choice of thermal package dictates the maximum duration of a sprint [Shao et al. 2014]. Whereas water, air, and foam enable sprint durations on the order of seconds [Raghavan et al. 2013a], PCMs enable durations on the order of minutes if not hours [Raghavan et al. 2013b; Volle et al. 2010; Shao et al. 2014]. Our sprint architecture employs paraffin wax, which is attractive for its high thermal capacitance and tunable melting point when blended with polyolefins [Raghavan 2013]. We estimate a chip with paraffin wax can sprint with durations on the order of 150s.

After a sprint, the thermal package must release its heat before the chip can sprint again. The average cooling duration, denoted as Δt_{cool}, is the time required before the PCM returns to ambient temperature. The rate at which the PCM dissipates heat depends on its melting point and the thermal resistance between the material and the ambient [Raghavan 2013]. Both factors can be engineered and, with paraffin wax, we estimate a cooling duration on the order of 300s, twice the sprint's duration.

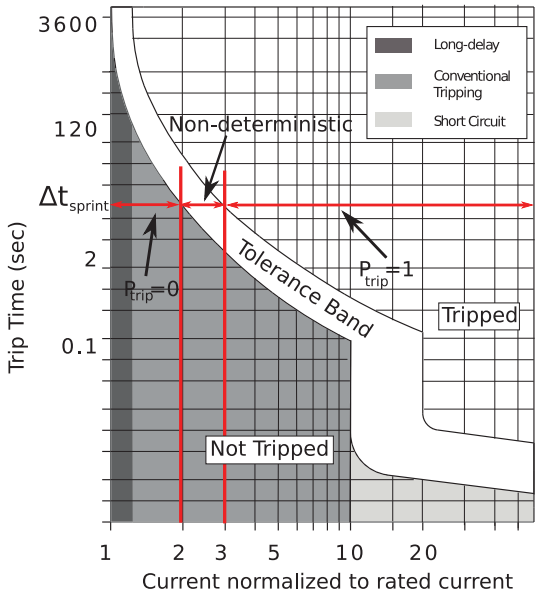

Fig. 2. Typical trip curve of a circuit breaker [Fu et al. 2011].

Different types of workloads may demand different sprint durations. Sprints for online queries requires tens of milliseconds or less [Hsu et al. 2015]. Sprints for parallel workloads requires seconds or more [Raghavan et al. 2013a], and those for warehouse-scale thermal management require support for hours [Skach et al. 2015]. In this article, we study data analytics applications that would prefer to sprint indefinitely. In this setting, the primary determinant of a sprint's duration is the thermal package.

2.2. Datacenter Support

At scale, servers within the same rack share a power supply. Chip multiprocessors draw current from a shared power distribution unit (PDU) that is connected to a branch circuit and protected by a circuit breaker (CB). Datacenter architects deploy servers to oversubscribe branch circuits for efficiency. Oversubscription utilizes a larger fraction of the facility's provisioned power for computation. But it relies on power capping and varied computational load across servers to avoid tripping circuit breakers or violating contracts with utility providers [Fan et al. 2007; Fu et al. 2011]. Although sprints boost computation for complex queries and during peak loads [Hsu et al. 2015; Zheng and Wang 2015], the risk of a power emergency increases with the number of sprinters in a power capped datacenter.

Circuit Breakers and Trip Curves. Figure 2 presents the circuit breaker's trip curve, which specifies how sprint duration and power combine to determine whether the breaker trips. The trip time corresponds to the sprint's duration. Longer sprints increase the probability of tripping the breaker. The current draw corresponds to the number of simultaneous sprints as each sprinter contributes to the load above rated current. Higher currents increase the probability of tripping the breaker. Thus, the tolerance for sprints depends on their duration and power. The breaker dictates the number of sprinters supported by the datacenter rack.

Figure 3 associates the number of sprinters to the tripping probability for a given trip time. Let n_S denote the number of sprinters, and let P_{trip} denote the probability of tripping the breaker. The breaker occupies one of the following regions:

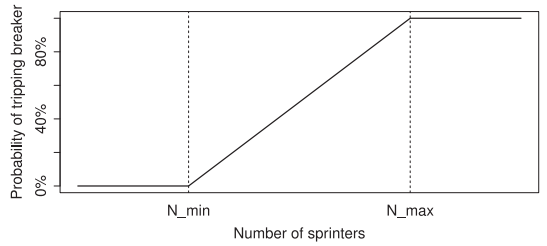

Fig. 3. Probability of tripping the rack's circuit breaker.

—**Non-Tripped.** P_{trip} is zero when $n_S < N_{\min}$
—**Non-Deterministic.** P_{trip} is a non-decreasing function of n_S when $N_{\min} \leq n_S < N_{\max}$
—**Tripped.** P_{trip} is one when $n_S \geq N_{\max}$

Note that $N_{\min}$ and $N_{\max}$ depend on the breaker's trip curve and the application's demand for power when sprinting.

Suppose a sprinter dissipates twice as much power as a non-sprinter, as in Spark applications on chip multiprocessors. We find that the breaker does not trip when less than 25% of the chips sprint and definitely trips when more than 75% of the chips sprint. In other words, $N_{\min} = 0.25N$ and $N_{\max} = 0.75N$. We consider UL489 circuit breakers from Rockwell Automation, which can be overloaded to 125–175% of rated current for a 150s sprint [Zheng and Wang 2015; Wang et al. 2012; Allen-Bradley 2016].

Uninterruptible Power Supplies. When the breaker trips and resets, power distribution switches from the branch circuit to the uninterruptible power supply (UPS) [Govindan et al. 2011, 2012]. The rack augments power delivery with batteries to complete sprints in progress. Lead acid batteries support discharge times of 5–120min, long enough to support the duration of a sprint. After completing sprints and resetting the breaker, servers resume computation on the branch circuit.

However, servers are forbidden from sprinting again until UPS batteries have been recharged. Sprints before recovery would compromise server availability and increase vulnerability to power emergencies. Moreover, frequent discharges without recharges would shorten battery life. The average recovery duration, denoted by $\Delta t_{\text{recover}}$, depends on the UPS discharge depth and recharging time. A battery can be recharged to 85% capacity in 8–10$\times$ the discharge time [Ametek 2016], which corresponds to 8–10$\times$ the sprint duration.

Servers are permitted to sprint again after recharge and recovery. However, if every chip multiprocessor in the rack were to sprint simultaneously and immediately after recovery, they would trigger another power emergency. The rack must stagger the distribution of sprinting permissions to avoid dI/dt problems.

2.3. Power Management

Figure 4 illustrates the management framework for a rack of sprinting chip multiprocessors. The framework supports policies that pursue the performance of sprints while avoiding system instability. Unmanaged and excessive sprints may trip breakers, trigger emergencies, and degrade performance at scale. The framework achieves its objectives with strategic agents and coarse-grained coordination.

Users and Agents. Each user deploys three runtime components: executor, agent, and predictor. Executors provide clean abstractions, encapsulating applications that could employ different software frameworks [Hindman et al. 2011]. The executor supports task-parallel computation by dividing an application into tasks, constructing a task dependence graph, and scheduling tasks dynamically based on available resources.

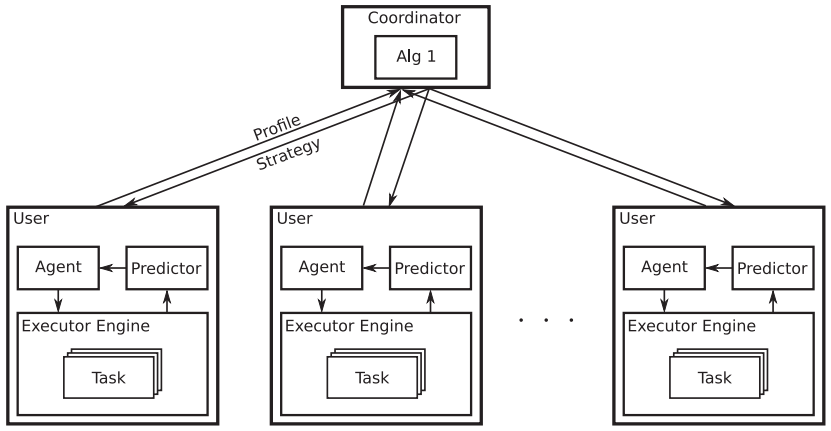

Fig. 4. Users deploy task executors and agents that decide when to sprint. Agents send performance profiles to a coordinator and receives optimized sprinting strategies.

Task scheduling is particularly important as it increases parallelism when sprinting powers-on cores and tolerates faults when cooling and recovery powers-off cores.

Agents are strategic and selfish entities that act on users' behalf. They decide whether to sprint by continuously analyzing fine-grained application phases. Because sprints are followed by cooling and recovery, an agent sprints judiciously and targets application phases that benefit most from extra capability. Agents use predictors that estimate utility from sprinting based on software profiles and hardware counters. Each agent represents a user and her application on a chip multiprocessor.

