DOI:10.1145/3299885

Distributed Strategies for Computational Sprints

By Songchun Fan,† Seyed Majid Zahedi,† and Benjamin C. Lee

Abstract

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.

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.[4] Such a strategy requires responsive mechanisms for delivering power to the computation that needs it most.

Computational sprinting is a class of mechanisms that supply 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,[13,14] sprinting has found numerous applications in datacenter systems. It can accelerate computation for complex tasks or accommodate transient activity spikes.[16,21]

The system architecture determines sprint duration and frequency. Sprinting multiprocessors generate extra heat, absorbed by thermal packages and phase change materials (PCMs),[14,16] 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, 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 her optimal strategy. The datacenter relies on agents' incentives to decentralize management as each agent self-enforces 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.

2. THE SPRINTING ARCHITECTURE

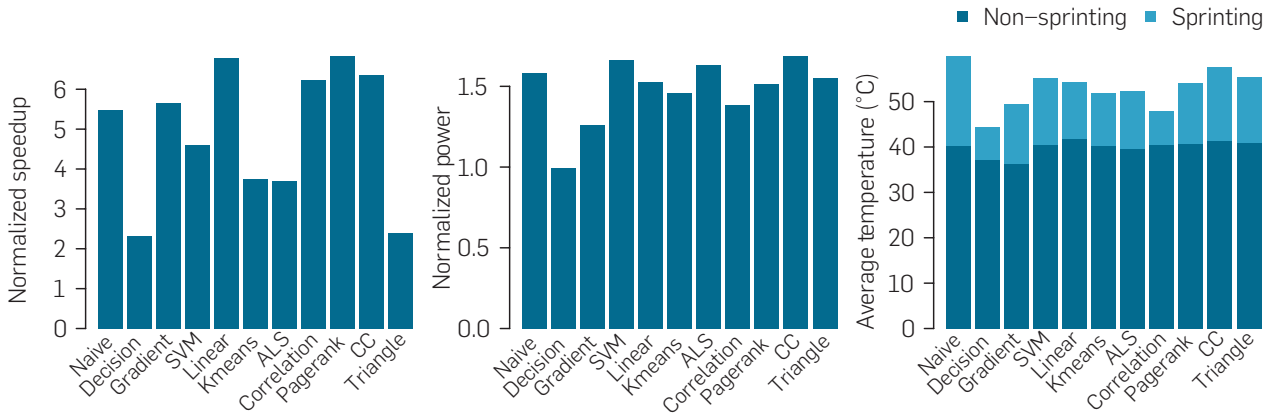
We present a sprinting architecture for chip multiprocessors in datacenters. Multiprocessors sprint by activating additional cores and increasing their voltage and frequency. Datacenter applications, with their abundant task parallelism, scale across additional cores as they become available. In Figure 1, Spark benchmarks perform 2–7× better on a sprinting multiprocessor, but dissipates 1.8× the power. Power produces heat.

Sprinters require infrastructure to manage heat and power. First, the chip multiprocessor's thermal package

The original version of this paper is entitled "The Computational Sprinting Game" and was published in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2016), ACM, NY.

† These authors contributed equally to this work.

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



and heat sink must absorb surplus heat during a sprint.[14, 15] 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. System architecture

