A Flexible Framework for Throttling-Enabled Multicore Management (TEMM)

Xiao Zhang, Rongrong Zhong, Sandhya Dwarkadas, and Kai Shen
Department of Computer Science, University of Rochester
Email: {xiao, rzhong, sandhya, kshen}@cs.rochester.edu

***Abstract*— Hardware execution throttling mechanisms such as duty cycle modulation and voltage/frequency scaling can effectively control core or chip-level resource consumption and hence have been advocated to manage multicore resource competition. However, finding the right throttle setting is challenging since the configuration space grows exponentially as the number of cores increases, making the naive approach of exhaustive search untenable. This paper proposes a flexible framework for Throttling-Enabled Multicore Management (TEMM) that efficiently finds a high-quality hardware execution throttling configuration for a user-specified resource management objective. In a manner similar to the Newton-Raphson method in numerical analysis, TEMM employs an iterative method to continuously improve the configuration search quality by leveraging the search results from previous iterations. Within each iteration, TEMM extrapolates the effects of throttling from reference configurations, searches for a high-quality throttling configuration based on model predictions (accelerated by hill climbing), sample-runs the selected configuration, and adds the measured performance and recorded execution statistics of interest as a new reference. Our evaluations show TEMM can quickly arrive at the exact or close to optimal throttling configuration.**

I. Introduction

Today's server markets are increasingly turning toward consolidation of resources using, for example, virtualization in cloud computing systems. With the dominance of multicore chips in today's cloud computing systems, performance isolation and quality of service (QoS) for the resulting multi-programmed workloads has become an increasing challenge. Largely due to contention for shared chip-level resources like the last-level cache and the off-chip memory bandwidth, a process may exhibit unexpected low performance because other simultaneously executing processes monopolize the shared resource. Malicious users could also take advantage of this system vulnerability to launch chip-level denial-of-service attacks [1].

Recent studies [2], [3], [4] advocated multicore resource management using hardware execution throttling. Specifically, commodity processors are deployed with mechanisms such as duty cycle modulation and dynamic voltage and frequency scaling (DVFS), originally designed for power/thermal management [5], to slow down (throttle) execution speed. By throttling down the execution speed of some of the cores, we can control an application's relative resource utilization to achieve a desired fairness or other quality-of-service objective. Execution throttling enables much finer-grain resource control compared to alternatives like page-coloring-based cache partitioning [6], [7], [8], [9] and scheduling quantum adjustment [10]. It also does not suffer from page coloring's problems of high recoloring costs and artificial memory pressure.

Despite the promises of execution throttling-enabled multicore management, identifying the appropriate throttling configuration (duty-cycle level and frequency) for a given resource management objective is challenging. First, execution throttling affects resource allocation indirectly and a throttling configuration may not obviously map to specific management objectives, including fairness, quality-of-service, overall performance, and power efficiency. Second, the space of possible throttling configurations grows exponentially with the number of CPU cores (for instance, eight duty-cycle levels per core allows 8^4=4096 throttling choices on a quad-core machine and $8^{12}\approx$69 billion choices on a 12-core machine). Searching for a high-quality configuration in a large, multi-dimensional space is challenging.

This paper presents a software system framework, called *TEMM*, for Throttling-Enabled Multicore Management. TEMM can automatically and quickly determine an optimal (or close to optimal) hardware throttling configuration given a user-specified service-level objective (SLO). The SLO could be an unfairness bound that specifies equal or proportional progress for concurrently executing programs, an absolute performance guarantee for a particular application, or a guaranteed resource allocation. TEMM models the effects of execution throttling from reference configurations, searches for a high-quality configuration based on model predictions, and iteratively refines the search with a broadening set of measured references. To enable online deployment with low overhead, we further develop a hill-climbing optimization to accelerate configuration search without exhaustive checking.

II. Background

A. Hardware Execution Throttling Mechanisms

Duty cycle modulation [5] is a hardware feature introduced for thermal management on Intel processors. The operating

Work was done while the first two authors were at the University of Rochester. Zhang is currently affiliated with Google and Zhong is currently affiliated with Yahoo!. This work was supported in part by the National Science Foundation (NSF) grants CNS-0834451, CCF-0702505, CNS-0615139, CCF-0937571, CNS-0509270, and CCF-1016902. Shen was also supported by a Google Research Award.

system can specify the fraction (as a multiplier of 1/8 or 1/16) of total CPU cycles during which the CPU is on duty, i.e., executing, by writing to the logical processor's IA32_CLOCK_MODULATION register. The processor is effectively halted during non-duty cycles for a duration of ∼3 microseconds [11]. Different fractional duty cycle ratios are achieved by keeping the time for which the processor is halted at a constant duration of ∼3 microseconds and adjusting the time period for which the processor is enabled. The microsecond granularity of duty cycle modulation ensures that memory bandwidth utilization is reduced, since any backlog of requests is drained quickly, so that no memory and cache requests are made for most of the 3 microsecond non-duty cycle duration. Thus, duty cycle modulation has a direct influence on memory bandwidth consumption [2] and can be used to control resource utilization. Duty cycle modulation can be applied on a per-core basis and has been used to simulate an asymmetric CMP [12] or artificially slow down application execution speed to measure its cache miss ratio curve [13] on multicore processors.

Dynamic voltage/frequency scaling (DVFS) is mainly designed for power management purposes. Since most current processors use off-chip voltage regulators (or a single on-chip regulator for all cores), they require that all sibling cores be set to the same voltage level. Therefore, a single frequency setting applies to the entire multicore chip on Intel processors [14], [15]. Compared to duty cycle modulation, DVFS is less effective at throttling memory bandwidth utilization since it operates only on the CPU and not on memory. The effect of DVFS is that throttled cores slow their rate of computation at a fine per-cycle granularity, although outstanding memory accesses continue to be processed at regular speed. On applications with high demand for memory bandwidth, the resulting effect is that of matching processor speed to memory bandwidth rather than that of throttling memory bandwidth utilization.

Re-configuring duty cycle or voltage/frequency level requires manipulation of platform-specific registers, which incurs very small overhead (about hundreds of cycles on our experimental platforms).

