DynaSprint: Microarchitectural Sprints with Dynamic Utility and Thermal Management

Ziqiang Huang
University of Waterloo
ziqiang.huang@uwaterloo.ca

José A. Joao
Arm Research
jose.joao@arm.com

Alejandro Rico
Arm Research
alejandro.rico@arm.com

Andrew D. Hilton
Duke University
adhilton@ee.duke.edu

Benjamin C. Lee
Duke University
benjamin.c.lee@duke.edu

ABSTRACT

Sprinting is a class of mechanisms that provides a short but significant performance boost while temporarily exceeding the thermal design point. We propose DynaSprint, a software runtime that manages sprints by dynamically predicting utility and modeling thermal headroom. Moreover, we propose a new sprint mechanism for caches, increasing capacity briefly for enhanced performance. For a system that extends last-level cache capacity from 2MB to 4MB per core and can absorb 10J of heat, DynaSprint-guided cache sprints improve performance by 17% on average and by up to 40% over a non-sprinting system. These performance outcomes, within 95% of an oracular policy, are possible because DynaSprint accurately predicts phase behavior and sprint utility.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Hardware** → **Power and energy**.

KEYWORDS

performance optimization, power/thermal management, caches

ACM Reference Format:
Ziqiang Huang, José A. Joao, Alejandro Rico, Andrew D. Hilton, and Benjamin C. Lee. 2019. DynaSprint: Microarchitectural Sprints with Dynamic Utility and Thermal Management. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358301

1 INTRODUCTION

Computational sprinting [34, 35] is a mechanism for dark silicon, which describes systems that can only power a fraction of their peak resources [16, 47]. Sprints provide a short but significant performance boost by activating reserve cores and/or increasing frequency and voltage. Sprints draw extra power and temporarily increase chip temperature beyond the thermal design point (TDP). Sprints often require thermal packages that deploy phase change material to increase the system's thermal capacitance, buffer heat during a sprint, and dissipate that heat during normal operation.

Because sprints cannot be sustained, the system needs a mechanism to decide when to start and stop a sprint. Such a mechanism needs to assess both the benefit (*i.e.*, performance gains) and the cost (*i.e.*, thermal headroom consumed). Prior work has either relied on an obvious trigger (*e.g.*, parallel code region activates additional cores [35]) or offline profiles to estimate utility [17]. However, such approaches are limited to certain types of sprint and cannot adapt to workload dynamics.

We propose DynaSprint, a utility and thermal aware run-time that determines when to initiate and terminate a sprint. Based on hardware support available in current processors, we present a software framework that evaluates utility and makes sprinting decisions. This framework integrates new approaches to online phase classification with prior approaches in phase prediction. In every management epoch, the framework predicts the workload's utility from sprinting and assesses the system's thermal profile. We show that these predictions are accurate and permit judicious sprints over time.

We demonstrate DynaSprint for a new class of sprints that increase microarchitectural capacity. Specifically, we propose *cache sprints* that expand last-level cache capacity and exceed TDP when that capacity is most useful. We use the cache as an example, but microarchitectural sprints could apply to resources such as the reorder buffer, issue queue, etc. Prior work in LLC control minimizes interference [25, 28, 55], maximizes throughput [9, 15, 33], or improves fairness [27, 31, 49, 52]. However, sprints present new challenges. Because sprint decisions made in the present affect those in the future, systems must pace sprints to maximize long-run performance under thermal constraints.

Our study of cache sprinting demonstrates its viability. We find that utility from extra cache capacity varies across program execution. Such phase behavior permits judicious sprints that increase capacity when utility is high and cool the system when utility is low. Furthermore, our power models indicate that cache sprint durations can be long enough to capture lengthy, high-utility phases. Longer durations are possible because increasing cache capacity is less power-intensive than activating cores or scaling voltage/frequency.

We implement DynaSprint in software and evaluate it using cache-sensitive applications. We prototype the run-time system on an Intel Xeon Broadwell that supports last-level cache allocation via Intel Cache Allocation Technology [20]. We show that DynaSprint-guided cache sprints improve performance by 17% on average and by up to

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358301

40%. Moreover, DynaSprint outperforms a greedy policy that sprints at every opportunity and performs within 95% of oracular policies that use perfect knowledge of sprint utility and thermal conditions.

2 DYNASPRINT DESIGN

Computational sprinting temporarily exceeds a chip's sustainable thermal budget to boost performance. For example, it activates additional cores and/or boosts frequency, exceeding the processor's thermal design point (TDP) by an order of magnitude or more. Starting and stopping these sprints can be straight-forward. Additional cores are activated at the beginning of a parallel region. Benefits from frequency boosts are relatively easy to estimate. However, managing sprints for microarchitectural resources to maximize long-run performance is challenging. Thus, we design a general framework called DynaSprint that can (1) manage microarchitectural sprints, (2) identify sprint opportunities accurately, and (3) minimize hardware and management overhead.

Broad Application. To find broad application to varied microarchitectural sprint mechanisms, DynaSprint looks beyond simple triggers like those proposed for processor core allocation. Instead, it integrates new approaches to online phase classification with prior work in phase prediction to sprint based on application phase characteristics.

High Accuracy. DynaSprint must "sprint when it really matters". Sprinting without justified benefit in the present not only wastes energy but can potentially hinder future sprints. DynaSprint uses multiple history-based predictors to ensure each sprint decision is made with high confidence.

Low Overhead. Since sprints are usually applied to power and thermal constrained systems, the management framework must incur low overheads. DynaSprint is a software runtime system that utilizes hardware support already available on modern processors. Moreover, DynaSprint works well with coarse-grained epochs that span 100M instructions, making management overheads negligible.

Figure 1 motivates these design goals with a snapshot of program execution split into phases. At "current time," DynaSprint's four tasks are to (1) confirm that the previous epoch was actually "phase B", (2) predict that the next epoch is "phase A", (3) predict the utility from sprinting in "phase A", and (4) determine whether the thermal headroom in the next epoch permits a sprint during phase A. DynaSprint must complete these tasks accurately and efficiently.

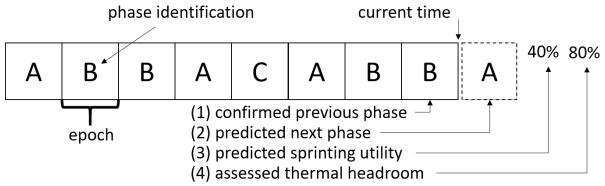

Figure 1: Example program execution in phases

2.1 Management Architecture

Figure 2 presents an overview of DynaSprint. DynaSprint manages sprinting by collecting data about the previous epoch. When an epoch ends, the phase classifier assesses hardware performance counters, which supply the phase signature, to decide whether the epoch corresponds to a previously observed phase or a new one (Section 2.2).

DynaSprint makes a series of predictions about the next epoch to decide whether to sprint. First, the phase predictor uses the most recently completed epoch's phases to query the phase history pattern table and predict the next epoch's phase (Section 2.3). Second, the utility and energy predictor uses the next epoch's phase as well as its historical performance and energy measurements to predict the effects of sprinting (Section 2.4). Third, the thermal tracker uses the previous epoch's energy consumption to determine how much thermal headroom remains in the system (Section 2.5). Finally, the sprinting coordinator aggregates predictions to decide whether to sprint in the next epoch (Section 2.6).

2.2 Phase Classifier

DynaSprint classifies epochs into phases to facilitate the prediction of utility from sprints. Epochs associated with the same phase exhibit similar characteristics such as instruction level parallelism, memory intensity, branch prediction accuracy, etc. DynaSprint captures these characteristics using hardware performance counters available on most modern processors. DynaSprint uses a phase signature, defined by data from a few counters, to classify each epoch into a corresponding phase.

Phase Signature. DynaSprint constructs a phase signature from three performance counters: the number of L1 and L2 cache misses per thousand instructions and the number of branch mispredictions per thousand instructions (abbreviated L1, L2, BR MPKI).

These hardware counters present a number of advantages. First, these counters are good indicators of program phase and cache utility. They measure activity in both datapath and caches, allowing them to distinguish between program phases. Moreover, these counters are independent of last-level cache capacity, producing the same phase signatures for the same instructions in both normal and cache sprinting modes. Finally, these counters are available on most processors, making DynaSprint compatible and portable.

Phase Signature Boundaries. DynaSprint tracks phases and their corresponding signatures during program execution. For each epoch, DynaSprint determines whether its signature matches a previously observed phase or differs enough from prior signatures to describe a new phase. DynaSprint makes this determination by specifying boundaries around phases and their corresponding signatures. When a measured signature lies within an existing phase's boundaries, the epoch is associated with that phase. When it lies beyond any existing boundary, the epoch is associated with a new phase.