Coordination. The coordinator collects profiles from all agents and assigns tailored sprinting strategies to each agent. The coordinator interfaces with strategic agents who may attempt to manipulate system outcomes by misreporting profiles or deviating from assigned strategies. Fortunately, our game-theoretic mechanism guards against such behavior.

First, agents will truthfully report their performance profiles. In large systems, game theory provides incentive compatibility, which means that agents cannot improve their utility by misreporting their preferences. The coordinator assigned a tailored strategy to each agent based on system conditions. An agent who misreports his or her profile has little influence on conditions in a large system. Not only does he or she fail to affect others, an agent who misreports suffers degraded performance as the coordinator assigns him or her a poorly suited strategy based on inaccurate profiles.

Second, agents will implement their assigned strategies because the coordinator optimizes those strategies to produce an equilibrium. In equilibrium, every agent implements his or her strategy and no agent benefits when deviating from it. An equilibrium has compelling implications for management overheads. If each agent knows that every other agent is playing his or her assigned strategy, then he or she will do the same without further communication with the coordinator. Global communication between agents and the coordinator is infrequent and occurs only when system profiles change. Local communication between each user's runtime components (i.e., executor, agent, predictor) is frequent but employs inexpensive, inter-process mechanisms. In effect, an equilibrium permits the distributed enforcement of sprinting policies.

In contrast, the centralized enforcement of coordinated policies poses several challenges. First, it requires frequent and global communication as each agent decides whether to sprint by querying the coordinator at the start of each epoch. The length of an epoch is short and corresponds to sprint duration. Moreover, without equilibria,

agents with kernel privileges could ignore prescribed policies, sprint at will, and cause power emergencies that harm all agents. Avoiding such outcomes in a multi-tenant datacenter would require a distributed runtime. The runtime, not the agent, would have kernel privileges for power management, introducing an abstraction layer and overheads.

3. THE SPRINTING GAME

We present a computational sprinting game that governs demands for power and manages system dynamics. We design a dynamic game that divides time into epochs and asks agents to play repeatedly. Agents represent chip multiprocessors that share a power supply. Each agent chooses to sprint independently, pursuing benefits in the current epoch and estimating repercussions in future epochs. Multiple agents can sprint simultaneously, but they risk tripping the circuit breaker and triggering power emergencies that harm global performance.

3.1. Game Formulation

The game considers N agents who run task-parallel applications on N chip multiprocessors. Each agent computes in either normal or sprinting mode. The normal mode uses a fraction of the cores at low frequency, whereas sprints use all cores at high frequency. Sprints rely on the executor to increase task parallelism and exploit extra cores. In this article, for example, we consider 3 cores at 1.2GHz in normal mode and 12 cores at 2.7GHz in a sprint.

The repeated game divides time into epochs. The duration of an epoch corresponds to the duration of a safe sprint, which neither overheats the chip nor trips the circuit breaker. An agent's utility from a sprint varies across epochs according to his or her application's phases. Agents apply a discount factor $\delta < 1$ to future utilities as, all else being equal, they prefer performance sooner rather than later.

3.2. Agent States

At any given time, an agent occupies one of three states—active (A), chip cooling (C), and rack recovery (R)—according to his or her actions and those of others in the rack. An agent's state describes whether he or she can sprint and describes how cooling and recovery impose constraints on his or her actions.

Active (A) – Agent can safely sprint. By default, an agent in an active state operates his or her chip in normal mode, with a few processor cores running at low frequency. The agent has an option to sprint, which deploys additional cores and raises the frequency. He or she decides whether to sprint by comparing a sprint's benefits in the current epoch against benefits from deferring the sprint to a future epoch. If the agent sprints, then his or her state in the next epoch is cooling.

Chip Cooling (C) – Agent cannot sprint. After a sprint, an agent remains in the cooling state until excess heat has been dissipated. Cooling requires a number of epochs Δt_{cool}, which depends on the chip's thermal conductance and resistance, the heat sink and cooling technology, and the ambient temperature. An agent in the cooling state stays in this state with probability p_c and returns to the active state with probability $1 - p_c$. Probability p_c is defined so $1/(1 - p_c) = \Delta t_{\text{cool}}$.

Rack Recovery (R) – Agent cannot sprint. When multiple chips sprint simultaneously, their total current draw may trip the rack's circuit breaker, trigger a power emergency, and require supplemental current from batteries. After an emergency, all agents remain in the recovery state until batteries recharge. Recovery requires a number of epochs $\Delta t_{\text{recover}}$, which depends on the rack's power supply and its battery capacity. Agents in the recovery state stay in this state with probability p_r and return to the active state with probability $1 - p_r$. Probability p_r is defined so $1/(1 - p_r) = \Delta t_{\text{recover}}$.

In summary, the states describe and enforce system constraints. A chip that sprints must cool before sprinting again. A rack that supports sprints with batteries must recharge those batteries before doing so again. Agents in cooling or recovery states are constrained, but those in active states will sprint strategically.

3.3. Agent Actions and Strategies

Agents have two possible actions—sprint or do not sprint. Strategic agents decide between these actions to maximize their utilities. Each agent's sprinting strategy depends on various factors, including

—agent's state and her utility from sprinting,
—agent's history of sprinting,
—other agents' states,
—other agents' utilities, strategies, and histories.

Sprinting strategies determine the game's performance. Agents who greedily sprint at every opportunity produce several sub-optimal outcomes. First, chips and racks would spend many epochs in cooling and recovery states, respectively, degrading system throughput. Moreover, agents who sprint at the first opportunity constrain themselves in future epochs, during which sprints may be even more beneficial.

In contrast, sophisticated strategies improve agent utility and system performance. Strategic agents sprint during the epochs that benefit most from additional cores and higher frequencies. Moreover, they consider other agents' strategies because the probability of triggering a power emergency and entering the recovery state increases with the number of sprinters. We analyze the game's governing dynamics to optimize each agent's strategy and maximize his or her performance.

4. GAME DYNAMICS AND AGENT STRATEGIES

A comprehensive approach to optimizing strategies considers each agent—his or her state, utility, and history—to determine whether sprinting maximizes his or her performance given his or her competitor's strategies and system state. In practice, however, this optimization does not scale to hundreds or thousands of agents.

For tractability, we analyze the population of agents by defining key probability distributions on population behavior. This approach has several dimensions. First, we reason about population dynamics in expectation and consider an "average" agent. Second, we optimize each agent's strategy in response to the population rather than individual competitors. Third, we find an equilibrium in which no agent can perform better by deviating from his or her her optimal strategy.

4.1. Mean-Field Equilibrium

The mean-field equilibrium (MFE), a concept drawn from economic game theory, is an approximation method used when analyzing individual agents in a large system is intractable [Adlakha and Johari 2013; Adlakha et al. 2013, 2010; Iyer et al. 2011], and [Gummadi et al. 2012]. With the MFE, we can characterize expected behavior for a population of agents and then optimize each agent's strategy against that expectation. We can reason about the population and neglect individual agents because any one agent has little impact on overall behavior in a large system.

The mean-field analysis for the sprinting game focuses on the sprint distribution, which characterizes the number of agents who sprint when the rack is not in the recovery state. In equilibrium, the sprint distribution is stationary and does not change across epochs. In any given epoch, some agents complete a sprint and enter the cooling state while others leave the cooling state and begin a sprint. Yet the number of agents who sprint is unchanged in expectation.

The stationary distribution for the number of sprinters translates into stationary distributions for the rack's current draw and the probability of tripping the circuit breaker—see Figure 3. Given the rack's tripping probability, which concisely describes population dynamics, an agent can formulate his or her best response and optimize his or her sprinting strategy to maximize performance.

We find an equilibrium by characterizing a population's statistical distributions, optimizing agents' responses, and simulating game play to update the population. We specify an initial value for the probability of tripping the breaker and iterate as follows.

—**Optimize Sprint Strategy (Section 4.2).** Given the probability of tripping the breaker P_{trip}, each agent optimizes his or her sprinting strategy to maximize his or her performance. He or she sprints if performance gains from doing so exceed some threshold. Optimizing his or her strategy means setting his or her threshold u_T.

—**Characterize Sprint Distribution (Section 4.3).** Given that each agent sprints according to his or her threshold u_T, the game characterizes population behavior. It estimates the expected number of sprinters n_S, calculates their demand for power, and updates the probability of tripping the breaker P'_{trip}.

—**Check for Equilibrium.** The game is in equilibrium if $P'_{\text{trip}} = P_{\text{trip}}$. Otherwise, iterate with the new probability of tripping the breaker.

4.2. Optimizing the Sprint Strategy

An agent considers three factors when optimizing his or her sprinting strategy: the probability of tripping the circuit breaker P_{trip}, his or her utility from sprinting u, and his or her state. An agent occupies the active (A), cooling (C), or recovery (R) state. To maximize expected value and decide whether to sprint, each agent optimizes the following Bellman equation:

$$V(u, A) = \max\{V_S(u, A), V_{\neg S}(u, A)\}. \tag{1}$$

The Bellman equation quantifies value when an agent acts optimally in every epoch. V_S and $V_{\neg S}$ are the expected values from sprinting and not sprinting, respectively. If $V_S(u, A) > V_{\neg S}(u, A)$, then sprinting is optimal. The game solves the Bellman equation and identifies actions that maximize value with dynamic programming.