B. Resource Management Objectives

This paper presents a flexible multicore resource management framework that can support a variety of objectives for fairness, quality-of-service, overall performance, and power optimization. Here we describe two specific examples of service-level objectives (SLOs) for multicore resource management:

- The *fairness-centric* objective specifies roughly equal performance progress among multiple applications. We are aware that there are several possible definitions of fair use of shared resources [16]. The particular choice of fairness measure should not affect the main purpose of our work. We take fairness as equal performance degradation compared to a standalone run for the application. Based on this fairness goal, we define an *unfairness factor* metric as the coefficient of variation (standard deviation divided by the mean) of all applications' performance normalized to that of their individual standalone run. At perfect fairness, the unfairness factor should be zero.
- The *QoS-centric* objective specifies a guarantee of a certain level of performance to a high priority application. In this case, we call this application the *QoS application* and we call the CPU core that the QoS application runs on the *QoS core*.

Given one such service-level objective, the best configuration should maximize performance (or power efficiency in the case of DVFS) while satisfying the objective. In the rare case that no possible configuration can meet the objective, we deem the closest one as the best. For example, for configurations $\mathcal{C}_1$ and $\mathcal{C}_2$, if both $\mathcal{C}_1$ and $\mathcal{C}_2$ meet the objective, but $\mathcal{C}_1$ has better performance than $\mathcal{C}_2$ does, then we deem $\mathcal{C}_1$ to be a better configuration than $\mathcal{C}_2$. Also, if neither of the two configurations meet the objective but $\mathcal{C}_1$ is closer to the target than $\mathcal{C}_2$, then we deem $\mathcal{C}_1$ to be a better configuration than $\mathcal{C}_2$.

III. Configuration Search Using Model-Driven Iterative Refinement

A multicore machine allows different combinations of throttling levels at its CPU cores and each such throttling configuration affects cross-core relative utilization of shared resources in a certain way. The foundation for TEMM is a search method that identifies a high-quality throttling configuration that meets a specified SLO while achieving high performance or power efficiency. TEMM's configuration search employs an iterative refinement framework using performance models of execution throttling mechanisms.

A. Iterative Refinement

TEMM's configuration search approach is in part motivated by the Newton-Raphson method in numerical analysis. To find an approximate root to a function, the Newton-Raphson method iteratively identifies approximations using sampled function points. While each iteration is driven by an inaccurate linear tangent line from a previously sampled function point (or a reference), the approximation quickly becomes more accurate as sampled reference function points move closer to a real root.

Specifically, our approach identifies a high-quality throttling configuration by iteratively repeating the following routine. At each iteration:

- a reference-based *throttling performance model* is used to estimate per-core performance and calculates whole-system performance/fairness/power metrics for each candidate throttling configuration;
- a "best" configuration is chosen based on the model-estimated performance metrics and the SLO;
- the system is run using the selected configuration for a sampling duration; the new sample is added to the reference base.

The search continuously improves over iterations because we use the results from measuring/sampling the performance and hardware execution statistics of the selected configuration at each iteration to improve the throttling performance model at the next iteration. Specifically, our throttling performance models are based on *references*—previously executed configuration samples. By leveraging the measured statistics at references, rather than directly predicting performance, the model needs to estimate only the difference between the target configuration and a reference. The closer the target and reference are, the easier it is to model their difference accurately. Since each iteration adds a new reference in the neighborhood of some high-quality throttling configuration, the throttling performance model is improved for future iterations with better search results.

The iterative refinement maintains a broadening set of measured references (we call it the *reference set*). The refinement ends when a predicted best configuration is already a previously executed configuration sample in the reference set (and therefore would not lead to a growth of the reference set or better configuration search). In some cases, such an ending condition may lead to too many configuration samples with associated cost and little benefit. To maintain stability, we introduce an early ending condition so that refinement stops when no better configurations (as defined in Section II-B) are identified after several steps.

B. Reference-Based Throttling Performance Model

Recall that each iteration of our approach utilizes a reference-based throttling performance model. While reference-based performance models exist for other complex systems such as the I/O system [17], the modeling of CPU throttling configuration differences requires new methods. We present our solutions for two throttling mechanisms, as well as an approach to predict the performance of a hybrid configuration involving both mechanisms. We use cycles-per-instruction (CPI) [1] as a performance indicator to guide runtime throttling and use normalized execution time or throughput (normalized to running-alone performance) for final SLO evaluation.

1) Duty Cycle Modulation: We consider an n-*core* system and each core hosts a computation intensive application. Our model utilizes performance at a set of reference configurations as input. At a minimum, the set contains $n+1$ configurations—n single-core running-alone configurations (i.e., ideal performance) [2] and a configuration of all cores running at full speed (i.e., default performance without any throttling). More reference sample configurations may become available as the iterative refinement progresses.

We represent a throttling configuration as $s = (s_1, s_2, ..., s_n)$ where s_i's correspond to individual cores' duty-cycle levels. We collect the CPI of each running application and calculate app_i's (the application running of core i) normalized performance, P_i^s. P_i^s is the ratio between the CPI when app_i runs alone (without resource contention and at full speed) and its CPI when running at configuration s.

Generally speaking, an application will suffer more resource contention if its sibling cores run at higher speed. To quantify this, we define the *sibling pressure* of application app_i under configuration $s = (s_1, s_2, ..., s_n)$ as $\mathcal{B}_i^s = \sum_{j=1, j\neq i}^{n} s_j$. We assume an application's performance degrades linearly with respect to its *sibling pressure*. Under this assumption, the linear coefficient k can be approximated as:

$$k = \frac{P_i^{ideal} - P_i^{default}}{\mathcal{B}_i^{ideal} - \mathcal{B}_i^{default}}, \tag{1}$$

where P_i^{ideal} is app_i's performance under ideal conditions (i.e., running alone at full speed without duty cycle modulation), and $P_i^{default}$ is app_i's performance when all sibling applications and app_i run at full speed with no duty cycle modulation. Since ideal is app_i running alone, $\mathcal{B}_i^{ideal}$ equals 0.

For a given configuration $t = (t_1, t_2, ..., t_n)$ that is the target of performance prediction, we need to choose a reference configuration $r = (r_1, r_2, ..., r_n)$ that is the closest to t in our reference set. We introduce *sibling Manhattan distance* between configuration r and t w.r.t app_i as:

$$D_i(r, t) = \sum_{j=1, j\neq i}^{n} |r_j - t_j|. \tag{2}$$

The closest reference r would be one with minimum such distance.