An epoch with a phase signature of N counters can be viewed as a point in a N-dimensional space with each counter being its respective coordinate. Figure 3 shows a 2D example in which a phase signature only consists of two counters: L1 MPKI and BR MPKI. In this example, the phase classifier has already identified three phases: A, B and C, represented by the rectangles with the solid lines. The dots within each rectangle represent the program epochs that have been classified into the corresponding phase. Suppose the coordinates of the next epoch "x" are within the rectangle of phase B, then x is classified into phase B, and phase B's boundary is updated after

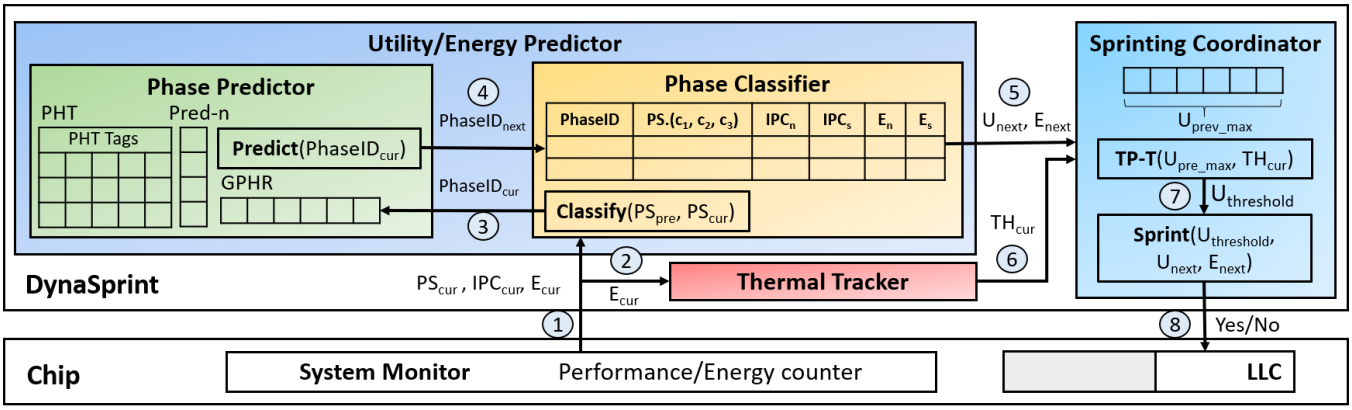

Figure 2: DynaSprint overview: (1) phase classifier reads the phase signature, IPC and energy counters for the current epoch from system monitor (2) thermal tracker reads the energy counters (3) phase classifier identifies the phase for the current epoch (4) phase predictor predicts the next phase (5) utility/energy predictor predicts the sprint utility/energy for next epoch (6) thermal tracker calculates the current thermal headroom (7) sprinting coordinator sets the sprint utility threshold (8) sprinting coordinator decides whether to sprint or not for the next epoch

including x. However, suppose the coordinates of the next epoch "y" are not within any of the existing rectangles, then y forms a new phase D.

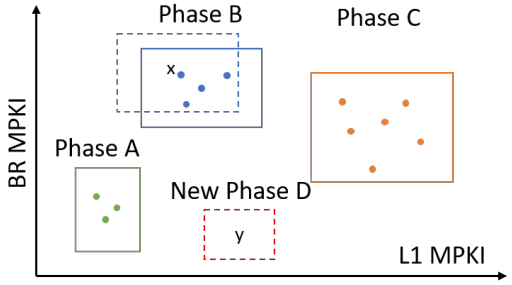

Figure 3: Phase Classifier Example

Algorithm 1 details the analysis of phase and signature boundaries. When an epoch ends, DynaSprint collects a signature for the current epoch PS_{cur}. The algorithm determines whether that signature lies in the neighborhood of signatures for previous observed phases $PS_{pre}[i]$.

Phase signature PS_1 is within the boundary of PS_2 if, for every counter C_j in the signature, the relative difference between $PS_1.C_j$ and $PS_2.C_j$ is smaller than boundary parameter B_{rd} or the absolute difference between $PS_1.C_j$ and $PS_2.C_j$ is smaller than parameter B_{ad}. We assess both absolute and relative differences because comparisons between small values (e.g., those less than one) can produce large relative differences. DynaSprint associates the current epoch with the nearest signatures and phases based on Euclidean distance.

Boundary Parameters. The values of B_{rd} and B_{ad} determine phase granularity, which ultimately determines the total number of phases that describe the program. The bigger the boundaries, the smaller the number of phases. On one hand, fewer phases mitigate classifier overheads, which are linear in the number of phases. On the other hand, more phases increase analysis granularity and permit more accurate predictions of utility and energy. Finer granularities lower variance and improve prediction for utility and energy for each phase. We set boundary parameters to $B_{rd} = 0.30$ and $B_{ad} = 2$ and show, in Section 5.4, that these parameters require a small number of phases yet predict utility and energy accurately.

Algorithm 1 Boundary-based Phase Classification

```
1:  procedure CLASSIFY(PS_pre[ ], PS_cur)
2:      PhaseID = -1
3:      closest = INT_MAX
4:      for PS_i ∈ PS_pre do
5:          if for every C_j ∈ PS: |PS_i.C_j − PS_cur.C_j| / PS_i.C_j < B_rd or
6:                                 |PS_i.Cj − PS_cur.Cj| < B_ad then
7:              dist = EuclideanDistance(PS_cur, PS_i)
8:              if dist < closest then
9:                  closest = dist
10:                 PhaseID = PS_i.PhaseID
11:     if PhaseID == -1 then
12:         PhaseID = NewPhaseID
13:     PS_pre[PhaseID] = PS_cur
```

2.3 Phase Predictor

DynaSprint implements phase prediction using a Global Phase History Table (GPHT) predictor [24]. First, the classifier assigns the just-completed epoch to a phase. Then, it uses the classified phase to index and update the history table. Finally, it uses the table to predict the next epoch's phase.

The green box in Figure 2 details the Global Phase History Table (GPHT). Its main structure consists of a global shift register, called the Global Phase History Register (GPHR), that tracks the last few observed phases. GPHR contents are used to index into a Pattern History Table (PHT), which caches several previously observed phase patterns (PHT Tags), their corresponding predictions for next

phase (PHT Pred-n), as well as recency information (Age/Invalid). When GPHR and PHT tags do not match, the predictor falls back to Last Value Prediction, predicting the last observed phase, stored in GPHR[0], as the next phase. Such table-based history predictors are much more effective than simple statistical predictors, particularly for highly variable programs [13, 24].

This phase prediction strategy exploits the fact that many programs consist of phases with similar characteristics and behaviors. Accurately predicting phases as the program runs benefits various online optimizations such as hardware reconfiguration, voltage and frequency scaling (DVFS), thermal management, and hotcode optimization [5, 7, 11, 21, 22, 24, 29, 45, 51]. Phase behaviors are repetitive and prior work has proposed table-based history predictors to capture past phase patterns for future prediction [13, 24].

2.4 Utility and Energy Predictor

DynaSprint uses historical data, recorded in the phase classifier, to predict the utility from sprinting in the next epoch. The classifier tracks, for each phase, four measures—instruction throughput and processor energy when running in normal and sprinting modes (IPC_n, IPC_s, E_n, E_s).

Updating the Predictor. When an epoch is classified and assigned to a phase, its performance and energy profile updates the classifier's data corresponding to its phase and mode. Initially, when the classifier encounters the first epoch observed for a phase, it will record the performance and energy profile for either the normal or sprinting mode. Later, when the classifier encounters the second epoch for the same phase, the sprinting coordinator collects data for the other mode by initiating or halting a sprint.

This mechanism ensures the classifier collects data for normal and sprinting modes at least once for each phase, thereby exploring the utility from sprinting. After the classifier has initialized a phase's entry with its first two epochs, performance and energy histories are updated automatically as the sprinting coordinator makes decisions at run-time.

Invoking the Predictor. The phase predictor forecasts the next epoch's phase, producing a phase ID. This ID indexes into the phase classifier's table to produce corresponding historical data for instruction throughput and processor energy, which serve as the predicted utility and cost from sprinting in the next epoch.

2.5 Thermal Tracker

The thermal tracker monitors power dissipation and thermal headroom to constrain sprint decisions. The tracker can measure power with live measurements or power models [2]. DynaSprint measurements drive a thermal model that calculates headroom, which quantifies the heat (measured in Joules) that can be expended before exceeding the processor package's thermal capacitance. When the package uses a phase change material [35], the thermal model uses the following parameters:

- C_{pcm}: The phase change material's thermal capacitance, which determines the heat that can be buffered.
- R_{pcm}: The phase change material's thermal resistance, which determines maximum power dissipated during a sprint.
- R_{package}: The processor package's thermal resistance, which determines how quickly heat transfers from the phase change material into the ambient after a sprint.
- R_{total}: The system's total thermal resistance, which determines the processor's maximum sustained power.