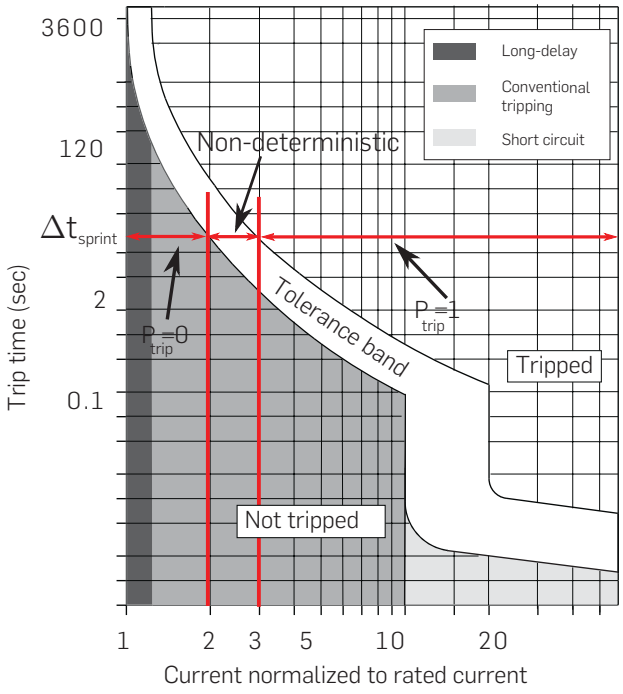
Chip multiprocessors and thermal packages. The quality of the multiprocessor's thermal package, measured by its thermal capacitance and conductance, determines the chip's maximum power level and dictates the duration of a sprint.[13, 15] More expensive heat sinks employ PCMs, which increase thermal capacitance, and permit sprint durations on the order of minutes if not hours. 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. Both factors can be engineered and, with paraffin wax, we estimate a cooling duration on the order of 300s, twice the sprint's duration.

Power delivery and circuit breakers. Datacenter architects deploy servers and multiprocessors to oversubscribe power distribution units for efficiency. Oversubscription utilizes a larger fraction of the facility's provisioned power. But it relies on power capping and varied computational load across servers to avoid tripping circuit breakers or violating contracts with utility providers.[4] Although sprints can boost computation, the risk of a power emergency increases with the number of sprinters in a power capped datacenter.

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

Figure 2. Typical trip curve of a circuit breaker.[5]



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.

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:

- **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. For Spark 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 circuit breakers that can be overloaded to 125–175% of rated current for a 150s sprint.[18, 21]

Uninterruptible power supplies. When the breaker trips and resets, power distribution switches from the branch circuit to the uninterruptible power supply (UPS).[7] 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.

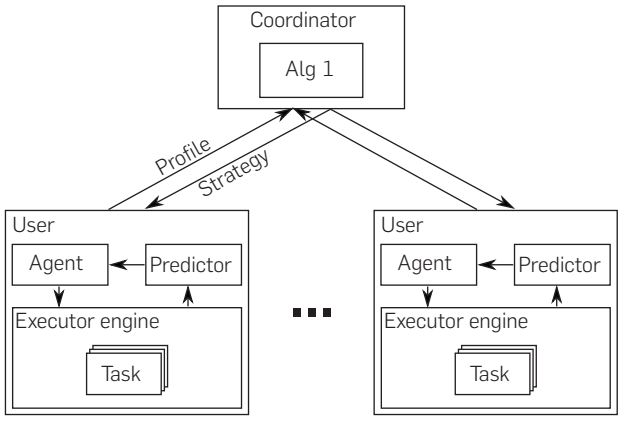
Servers are forbidden from sprinting again until UPS batteries are recharged. Sprints before recovery compromises server availability and increases vulnerability to power emergencies. Moreover, frequent discharges without recharges shorten battery life. The average recovery duration, denoted by $\Delta t_{recover}$, depends on the UPS discharge depth and recharging time. A battery can be recharged to 85% capacity in 8–10× the discharge time, which corresponds to 8–10× the sprint duration.

2.2 Management architecture

Figure 3 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 run-time components: executor, agent, and predictor. Executors provide clean abstractions, encapsulating applications that could employ different software frameworks.[10] 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. Task scheduling is particularly important as it increases parallelism when sprinting powers-on cores and tolerates faults when cooling and recovery powers-off cores.

Figure 3. Users deploy task executors and agents that decide when to sprint. Agents send performance profiles to a coordinator and receives optimized sprinting strategies.