Value in Active State. Sprinting defines a repeated game in which an agent acts in the current epoch and encounters consequences of that action in future epochs. Accordingly, the Bellman equation is recursive and expresses an action's value in terms of benefits in the current epoch plus the discounted value from future epochs.

Suppose an agent in the active state decides to sprint. Her value from sprinting is her immediate utility u plus her discounted utility from future epochs. When she sprints, her future utility is calculated for the chip cooling state $V(C)$ or calculated for the rack recovery state $V(R)$ when her sprint trips the circuit breaker,

$$V_S(u, A) = u + \delta\left[V(C)(1 - P_{\text{trip}}) + V(R)P_{\text{trip}}\right]. \tag{2}$$

On the other hand, an agent who does not sprint will remain in the active state unless other sprinting agents trip the circuit breaker and trigger a power emergency that requires recovery,

$$V_{\neg S}(u, A) = \delta\left[V(A)(1 - P_{\text{trip}}) + V(R)P_{\text{trip}}\right]. \tag{3}$$

We use $V(A)$ to denote an agent's expected value from being in the active state, which depends on an agent's utility from sprinting. The game profiles an application and its time-varying computational phases to obtain a probability density function $f(u)$, which characterizes how often an agent derives utility u from sprinting. With this density,

the game estimates expected value,

$$V(A) = \int V(u, A) f(u) du. \tag{4}$$

Value in Cooling and Recovery States. An active agent transitions into cooling and recovery states when he or she and/or others sprint. Because agents cannot sprint while cooling or recovering, their expected values from these states do not depend on their utility from sprinting,

$$\begin{aligned} V(C) = {} & \delta \left[V(C) p_c + V(A)(1 - p_c) \right] \left(1 - P_{\text{trip}}\right) \\ & + \delta V(R) P_{\text{trip}}, \end{aligned} \tag{5}$$

$$V(R) = \delta \left[V(R) p_r + V(A)(1 - p_r) \right]. \tag{6}$$

Parameters p_c and p_r are technology-specific probabilities of an agent in cooling and recovery states staying in those states. An agent in cooling will remain in this state with probability p_c and become active with probability $1 - p_c$, assuming the rack avoids a power emergency. If the circuit breaker trips, then an agent enters recovery. An agent remains in recovery with probability p_r and becomes active with probability $1 - p_r$. The game tunes these parameters to reflect the time required for chip cooling after a sprint and for rack recovery after a power emergency—see Section 2.

Threshold Strategy. An agent should sprint if his or her utility from doing so is greater than not. But when is this the case? Equation (8), which follows from Equations (2) and (3), states that an agent should sprint if his or her utility u is greater than his or her optimal threshold for sprinting u_T,

$$V_S(u, A) > V_{\neg S}(u, A), \tag{7}$$

$$u > \underbrace{\delta \left(V(A) - V(C) \right) \left(1 - P_{\text{trip}}\right)}_{u_T}. \tag{8}$$

Thus, an agent uses threshold u_T to test a sprint's utility. If sprinting improves performance by more than the threshold, then an agent should sprint. Applying this strategy in every epoch maximizes expected value across time in the repeated game.

4.3. Characterizing the Sprint Distribution

Given threshold u_T for his or her strategy, an agent uses his or her density function on utility to estimate the probability that he or she sprints, p_s, in a given epoch,

$$p_s = \int_{u_T}^{u_{\max}} f(u) du. \tag{9}$$

The probabilities of sprinting (p_s) and cooling (p_c) define a Markov chain that describes each agent's behavior, see Figure 5, which assumes that the agent is not in recovery. As agents play their strategies, the Markov chain converges to a stationary distribution in which each agent is active with probability p_A. If N agents play the game, then the expected number of sprinters is

$$n_S = p_s \times p_A \times N. \tag{10}$$

As the number of sprinters increases, so does the rack's current draw and the probability of tripping the breaker. Given the expected number of sprinters, the game updates the probability of tripping the breaker according to its trip curve (e.g., Figure 3).

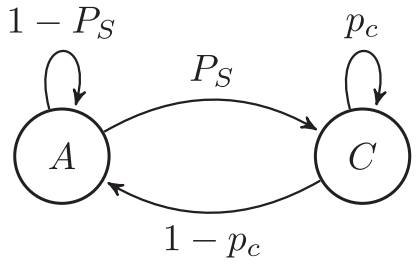

Fig. 5. State transitions when agent sprints, chip cools. Assumes an agent is not in recovery.

ALGORITHM 1: Optimizing the Sprint Strategy

input: Probability density function for sprinting utilities ($f(u)$)
output: Optimal sprinting threshold (u_T)
$j \leftarrow 1$
$P^0_{\text{trip}} \leftarrow 1$
while P^j_{trip} *not converged* **do**
  $u^j_T \leftarrow$ DP solution for Equations (1)–(8) with P^j_{trip}
  $p^j_S \leftarrow$ Equation (9) with $f(u)$, u^j_T
  $n^j_S \leftarrow$ Equation (10) with MC solution and P^j_S
  $P^{j+1}_{\text{trip}} \leftarrow$ Equation (11)
  $j \leftarrow j + 1$
end

Mathematically, the curve is described as follows:

$$P_{\text{trip}} = \begin{cases} 0 & \text{if } n_S < N_{\min} \\ \frac{n_S - N_{\min}}{N_{\max} - N_{\min}} & \text{if } N_{\min} \le n_S \le N_{\max}, \\ 1 & \text{if } n_S > N_{\max} \end{cases} \tag{11}$$

P_{trip} determines n_S, which determines P'_{trip}. If $P_{\text{trip}} = P'_{\text{trip}}$, then agents are playing optimized strategies that produce an equilibrium.

4.4. Finding the Equilibrium

When the game begins, agents make initial assumptions about population behavior and the probability of tripping the breaker. Agents optimize their strategies in response to population behavior. Strategies produce sprints that affect the probability of tripping the breaker. Over time, population behavior and agent strategies converge to a stationary distribution, which is consistent across epochs. The game is in equilibrium if the following conditions hold.

—Given tripping probability P_{trip}, the sprinting strategy dictated by threshold u_T is optimal and solves the Bellman equation in Equations (1)–(3).
—Given sprinting strategy u_T, the probability of tripping the circuit breaker is P_{trip} and is calculated by Equations (9)–(11).

In equilibrium, every agent plays his or her optimal strategy and no agent benefits when deviating from his or her strategy. In practice, the coordinator in the management framework finds and maintains an equilibrium with a mix of offline and online analysis.

Offline Analysis. Agents sample epochs and measure utility from sprinting to produce a density function $f(u)$, which characterizes how often an agent sees utility u from

sprinting. The coordinator collects agents' density functions, analyzes population dynamics, and tailors sprinting strategies for each agent. Finally, the coordinator assigns optimized strategies to support online sprinting decisions.

Algorithm 1 describes the coordinator's offline analysis. It initializes the probability of tripping the breaker. Then it iteratively analyzes population dynamics to find an equilibrium. Each iteration proceeds in three steps. First, the coordinator optimizes sprinting threshold u_T by solving the dynamic program defined in Equations (1)–(8). Second, it estimates the number of sprinters according to Equation (10). Finally, it updates the probability of tripping the breaker according to Equation (11). The algorithm terminates when thresholds, number of sprinters, and tripping probability are stationary.

The offline algorithm has no performance overhead. The analysis runs periodically to update sprinting strategies and the tripping probability as application mix and system conditions evolve. It does not affect an application's critical path as agents use updated strategies when they become available but need not wait for them.

The algorithm requires little computation. It solves the dynamic program with value-iteration, which has a convergence rate that depends on the discount factor δ. The number of iterations grows polynomially in $(1-\delta)^{-1}$. We implement and run the algorithm on an Intel Core i5 processor with 4GB of memory. The algorithm completes in less than 10s, on average.

Online Strategy. An agent decides whether to sprint at the start of each epoch by estimating a sprint's utility and comparing it against his or her threshold. Estimation could be implemented in several ways. An agent could use the first few seconds of an epoch to profile his or her normal and sprinting performance. Alternatively, an agent could use heuristics to estimate utility from additional cores and higher clock rates. For example, task queue occupancy and cache misses are associated with a sprint's impact on task parallelism and instruction throughput, respectively. Comparisons with a threshold are trivial. If an agent decides to sprint, it turns on otherwise disabled cores using CPU-hotplug and increases clock rates using ACPI [Mwaikambo et al. 2004].

5. EXPERIMENTAL METHODOLOGY

Servers and Sprints. The agent and its application are pinned to a chip multiprocessor, an Intel Xeon E5-2697 v2 that can run at 2.70GHz. Two multiprocessors share 128GB of main memory within a server. An agent runs in normal or sprinting mode. In normal mode, the agent uses 3 1.2GHz cores. In sprinting mode, the agent uses 12 2.7GHz cores. We turn cores on and off with Linux `sysfs`. In principle, sprinting represents any mechanism that performs better but consumes more power.