Let us denote the system's nominal power as P_n, its sprinting power as P_s, and its thermal design point as P_t. Denote the system's thermal headroom as H_t at time t. Suppose the system sprints from t_1 to t_2 without exhausting the thermal headroom and then operates in normal mode from t_2 to t_3. We can estimate the thermal headroom at these points in time.

$$\begin{aligned} H_2 &= H_1 - (P_s - P_t) \times (t_2 - t_1) \\ H_3 &= H_2 + (P_t - P_n) \times (t_3 - t_2) \end{aligned}$$

2.6 Sprinting Coordinator

The sprinting coordinator initiates and terminates sprints. The decision to sprint must balance utility in the present and the future, a trade-off encapsulated by two questions. First, how significant is utility in the next epoch U_{next} compared to potential utilities further into the future? Second, if the system sprints and reduces thermal headroom in the present, will it be able to sprint and exploit high-utility epochs in the near future? Answers to these questions would permit the system to use thermal headroom judiciously and enhance performance over the long run.

The coordinator pursues these objectives given utility predictions and thermal models for the next epoch. If the coordinator is aggressive and sprints when utility is low, it may deplete the system's thermal headroom that might have been helpful in the future. If the coordinator is conservative and does not sprint when utility is low, it may lose an opportunity to translate thermal headroom into performance.

Thermal Proportional Threshold (TP-T). We propose a policy, TP-T, that determines whether the coordinator initiates a sprint in the next epoch. Shown in Equation 1, TP-T sets a threshold for initiating a sprint $U_{\text{threshold}}$ by multiplying the fraction of the thermal headroom already consumed by the maximum utility U_{max} derived from recent sprints (*e.g.*, over the last ten epochs). The coordinator initiates a sprint if predicted utility exceeds the threshold and the system possesses sufficient thermal headroom.

$$U_{\text{threshold}} = U_{\text{max}} \times \frac{H_{\text{consumed}}}{H_{\text{total}}} \tag{1}$$

In effect, the coordinator treats thermal headroom as a resource whose value varies depending on the amount available and it treats utility as the return from spending that resource. As the amount of thermal headroom decreases, it becomes more valuable and the coordinator requires a higher return when consuming it.

3 CACHE SPRINTING ARCHITECTURE

We highlight the advantages of DynaSprint for cache sprints, an instance of a new class of techniques we call "microarchitectural capacity sprinting." Cache sprinting briefly expands the processor's last-level cache (LLC) capacity beyond what is dictated by the thermal design point. During normal operation, half of the LLC ways are powered off and unused. During a sprint, the remaining ways are powered on. When sprinting, the LLC dissipates more dynamic

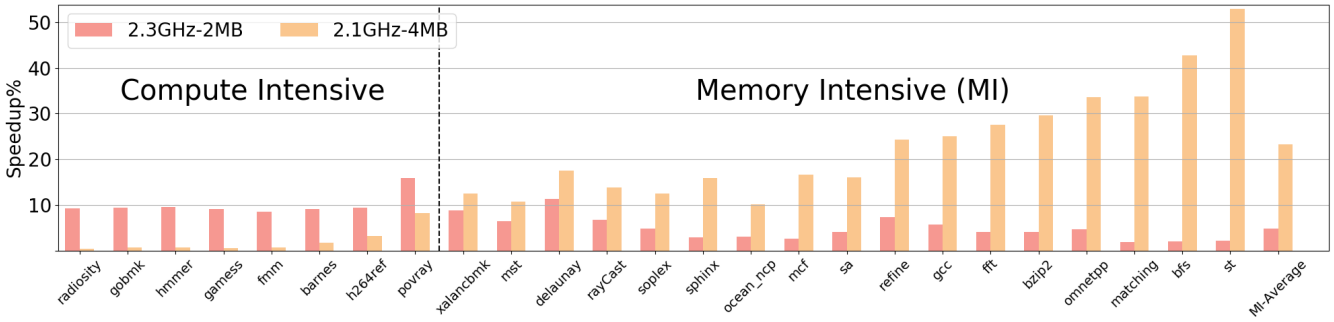

Figure 4: Performance comparison: scaling frequency vs increasing LLC capacity, baseline 2.1GHz with 2MB LLC

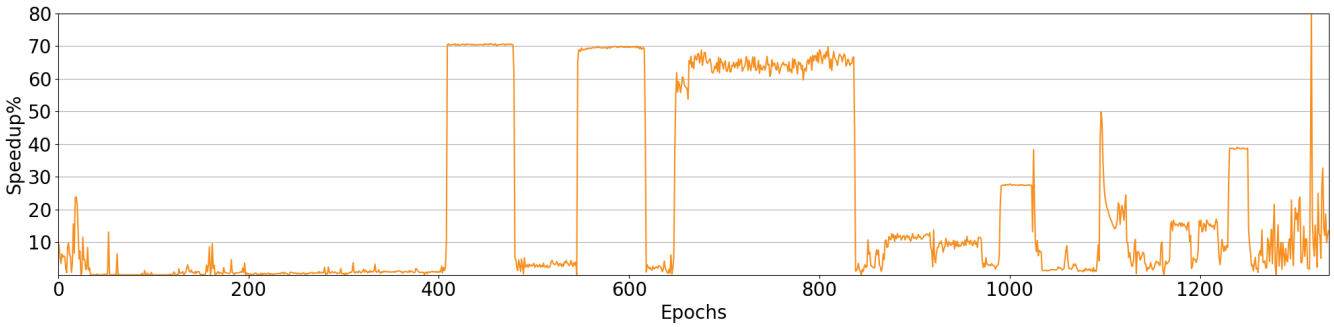

Figure 5: Performance gain: gcc-s04 running with 4MB LLC over baseline 2MB

power because each access must check twice the number of tags and the cache supplies more data in response to hits. The LLC also dissipates more static power.

3.1 Case for Cache Sprinting

Cache sprinting complements prior mechanisms in computational sprinting that activate additional cores and boost their frequencies. Sprinting with the last-level cache serves memory-bound applications that do not benefit from boosted processor cores.

High Performance Impact. Figure 4 shows the performance gains for two alternative sprint mechanisms that consume 2W of extra power—scaling frequency and increasing LLC capacity—on an Intel Xeon Broadwell processor. Scaling frequency from 2.1GHz to 2.3GHz improves performance for compute-intensive applications by 10% on average but has little effect for memory-intensive applications. In contrast, increasing cache capacity from 2MB to 4MB improves performance for memory-intensive applications significantly. Although the potential benefits of LLC sprints are significant, the management mechanism must exploit that potential.

Time-Varying Utility. Intelligent sprints make sense only if utility from cache capacity varies across phases, permitting sprints when utility is high and cooling when utility is low. This variance exists in most workloads and we present gcc as an example in Figure 5. The x-axis shows epochs of 100M instructions and the y-axis shows speedup with a 4MB LLC over 2MB LLC in each epoch. Utility varies from 0% (excellent times to cool) to 80%, with sustained periods of 70% (excellent times to sprint).

Low Power Density. LLC power density is much lower than that of processor cores used in prior sprint studies [47]. Cache sprints exhibit smaller power deltas due to the cache's lower power density and smaller difference between max and average power. Smaller deltas permit longer, more aggressive sprints before hitting thermal constraints. Although cache sprints generate extra heat, primarily due to static power, the LLC is unlikely to become a hotspot. Datapath structures, such as the register file and ALUs, exhibit much higher temperatures [26]. Thus, cache sprints will be constrained by the processor's global heat profile rather than any local hot spots.

Note that our framework generalizes beyond LLC sprints to other memory microarchitectures or technologies. For example, 3D-DRAM could offer hundreds of MB of LLC but would benefit from a sprint framework that exploits its huge capacity while managing its limited cooling capabilities.

3.2 System Architecture

DynaSprint is a software runtime system that controls cache sprints. Software decides when to sprint and communicates that decision to hardware via a control register. Although full hardware control is possible and permits decisions at finer granularities, software control permits more sophisticated management policies. The operating system can track information about each process's behavior and perform complex computation at coarse granularities to make informed sprinting decisions that reflect program behavior.

When a sprint starts, the additional cache ways are empty and fill over time. If the sprint is timely, the ways fill with useful data,

the hit rate increases, and performance improves. As the sprint progresses, it consumes the chip's thermal headroom by raising the processor's temperature toward safety tolerances or transforming the phase change material into an amorphous state. The sprint ends when the software finds little utility from extra cache capacity or the hardware finds that the thermal headroom has been exhausted.

Sprints must end with sufficient time and thermal headroom to write dirty data to memory before powering down. In an extreme case, every block in the additional ways is dirty and block addresses are distributed poorly across memory pages. For example, writing back 16MB of dirty data would require less than 500 microseconds, a negligible transition delay given sprints that span several seconds.

Although software can make decisions about utility, it cannot be trusted with the chip's physical safety. Hardware must monitor thermal headroom and halt sprints, even if it means overriding the operating system, when headroom is exhausted. Specifically, hardware must set the machine state register's (MSR's) sprint bit to zero and ignore writes to that register until thermal headroom has been restored so that the cache can sprint without jeopardizing thermal budgets. In the event of a thermal emergency, the hardware halts processor cores until cache ways are powered off.