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. An agent who misreports her profile has little influence on conditions in a large system. Not only does she fail to affect others, an agent who misreports suffers degraded performance as the coordinator assigns 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 her strategy and no agent benefits by deviating from it. An equilibrium has compelling implications for management overheads. If each agent knows that every other agent is playing her assigned strategy, 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. In effect, an equilibrium permits the distributed enforcement of sprinting policies.

Equilibria are especially compelling when compared to the centralized enforcement of coordinated policies, which poses several challenges. First, centralized enforcement 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.

3. THE SPRINTING GAME

We design a sprinting game to govern power supply and manage system dynamics. The game divides time into epochs and asks agents to play repeatedly. Agents represent chip multiprocessors that share power. Each agent chooses to sprint independently, pursuing benefits in the current epoch and estimating repercussions in future epochs. An agent's utility from sprinting varies across epochs according to her application's phases. Multiple agents can sprint

simultaneously, but they risk tripping the circuit breaker and triggering power emergencies that harm global performance.

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, we consider three cores at 1.2GHz in normal mode and twelve cores at 2.7GHz in a sprint.

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

Active (A) – Agent can safely sprint. An agent in the active state operates her chip in normal mode by default. The agent may decide to sprint by comparing benefits in the current epoch against benefits from deferring the sprint to a future epoch. If the agent sprints, 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 package. 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 that $1/(1 - p_c) = \Delta t_{\text{cool}}$.

Rack recovery (R) – Agent cannot sprint. When multiple chips sprint simultaneously, total current draw may trip the 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 power supply and 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 that $1/(1 - p_r) = \Delta t_{\text{recover}}$.

4. GAME DYNAMICS AND STRATEGIES

Strategic agents decide between sprinting or not to maximize utilities. Sophisticated strategies produce several desirable outcomes. Agents sprint during the epochs that benefit most from additional cores and higher frequencies. Moreover, agents 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 dynamics to optimize each agent's strategy for her performance. A comprehensive approach to optimizing strategies considers each agent—her state, utility, and history—to determine whether sprinting maximizes her performance given her competitor's strategies and system state. In practice, however, this optimization is intractable for hundreds or thousands of agents.

4.1 Mean field equilibrium

The mean field equilibrium (MFE) is an approximation method when analyzing individual agents in a large system is intractable.[1] First, we define key probability distributions on population behavior. 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 her optimal strategy. Thus, we 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 system is not in recovery. 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. Given the tripping probability, which concisely describes population dynamics, an agent can formulate her best response and optimize her sprinting strategy to maximize performance. We find an equilibrium by specifying an initial value for the tripping probability and iterating.

- **Optimize sprint strategy (§4.2).** Given the probability of tripping the breaker P_{trip}, each agent optimizes her sprinting strategy to maximize her performance. She sprints if performance gains from doing so exceed some threshold. Optimizing her strategy means setting her threshold u_T.
- **Characterize sprint distribution (§4.3).** Given that each agent sprints according to 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

Sprinting defines a repeated game in which an agent acts in the current epoch and encounters consequences of that action in future epochs. An agent optimizes her sprinting strategy accounting for the probability of tripping the circuit breaker P_{trip}, her utility from sprinting u, and her state. To decide whether to sprint, each agent optimizes the following Bellman equation.

$$V(u, A) = \max\{V_s(u, A), \quad V_{\neg s}(u, A)\} \tag{1}$$

The 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. An action's value depends on 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 future utility. When she sprints, future utility is calculated for the cooling state $V(C)$ or the recovery state $V(R)$ when her sprint trips the breaker.

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

However, an agent who does not sprint will remain in the active state unless other sprinting agents trip the circuit breaker and require recovery.

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

$V(A)$ denotes an agent's expected value from being in the active state. The game profiles an application and its time-varying computational phases to obtain a 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 she and/or others sprint.

$$V(C) = \delta\ [V(C)p_c + V(A)(1-p_c)](1-P_{\text{trip}}) + \delta\ V(R)P_{\text{trip}} \tag{5}$$

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

Parameters p_c and p_r are technology-specific probabilities of an agent in cooling and recovery states staying in those states. The game tunes these parameters to reflect the time required for chip cooling after a sprint and for rack recovery after a power emergency.