Ignoring changes in sibling pressure, we assume the application's performance is linear to its duty-cycle level. In order to incorporate the effect of sibling pressure changes, we apply the linear coefficient k to the sibling Manhattan distances. Hence, app_i's performance under configuration t can be estimated as:

$$E(P_i^t) = P_i^r \cdot \frac{t_i}{r_i} + k \cdot (\mathcal{B}_i^t - \mathcal{B}_i^r). \tag{3}$$

Equation (3) assumes that an application's performance is affected by two main factors: the duty-cycle level of the application itself and *sibling pressure* from its sibling cores. The first part of the equation assumes a linear relationship between the application's performance and its duty-cycle level. The second part assumes that performance degradation caused by inter-core resource contention is linear to the sum of duty-cycle levels of sibling cores. While these assumptions may not be precise (just like the inaccurate linear tangent line used in each step of the Newton-Raphson numerical analysis method), they are good approximations when the target and reference configurations are similar (small Manhattan distance). Our iterative refinement utilizes the reference-based model to continuously sample more references in the neighborhood of high-quality configurations and consequently allow better reference-

[1] Cycles are captured by the CPU_CLK_UNHALTED.REF performance counter, which uses a fixed reference frequency that is invariant to CPU frequency change to count cycles.

[2] We require running-alone performance since normalized performance is used in our fairness and QoS objectives. If the SLO is an absolute performance target that does not rely on run-alone performance, this configuration would not be necessary.

based configuration search in later iterations.

2) Voltage/Frequency Scaling: We use a simple frequency-to-performance model that we devised in previous work [18]. Specifically, it assumes that the execution time is dominated by memory and cache access latencies, and accesses to off-chip memory are not affected by frequency scaling while on-chip cache access latencies are linearly scaled with the CPU frequency. Let F be the maximum frequency and f a scaled frequency, and $T(F)$ and $T(f)$ be execution times of an application when the CPU runs at frequency F and f. The performance at f (normalized to running at the full frequency F) is defined as:

$$\frac{T(F)}{T(f)} = \frac{L_{\text{cache}} + R_{\text{F}} \cdot L_{\text{memory}}}{\frac{F}{f} \cdot L_{\text{cache}} + R_{\text{f}} \cdot L_{\text{memory}}} \quad (4)$$

L_{cache} and L_{memory} are access latencies to the cache and memory respectively measured at full speed, which we assume are platform-specific constants. R_{f} and R_{F} are run-time cache miss ratios measured by performance counters at frequency f and F. Since DVFS is applied to the whole chip, shared cache space competition among sibling cores on the same chip can be assumed to change very little under DVFS. We therefore assume R_{F} equals R_{f} as long as all cores' duty cycle configurations are the same for two different runs.

3) A Hybrid Model: Our approach requires finding a reference configuration to estimate the normalized performance of a target configuration. After adding DVFS, we have two components (duty cycle and DVFS) in a configuration setting. Thus, when we pick a closest reference configuration, we first find the set of samples with the closest DVFS configuration on app_i, then we pick the one with minimum *sibling Manhattan distance* on the duty cycle configuration. When we estimate the performance of the target, if the reference has the same DVFS settings as the target, the estimation is exactly the same as Equation (3). Otherwise, we first estimate the reference's performance at the target's DVFS settings using Equation (4), and then use the estimated reference performance to predict the performance at the target configuration.

C. Hill Climbing-Based Search

Our throttling performance model in Section III-B can estimate the performance at any duty cycle configuration. At each iteration of our configuration search, we can apply the model to all possible configurations and choose the best according to the desired SLO. However, such an exhaustive check is not scalable as an n-core system with a maximum of m throttling levels per core allows m^n possible configurations. On the 2.27 GHz CPU of our test platform, it takes about 10 microseconds to estimate the performance of a configuration. Applying the model on 8^4 configurations (quad-core platform) would lead to an excessive cost of about 41 milliseconds while applying it on 8^{12} configurations (12-core platform) is clearly infeasible.

To reduce computation overhead, we apply a hill climbing algorithm to prune the m^n search space. Using our quad-core Nehalem platform as an example, assuming we are currently at a configuration (x, y, z, u), we calculate (or fork) 4 children configurations: $(x-1, y, z, u)$, $(x, y-1, z, u)$, $(x, y, z-1, u)$, and $(x, y, z, u-1)$. The best one of the 4 configurations will be chosen as the next fork position. Note that the sum of the throttling level of the next fork position (x', y', z', u') is 1 smaller than the sum of the current fork position (x, y, z, u):

$$x' + y' + z' + u' = x + y + z + u - 1.$$

In our example, the first fork position is $(8, 8, 8, 8)$ (default configuration with every core running at full speed). The end condition is that we either cannot fork any more or find a configuration that meets our unfairness or QoS constraint. As with any hill climbing algorithm, the caveat is that there is the possibility of finding a local rather than a global minimum.

Under this hill climbing algorithm, the worst-case search cost for a system with n cores and m throttling levels occurs when forking from $(m, m, ..., m)$ to $(1, 1, ..., 1)$. Since the difference between the sum of the throttling levels of two consecutive fork positions is 1, and the first fork position has a configuration sum of $m \cdot n$ while the last one has a configuration sum of n, the total possible fork positions is $m \cdot n - n$. Each of these fork positions will probe at most n children. So, we examine $(m-1)n^2$ configurations in the worst case, which is substantially cheaper than enumerating all m^n configurations.

D. Dynamic Environments

While our approach can identify a high-quality throttling configuration for a stable multicore execution, dynamic changes in a real system require adaptation of the throttling configuration. Application execution characteristics may change due to a change in phase behavior, requiring different throttling levels for fairness or high performance. TEMM supports continuous configuration search in which recency is reflected by replacing an old measurement sample with a new sample at the same configuration. By doing so, our iterative framework takes a phase change as a mistaken prediction and automatically incorporates the behavior in the current phase into the model to correct the next round of prediction. In addition to phase changes within applications, an operating system context switch also affects TEMM's effectiveness since it requires a new configuration search for the new set of co-running applications.

IV. Evaluation Results

A. System Implementation

We implemented the necessary kernel support for performance counter monitoring, duty cycle modulation (8 duty-cycle levels), and DVFS in Linux 2.6.30. TEMM's configuration search and policy control are implemented in a user-level daemon thread. The kernel maintains a per-core data structure and exports a system call interface allowing the daemon to query per-core hardware counter metrics and to update the

current configuration. The per-core data structure is asynchronously updated (and checked for current configuration) at each kernel clock tick (which is 1 millisecond by default). If a configuration change is required, the kernel writes to its local configuration at that time and each core reads and updates its configuration at the next tick. By default, configurations are sampled and changed (if necessary) by the daemon at 1 second intervals.