4 EXPERIMENTAL METHODOLOGY

System Setup. We emulate LLC sprints on an off-the-shelf chip multiprocessor, restricting its LLC capacity in nominal mode and restoring its capacity in sprint mode. We focus our sprint evaluation on serial workloads running on a single core and size LLC capacity accordingly.

Table 1 summarizes the system configuration. We conduct experiments on physical hardware using an 8-core Intel Broadwell Xeon E5-2620 v4 processor. The Xeon processor has a 20MB, 20-way last-level cache, which corresponds to 2.5MB per core. In normal mode, we configure Intel's Cache Allocation Technology(CAT) [20] to restrict a core and its single-threaded program to use 2MB (not 2.5MB because CAT allocates cache capacity at one-way, 1MB granularity). In sprinting mode, we permit a core to use 4MB of cache.

We configure the system to minimize interference and ensure deterministic results. First, we disable C-states and DVFS, fixing all cores' frequencies to 2.1GHz. Second, we disable the Watchdog hang timer as well as the Address Space Layout Randomization. Third, we stop all non-essential system daemons/services. Fourth, we use cset command for CPU shielding so that other processes will not run on the core targeted for experiments. Finally, we run each application three times and use the average number for all reported performance values. We collect the relevant hardware counters' values for phase signatures, from Section 2.2, using Performance Application Programming Interface (PAPI) [48]. These numbers are collected every 100 million instructions using the PAPI_overflow() function.

Applications. We evaluate SPECCPU2006 [19], SPLASH2 [50] and PBBS [43] benchmark suites. We focus on the 17 LLC-sensitive applications from Figure 4, which exhibit higher performance gains from larger cache capacity than with processor frequency scaling. Applications in SPECCPU2006 and SPLASH2 are run to completion with the largest input size, which usually takes minutes. Applications in PBBS are much smaller and are run multiple times, with different inputs, until 100B instructions have been executed.

Table 1: System Specification

Field	Value
Core	8-core Broadwell, 2.1GHz
L1 Caches	32KB, private, split D/I, 8-way
L2 Caches	256KB, private, 8-way
L3 Cache	20MB, shared, 20-way Way-partitioning with Intel CAT [20]
Memory	8GB, RDIMM, DDR4 2400MT/s
OS	Red Hat Enterprise Linux 7.5 Linux kernel version 3.10.0

Power Model. We estimate our system's power consumption using Intel's Running Average Power Limit (RAPL) driver [2]. Static power is estimated by measuring a microbenchmark that only calls the sleep() function while dynamic power is measured during the execution (minus estimated static power).

On our system, the available RAPL domains only provide power for the package (cores + LLC + memory controller) as a whole. We use power breakdowns from prior work [10] to estimate the individual power of cores and LLC. In summary, we estimate the nominal and sprinting power of a scaled system (single core + 2MB LLC) to be 6W and 8W, respectively, in which LLC consumes about 1W per MB.

Thermal Model. We use a thermal model like those in prior sprint studies, which use PCM to increase thermal capacitance [34, 35]. We consider heat spreaders with low thermal resistance that take heat from the processor die and present a uniform temperature distribution to the heat sink [26]. We consider an amount of PCM that absorbs 10J of heat before melting completely, enabling a sprint spanning tens of seconds. Note that PCM parameters affect the run-time's thermal model because we emulate systems with lower TDPs that require intelligent sprints. The baseline system has a TDP of 7W for 1 core and 2MB LLC. We evaluate sensitivity to these parameters in Section 5.5.

5 EVALUATION

We evaluate DynaSprint and its thermal-proportional threshold policy against several alternatives. Since there is no prior work in managing LLC sprinting, we compare against three other policies that vary in their knowledge of sprint utility and thermal headroom.

Greedy (G) initiates a sprint whenever thermal headroom is available, ignoring utility. Once thermal headroom is exhausted (i.e., PCM transition to amorphous state completes), it waits for the system to cool and restore thermal headroom before starting the next sprint.

Local-Oracle (LO) has perfect knowledge of sprint utility from all prior epochs (U_{history}) as well as the next epoch (U_{next}). It initiates a sprint if U_{next} is greater than the average of $U_{history}$ and stops otherwise. LO is an oracular policy, which represents heuristics that only incorporate local information obtained through online profiling or prediction. The sprint threshold is set based on utility alone.

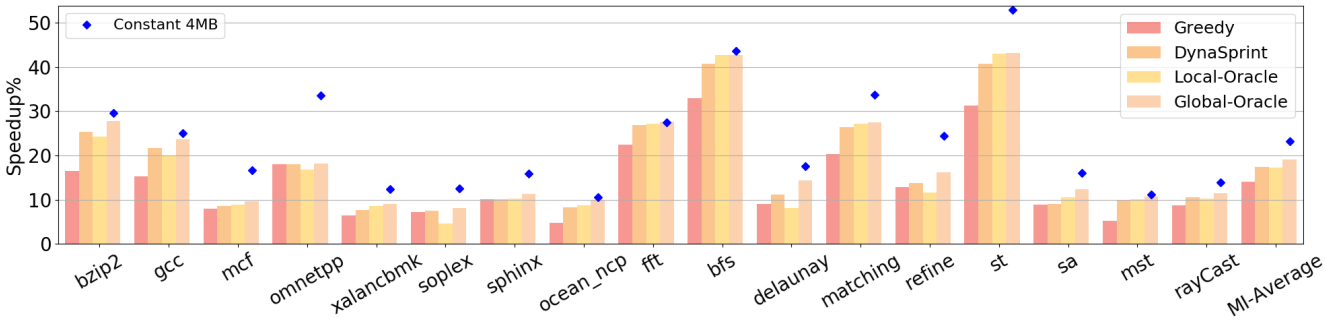

Figure 6: DynaSprint Performance: compared against various policies

Global-Oracle (GO) has perfect knowledge of sprint utility from all past and future epochs in the program's execution. It sorts epochs based on their sprint utilities and iteratively selects the epoch with the highest utility for sprinting execution. If the selected epoch cannot sprint due to lack of thermal headroom, the epoch is marked for normal execution and GO proceeds to the next epoch. GO uses global information obtained through offline profiling and cannot be implemented online. However, this policy provides an upper bound on performance from cache sprinting.

Thermal-Proportional Threshold (TP-T) predicts the sprinting utility of the next epoch (U_{next}) and tracks the maximum utility observed from a sprint in the recent past (U_{max}). Then it sets the sprint utility threshold ($U_{\text{threshold}}$) by multiplying the percentage of the already consumed thermal headroom and U_{max} as shown in Equation 1. Finally it initiates a sprint if U_{next} exceeds $U_{\text{threshold}}$.

5.1 Performance

Figure 6 shows performance gains from LLC sprints over a baseline that operates under nominal power. (The blue dots indicate upper bounds on performance given unlimited thermal headroom and a continuous 4MB sprint.) DynaSprint-guided LLC sprints improve performance by 17%, on average, and by up to 40%. When compared with alternatives, DynaSprint's TP-T policy outperforms greedy heuristics and is competitive with oracular policies.

Comparison against Greedy (G). TP-T significantly outperforms G for 8 of 17 benchmarks because its sprints are more judicious and timely. Figure 7(a) shows the probability density of sprint utility for three of these applications: gcc, ocean and bfs (one from each benchmark suite). Their sprint utility distribution often appears bimodal, revealing many epochs that benefit greatly from sprinting and many that do not. The large variance in sprint utility offers TP-T an opportunity to spend its limited thermal budget wisely to maximize performance gains. For instance, ocean frequently achieves large performance gains between 50 to 60% and also achieves small performance gain between 0 to 20%. TP-T is more likely to reserve its sprints for the large gains. Greedy, on the other hand, could waste its thermal budget in low-utility epochs and have insufficient headroom for high-utility ones.

Greedy performs comparably to TP-T for several benchmarks. Figure 7(b) shows the probability density of sprint utility for three of these applications. They often exhibit very small variance in sprint

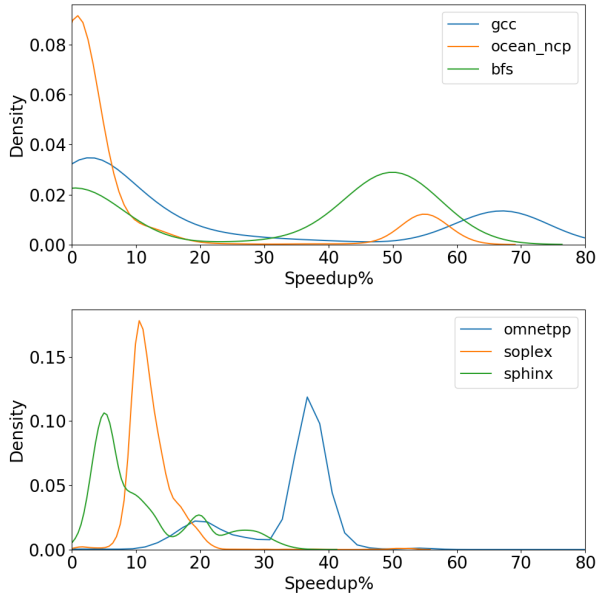

Figure 7: PDF for utility: (a) bimodal, (b) unimodal

utility and all epochs benefit similarly from sprinting. For example, most of soplex's epochs report utility between 10 to 20%. There is not much opportunity to prioritize sprints for high-utility epochs. As a result, TP-T's threshold will be lower than the average sprint utility for most epochs when thermal headroom is available. In effect, TP-T behaves very similarly to greedy.