Workloads. We evaluate Apache Spark workloads [Zaharia et al. 2010]. The Spark runtime engine dynamically schedules tasks to use available cores and maximize parallelism, adapting as sprints cause the number of available cores to vary across epochs. Each agent runs a Spark application on representative datasets as shown in Table I.

Profiling Methods. We collect system profiles that measure power and temperature, using the Intel Performance Counter Monitor 2.8 to read MSR registers once every second. We collect workload profiles by modifying Spark (v1.3.1) to log the IDs of jobs, stages, and tasks on their completion.

We measure application performance in terms of the number of tasks completed per second (TPS). Each application defines a number of jobs, and each job is divided into tasks that compute in parallel. Jobs are completed in sequence while tasks can be completed out of order. The total number of tasks in a job is constant and independent of the available hardware resources. Thus, TPS measures performance for a fixed amount of work.

Table I. Spark Workloads

Benchmark	Category	Dataset	Data Size
NaiveBayesian	Classification	kdda2010 [Stamper et al. 2010]	2.5G
DecisionTree	Classification	kdda2010	2.5G
GradientBoostedTrees	Classification	kddb2010 [Stamper et al. 2010]	4.8G
SVM	Classification	kdda2010	2.5G
LinearRegression	Classification	kddb2010	4.8G
k means	Clustering	uscensus1990 [Lichman 2013]	327M
ALS	Collaborative Filtering	movielens2015 [Harper and Konstan 2015]	325M
Correlation	Statistics	kdda2010	2.5G
PageRank	Graph Processing	wdc2012 [Meusel et al. 2012]	5.3G
ConnectedComponents	Graph Processing	wdc2012	5.3G
TriangleCounting	Graph Processing	wdc2012	5.3G

Table II. Experimental Parameters

Description	Symbol	Value
Min # sprinters	$N_{\min}$	250
Max # sprinters	$N_{\max}$	750
Prob. of staying in cooling	p_c	0.50
Prob. of staying in recovery	p_r	0.88
Discount factor	δ	0.99

We trace TPS during an application's end-to-end execution in normal and sprinting modes. Since execution times differ in the two modes, comparing traces requires some effort. For every second in normal mode, we measure the number of tasks completed and estimate the number of tasks that would have been completed in the sprinting mode. For our evaluation, we estimate a sprint's speedup by comparing the measured non-sprinting trace and the interpolated sprinting trace. In a practical system, online profiling and heuristics would be required.

Simulation Methods. We simulate 1,000 users and evaluate their performance in the sprinting game. The R-based simulator uses traces of Spark computation collected in both normal and sprinting modes. The simulator models system dynamics as agents sprint, cool, and recover.

One set of simulations evaluates homogeneous agents who arrive randomly and launch the same type of Spark application; randomized arrivals cause application phases to overlap in diverse ways. A second set of simulations evaluates heterogeneous agents who launch different types of applications, further increasing the diversity of overlapping phases. Diverse phase behavior exercises the sprinting game as agents and their processors optimize strategies in response to varied competitors'.

Table II summarizes technology and system parameters. Parameters $N_{\min}$ and $N_{\max}$ are set by the circuit breaker's tripping curve. Parameters p_c and p_r are set by the chip's cooling mechanism and the rack's UPS batteries. These probabilities decrease as cooling efficiency and recharge speed increase—see Section 2.

6. EVALUATION

We evaluate the sprinting game and its equilibrium threshold against several alternatives. Although there is little prior work on managing sprints, we compare against four policies that represent broader perspectives on power management. First, greedy heuristics focus on the present and neglect the future [Zheng and Wang 2015]. Second, control-theoretic heuristics are reactive rather than proactive [bro 2001; ska 2002]. Third, round-robin methods focus on fairness and neglect individual sprinters' utilities. Fourth, centralized heuristics focus on the system and neglect individuals. Unlike

these approaches, the sprinting game anticipates the future and emphasizes strategic agents who participate in a shared system.

Greedy (G) permits agents to sprint as long as the chip is not cooling and the rack is not recovering. This mechanism may frequently trip the breaker and require rack recovery. After recovery, agent wake-ups and sprints are staggered across two epochs. Greedy produces a poor equilibrium—knowing that everyone is sprinting, an agent's best response is to sprint as well.

Exponential Backoff (E-B) throttles the frequency at which agents sprint. An agent sprints greedily until the breaker trips. After the first trip, agents wait 0–1 epoch before sprinting again. After the second trip, agents wait 0–3 epochs. After the tth trip, agents wait for some number of epochs drawn randomly from $[0, 2^t - 1]$. The waiting interval contracts by half if the breaker has not been tripped in the past 100 epochs.

Round-Robin (R-R) randomly and uniformly chooses a fixed number of active agents to sprint. Since agents are selected to sprint at random, they may sprint when their benefit from sprinting is low or they may fail to sprint when their benefit is high. On average, all agents are allocated equal sprinting time.

Cooperative Threshold (C-T) assigns each agent the globally optimal threshold for sprinting. The coordinator exhaustively searches for the threshold that maximizes system performance. The coordinator enforces these thresholds although they do not reflect agents' best responses to system dynamics. These thresholds do not produce an equilibrium but do provide an upper bound on performance.

Equilibrium Threshold (E-T) assigns each agent her optimal threshold from the sprinting game. The coordinator collects performance profiles and implements Algorithm 1 to produce thresholds that reflect agents' best responses to system dynamics. These thresholds produce an equilibrium and agents cannot benefit by deviating from their assigned strategy.

6.1. Sprinting Behavior

Figure 6 compares sprinting policies and resulting system dynamics as 1,000 instances of *Decision Tree*, a representative application, computes for a sequence of epochs. Sprinting policies determine how often agents sprint and whether sprints trigger emergencies. Ideally, policies would permit agents to sprint up until they trip the circuit breaker. In this example, 250 of the 1,000 agents for *Decision Tree* can sprint before triggering a power emergency.

Greedy heuristics are aggressive and inefficient. A sprint in the present precludes a sprint in the near future, harming subsequent tasks that could have benefited more from the sprint. Moreover, frequent sprints risk power emergencies and require rack-level recovery. G produces an unstable system, oscillating between full-system sprints that trigger emergencies and idle recovery that harms performance. G staggers the distribution of sprinting permissions after recovery to avoids dI/dt problems, which reduces but does not eliminate instability.

Control-theoretic approaches are more conservative, throttling sprints in response to power emergencies. E-B adaptively responds to feedback, producing a more stable system with fewer sprints and emergencies. Indeed, E-B may be too conservative, throttling sprints beyond what is necessary to avoid tripping the circuit breaker. The number of sprinters is consistently lower than N_{min}, which is safe but leaves sprinting opportunities unexploited. Thus, in neither G nor E-B do agents sprint to full advantage.

Round-robin methods are designed to be fair, in the sense that all agents are allocated equal sprinting time (on average). At each epoch, R-R(X) randomly and uniformly chooses X active agents to sprint. We now discuss system dynamics for R-R(250) and

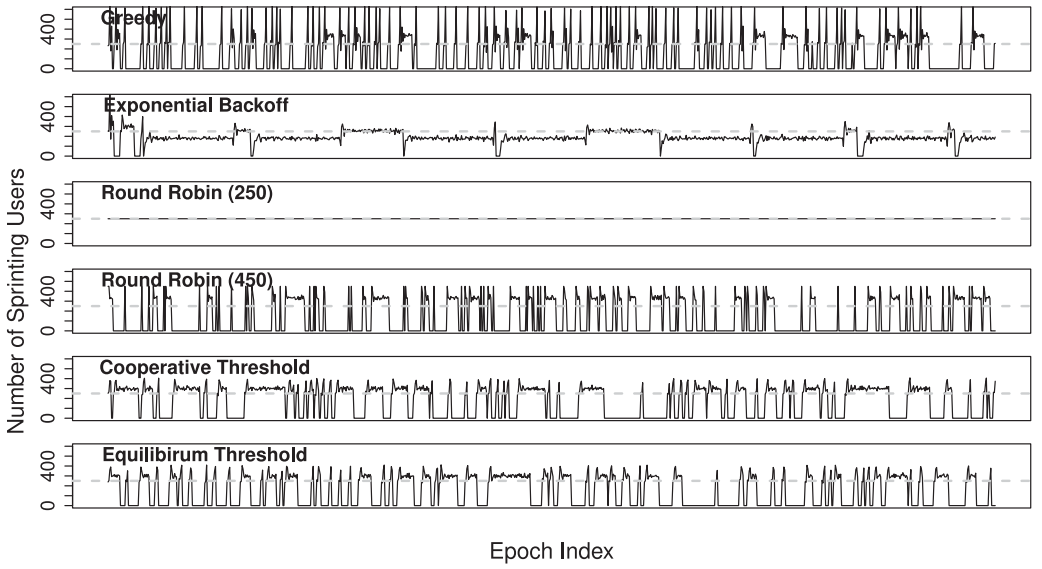

Fig. 6. Sprinting behavior for a representative application, Decision Tree. The black line denotes number of sprinters. The gray line denotes the point at which sprinters risk a power emergency, $N_{\min}$.