The online system uses CPI as run-time performance guidance. In addition, it takes baseline performance (running each application alone) and SLOs as inputs.

B. Experimental Setup

Our evaluation is conducted on three platforms. The first is an Intel Xeon E5520 2.27 GHz "Nehalem" quad-core processor. The second is a 2-chip NUMA machine with Intel Xeon L5640 2.27 Ghz "Westmere" six-core processors (hyperthreading disabled, 12 cores in total). The last is a 2-chip SMP machine with Intel Xeon 5160 3.0 Ghz "Woodcrest" dual-core processors (4 cores in total). All platforms run our modified 2.6.30 Linux kernel configured with 8 duty-cycle levels and Woodcrest is additionally configured with 4 DVFS levels (3/2.67/2.33/2 Ghz).

In order to test our framework, we focus on multiprogrammed workloads. For the 4-core platforms (Nehalem and Woodcrest), we use sets of four co-running applications selected from the SPECCPU2000 benchmarks that show significant resource contention. The five workloads used in our experiments are:
set-1 = {*mesa, art, mcf, equake*},
set-2 = {*swim, mgrid, mcf, equake*},
set-3 = {*swim, art, equake, twolf*},
set-4 = {*swim, applu, equake, twolf*},
set-5 = {*swim, mgrid, art, equake*}.
For the 12-core Westmere platform, we run 12 representative SPECCPU2006 benchmarks concurrently:
{*leslie3d, dealII, soplex, povray, GemsFDTD, lbm, mcf, hmmer, libquantum, h264ref, omnetpp, astar*}.
The benchmarks are compiled using gcc 4.4.1 at the -O3 optimization level.

We also include 4 server-style applications (for platforms with 4 cores):
{*TPC-H, WebClip, SPECWeb, SPECJbb2005*}.
TPC-H runs on the MySQL 5.1.30 database. Both WebClip and SPECWeb use independent copies of the Apache 2.0.63 web server. WebClip hosts a set of video clips, synthetically generated following the file size and access popularity distribution of the 1998 World Cup workload [19]. SPECWeb hosts a set of static web content following a Zipf distribution. SPECJbb2005 runs on IBM 1.6.0 Java. All server applications are configured with 300∼400 MB footprints so that they can fit into the memory on our test platforms. We do not use any I/O bound applications since our focus is on CPU/memory-intensive workload configuration.

C. Evaluation of Configuration Search Effectiveness

1) Comparison to Exhaustive Search: In order to provide a baseline for comparison, we populate the performance of all possible configurations of the 5 SPECCPU2000 sets on the Nehalem platform. Since DVFS is only applied on a per-chip basis, we only consider duty cycle modulation in this first experiment. Our 8 duty-cycle levels result in a total of 8^4 possibilities. Since the configurations with lower duty cycles will have very long execution times and are unlikely to provide reasonable performance even if SLO objectives are met, we limit our experimental time for the exhaustive search by only populating duty-cycle levels from 8 (full speed) to 4 (half speed). We also avoid configurations in which all cores are throttled (i.e., we want at least one core to run at full speed). In total, we try $5^4 - 4^4 = 369$ configurations for each set. Since each application executes for different lengths of time, we run each configuration for tens of minutes and use the average execution time of each application within this run in determining performance. In total, it took us two weeks to populate the configuration space for the 5 test sets. We use this data to determine the optimal configuration for the *Oracle* method (see Section IV-C4).

2) SLO metrics: Our examined SLOs are the two discussed in Section II-B. For the fairness-centric tests, we consider unfairness values of 0.05, 0.10, 0.15, and 0.20 as objectives. For the QoS-centric tests, we consider a normalized (to running alone) performance of 0.50, 0.55, 0.60, and 0.65 as targets for a selected application in each set. We chose mcf in set-1 and set-2, twolf in set-3 and set-4, and art in set-5 as the high-priority QoS application, because they are the most negatively affected application in the full-speed configuration (i.e., no throttling at any core) of the corresponding test set.

Since there may be multiple configurations satisfying a SLO target, we use an *overall performance* metric to compare their quality. For a set of applications, *overall performance* is defined as the geometric mean of their normalized performance. We use execution time as the performance metric for SPECCPU2000 applications, and throughput (transactions per second) for server applications. For the fairness-centric test, overall performance includes all co-running applications. For the QoS-centric test, overall performance only includes the non-prioritized applications (those with no QoS guarantees). Our goal is therefore to find a configuration that maximizes overall performance while satisfying SLOs.

We also compare the convergence speed of different methods, i.e., the number of configurations sampled before selecting a configuration that meets the constraints. The performance samples of the applications' standalone runs are not counted in the number of samples (they can be collected independently and a priori).

3) Effectiveness of Iterative Refinement: Given a service level target, TEMM iteratively samples configurations toward the region where a high-quality configuration resides. We show four examples of real tests on the Nehalem platform in Figure 1. We present configurations as a quad-tuple (u, v, w, z)