Threshold strategy. An agent should sprint if her utility from doing so is greater than not. Equation (7), which follows from Equations (2) and (3), states that an agent should sprint if her utility u is greater than her optimal threshold for sprinting u_T. Applying this strategy in every epoch maximizes expected value across time in the repeated game.

$$\begin{aligned} V_S(u,\ A) &> V_{\neg S}(u,\ A) \\ u &> \underbrace{\delta\ (V(A)-V(C))\ (1-P_{\text{trip}})}_{u_T} \end{aligned} \tag{7}$$

4.3 Characterizing the sprint distribution

Given threshold u_T, an agent estimates the probability that she sprints, p_s, in a given epoch.

$$p_s = \int_{u_T}^{u_{\max}} f(u)\, du \tag{8}$$

The probabilities of sprinting (p_s) and cooling (p_c) define a Markov chain that describes each agent's behavior. As agents play their strategies, the Markov chain converges to a stationary distribution in which each agent is active with probability p_A. Given N agents, the expected number of sprinters is

$$n_S = p_s \times p_A \times N \tag{9}$$

Given the expected number of sprinters, the game updates the probability of tripping the breaker according to its trip curve (*e.g.*, Figure 2).

$$P_{\text{trip}} = \begin{cases} 0 & \text{if } n_S < N_{\min} \\ \dfrac{n_S - N_{\min}}{N_{\max} - N_{\min}} & \text{if } N_{\min} \le n_S \le N_{\max} \\ 1 & \text{if } n_S > N_{\max} \end{cases} \tag{10}$$

P_{trip} may change u_T and n_S, which may produce a new 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. 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 (8)–(10).

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

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)–(7). Second, it estimates the number of sprinters according to Equation (9). Finally, it updates the probability of tripping the breaker according to Equation (10). The algorithm terminates when thresholds, number of sprinters, and tripping probability

Algorithm 1: Optimizing the Sprint Strategy

input : Density for sprinting utilities ($f(u)$)
output: Optimal sprinting threshold (u_T)
$j \leftarrow 1$
$P^0_{lstrip} \leftarrow 1$
while P^j_{trip} *not converged* **do**
  $u^j_T \leftarrow$ DP solution for Equations (1)–(7) with P^j_{trip}
  $p^j_S \leftarrow$ Equation (8) with $f(u), u^j_T$
  $n^j_S \leftarrow$ Equation (9) with MC solution and P^j_S
  $P^{j+1}_{\text{trip}} \leftarrow$ Equation (10)
  $j \leftarrow j+1$
end

are stationary.

The analysis runs periodically to update sprinting strategies and the tripping probability as application mix and system conditions evolve. The analysis does not affect an application's critical path as agents use updated strategies when they become available but need not wait for them. On an Intel® Core™ i5 processor with 4GB of memory, the analysis completes in less than 10s, on average.

Online play. An agent decides whether to sprint at the start of each epoch by estimating a sprint's utility and comparing it against her threshold. Estimation could be implemented in several ways. An agent could use the first few seconds of an epoch to profile 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.

5. EXPERIMENTAL METHODOLOGY

Server measurements. The agent and its application are pinned to a chip multiprocessor, an Intel® Xeon® E5-2697 v2. In normal mode, the agent uses three 1.2GHz cores. In sprinting mode, the agent uses twelve 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.

We evaluate Apache Spark workloads. 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. We profile workloads by modifying Spark (v1.3.1) to log the IDs of jobs, stages, and tasks as they complete. We profile system and power temperature using the Intel® Performance Counter Monitor 2.8.