R-R(450). Since $N_{min} = 250$, power emergencies cannot occur under R-R(250) and the system does not spend any time in the recovery state. However, power emergencies are possible for R-R(450). In the first epoch after recovery from a power emergency, all 1,000 agents are active and 450 are selected sprint. If an emergency is not triggered, then R-R chooses another 450 agents to sprint from among the 550 active agents. In the third epoch, there are fewer than 450 active users because many who sprinted in the first epoch are still cooling. Note that R-R is oblivious to agents' utilities, in contrast to C-T and E-T.

E-T produces an equilibrium in which agents play their optimal strategies and converge to a stationary distribution. In equilibrium, the number of sprinters is just slightly above $N_{min} = 250$, the number that causes a breaker to transition from the non-tripped region to the tolerance band. After emergency and recovery, the system quickly returns to equilibrium. Note that E-T's system dynamics are similar to those from the high-performance, cooperative C-T policy.

Figure 7 shows the percentage of time an agent spends in active, cooling, and recovery states. The analysis highlights G and E-B's limitations. With G, an agent spends more than 50% of its time in recovery, waiting for batteries to recharge after an emergency. With E-B, an agent spends nearly 40% of its time in active mode but not sprinting. Power emergencies do not occur in R-R(250), so each agent spends no time in recovery and 25% of its time sprinting. Power emergencies occur frequently in R-R(450) and each agent spends more than 50% of its time in recovery.

Agents spend comparable shares of their time sprinting in each policy (except R-R(250). However, this observation understates the sprinting game's advantage. G and E-B sprint at every opportunity and ignore transitions into cooling states, which preclude sprints in future epochs. R-R, on the other hand, neglects agents' utilities and randomly picks sprinters. In contrast, E-T and C-T's sprints are more timely as strategic agents sprint only when estimated benefits exceed an optimized threshold. Thus, a sprint in E-T or C-T contributes more to performance than one in G, E-B, or R-R.

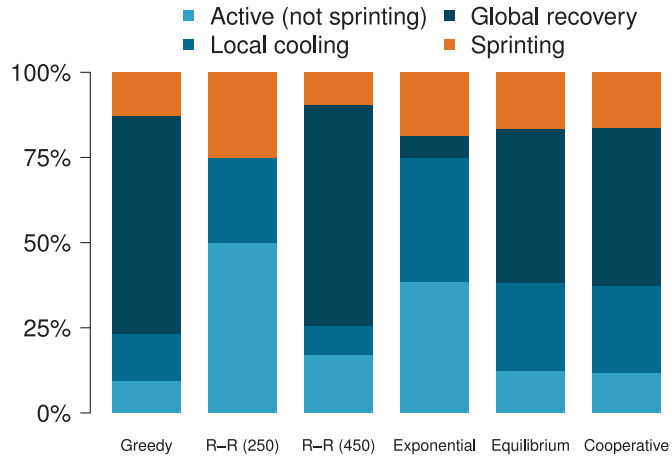

Fig. 7. Percentage of time spent in agent states for a representative application, Decision Tree.

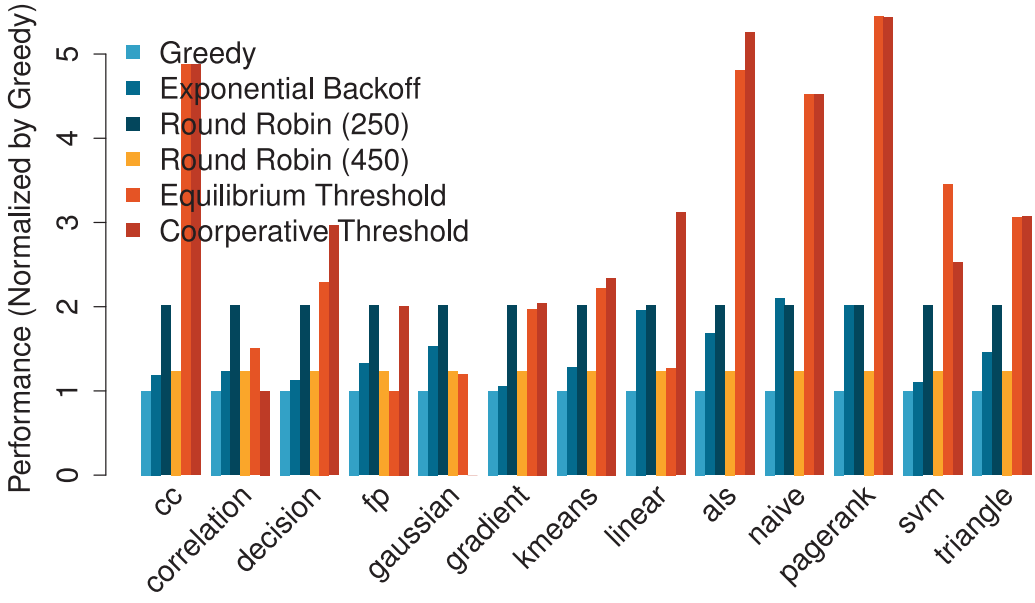

Fig. 8. Performance, measured in task throughput and normalized against greedy, for single application type.

6.2. Sprinting Performance

Figure 8 shows task throughput under varied policies. The sprinting game outperforms greedy heuristics and is competitive with globally optimized heuristics. Rather than sprinting greedily, E-T uses equilibrium thresholds to select more profitable epochs for sprinting. E-T outperforms G, E-B, and R-R by up to 6.8×, 4.8×, and 2.5×, respectively. Agents who use their own strategies to play the game competitively produce outcomes that rival expensive cooperation. E-T's task throughput is 90% that of C-T's for most applications. R-R(250) outperforms R-R(450) by avoiding power emergencies, and thereby allowing more agents to sprint over time. It also outperforms E-T for *Linear Regression* and *Correlation*.

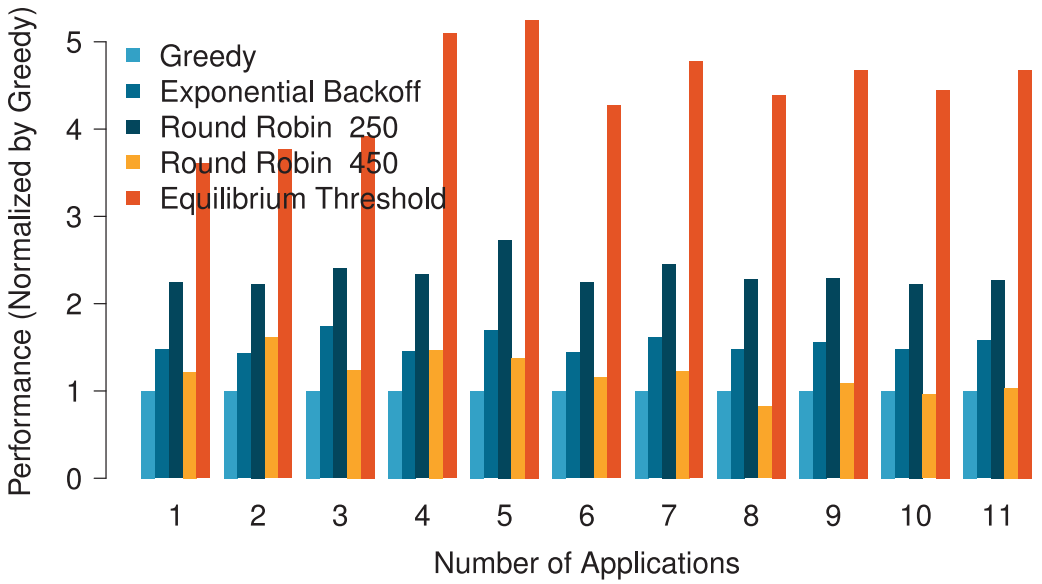

Fig. 9. Performance, measured in task throughput and normalized against greedy, for multiple application types.

Linear Regression and *Correlation* are outliers, achieving only 36% and 65% of cooperative performance. For these applications, E-T performs as badly as G and E-B because the applications' performance profiles exhibit little variance and all epochs benefit similarly from sprinting. When an agent cannot distinguish between epochs, she sets a low threshold and sprints for every epoch. In effect, for such applications, E-T produces a greedy equilibrium.

Thus far, we have considered agents for applications of the same type that compute together. When agents represent different types of applications, E-T assigns different sprinting thresholds for each type. Figure 9 shows performance as the number of application types increases. We evaluate performance for a system with k types by randomly selecting k applications, finding each agent's strategy under an E-T policy, and repeating 10 times to report an average. As before, E-T performs much better than G, E-B, and R-R. We do not evaluate C-T because searching for optimal thresholds for multiple types of agents is computationally hard.