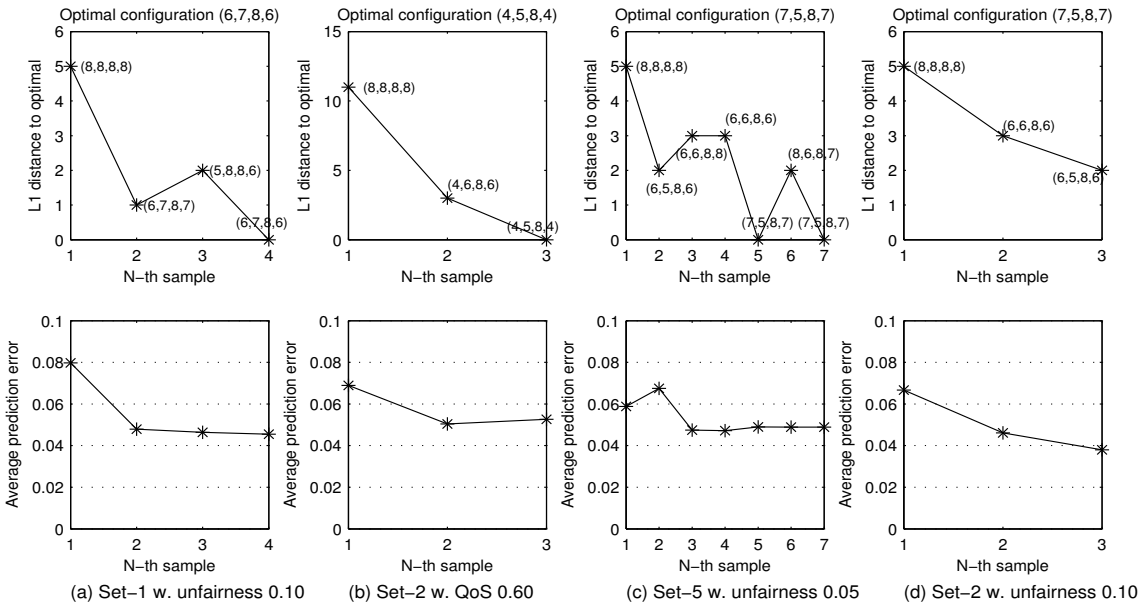

Fig. 1: Examples of our iterative refinement for some real tests. X-axis shows the N-th sample. For the top half of the sub-figures, the Y-axis is the L_1 distance (or Manhattan distance) from the current sample to the optimal. Configuration is represented as a quad-tuple (u, v, w, z) with each dimension indicating the duty-cycle level of the corresponding core. For the bottom half of the sub-figures, the Y-axis is the average performance prediction error of all considered points over applications in the set. Here, considered points are selected according to the hill climbing algorithm in Section III-C.

with each letter indicating the duty-cycle level of the corresponding core. The top half of Figure 1 shows the Manhattan or L_1 distance of the selected configuration from that predicted by Oracle. The first sample $(8, 8, 8, 8)$ (i.e., full-speed configuration) is usually not close to the optimal configuration, but TEMM automatically adjusts subsequent samples toward the optimal region (represented by a smaller L_1 distance). The iterative procedure terminates when the predicted best configuration stabilizes, which is the optimal in Figures 1(a) and (b). It is possible that TEMM will terminate at a different configuration from the optimal (as in Figure 1(d), where the L_1 distance is not zero when the algorithm terminates) by discovering a local minimum, although the SLO is satisfied. TEMM may also continue sampling even after discovering a satisfying configuration in the hope of discovering a better configuration: in Figure 1(c), it finds the optimal configuration $(7, 5, 8, 7)$ at the 5th sample, but continues to explore $(8, 6, 8, 7)$. If the next prediction is within the set of sampled configurations ($(7, 5, 8, 7)$ in this case), the algorithm concludes that a better configuration will not be found and stops exploration.

4) Comparison of Different Methods: In order to isolate and identify the effectiveness of TEMM's configuration search method, we compare it with several alternatives in an offline manner, using whole application performance at the requisite configuration as input. *1-iteration TEMM* is the same as TEMM but without iterative refinement. *Oracle* uses exhaustive search to always use the optimal configuration — the one with the best overall performance (geometric mean of application performance on all cores) while satisfying the fairness or QoS objective. *Random* search randomly samples 15 configurations and picks the best.

We also consider a *greedy* configuration search for comparison. It begins by sampling the configuration with every CPU running at full speed. Greedy search bears similarity to the hill climbing approach, but without the help of a performance model to guide the search. At each step of the greedy search, we lower one core's throttling level by one and sample its overall performance. The choice of the throttled core depends on the resource management objective. For resource management with a fairness-centric objective, the throttled core at each greedy step is the one with the the highest CPI ratio (the ratio between the CPI when the application runs alone and the CPI when it runs along with other applications). The rationale is that this core is the most aggressive in competing for shared cache and memory bandwidth resources, and therefore slowing it down would most likely lead to fairness improvement. For resource management with a QoS-centric objective, the throttled core at each greedy step is the core with the highest CPI ratio among the non-QoS cores. By slowing down this core, a high-priority core has a better chance of meeting its QoS target with fewer duty cycle adjustments. The greedy search stops when the QoS objective is met.

Figure 2 shows the results using a 0.10 unfairness threshold. From Figure 2(a), we can see that only Oracle and TEMM satisfy the objectives for each experiment (indicated by unfairness below the horizontal solid line). Figure 2(b) shows the corresponding overall performance normalized to the performance of Oracle. In some tests, 1-iteration TEMM, greedy search, and random search show better performance than Oracle, but in each case, they fail to meet the unfairness

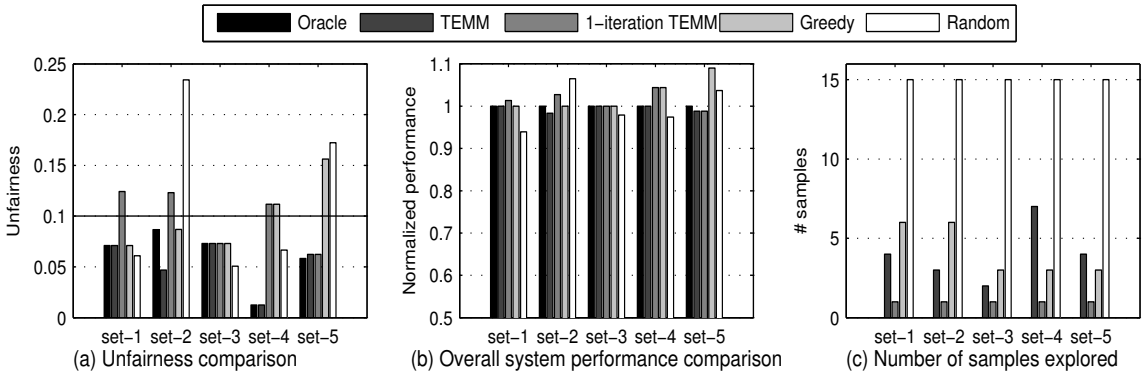

Fig. 2: Comparison of methods for the fairness-centric tests with unfairness ≤ 0.10. In (a), the unfairness target threshold is indicated by a solid horizontal line (lower is good). In (b), performance is normalized to that of Oracle. In (c), Oracle requires zero samples. The optimization metric is to find a configuration that first satisfies the unfairness threshold (0.10) and then maximizes overall performance.