We measure workload performance in terms of tasks completed per second (TPS). The total number of tasks in a job is constant and independent of the available hardware resources such that TPS measures performance for a fixed amount of work. In our experiments, we trace TPS during application execution in normal and sprinting modes and we estimate speedups by comparing the two traces, epoch by epoch. In a practical system, online profiling and heuristics would be required to estimate speedups.

Datacenter simulation. We simulate 1000 users and evaluate their performance in the sprinting game. The simulator uses server traces and models system dynamics as agents sprint, cool, and recover. Simulations evaluate homogeneous agents who arrive randomly and launch the same type of Spark application; randomized arrivals cause application phases to overlap in diverse ways. Diverse phase behavior exercises the sprinting game as agents optimize strategies in response to varied competitors'.

Table 1 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 system's batteries. These probabilities decrease as cooling efficiency and recharge speed increase.

6. EVALUATION

We evaluate the sprinting game and its equilibrium threshold against several alternatives that represent broader perspectives on power management. First, greedy heuristics focus on the present and neglect the future.[21] Second, control-theoretic heuristics are reactive rather than proactive.[2] Third, centralized heuristics focus on the system and neglect individual users. Unlike these approaches, the sprinting game anticipates the future and models strategic agents 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. 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 t-th trip, agents wait for some number of epochs drawn randomly from $[0, 2^t - 1]$ before sprinting again. The waiting interval contracts by half if the breaker has not been tripped in the past 100 epochs.

Cooperative Threshold (C-T) assigns each agent the globally optimal sprinting threshold. The coordinator identifies and enforces thresholds that maximize system performance. Although these thresholds provide an upper bound on performance, they do not produce an equilibrium because thresholds do not reflect agents' best responses to system dynamics.

Equilibrium Threshold (E-T) assigns each agent her optimal threshold from the sprinting game. The coordinator collects performance profiles and finds thresholds that

Table 1. 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

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 4 compares sprinting policies and resulting system dynamics as 1000 instances of *Decision Tree*, a representative application, computes across over time. 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 1000 agents 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.

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. In neither G nor E-B do agents sprint to full advantage.

In contrast, the computational sprinting game performs well by embracing agents' strategies. 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}, 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.

Figure 5 shows the percentage of time an agent spends in each state. E-T and C-T sprints are timely as strategic agents sprint only when estimated benefits exceed an optimized threshold. A sprint in E-T or C-T contributes more to performance than one in G or E-B. Moreover, G and E-B ignore the consequences of a sprint. 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.

6.2 Sprinting performance

Figure 6 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 and E-B by up to 6.8× and 4.8×, 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.

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

Figure 4. Sprinting behavior for a representative application, Decision Tree. Black line denotes number of sprinters. Gray line denotes the point at which sprinters risk a power emergency, N_{min}.