Comparison against Local Oracle (LO). TP-T's performance is close to LO's. Indeed, TP-T performs better than LO for 7 of 17 benchmarks. Although LO has perfect knowledge of local utilities, its sprint thresholds neglect the system's current thermal conditions. As a result, LO can sprint too conservatively when thermal headroom is abundant and too aggressively when thermal headroom is scarce. TP-T is aware of thermal constraints and paces its sprints according to available headroom.

There are cases where LO outperforms TP-T and is close to GO (e.g., bfs, st) due to two reasons. First, because of their bimodal utility distribution, when LO sets sprint thresholds based on average

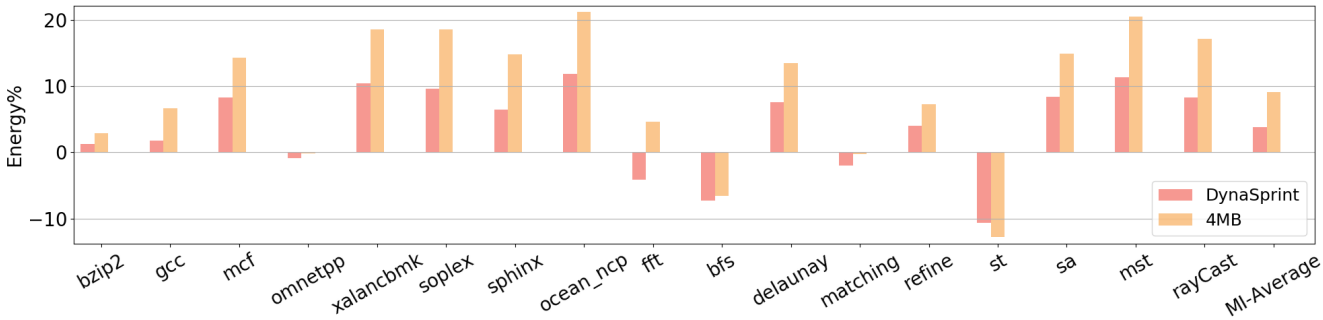

Figure 8: Energy: DynaSprint and 2.1GHz-4MB configurations relative to 2MB baseline

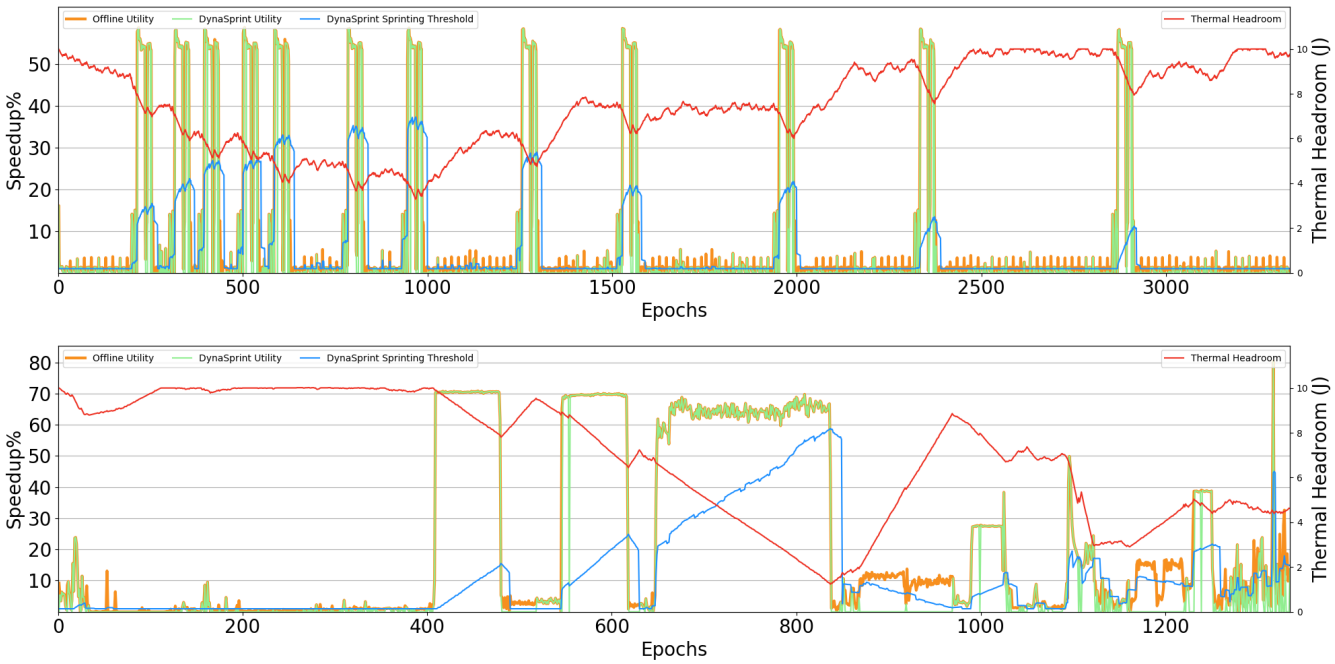

Figure 9: DynaSprint sprinting behavior: (a) ocean_ncp, (b) gcc-s04

of local utilities, the high-utility epochs will also exhibit utilities greater than the thresholds. Second, these applications tend to have a small number of recurring phases such that local information captures global behavior.

Comparison against Global Oracle (GO). On average, TP-T performs within 95% of the performance upper bound set by GO. GO has perfect knowledge of sprint utility for every epoch and perfect knowledge of the system's thermal condition. This oracular knowledge persists even as epochs are selected for normal or sprinting computation. The small performance gap between TP-T and GO indicates that DynaSprint accurately predicts sprint utility and that the TP-T thresholds are effective for pacing sprints to maximize performance gains over the long run.

5.2 Energy

Figure 8 shows DynaSprint's energy consumption compared with the baseline configuration that uses only 2MB of LLC. The energy overhead of the configuration that constantly uses 4MB of LLC is also plotted for reference. On average, DynaSprint increases energy by less than 5%. For several applications, such as bfs and st, DynaSprint even reduces energy by about 10% due to significant reductions in execution time. Although the goal of sprints is not improving energy efficiency but maximizing performance under thermal constraints, these results show that sprints can improve efficiency.

5.3 Sprinting and System Dynamics

Figure 9(a) shows system dynamics during the complete execution of ocean_ncp. The orange line plots offline profiles of sprint utility. The green line plots online measurements of sprint utility achieved