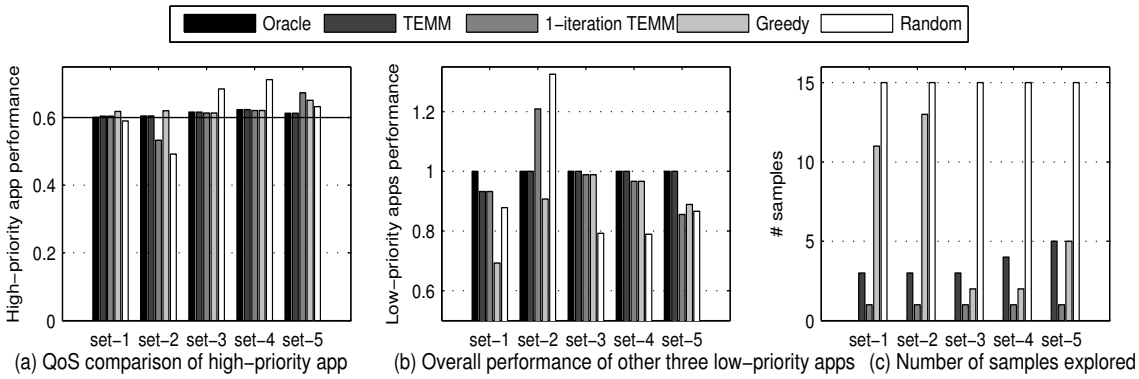

Fig. 3: Comparison of methods for the QoS-centric tests with high-priority thread performance normalized to running alone ≥ 0.60. In (a), the QoS target is indicated by a horizontal line (higher is good). In (b), performance is normalized to that of Oracle. In (c), Oracle requires zero samples. The optimization metric is to find a configuration that first maintains a performance target of 0.60 for the QoS core and then maximizes overall performance for the non-QoS cores.

target. Only TEMM meets all unfairness requirements and is very close to (less than 2% away from) the performance of Oracle. Figure 2(c) shows the number of samples before a method settles on a configuration.

Figure 3 shows results of QoS tests with a performance target (normalized to running alone) of 0.6 for a selected high-priority application. From Figure 3(a), we can see that Oracle, TEMM, and greedy search all meet the QoS target (equal to or higher than the 0.6 horizontal line). However, TEMM consistently achieves better performance than greedy search: TEMM is within 7% of Oracle while greedy search could lose 30%. For set-2, 1-iteration TEMM and random search achieve good performance in Figure 3(b) but they fail to meet the QoS target in Figure 3(a). For set-1, random search shows lower performance while also failing to meet the QoS target.

Figures 2(c) and 3(c) show that TEMM has more stable convergence speed (3∼5 samples) than greedy search (2∼13 samples). The convergence speed of greedy search is largely determined by how far away the satisfying configuration is from the starting point since it only moves one duty-cycle level each time. This could be a serious limitation for systems with many cores and more configurations. TEMM converges quickly because it has the ability to estimate the whole search space at each step.

In total, we have 8 tests (4 parameters for both unfairness and QoS) for each of the five co-running workloads and we summarize the 40 tests in Table I. TEMM meets SLOs in all cases but one (set-2 with QoS target ≥ 0.65) where there is no configuration in our search space (duty-cycle levels from 4 to 8) that can meet the target (i.e., even Oracle failed on this one). We compare the overall performance (normalized to that of Oracle) in 2 ways: 1) we pick 18 common tests for which all methods meet the SLOs in order to provide a fair comparison; 2) we include any passing test of a method in the performance calculation for that method. In both cases, TEMM shows the best results, achieving 99% of Oracle's performance.

D. Scalability Evaluation

In order to evaluate the scalability of our iterative method, we ran 12 SPECCPU2006 benchmarks on our 2-chip (12

Method	# tests meeting SLO	# samples	Perf1	Perf2
The Oracle	39/40	0	100%	100%
TEMM	39/40	4.1	99.6%	99.4%
1-iteration TEMM	23/40	1	94.1%	95.0%
Greedy search	33/40	4.2	98.1%	96.8%
Random search	25/40	15	90.9%	91.1%

TABLE I: Summary of the comparison among methods. Here Perf1 is average normalized performance of 18 common tests for which all methods meet the SLO. Perf2 is average normalized performance for all tests of a method that meet the SLO.

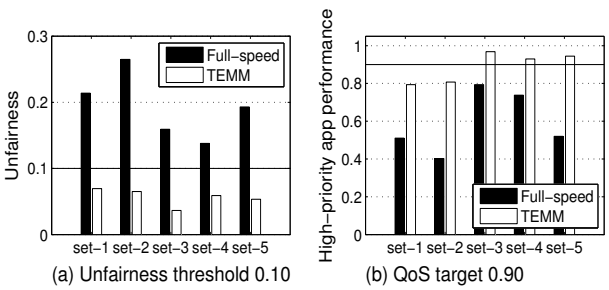

Fig. 4: Dynamic system evaluation results for 5 SPEC-CPU2000 sets. Only duty cycle modulation is used by TEMM as the throttling mechanism.

Method	# samples	Unfairness (target 0.10)	Chosen configuration
TEMM	2	0.13	(8,6,8,5,8,8,8,5,8,5,8,6)
Full-speed	1	0.35	(8,8,8,8,8,8,8,8,8,8,8,8)
Random	50	0.26	(8,8,8,6,7,7,8,6,8,5,6,6)

Method	# samples	QoS (target 0.6)	Chosen configuration
TEMM	8	0.61	(8,6,8,5,8,8,8,5,8,5,8,6)
Full-speed	1	0.38	(8,8,8,8,8,8,8,8,8,8,8,8)
Random	50	0.55	(8,8,8,6,4,5,4,7,4,7,4,6)

TABLE II: Scalability results on 12-core Westmere platform for 12 SPECCPU2006 benchmarks.

cores total) "Westmere" platform. We configured the NUMA setup such that memory allocation is interleaved between the two memory nodes. While our throttling-based resource management should work under different NUMA setups, the interleaved allocation is more relevant because it stresses cross-chip memory controller contention and introduces new resource management complexity. In addition, interleaved memory allocation eliminates the uncertainty in the location of shared libraries and therefore leads to more stable measurements.

We set the lowest duty-cycle level to 4 (half speed) to limit our experimental time (many SPECCPU2006 benchmarks take tens of minutes to finish a single run even at full speed). Even with limited levels, it is impossible to exhaustively populate the whole search space for 12 cores. Hence we only compare the results of our algorithm against a random method (choosing the best out of 50 randomly sampled configurations) and the default full-speed configuration (no throttling at any core). We chose an unfairness threshold of 0.10 and QoS of 0.6 for the 3rd core (since soplex running on the 3rd core is the most severely affected application) as SLOs. Results shown in Table II suggest that our algorithm can quickly converge to a good configuration on a 12-core platform.

E. Evaluation on Dynamic Systems

The results presented in the previous sections assume stable behaviors for each test configuration. In this section, we evaluate the TEMM system in an online dynamic system environment with effects of application phase behavior changes.