6.3. Sprinting Strategies

Figure 10 uses kernel density plots for two representative applications, *Linear Regression* and *PageRank*, to show how often and how much their tasks benefit from sprinting. *Linear Regression* presents a narrower distribution and performance gains from sprinting vary in a band between 3× and 5×. In contrast, *PageRank*'s performance gains can often exceed 10×.

The coordinator uses performance profiles to optimize threshold strategies. *Linear Regression*'s strategy is aggressive and uses a low threshold that often induces sprints. This strategy arises from its relatively low variance in performance gains. If sprinting's benefits are indistinguishable across tasks and epochs, an agent sprints indiscriminately and at every opportunity. *PageRank*'s strategy is more nuanced and uses a high threshold, which cuts her bimodal distribution and implements judicious sprinting. She sprints for tasks and epochs that benefit most (i.e., those that see performance gains greater than 10×).

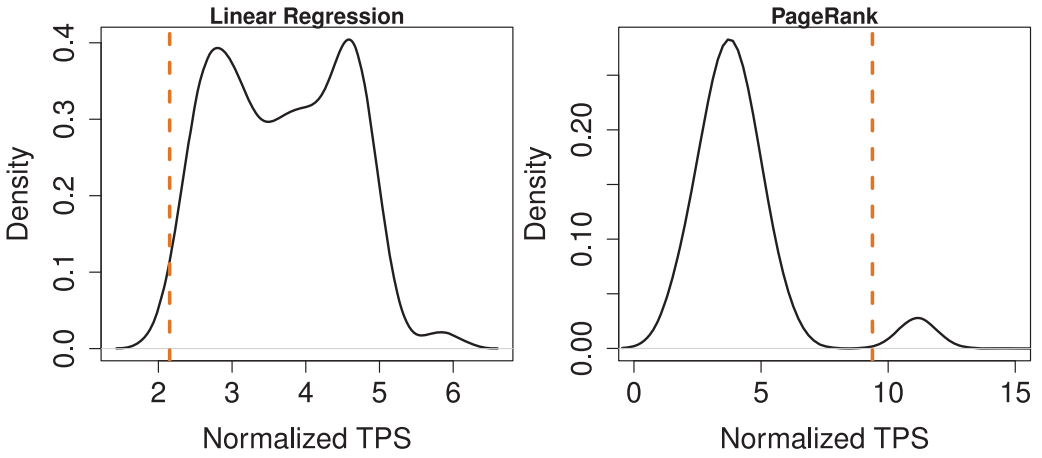

Fig. 10. Probability density for sprinting speedups.

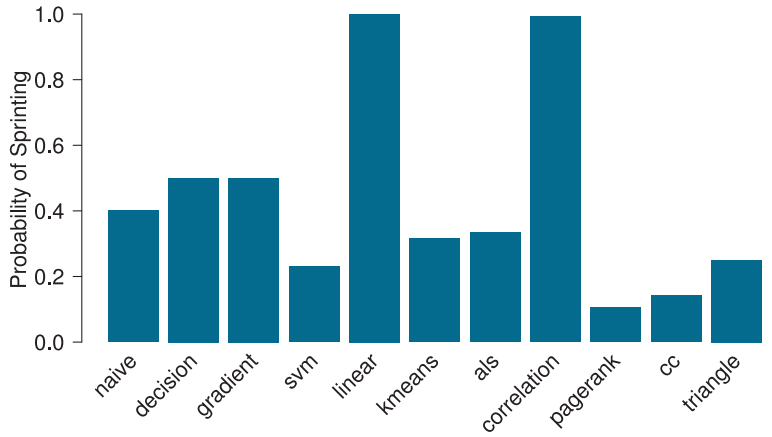

Fig. 11. Probability of sprinting.

Figure 11 illustrates diversity in agents' strategies by reporting their propensities to sprint. *Linear Regression* and *Correlation*'s narrow density functions and low thresholds cause these applications to sprint at every opportunity. The majority of applications, however, resemble *PageRank* with higher thresholds and judicious sprints.

6.4. Equilibrium versus Cooperation

Sprinting thresholds from equilibria are robust to strategic behavior and perform well. However, cooperative thresholds that optimize system throughput can perform even better. Our evaluation has shown that the sprinting game delivers 90% of the performance from cooperation. But we find that the game performs well only when the penalties from non-cooperative behavior are low. To understand this insight, let us informally define efficiency as the ratio of game performance from equilibrium thresholds (E-T) to optimal performance from cooperative thresholds (C-T).[1]

[1]We are informal because the domain of strategies is huge and we consider only the best cooperative threshold. A non-threshold strategy might provide even better performance.

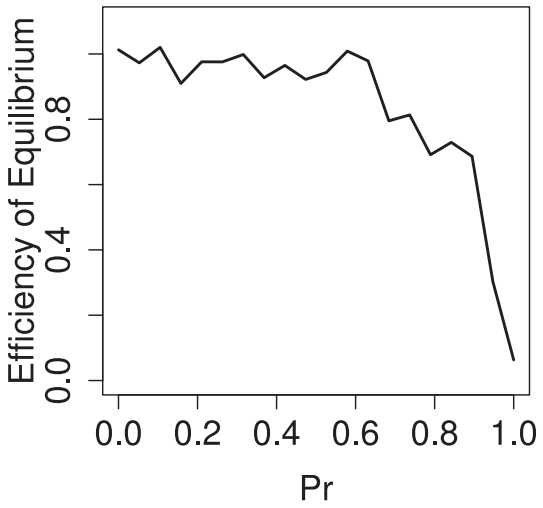

Fig. 12. Efficiency of equilibrium thresholds.

The sprinting game produces efficient equilibria because the penalty for non-cooperative behavior is triggering a power emergency. In the sprinting architecture, recovery is relatively inexpensive as batteries recharge and normal system operation resumes in 10 epochs or fewer. However, higher penalties for non-cooperative behavior would degrade the game's performance from equilibrium strategies. Figure 12 shows how efficiency falls as recovery from power emergencies become increasingly expensive. Recall that p_r is the probability an agent in recovery stays in that state.

Prisoner's Dilemma. The sprinting game fails when an emergency requires indefinite recovery and p_r is 1. In this extreme scenario, we would like the game to produce an equilibrium in which agents sprint yet avoid tripping the breaker. Unfortunately, the game has no equilibrium that avoids tripping the breaker and triggering indefinite recovery. If a strategic agent were to observe system dynamics that avoid tripping the breaker, which means P_{trip} is zero, then he or she would realize that other agents have set high thresholds to avoid sprints. His or her best response would be to lower his or her threshold and sprint more often. Others would behave similarly and drive P_{trip} higher.

In equilibrium, P_{trip} would rise above zero and agents would eventually trip the breaker, putting the system into indefinite recovery. Thus, selfish agents would produce inefficient equilibria—the Prisoner's Dilemma in which each agent's best response performs worse than a cooperative one.

Enforcing Non-Equilibrium Strategies. The Folk theorem guides agents to a more efficient equilibrium by punishing agents whose responses harm the system. The coordinator would assign agents the best cooperative thresholds to maximize system performance from sprinting. When an agent deviates, he or she is punished such that performance lost exceeds performance gained. When applied to our previous example, punishments would allow the system to escape inefficient equilibria as agents are compelled to increase their thresholds and ensure P_{trip} remains zero.

Note that threat of punishment is sufficient to shape the equilibrium. Agents would adapt strategies based on the threat to avoid punishment. The coordinator could monitor sprints, detect deviations from assigned strategies, and forbid agents who deviate from ever sprinting again. Alternatively, agents could impose collective punishment by continuously sprinting, triggering emergencies, and degrading everyone's performance.

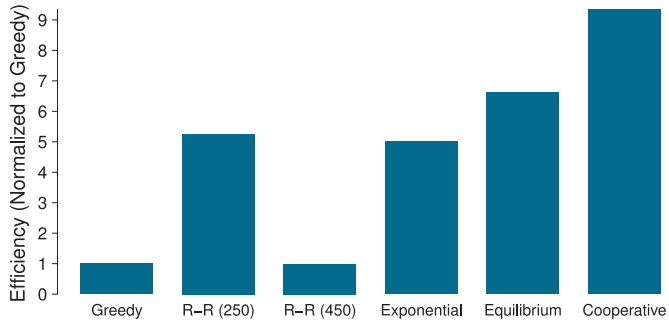

Fig. 13. A comparison of the average gains of each sprinting policy per extra dollar spent.

The threat of collective action deters agents who would deviate from the cooperative strategy.

6.5. Efficiency and Cost of Ownership

Let C_{nominal} denote components of the yearly total cost of ownership (TCO) that do not vary with the sprinting technique employed. In other words, if all multiprocessors operate nominally, TCO would be C_{nominal}. We model TCO as follows:

$$TCO = C_{\text{nominal}} + C_{\text{sprintPwr}} + C_{\text{batteries}} + C_{\text{chip}},$$

where $C_{\text{sprintPwr}}$, $C_{\text{batteries}}$, and C_{chip} denote the power cost of sprinting, maintenance cost of batteries, and maintenance cost of chip multiprocessors, respectively.