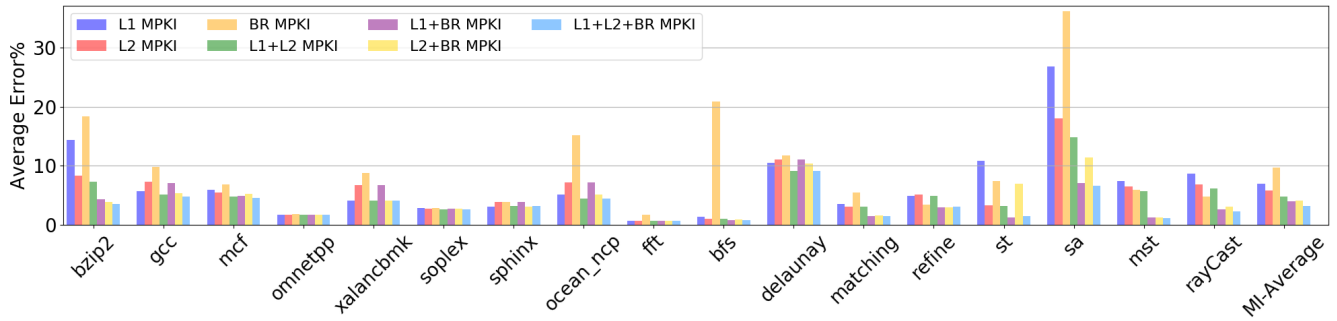

Figure 10: Impact of phase signature definition on phase classification accuracy

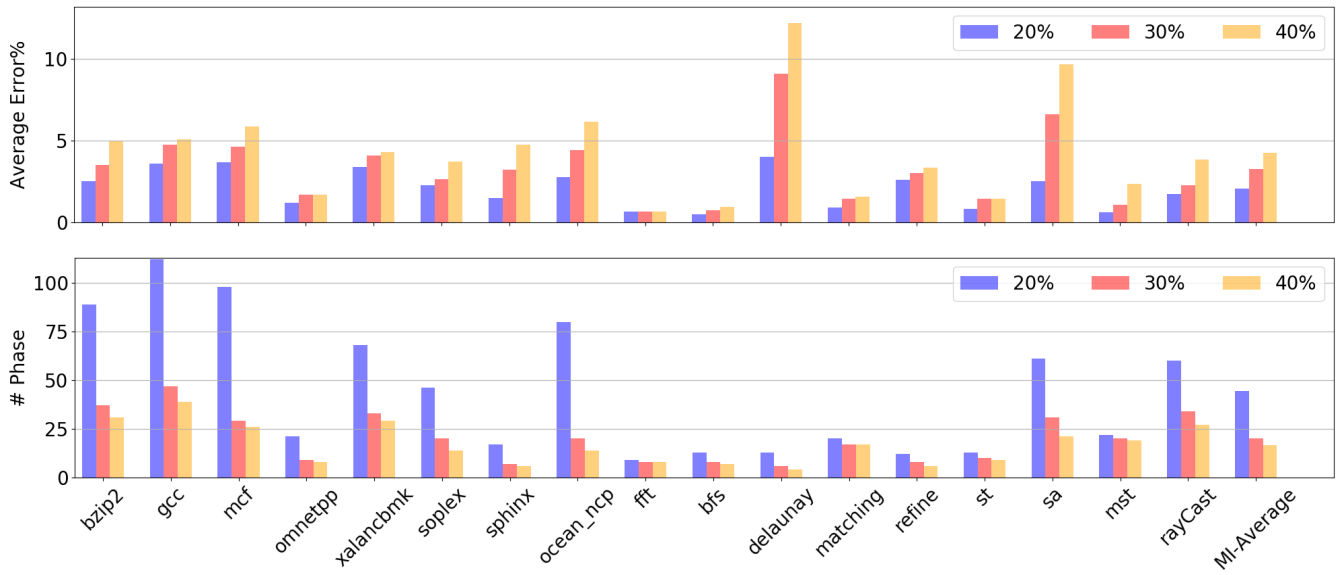

Figure 11: Impact of signature boundary on: (a) phase classification accuracy, (b) total number of phases

from DynaSprint's TP-T policy. Note that the orange and green lines align in most cases. The red line shows the available thermal headroom. The blue line shows DynaSprint's threshold for deciding to sprint.

The figure illustrates two major system behaviors. First, DynaSprint accurately identifies high-utility epochs and executes sprints during these times. For epochs that benefit most from sprints, the orange line overlaps with the green line. Second, DynaSprint dynamically adjusts its sprint threshold according to recently observed performance data and the available thermal headroom according to the TP-T policy.

Ocean_ncp is representative of applications that have a few highly-repetitive phases and have a bimodal utility distribution. DynaSprint can manage sprinting effectively for these applications because its phase classifier and predict can capture phase behaviors effectively. Moreover, its TP-T policy can easily identify high-utility epochs and trigger timely sprints.

Figure 9(b) shows the same system dynamics for a different type of application, gcc. Compared to ocean_ncp, gcc has many more phases and those phases are much less repetitive. Gcc is representative of applications that go through many different stages during their execution. This type of application is harder to manage as its less predictable. However, we see that DynaSprint still performs well, sprinting at most of the high-utility epochs and dynamically adapting the threshold to phases and thermal budget. The gap between TP-T and GO is mainly caused by phase and utility mispredictions.

5.4 Prediction Accuracy

Phase Classification. DynaSprint assigns epochs to phases as the first step in predicting utility, which is critical to the effectiveness of the TP-T policy. We measure accuracy by assigning an epoch to a phase tracked by the classifier and determining how closely that phase's performance predicts the epoch's performance. Error is the percentage difference between the phase's instruction throughput and the epoch's actual throughput. Two design elements affect accuracy:

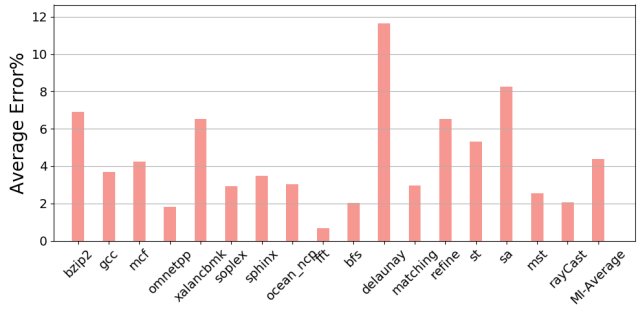

Figure 12: Utility prediction accuracy

the definition of the phase signature and the boundaries that define neighborhoods around phase signatures.

Figure 10 shows classification error when including varied hardware counters in the phase signature. Including more counters in the signature improves accuracy. A single counter can perform poorly (e.g., bzip2, sa) and a second counter significantly improves accuracy. Moreover, using diverse measures of activity improves accuracy. Among signatures defined by two counters, combining either L1 or L2 MPKI with BR MPKI performs better than combining L1 and L2 MPKI. L1 and L2 MPKI both characterize memory intensity while BR MPKI characterizes datapath activity. DynaSprint uses all three counters to define phase signatures that achieve high accuracy across all applications.

Figure 11(a) shows classification error when using varied boundaries (B_{rd}) around phase signatures. Smaller boundaries capture finer-grained phases and improve accuracy. However, Figure 11(b) shows that finer-grained phases increases the number of phases the classifier tracks, which increases overhead. DynaSprint sets $B_{rd} = 0.3$ to limit the number of phases yet accurately classify performance to within 97% of the actual value. Although narrowing boundaries ($B_{rd} = 0.20$) improves accuracy slightly, it significantly increases the number of phases for a few applications. On the other hand, broadening boundaries ($B_{rd} = 0.40$) noticeably reduces accuracy without much reducing the number of phases.

Utility Prediction. Figure 12 assesses how the phase classifier's accuracy translates into utility predictor's accuracy. Error is the percentage difference between predicted and actual speedups from a cache sprint where speedup is predicted from instruction throughputs reported by the classifier. Overall, DynaSprint accurately predicts utility and sprint speedups to within 95% of actual values.

Utility prediction accuracy exhibits larger variance across applications than phase classification accuracy. Some applications achieve higher accuracy because of their relatively stable phases and sprint utility (e.g., omnetpp and fft). But some applications are harder to predict because of high variance in their phases (e.g., bzip2) and sprint utility (e.g., xalancbmk). The delaunnay and sa benchmarks report the largest errors, for both phase classification and utility prediction, because they have many epochs with similar phase signatures but different performance profiles.

5.5 Sensitivity Analysis

Thermal Design Power (TDP). We have been evaluating a system that operates nominally at 6W, sprints at 8W, and is constrained by a 7W thermal design point. For this system, the cooling duration that restores the thermal headroom must match the sprint duration. Time spent generating heat at 1W above TDP must be matched by time spent dissipating heat at 1W below TDP; the sprint-to-cool ratio is 1:1.

Figure 13(a) presents performance under tighter thermal constraints where TDP is set to 6.5W, closer to nominal power. As the gap between nominal and thermal design power narrows, the system requires more time to cool after sprint. For our parameters, the cooling duration must be 3× longer than the sprinting duration; the sprint-to-cool ratio is 1:3. Sprints become more expensive, degrading sprint performance across all policies.

Costly sprints reduce the system's tolerance to poor decisions. TP-T's advantage over G grows under tighter thermal constraints (e.g., fft, bfs, mst). TP-T identifies high-utility epochs and sprints judiciously whereas G luckily sprints during high-utility epochs when thermal headroom was generous but suffers from poor decisions when headroom becomes scarce. As sprints become more expensive, prediction accuracy becomes more important and TP-T cannot compete wth oracular policies. TP-T underperforms LO, which benefits from perfect knowledge of sprint utility (e.g., bzip2, gcc).

Total Thermal Headroom (TTH). We have been evaluating a system with PCM that provides 10J of thermal headroom. Figure 13(b) presents performance when headroom is halved to 5J. Performance is unaffected for most applications. Headroom determines the duration of a full sprint. For example, 10J permits our system to sprint for tens of seconds. But when applications prefer sprints that exceed the full duration, headroom impacts performance less than TDP because it does not change the maximum sustainable sprint-to-cool ratio. As long as the application derives varied utilities from sprints across time, TP-T will exploit low-utility epochs for cooling and judiciously exploit thermal headroom by dynamically setting the thresholds for sprints.

Sprinting Intensity (SI). We have been evaluating a sprint that doubles LLC capacity from 2MB to 4MB. Figure 13(c) presents performance when sprints triple capacity to 6MB. More intense sprints do not necessarily translate into long-run performance and only three applications benefit. As sprint intensity increases to 6MB, sprint power increases to 10W and the maximum sustainable sprint-to-cool ratio falls to 1:3.