1) Evaluation of SPECCPU Benchmarks: In the first set of dynamic system experiments, we run one application per core so that no context switch is involved. This emulates batch mode scheduling for non-interactive applications.

We evaluate the full TEMM system using an unfairness objective of 0.10 and a QoS target of 0.90 for the 5 SPEC-CPU2000 sets as described in Section IV-B on the Nehalem platform. Figure 4 shows the results of the online tests. The full-speed configuration exhibits poor fairness among applications and has no control over providing QoS for the selected applications. TEMM meets targets for sets 3-5 but fails to provide the QoS target of 0.90 for mcf in set-1 and set-2. The reason is that the current duty-cycle modulation on our platform can only throttle the CPU to a minimum of 1/8—we do not attempt to de-schedule any application (i.e., virtually throttle CPU to 0 speed), which would be necessary to give mcf enough of the shared resource to maintain 90% of its ideal performance. Nevertheless, TEMM manages to keep mcf's performance fairly close to the target (within 10%).

The runtime overhead of our approach mainly comes from the computation load of predicting the best configuration based on the existing reference pool (reading performance counters and setting the modulation level only takes several microseconds). Recall that we introduced a hill climbing algorithm in Section III-C, which significantly reduces the worst-case number of evaluated configurations from m^n to $(m-1)n^2$ for an n core system with a maximum of m modulation levels. As shown in Table III, the hill climbing optimization reduces computation overhead by 20$\sim$60x and mostly incurs less than 1 millisecond overhead in our tests.

2) Evaluation of Server Benchmarks: In order to demonstrate the more general applicability of our approach, we add DVFS as another source of throttling, and change the management objective from overall performance to power efficiency. Note that DVFS is applied to the whole chip and not per core on our Intel processors. We test this new model only on the 2-chip Woodcrest platform.

We run 4 server benchmarks together on Woodcrest (2 dual-core chips, 4 cores in total) and bind each server application to one core (simulating encapsulation of each server in a

Set	Target	Hill-Climbing	Exhaustive
#1	Unfairness ≤ 0.1	0.32 ms	15.94 ms
	QoS ≥ 0.9	1.06 ms	27.48 ms
#2	Unfairness ≤ 0.1	0.49 ms	31.64 ms
	QoS ≥ 0.9	1.28 ms	66.14 ms
#3	Unfairness ≤ 0.1	0.18 ms	9.93 ms
	QoS ≥ 0.9	0.88 ms	21.92 ms
#4	Unfairness ≤ 0.1	0.21 ms	6.54 ms
	QoS ≥ 0.9	1.81 ms	35.03 ms
#5	Unfairness ≤ 0.1	0.19 ms	10.04 ms
	QoS ≥ 0.9	1.33 ms	28.82 ms

TABLE III: Average runtime overhead in milliseconds of calculating the best duty cycle configuration in dynamic system evaluation. To choose a sampling configuration at each TEMM iteration, *Exhaustive* searches and compares all possible configurations while *Hill-Climbing* limits calculation to a subset.

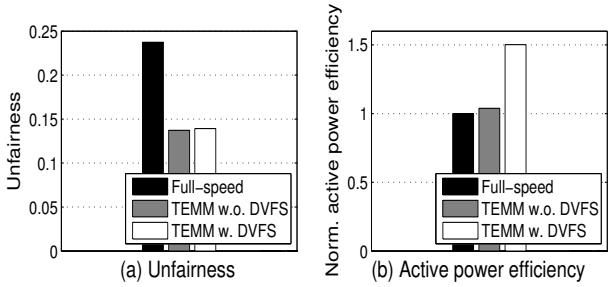

Fig. 5: Dynamic system evaluation on active power efficiency (performance per watt). TEMM without DVFS only uses duty cycle modulation as the throttling mechanism. TEMM with DVFS combines two throttling mechanisms (duty cycle modulation and voltage/frequency scaling).

single-core virtual machine). The pairing is randomly selected: TPC-H and WebClip run together on one chip, and SPECWeb and SPECJbb2005 run on the other chip. We first consider only duty cycle modulation as the throttling mechanism. The goal here is to maximize power efficiency while limiting unfairness (threshold target 0.10). We are mainly interested in active power (whole system operating power minus idle power) in this test and define *active power efficiency* as performance divided by active power in watts. We empirically determine active power to be quadratically proportional to frequency (a result of the limited range for voltage scaling as well as activity outside the CPU such as at memory) and linearly proportional to duty-cycle levels, and create a model accordingly. Performance is calculated in terms of throughput (i.e., transaction per second) although our TEMM implementation is guided by the CPI metric. This might be problematic for applications whose instructions mutate during different runs, but this is not the case in our experiments.

Figure 5 shows unfairness and active power efficiency under the default system (Full-speed) and under TEMM with and without DVFS. We can see that TEMM with DVFS achieves much better active power efficiency while providing good fairness. This experiment also demonstrates that our framework can be applied to different resource management scenarios.

V. Related Work

There has been considerable focus on the issue of quality of service for applications executing on multicore processors [20], [21], [22], [23], [24], [25], [26], [27]. Suh et al. [20] use hardware counters to estimate marginal gains from increasing cache allocations to individual processes. Zhao et al. [23] propose the CacheScouts architecture to determine cache occupancy, interference, and sharing of concurrently running applications. Tam et al. [24] use the data sampling feature available in the Power5 performance monitoring unit to sample data accesses. Awasthi et al. [26] use an additional layer of translation to control the placement of pages in a multicore shared cache. Mutlu et al. [25] propose parallelism-aware batch scheduling at the DRAM level in order to reduce inter-thread interference at the memory level. These techniques are orthogonal and complementary to controlling the amount of a resource utilized by individual threads. Without extra hardware support, software page coloring [28], [6], [7], [8], [9] is an effective mechanism to achieve cache partitioning. However, cache partitioning alone does not take into account contention on other on-chip resources such as the on-chip interconnect and memory controllers. Hardware execution throttling can control the number of accesses to the cache, thereby affecting cache reference pressure and indirectly the cache space sharing as well as memory bandwidth consumption.

Existing scheduling quantum adjustment [10] at the operating system level could be used for the purpose of execution throttling. To better guide scheduling quantum adjustment, West et al. [13] introduce an analytical model to estimate an application's cache occupancy on-the-fly. However, CPU scheduling quantum adjustment suffers from its inability to provide fine-grained quality of service guarantees [3]. The coarser throttling granularity results in higher performance fluctuations, especially for fine-granularity tasks such as individual requests in a server system.