Power Cost of Sprinting. Sprints increase power draw, which increase power costs. We model the average increase in power cost as follows:

$$C_{\text{sprintPwr}} = NC_{\text{W-s}}F_{\text{sprint}}\left(P_{\text{sprint}} - P_{\text{nominal}}\right)T.$$

N is the total number of chip multiprocessors and $C_{\text{W-s}}$ is the average cost (in dollars) of one Watt-second of energy. P_{sprint} and P_{nominal} are average sprinting and nominal power (in Watts), respectively, of a chip multiprocessor. We use a conservative value of \$0.13/kW-h for $C_{\text{W-s}}$ [Skach et al. 2015]. We use average values of 83 and 121W for P_{sprint} and P_{nominal}, respectively, based on our system measurements across workloads.

F_{sprint} denotes the fraction of time a chip multiprocessor spends sprinting, on average, which is defined as follows:

$$F_{\text{sprint}} = \frac{\text{avg \# of sprints over epochs}}{\text{total \# of sprinters}}.$$

Finally, T is one year, the duration over which we calculate TCO.

Maintenance Cost of UPS Batteries. Different sprinting policies trigger different numbers of power emergencies, and therefore degrade UPS batteries at different rates. The cost due to replacing UPS batteries is as follows:

$$C_{\text{batteries}} = N_b\left(C_b\frac{R_b}{M_b}\right)T.$$

N_b is the total number of batteries, C_b is the replacement cost of one battery, M_b is the maximum number of power emergencies a battery can experience before needing replacement, and R_b is the average rate at which power emergencies occur.

We assume that the backup power for the UPS unit is composed of N_b batteries, wired in parallel, to deliver the power required to complete sprints when a power emergency occurs. The modeled battery [Battery 2016] retails for about \$40. In the theoretical

worst-case scenario—an emergency triggered by 750 sprinters at the start of their sprints and 250 non-sprinting multiprocessors—the batteries must supply 111.5kW for the entire sprint duration of 150s. Batteries are estimated to supply 730W for 7min. Therefore the system needs at least 153 such batteries ($N_b = 153 > 111.5\text{kW}/730\text{W}$). Note that 7min supply rates are conservative for a system that requires sprint power for 150s.

Therefore, we estimate battery costs to be $C_b = 153 \times \$40 = \$6{,}120$. We do not account for losses due to voltage transformation (12VDC to 120VAC). If these losses are significant, extra batteries must be provisioned. We estimate M_b to be greater than 3,000 cycles [Battery 2016]. We expect to discharge batteries to less than 30% of capacity during each power emergency. Finally, we model R_b as follows:

$$R_b = \frac{\text{avg \# of emergencies over epochs}}{\text{length of an epoch}}.$$

Maintenance Cost of Chip Multiprocessors. Different sprinting policies permit multiprocessors to sprint with different frequencies and therefore incur PCM heating-cooling cycles with different frequencies. The cost due to PCM replacements can be modeled as

$$C_{\text{chip}} = N\left(C_w \frac{R_w}{M_w}\right) T,$$

C_w is the cost to replace the paraffin wax for one multiprocessor, M_w is the maximum number of heating-cooling cycles (or sprints) a PCM can undergo before needing replacement, and R_w is the average rate at which heating-cooling cycles (or sprints) occur. We quantify C_w as follows:

$$C_w = \frac{\text{dollars}}{\text{gram of paraffin}} \times \frac{\text{grams of paraffin}}{\text{PCM unit}}.$$

We use a conservative value \$2,000/ton of paraffin [Skach et al. 2015], and we consider a 200J/g paraffin PCM described in Raghavan [2013]. Since $P_{\text{sprint}} - P_{\text{nominal}} = 38\text{W}$ and the sprinting duration is 150s, the system requires a PCM mass of

$$(38\text{J/s} \times 150\text{s}) / (200\text{J/g}) = 28.5\text{g},$$

which we round to 30g. For M_w, we use 1,000 as a conservative value [Skach et al. 2015]. Finally, we model R_w as follows:

$$R_w = \frac{\text{avg \# of sprints per second}}{\text{total \# of sprinters}}.$$

Efficiency analysis. We model the efficiency of a sprinting policy as

$$E = \frac{P}{TCO - C_{nominal}},$$

where the nominator represents the performance gain of a sprinting policy, and the denominator represents the policy's extra costs due to sprinting. This ratio serves as a means of quantifying the cost effectiveness of a sprinting policy—the average gains of a sprinting policy per extra dollar spent.

Figure 13 demonstrates the cost effectiveness of different sprinting policies normalized to that of greedy. G and R-R(450) are relatively inefficient as both frequently trigger power emergencies and sprint without considering the utility of sprinting. By triggering fewer power emergencies, R-R(250) and E-B demonstrate the efficiency possible by constantly sprinting users. However, by neglecting the utilities of the sprinters, they do not maximize efficiency gains.

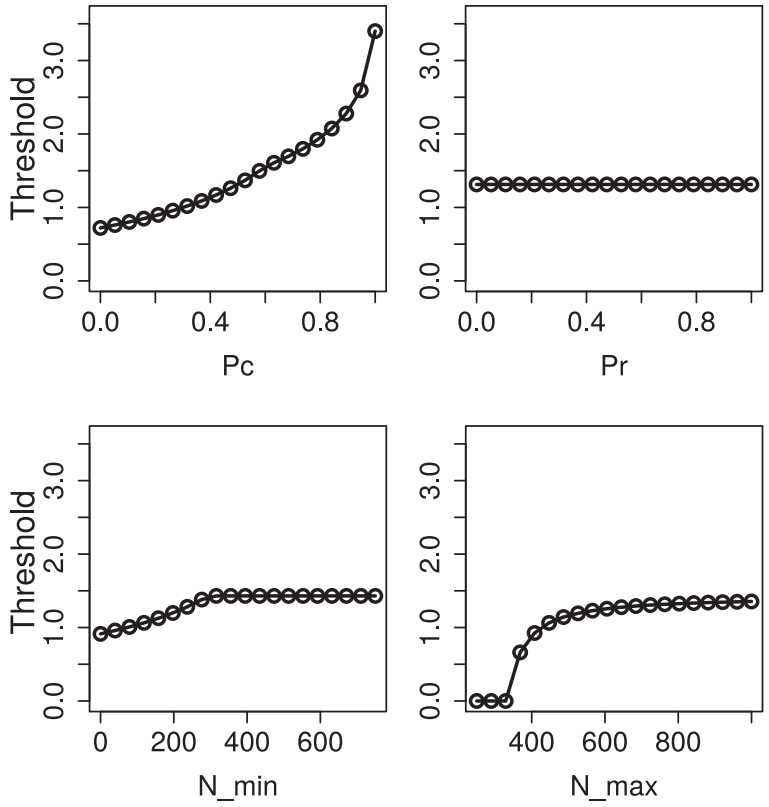

Fig. 14. Sensitivity of sprinting threshold to architectural parameters—probability of staying in cooling and recovery p_c, p_r and the tripping curve $N_{\min}$, $N_{\max}$.

Although the E-T and C-T strategies cause some power emergencies, by considering sprinting agents' utilities when choosing sprinters, they increase the gains from sprinting and thus produce the highest efficiencies. Since C-T assigns each agent the globally optimal threshold for sprinting, its agents sprint most effectively. As a result C-T results in the highest performance per dollar gains.

6.6. Sensitivity Analysis

Figure 14 shows the sprinting threshold's sensitivity to the game's parameters. In practice, server engineering affects cooling and recovery durations (p_c, p_r) as well as the breaker's trip curve ($N_{\min}$, $N_{\max}$).

As cooling duration increases, thresholds increase and agents sprint less. Agents are more cautious because sprinting in the current epoch requires many more epochs for cooling. The opportunity cost of sprinting mistakenly rises. As recovery duration increases, the cost of tripping the breaker increases. However, because each agent sprints to pursue her own performance while hoping others do not trip the breaker, thresholds are insensitive to recovery cost. When p_r is 1, we have shown how agents encounter the Prisoner's Dilemma, see Section 6.4.

When $N_{\min}$ and $N_{\max}$ are small, the probability of tripping the breaker is high. Ironically, agents sprint more aggressively and extract performance now because emergencies that forbid future sprints are likely. When $N_{\min}$ and $N_{\max}$ are big, each

agent sprints more judiciously as a sprint now affects the ability to sprint in the future.

7. RELATED WORK

The sprinting problem falls into the general category of datacenter power management, but we are the first to identify the problem and propose a game-theoretic approach. The sprinting problem is made interesting by modern approaches to datacenter provisioning.

To minimize total cost of ownership and maximize return on investment, datacenters oversubscribe their servers [Barroso et al. 2013; Chen et al. 2008], bandwidth [Ballani et al. 2011], branch circuits [Fu et al. 2011], and cooling and power supplies [Fan et al. 2007; Govindan et al. 2012]. In datacenters, dynamic power capping [Bhattacharya et al. 2013] adjusts the power allocation to individual servers, enabling a rich policy space for power and energy management. In servers, managers could pursue performance while minimizing operating costs, which are incurred from energy and cooling [Beloglazov et al. 2012; Lin et al. 2013; Berral et al. 2010; Goiri et al. 2015]. Researchers have sought to allocate server power to performance critical services via dynamic voltage and frequency scaling (DVFS) [Hsu et al. 2015; Lo et al. 2014].