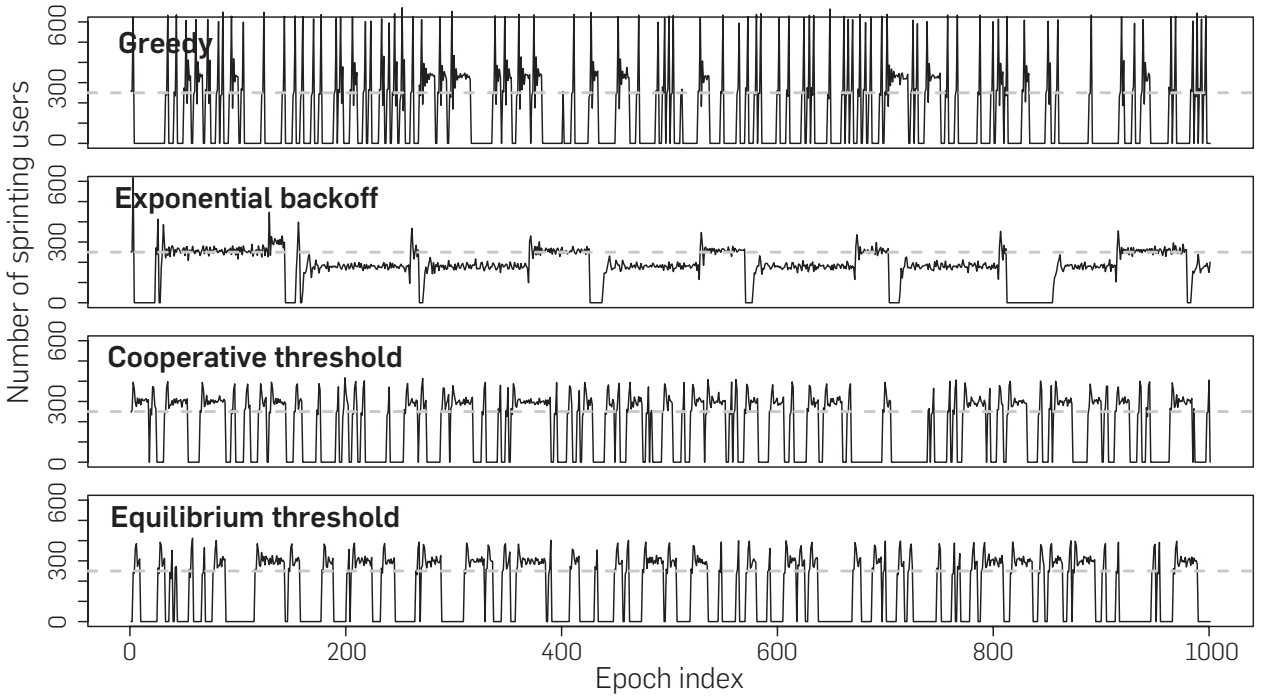


Figure 5. Percentage of time spent in agent states for a representative application, Decision Tree.

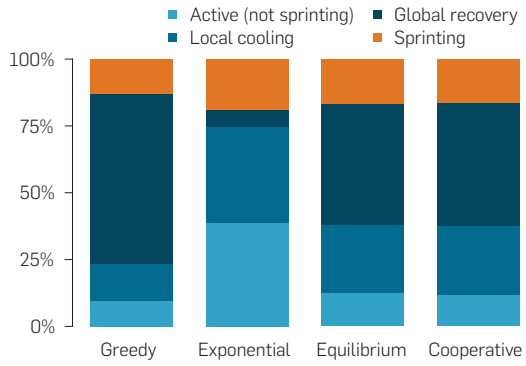


Figure 6. Performance, measured in tasks per second and normalized against greedy, for a single application type.

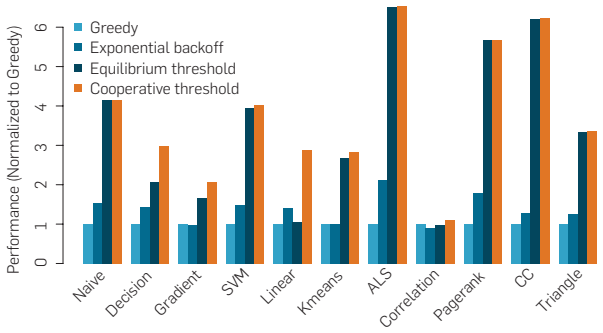


Figure 7. Probability density for sprinting speedups.