Figure 14 shows the time spent sprinting when a sprint expands cache capacity to 4MB or 6MB. Unless performance from 6MB is much greater than that from 4MB, the more intense sprint does not justify the extra power. Cache capacity suffers from diminishing marginal returns and most applications in our study do not benefit significantly more at 6MB compared to 4MB.

When designing a system for bimodal operation, sprint or normal, architects must understand applications' performance sensitivities to resource allocations. Ideally, the sprinting and normal modes are configured to maximize marginal performance gains given a marginal resource allocation. If applications' sensitivities vary, the system could support multiple sprint intensities. DynaSprint could easily

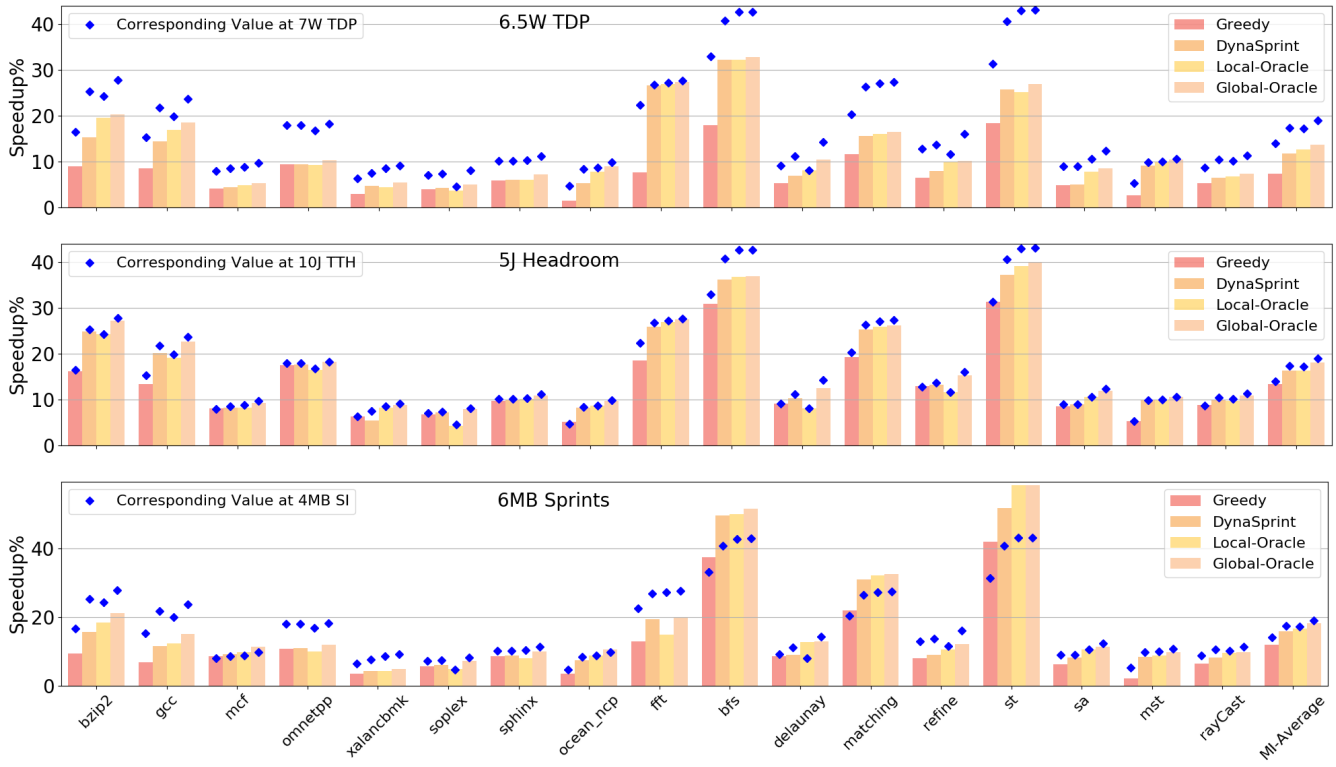

Figure 13: Performance sensitivity to (a) TDP (6.5W), (b) total thermal headroom (5J), (c) sprinting intensity (6MB)

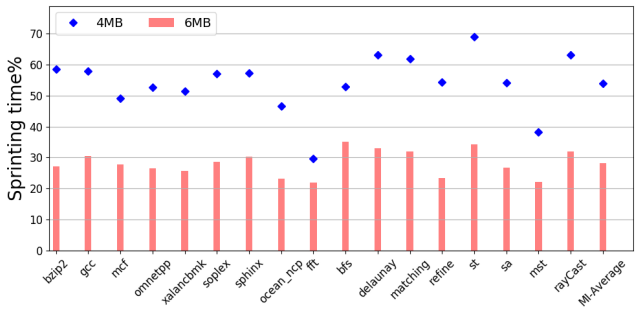

Figure 14: Impact of sprinting intensity on sprinting time

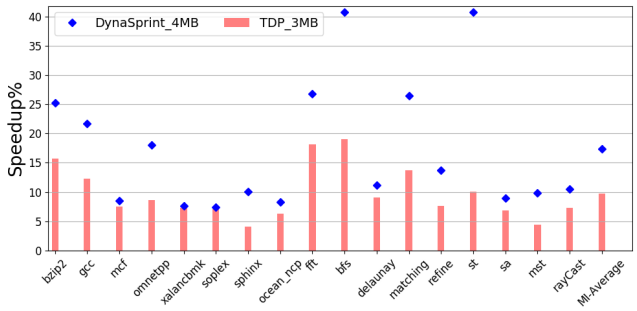

Figure 15: DynaSprint performance against perfect TDP

track performance histories for each sprint mode and application phase, deciding which mode improves performance most efficiently.

Comparison to Sustained Operation at TDP. We set our baseline, nominal power at 6W, which is 1W below the 7W thermal design point that dictates sustainable power draw. However, we also compare DynaSprint against an operating point that dissipates 7W, assuming that the processor uses power to operate exactly at the thermal design point. Because our models indicate that the system requires 1W of power per 1MB of cache capacity, sustained operation at 7W permits the sustained use of a 3MB LLC.

Figure 15 compares performance from sprinting (dynamic 2MB and 4MB modes) and sustained operation at TDP (static 3MB mode) relative to a baseline 2MB mode. For applications that exhibit high variance in sprint utility, DynaSprint significantly outperforms sustained operation at TDP. For applications that exhibit low variance, performance depends on application sensitivity to cache size. Some applications (*e.g.*, omnetpp) benefit more from a cache size much larger than permissible under TDP and prefer alternating between sprint and nominal modes. Others (*e.g.*, xalancbmk and soplex) are insensitive to cache sizes between 2MB and 4MB such that sustained operation with 3MB matches the performance of sprint-and-cool.

5.6 Discussion

Multi-core Implications. Although we only evaluate single-core applications in this paper, DynaSprint can naturally be extended to manage multi-program sprints. The phase predictor should remain effective because the framework can collect per-process performance counters. However, a program's sprinting strategy should account for system dynamics (i.e., other programs' decisions). Each program can request sprints based on its own utility, but the system grants sprints based on system performance and fairness metrics.

Multi-resource Implications. While we focus on managing sprints with a single resource (i.e., last-level cache)in this paper, DynaSprint can also be extended to manage sprints with multiple resource types. For example, DynaSprint could assess the compute and memory intensities within and across applications, deciding whether to boost core frequency or expand last-level cache capacity. DynaSprint could also apply various techniques to reduce the management overhead. For instance, if DynaSprint already knows that an epoch has high utility with cache sprinting, it should not boost core frequency because these two mechnisms complement each other.

6 RELATED WORK

Computational sprinting has been applied to datacenters [17, 44, 53]. At such scale, power constrains not only the processor chip but also the servers and clusters that share a power supply. Uncoordinated sprints risk tripping the circuit breakers. Although batteries can supply power to complete computation during power emergencies, future sprints are forbidden until batteries recharge.

Management policies have been proposed to partition the last-level cache to minimize interference [25, 28, 55], maximize system throughput [9, 15, 33], or improve fairness [27, 31, 49, 52]. Performance-centric partitioning mechanisms often construct miss rate curves (MRC), which characterize the miss rate as a function of the cache allocation. Utility-based Cache Partitioning (UCP) [33] uses simple hardware monitors to estimate the MRC, relying on the stack property of the LRU replacement policy [30]. Ubik [25] uses an analytical model that relies on transient behavior being analyzable and makes assumptions about future cache microarchitecture. Prior work has also estimated the MRC in software [8, 36, 37, 46], tracing memory addresses and using analytical models [8, 14]. Unfortunately, tracing memory addresses is expensive and introduces significant slowdowns over native execution.

Many hardware design techniques have been proposed to dynamically resize caches [4, 6, 18, 32]. These techniques focus on reducing leakage power to save energy subject to constraints on performance degradation. In contrast, sprinting seeks to maximize long-term performance gains under thermal constraints and must assess trade-offs between sprinting now and sprinting in the future.