Economics and game theory have proven effective in datacenter power and resource management [Chase et al. 2001]. Markets [Guevara et al. 2013, 2014] and price theory [Somu Muthukaruppan et al. 2014] have been applied to manage heterogeneous server cores. Demand response models have been proposed to handle power emergencies [Liu et al. 2013]. In addition to performance, fairness in game theory has been studied to incentivize users when sharing hardware in a cloud environment [Ghodsi et al. 2011; Zahedi and Lee 2014, 2015].

In this article, we treat the sprinting management problem as a repeated game and seek an equilibrium that leads sprinting servers to expected behavior. Similar approaches have been applied to power control in wireless communication systems [Le Treust and Lasaulce 2010]. But we are the first to consider game theory for datacenter management, especially in the context of computational sprinting and power capping.

8. CONCLUSIONS

We present a sprinting architecture in which many, independent chip multiprocessors share a power supply. When an individual chip sprints, its excess heat constrains future sprints. When a collection of chips sprint, its additional power demands raise the risk of power emergencies. For such an architecture, we present a management framework that determines when each chip should sprint.

We formalize sprint management as a repeated game. Agents represent chip multiprocessors and their workloads, executing sprints strategically on their behalf. Strategic behaviors produce an equilibrium in the game. We show that, in equilibrium, the computational sprinting game outperforms prior, greedy mechanisms by 4–6× and delivers 90% of the performance achieved from a more expensive, globally enforced mechanism.

REFERENCES

2001. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 171–182.

2002. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 17–28.

Sachin Adlakha and Ramesh Johari. 2013. Mean field equilibrium in dynamic games with strategic complementarities. *Operat. Res.* 61, 4 (2013), 971–989.

Sachin Adlakha, Ramesh Johari, and Gabriel Y. Weintraub. 2013. Equilibria of dynamic games with many players: Existence, approximation, and market structure. *J. Econ. Theory* (2013).

Sachin Adlakha, Ramesh Johari, Gabriel Y. Weintraub, and Andrea Goldsmith. 2010. On oblivious equilibrium in large population stochastic games. In *Proceedings of the 49th IEEE Conference on Decision and Control (CDC)*. IEEE, 3117–3124.

Allen-Bradley. 2016. Bulletin 1489 UL489 Circuit Breakers. (2016). http://literature.rockwellautomation.com/idc/groups/literature/documents/td/1489-td001_-en-p.pdf Online; accessed: 12-29-2016

Ametek. 2016. Selection and Sizing of Batteries for UPS Backup. (2016). http://www.solidstatecontrolsinc.com/knowledgecenter/~/media/85b8e51754c446bda1f38449f444471c.ashx Online; accessed: 12-29-2016

Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*. ACM, 242–253.

Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synth. Lect. Comput. Arch.* 8, 3 (2013), 1–154.

CSB Battery. 2016. EVH12150. (2016). http://www.csb-battery.com.tw/english/01product/02detail.php Online; accessed: 12-29-2016

Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Gen. Comput. Syst.* 28, 5 (2012), 755–768.

Josep Ll Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. 2010. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*. ACM, 215–224.

Arka A. Bhattacharya, David Culler, Aman Kansal, Sriram Govindan, and Sriram Sankar. 2013. The need for speed and stability in data center power capping. *Sust. Comput.: Inf. Syst.* 3, 3 (2013), 183–193.

Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. 2001. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)*. ACM, 103–116.

Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. 2008. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 337–350.

Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. 2007. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 13–23.

Xing Fu, Xiaorui Wang, and Charles Lefurgy. 2011. How much power oversubscription is safe and allowed in data centers. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)*. ACM, 21–30.

Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 323–336.

Íñigo Goiri, Thu D. Nguyen, Ricardo Bianchini, and Íñigo Goiri Presa. 2015. CoolAir: Temperature-and variation-aware management for free-cooled datacenters. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 253–265.

Sriram Govindan, Anand Sivasubramaniam, and Bhuvan Urgaonkar. 2011. Benefits and limitations of tapping into stored energy for datacenters. In *Proceeding of the 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 341–351.

Sriram Govindan, Di Wang, Anand Sivasubramaniam, and Bhuvan Urgaonkar. 2012. Leveraging stored energy for handling power emergencies in aggressively provisioned datacenters. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 75–86.

Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. 2013. Navigating heterogeneous processors with market mechanisms. In *Proceeding of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 95–106.

Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. 2014. Strategies for anticipating risk in heterogeneous system design. In *Proceeding of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 154–164.

Ramakrishna Gummadi, Ramesh Johari, and Jia Yuan Yu. 2012. Mean field equilibria of multiarmed bandit games. In *Proceedings of the 13th ACM Conference on Electronic Commerce (EC)*. ACM, 655–655.

Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 295–308.

Chang-Hong Hsu, Yunqi Zhang, Michael Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, Ronald G. Dreslinski, and others. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 271–282.

Krishnamurthy Iyer, Ramesh Johari, and Mukund Sundararajan. 2011. Mean field equilibria of dynamic auctions with learning. *ACM SIGecom Exch.* 10, 3 (2011), 10–14.

Mael Le Treust and Samson Lasaulce. 2010. A repeated game formulation of energy-efficient decentralized power control. *IEEE Trans. Wireless Commun.* 9, 9 (2010), 2860–2869.

Yingmin Li, Benjamin Lee, David Brooks, Zhigang Hu, and Kevin Skadron. 2006. CMP design space exploration subject to physical constraints. In *Proceedings of the 12th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 17–28.

M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml.

Minghong Lin, Adam Wierman, Lachlan L. H. Andrew, and Eno Thereska. 2013. Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Trans. Netw.* 21, 5 (2013), 1378–1391.

Zhenhua Liu, Adam Wierman, Yuan Chen, Benjamin Razon, and Niangjun Chen. 2013. Data center demand response: Avoiding the coincident peak via workload shifting and local generation. *Perf. Eval.* 70, 10 (2013), 770–791.

David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 301–312.

F. Maxwell Harper and Joseph A. Konstan. 2015. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4 (2015), 19.

Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2012. Web Data Commons - Hyperlink Graphs. (2012). http://webdatacommons.org/hyperlinkgraph/index.html Online; accessed: 12-29-2016

Zwane Mwaikambo, Ashok Raj, Rusty Russell, Joel Schopp, and Srivatsa Vaddagiri. 2004. Linux kernel hotplug CPU support. In *Linux Symposium*, Vol. 2.

Arun Raghavan. 2013. *Computational Sprinting: Exceeding Sustainable Power in Thermally Constrained Systems*. Ph.D. Dissertation. University of Pennsylvania.

Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2013a. Computational sprinting on a hardware/software testbed. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 155–166.

Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2013b. Utilizing dark silicon to save energy with computational sprinting. *IEEE Micro* 33, 5 (2013), 20–28.

Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2012. Computational sprinting. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 1–12.

Lei Shao, Arun Raghavan, Laurel Emurian, Marios C. Papaefthymiou, Thomas F. Wenisch, Milo M. K. Martin, and Kevin P. Pipe. 2014. On-chip phase change heat sinks designed for computational sprinting. In *Proceedings of the 30th Annual Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)*. IEEE, 29–34.

Matt Skach, Manish Arora, Chang-Hong Hsu, Qi Li, Dean Tullsen, Lingjia Tang, and Jason Mars. 2015. Thermal time shifting: Leveraging phase change materials to reduce cooling costs in warehouse-scale computers. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 439–449.

Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. 2014. Price theory based power management for heterogeneous multi-cores. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, 161–176.

J. Stamper, A. Niculescu-Mizil, S. Ritter, G. J. Gordon, and K. R. Koedinger. 2010. Algebra I 2006-2007. Challenge data set from KDD Cup 2010 Educational Data Mining Challenge. (2010). http://pslcdatashop.web.cmu.edu/KDDCup/downloads.jsp.

Fabien Volle, Suresh V. Garimella, Mark Juds, and others. 2010. Thermal management of a soft starter: Transient thermal impedance model and performance enhancements using phase change materials. *IEEE Trans. Power Electron.* 25, 6 (2010), 1395–1405.

Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. 2012. Ship: A scalable hierarchical power control architecture for large-scale data centers. *IEEE Trans. Parallel Distrib. Syst.* 23, 1 (2012), 168–176.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, Vol. 10. 10.

Seyed Majid Zahedi and Benjamin C. Lee. 2014. REF: Resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 145–160.

Seyed Majid Zahedi and Benjamin C. Lee. 2015. Sharing incentives and fair division for multiprocessors. *IEEE Micro* 35, 3 (2015), 92–100.

Wenli Zheng and Xiaorui Wang. 2015. Data center sprinting: Enabling computational sprinting at the data center level. In *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 175–184.

Received September 2016; accepted November 2016