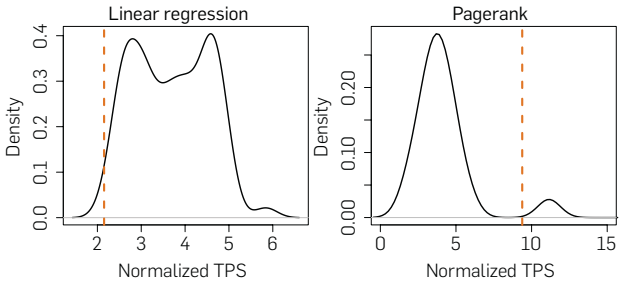
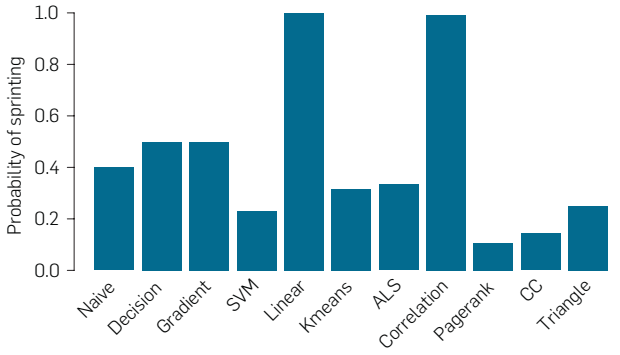


Figure 8. Probability of sprinting.



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.

6.3 Sprinting strategies

Figure 7 uses 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 density plots 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×).

Figure 8 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.

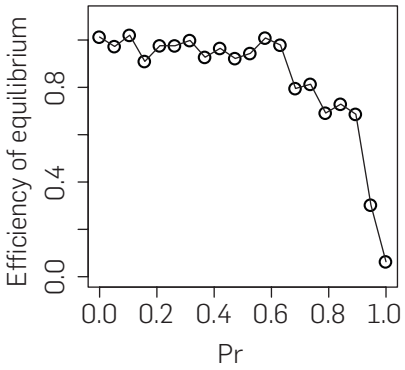
6.4 Equilibrium versus cooperation

Equilibrium thresholds are robust to strategic behavior and perform well, but cooperative thresholds can perform even better. The sprinting game's equilibrium delivers 90% of the performance from cooperation because the penalties from non-cooperative behavior are low. Figure 9 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.

The sprinting game fails when an emergency requires indefinite recovery and p_r is one. This 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, she would realize that other agents have set high thresholds to avoid sprints. Her best response would be lowering her threshold and sprinting 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.

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, she is punished such that

Figure 9. Efficiency of equilibrium thresholds.



performance lost exceeds performance gained. In our example, punishments would allow the system to escape inefficient equilibria as agents are compelled to increase their thresholds and ensure P_{trip} remains zero. The coordinator could monitor sprints, detect deviations from assigned strategies, and forbid agents who deviate from ever sprinting again. Note that threat of punishment is sufficient to shape the equilibrium.

7. CONCLUSION

Economics and game theory have proven effective in datacenter power and resource management. Game-theoretic notions of fairness can incentivize strategic users when sharing hardware.[6,12,19,20] Markets and price theory can allocate and manage heterogeneous servers.[8,9,17] Demand response models can handle power emergencies.[3,11]

We link system architecture and algorithmic economics to decentralize the allocation of shared resources to strategic users. The computational sprinting game is a management architecture that governs how independent chip multiprocessors share a power supply. The approach generalizes beyond datacenters and is relevant to systems that are distributed, heterogeneous, and dynamic. The game's approach to sprinting applies to any mechanism that briey accelerates performance using additional resources be they processor, memory, network, or power. The game's equilibrium highlights a path to scalable management because mean field analysis provides tractability when the number of system components is large. However, finding the equilibrium requires statistical distributions of agent behaviors and further research is needed to reduce offline profiling costs and accelerate online utility prediction.

Acknowledgments

This work is supported by National Science Foundation grants CCF-1149252, CCF-1337215, SHF-1527610, and AF-1408784. This work is also supported by STARnet, a SRC program, sponsored by MARCO and DARPA.