Ebrahimi et al. [4] propose a new hardware design to track contention at different cache/memory levels and throttle threads with unfair resource usage or disproportionate progress. We address the same problem but without requiring special hardware support.

Herdrich et al. [2] and Zhang et al. [3] show that duty cycle modulation is effective at controlling utilization of contended resources (last-level cache and off-chip bandwidth). While these studies focus on the characteristics of execution throttling mechanisms, this paper addresses the policy question—how to automatically and quickly identify a high-quality throttling configuration to achieve a desired SLO.

Our hill-climbing algorithm is similar in principle to that in [29] where the optimization target is a chip-wide DVFS setting with a polynomial search space. We study per-core throttling plus chip-wide DVFS and dramatically reduce the

search space from exponential to $O(n^2)$ (it can be further reduced to $O(n \times \log n)$ by binary search, though we did not evaluate it in this work). Besides this, our studied SLOs are more diversified and evaluation is done on real machines.

There are also feedback-driven models based on formal control theory. They usually require system parameter tuning [30], [31]. Our model is kept simple and intuitive, allowing easy portability across different platforms. Our iterative method allows us to acquire measurements gradually closer to the target and these near-target measurements eventually help overcome any model inaccuracy. Our evaluation shows that this approach works effectively.

VI. Conclusion

This paper presents TEMM, a software framework that manages multicore resources via controlled hardware execution throttling on selected CPU cores. It models the effects of duty cycle modulation and voltage/frequency scaling from reference configurations, searches for a high-quality throttling configuration based on model predictions, and iteratively refines the search with a broadening set of measured references. TEMM also employs a hill-climbing optimization to accelerate configuration search.

We evaluate TEMM using a set of SPECCPU2000/2006 benchmarks and 4 server-style applications. We test our approach on a variety of resource management objectives such as fairness, QoS, performance, and active power efficiency (in the case of DVFS) using three different multicore platforms for multiprogrammed workloads. Our results demonstrate that hardware throttling coupled with our iterative framework effectively supports multiple forms of service level objectives for multicore platforms in an efficient and flexible manner.

References

[1] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security Symp.*, Boston, MA, 2007, pp. 257–274.

[2] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses, "Rate-based QoS techniques for cache/memory in CMP platforms," in *23rd International Conference on Supercomputing (ICS)*, Yorktown Heights, NY, Jun. 2009.

[3] X. Zhang, S. Dwarkadas, and K. Shen, "Hardware execution throttling for multi-core resource management," in *USENIX Annual Technical Conf. (USENIX)*, Santa Diego, CA, Jun. 2009.

[4] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proc. of the 15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, PA, mar 2010, pp. 335–346.

[5] "IA-32 Intel architecture software developer's manual," 2008, http://www.intel.com/products/processor/manuals/.

[6] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared L2 caches on multicore systems in software," in *Workshop on the Interaction between Operating Systems and Computer Architecture*, San Diego, CA, Jun. 2007.

[7] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Salt Lake, UT, Feb. 2008, pp. 367–378.

[8] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer," in *41th Int'l Symp. on Microarchitecture (Micro)*, Lake Como, ITALY, Nov. 2008, pp. 258–269.

[9] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the Fourth EuroSys Conference*, Nuremberg, Germany, Apr. 2009.

[10] A. Fedorova, M. Seltzer, and M. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Brasov, Romania, Sep. 2007, pp. 25–36.

[11] "Intel core2 duo and dual-core thermal and mechanical design guidelines," 2009, http://www.intel.com/design/core2duo/documentation.htm.

[12] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Int'l Symp. on Computer Architecture*, 2005, pp. 506–517.

[13] R. West, P. Zaroo, C. Waldspurger, and X. Zhang, "Online cache modeling for commodity multicore processors," *Operating Systems Review*, vol. 44, no. 4, Dec. 2010.

[14] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar, "Power and thermal management in the Intel Core Duo processor," *Intel Technology Journal*, vol. 10, no. 2, pp. 109–122, 2006.

[15] "Intel turbo boost technology in intel core microarchitecture (Nehalem) based processors," Nov. 2008, http://download.intel.com/design/processor/applnots/320354.pdf.

[16] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource," in *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2006, pp. 13–22.

[17] K. Shen, C. Stewart, C. Li, and X. Li, "Reference-driven performance anomaly identification," in *ACM SIGMETRICS*, Seattle, WA, Jun. 2009, pp. 85–96.

[18] X. Zhang, K. Shen, S. Dwarkadas, and R. Zhong, "An evaluation of per-chip nonuniform frequency scaling on multicores," in *USENIX Annual Technical Conf. (USENIX)*, Boston, MA, Jun. 2010.

[19] M. Arlitt and T. Jin, "Workload Characterization of the 1998 World Cup Web Site," HP Laboratories Palo Alto, Tech. Rep. HPL-1999-35, 1999.

[20] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," in *The Journal of Supercomputing 28*, 2004, pp. 7–26.

[21] K. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *39th Int'l Symp. on Microarchitecture (Micro)*, Orlando, FL, Dec. 2006, pp. 208–222.

[22] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *39th Int'l Symp. on Microarchitecture (Micro)*, Orlando, FL, Dec. 2006, pp. 423–432.

[23] L. Zhao, R. Iyer, R. Illikkal, J. Moses, D. Newell, and S. Makineni, "CacheScouts: Fine-grain monitoring of shared caches in CMP platforms," in *16th Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Brasov, Romania, Sep. 2007, pp. 339–352.

[24] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/Eurosys European Conference on Computer Systems*, Lisbon, Portugal, Mar. 2007.

[25] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *International Symposium on Computer Architecture (ISCA)*, Beijing, China, Jun. 2008, pp. 63–74.

[26] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *15th Int'l Symp. on High-Performance Computer Architecture*, Raleigh, NC, Feb. 2009.

[27] K. Shen, "Request behavior variations," in *15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, Mar. 2010.

[28] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *39th Int'l Symp. on Microarchitecture (Micro)*, Orlando, FL, Dec. 2006, pp. 455–468.

[29] J. Li and J. F. Martnez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2006, pp. 77–87.

[30] X. Wang, C. Lefurgy, and M. Ware, "Managing peak system-leavel power with feedback control," IBM Research Tech Report RC23835, Tech. Rep., Nov. 2005.

[31] Y. Wang, K. Mai, and X. Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009.