Most dynamic phase analysis techniques observe that performance is strongly correlated with executed code [11, 12, 38, 40, 41]. These techniques often collect some form of *execution frequency vectors* (EFV) that identify the code executed at some point in time. Prior works have constructed EFVs with instructions [11, 38, 40] or basic blocks [12, 41]. Unfortunately, EFVs are expensive because they record multiple samples of instruction addresses for accuracy. Because of performance counter skid [3], recording the precise instruction address requires tools like Intel PEBS [1] which introduce additional overhead [38].

Prior approaches have used dynamic phase analysis to predict future application behavior [13, 23, 24, 39, 42, 54]. Observing that phases recur, prior studies use a table-based history predictor to capture past phase patterns [13, 24]. These techniques often tailor phase definitions and boundaries for a specific optimization. For example, prior work define phases based on the ratio of memory bus transactions to micro-ops and statically determines phase boundaries to guide DVFS [24]. More sophisticated dynamic resource management requires more careful phase definition.

7 CONCLUSION

We propose DynaSprint, a software runtime system that manages sprints intelligently based on the application's sprint utility and the system's thermal headroom. We also propose cache sprinting, which dynamically allocates last-level cache capacity. With a modest amount of thermal headroom provided by phase change materials, DynaSprint-guided cache sprinting can improve performance by 17% on average and up to 40% over a non-sprinting system. Moreover, the system performs within 95% of a globally optimized oracular policy.

ACKNOWLEDGMENTS

This work is supported by the Semiconductor Research Corporation's Global Research Collaboration (GRC) program under task 2821.001. This work is also partly supported by National Science Foundation grants CCF-1149252, CCF-1337215, SHF-1527610, and AF-1408784. 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.

REFERENCES

[1] [n. d.]. Intel 64 and IA-32 Architectures software developer's manual.
[2] [n. d.]. Intel Running Average Power Limit. url:https://01.org/blogs/2014/running-average-power-limit-rapl.
[3] [n. d.]. Intel VTune Amplifier 2018 User's Guide. url:https://software.intel.com/en-us/vtune-amplifier-help-hardware-event-skid.
[4] David H. Albonesi. 1999. Selective Cache Ways: On-demand Cache Resource Allocation. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32)*. IEEE Computer Society, Washington, DC, USA, 248–259. http://dl.acm.org/citation.cfm?id=320080.320119
[5] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. 2003. Dynamically Tuning Processor Resources with Adaptive Processing. *Computer* 36, 12 (Dec. 2003), 49–58. https://doi.org/10.1109/MC.2003.1250883
[6] Rajeev Balasubramonian, David H. Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. 2000. Dynamic Memory Hierarchy Performance Optimization.
[7] Frank Bellosa. 2000. The Benefits of Event: Driven Energy Accounting in Power-sensitive Systems. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System (EW 9)*. ACM, New York, NY, USA, 37–42. https://doi.org/10.1145/566726.566736
[8] Erik Berg and Erik Hagersten. 2005. Fast Data-locality Profiling of Native Execution. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*. ACM, New York, NY, USA, 169–180. https://doi.org/10.1145/1064212.1064232
[9] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal Cache Partition-Sharing. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP) (ICPP '15)*. IEEE Computer Society, Washington, DC, USA, 749–758. https://doi.org/10.1109/ICPP.2015.84

[10] Hsiang-Yun Cheng, Jia Zhan, Jishen Zhao, Yuan Xie, Jack Sampson, and Mary Jane Irwin. 2015. Core vs. Uncore: The Heart of Darkness. In *Proceedings of the 52Nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, Article 121, 6 pages. https://doi.org/10.1145/2744769.2747916

[11] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*. IEEE Computer Society, Washington, DC, USA, 233–244. http://dl.acm.org/citation.cfm?id=545215.545241

[12] Ashutosh S. Dhodapkar and James E. Smith. 2003. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 217–. http://dl.acm.org/citation.cfm?id=956417.956539

[13] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. 2003. Characterizing and Predicting Program Behavior and Its Variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*. IEEE Computer Society, Washington, DC, USA, 220–. http://dl.acm.org/citation.cfm?id=942806.943853

[14] D. Eklov and E. Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 55–65. https://doi.org/10.1109/ISPASS.2010.5452069

[15] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multi-cores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 104–117. https://doi.org/10.1109/HPCA.2018.00019

[16] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108

[17] Songchun Fan, Seyed Majid Zahedi, and Benjamin C. Lee. 2016. The Computational Sprinting Game. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 561–575. https://doi.org/10.1145/2872362.2872383

[18] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. 2002. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA '02)*. IEEE Computer Society, Washington, DC, USA, 148–157. http://dl.acm.org/citation.cfm?id=545215.545232

[19] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[20] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 657–668. https://doi.org/10.1109/HPCA.2016.7446102

[21] Michael C. Huang, Jose Renau, and Josep Torrellas. 2003. Positional Adaptation of Processors: Application to Energy Reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/859618.859637

[22] C. J. Hughes, J. Srinivasan, and S. V. Adve. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. 250–261. https://doi.org/10.1109/MICRO.2001.991123

[23] Canturk Isci, Alper Buyuktosunoglu, and Margaret Martonosi. 2005. Long-Term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro* 25, 5 (Sept. 2005), 39–51. https://doi.org/10.1109/MM.2005.93

[24] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. 2006. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 359–370. https://doi.org/10.1109/MICRO.2006.30

[25] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient Cache Sharing with Strict Qos for Latency-critical Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 729–742. https://doi.org/10.1145/2541940.2541944

[26] Y. Li, B. Lee, D. Brooks, , and K. Skadron. 2006. CMP design space exploration subject to physical constraints. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* 17–28. https://doi.org/10.1109/HPCA.2006.1598109

[27] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 367–378. https://doi.org/10.1109/HPCA.2008.4658653

[28] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 450–462. https://doi.org/10.1145/2749469.2749475

[29] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. 2009. Dynamic Performance Tuning for Speculative Threads. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 462–473. https://doi.org/10.1145/1555754.1555812

[30] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117. https://doi.org/10.1147/sj.92.0078

[31] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. 2009. FlexDCP: A QoS Framework for CMP Architectures. *SIGOPS Oper. Syst. Rev.* 43, 2 (April 2009), 86–96. https://doi.org/10.1145/1531793.1531806

[32] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. 2000. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED '00)*. ACM, New York, NY, USA, 90–95. https://doi.org/10.1145/344166.344526

[33] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 423–432. https://doi.org/10.1109/MICRO.2006.49

[34] Arun Raghavan, Laurel Emurian, Lei Shao, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M.K. Martin. 2013. Computational Sprinting on a Hardware/Software Testbed. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 155–166. https://doi.org/10.1145/2451116.2451135

[35] 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 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6169031

[36] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/1854273.1854286

[37] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. 2012. Phase Guided Profiling for Fast Cache Modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 175–185. https://doi.org/10.1145/2259016.2259040

[38] A. Sembrant, D. Eklov, and E. Hagersten. 2011. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 104–115. https://doi.org/10.1109/IISWC.2011.6114207

[39] Xipeng Shen, Yutao Zhong, and Chen Ding. 2004. Locality Phase Prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/1024393.1024414

[40] T. Sherwood, E. Perelman, and B. Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. 3–14. https://doi.org/10.1109/PACT.2001.953283

[41] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 45–57. https://doi.org/10.1145/605397.605403

[42] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*. ACM, New York, NY, USA, 336–349. https://doi.org/10.1145/859618.859657

[43] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, New York, NY, USA, 68–70. https://doi.org/10.1145/2312005.2312018

[44] 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 '15)*. ACM, New York, NY, USA, 439–449. https://doi.org/10.1145/2749469.2749474

[45] K. Skadron, M. R. Stan, W. Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and D. Tarjan. 2003. Temperature-aware microarchitecture. In *30th*

Annual International Symposium on Computer Architecture, 2003. Proceedings. 2–13. https://doi.org/10.1109/ISCA.2003.1206984

[46] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. 2009. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/1508244.1508259

[47] M. B. Taylor. 2012. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*. 1131–1136.

[48] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–173.

[49] Xiaodong Wang and José F. Martínez. 2016. ReBudget: Trading Off Efficiency vs. Fairness in Market-Based Multicore Resource Allocation via Runtime Budget Reassignment. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 19–32. https://doi.org/10.1145/2872362.2872382

[50] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/223982.223990

[51] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. 2005. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Washington, DC, USA, 271–282. https://doi.org/10.1109/MICRO.2005.7

[52] S. Zahedi and B. Lee. 2014. REF: Resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[53] W. Zheng and X. Wang. 2015. Data Center Sprinting: Enabling Computational Sprinting at the Data Center Level. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 175–184. https://doi.org/10.1109/ICDCS.2015.26

[54] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/1024393.1024415

[55] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 33–47. https://doi.org/10.1145/2872362.2872394