References

1. Adlakha, S., Johari, R. Mean field equilibrium in dynamic games with strategic complementarities. *Oper. Res. 61*, 4 (2013), 971–989.
2. Brooks, D. Martonosi, M. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th IEEE International Symposium on High Performance Computer Architecture (HPCA)* (Monterrey, Nuevo Leon, Mexico, 2001), 171–182.
3. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P. Managing energy and server resources in hosting centers. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP)* (Banff, Alberta, Canada, 2001), 103–116.
4. Fan, X., Weber, W.-D., Barroso, L.A. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)* (San Diego, CA, USA, 2007), 13–23.
5. Fu, X., Wang, X., Lefurgy, C. How much power oversubscription is safe and allowed in data centers. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)* (Karlsruhe, Germany, 2011), 21–30.
6. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)* (Boston, MA, USA, 2011), 323–336.
7. Govindan, S., Sivasubramaniam, A., Urgaonkar, B. Benefits and limitations of tapping into stored energy for datacenters. In *Proceeding of the 38th Annual International Symposium on Computer Architecture (ISCA)* (San Jose, CA, USA, 2011), 341–351.
8. Guevara, M., Lubin, B., Lee, B.C.. Navigating heterogeneous processors with market mechanisms. In *Proceeding of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)* (Shenzhen, China, 2013), 95–106.
9. Guevara, M., Lubin, B., Lee, B.C. Strategies for anticipating risk in heterogeneous system design. In *Proceeding of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)* (Orlando, FL, USA, 2014), 154–164.
10. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I. 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)* (Boston, MA, USA, 2011), 295–308.
11. Liu, Z., Wierman, A., Chen, Y., Razon, B., Chen, N. Data center demand response: Avoiding the coincident peak via workload shifting and local generation. *Perform. Eval. 70*, 10 (2013), 770–791.
12. Llull, Q., Fan, S., Zahedi, S.M., Lee, B.C. Cooper: Task colocation with cooperative games. In *Proceedings of the 23rd IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Austin, TX, USA, 2017), 421–432.
13. Raghavan, A., Emurian, L., Shao, L., Papaefthymiou, M., Pipe, K.P., Wenisch, T.F., Martin, M.M. Computational sprinting on a hardware/software testbed. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, USA, 2013), 155–166.
14. Raghavan, A., Luo, Y., Chandawalla, A., Papaefthymiou, M., Pipe, K.P., Wenisch, T.F., Martin, M.M.K. Computational sprinting. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)* (New Orleans, LA, USA, 2012), 1–12.
15. Shao, L., Raghavan, A., Emurian, L., Papaefthymiou, M.C., Wenisch, T.F., Martin, M.M., Pipe, K.P. On-chip phase change heat sinks designed for computational sprinting. In *Proceedings of the 30th Annual Semiconductor Thermal Measurement and Management Symposium* (San Jose, CA, USA, 2014),29–34.
16. Skach, M., Arora, M., Hsu, C.-H., Li, Q., Tullsen, D., Tang, L., Mars, J. 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)* (Portland, OR, USA, 2015), 439–449.
17. Somu Muthukaruppan, T., Pathania, A., Mitra, T. 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)* (Salt Lake City, UT, USA, 2014), 161–176.
18. Wang, X., Chen, M., Lefurgy, C., Keller, T.W. Ship: A scalable hierarchical power control architecture for large-scale data centers. *IEEE Trans. Parallel Distrib. Syst. 23*, 1 (2012), 168–176.
19. Zahedi, S.M., Lee, B.C. Sharing incentives and fair division for multiprocessors. *IEEE Micro 35*, 3 (2015), 92–100.
20. Zahedi, S.M., Llull, Q., Lee, B.C. Amdahl's Law in the datacenter era: A market for fair processor allocation. In *Proceedings of the 24rd IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Vienna, Austria, 2018).
21. Zheng, W., Wang, X. Data center sprinting: Enabling computational sprinting at the data center level. In *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS)* (Columbus, OH, USA, 2015), 175–184.

Songchun Fan† (songchun.fan@duke.edu), Duke University, California, USA.

Seyed Majid Zahedi† **and Benjamin C. Lee** ([seyedmajid.zahedi, benjamin.c.lee]@duke.edu), Duke University, Durham, NC, USA.