Request Behavior Variations *

Kai Shen
Department of Computer Science, University of Rochester
kshen@cs.rochester.edu

Abstract

A large number of user requests execute (often concurrently) within a server system. A single request may exhibit fluctuating hardware characteristics (such as instruction completion rate and on-chip resource usage) over the course of its execution, due to inherent variations in application execution semantics as well as dynamic resource competition on resource-sharing processors like multicores. Understanding such behavior variations can assist fine-grained request modeling and adaptive resource management.

This paper presents operating system management to track request behavior variations online. In addition to metric sample collection during periodic interrupts, we exploit the frequent system calls in server applications to perform low-cost in-kernel sampling. We utilize identified behavior variations to support or enhance request modeling in request classification, anomaly analysis, and online request signature construction. A foundation of our request modeling is the ability to quantify the difference between two requests' time series behaviors. We evaluate several differencing measures and enhance the classic dynamic time warping technique with additional penalties for asynchronous warp steps. Finally, motivated by fluctuating request resource usage and the resulting contention, we implement contention-easing CPU scheduling on multicore platforms and demonstrate its effectiveness in improving the worst-case request performance.

Experiments in this paper are based on five server applications—Apache web server, TPCC, TPCH, RUBiS online auction benchmark, and a user-content-driven online teaching application called WeBWorK.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Multiprocessing/multiprogramming/multitasking, Scheduling; D.4.8 [*Operating Systems*]: Performance—Measurements, Modeling and prediction

General Terms Design, Experimentation, Measurement, Performance, Reliability

Keywords Server system, Request modeling, Operating system adaptation, Multicore, Hardware counter

* This work was supported in part by NSF CAREER Award CCF-0448413, grants CNS-0615045, CCF-0621472, CNS-0834451, and by an IBM Faculty Award.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

1. Introduction

The characterization of server workload resource consumption is important for managing behavior in highly interactive and concurrent online services. Previous operating system research [10, 26, 32, 35, 36] has proposed software-level techniques to characterize resource consumption (mostly CPU usage) and adapt system management functions. At the same time, prior architectural characterization of online applications [11, 19, 27] advanced the understanding of lower-level hardware execution characteristics like instructions per cycle, cache miss rates, and branch prediction accuracy. These characterizations are essential for efficiently utilizing the increasingly ubiquitous hardware resource-sharing multiprocessors, and for managing rare but important anomalous behaviors to achieve high system dependability.

A server system execution is often composed of many concurrent requests. Specifically, we define a *request* or a *request execution* as the set of server activities to service a user call, like a request for retrieving a web object, or a request for processing an e-commerce transaction. The execution of a single request typically consumes a fraction of a second in CPU time. However, there may exist large variation of hardware characteristics during its course. A fine-grained characterization of such variation may lead to enhanced modeling and better management of a server application. The request behavior variation patterns are particularly interesting on multicore platforms. On the one hand, the inter-core resource sharing obfuscates the request execution performance, leading to increased behavior variation. On the other hand, request behavior variations motivate adaptive resource management for better sharing on multicores.

This paper characterizes request behavior variations by collecting per-request hardware performance counter metrics during realistic executions of server applications. We explore operating system (OS)-level techniques to track such variations online. In addition to metric sample collection during periodic interrupts, we exploit the frequent system calls in server applications to perform low-cost in-kernel sampling. Further, the semantics of system call events allow some to act as signals for impending behavioral transitions, leading to cost-effective targeted sampling. Our approach functions transparently at the OS (*i.e.*, requiring no change in software applications and no special assistance in the hardware). Transparent system management provides more general applicability and it is essential for third-party management environments such as service hosting platforms.

Online tracking of request hardware behavior variations can enable new and improve existing fine-grained server system request modeling. The variation pattern represents a strong request signature that is more descriptive than the average metric value [27]. Compared to software metric-only online request modeling [10], the information of hardware behavior variations is essential to capturing dynamic hardware resource competition on resource-sharing processors like multicores. Specifically, this paper demonstrates that our modeling helps classify requests into groups with common variation patterns, and at the same time detect anomalous patterns linked to worst-case performance and resource competition. It can also support better online request identification and resource usage prediction.

Further, it is known that different pairings of tasks on resource-sharing multiprocessors may result in unequal levels of resource contention and thus differences in performance. Previous research has proposed contention-easing CPU scheduling for non-server applications [14, 24, 31] or treating the whole server application as a scheduling unit [15, 38]. However, adaptive resource scheduling within a single server application can exploit fine-grained workload behavior variations. Request behavior variations identified in our research present an important foundation for such adaptive scheduling.

Our study on request behavior variations is reminiscent of the body of past work on application phase identification and recognition [12, 13, 17, 29, 30]. However, server applications possess different workload features from those of technical computing and workstation applications. In particular, frequent network and I/O operations and the increasingly componentized server architectures, coupled with high execution concurrency and frequent context switches, result in behavioral fluctuations that may not form long stable phases. Further, our work targets OS-level system management requiring no change in software applications or special hardware assistance. We only utilize control mechanisms and information available to the OS.

The rest of this paper is organized as follows. Section 2 provides an empirical characterization of request behavior variations using realistic executions of several server applications. Section 3 presents our operating system-level approach to track request execution behavior variations online. Section 4 demonstrates promising utilization cases of variation-driven request modeling. Section 5 then examines contention-easing CPU scheduling, showing both positive and negative results. Section 6 discusses previous work related to ours. Finally, Section 7 concludes the paper with a summary of findings.

2. Empirical Request Behavior Variations

We characterize request behavior variations by collecting per-request hardware performance counter metrics during realistic executions of several server applications. We pay attention to both behavior variations across different requests (*inter-request variations*) and variations over the course of individual request executions (*intra-request variations*). We are also interested in the impact of multicore platforms on request behavior variations. This empirical characterization serves as both background and motivation for the rest of the paper.

2.1 Empirical Setup and Statistics Collection Methodology

Our empirical evaluation employs the following server applications:

- *Web server*. We run the Apache 2.2.3 web server. As the workload, we set up the static content portion of the SPECweb99 benchmark [4]. It contains four classes of files with sizes ranging from 100 bytes to 900 KB. The total dataset size in our workload setup is 200 MB.
- *TPCC* [5] simulates a population of terminal operators executing Order-Entry transactions against a database. It contains five types of transactions: "new order", "payment", "order status", "delivery", and "stock level", constituting 45%, 43%, 4%, 4%, and 4% of all requests, respectively. The workload runs on the MySQL 5.0.18 database with the InnoDB storage engine.
- *TPCH* [6] is a database-driven decision support benchmark. The TPCH workload consists of 22 complex SQL queries. Some queries require an excessive amount of time to finish and thus they are not appropriate for interactive server workloads. We choose a subset of 17 queries in our experimentation: Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q11, Q12, Q13, Q14, Q15, Q17, Q19, Q20, and Q22. Our synthetic workload contains an equal proportion of requests of each query type. We chose a dataset scaling factor to generate a dataset of 361 MB. TPCH runs on the MySQL 5.0.18 database.
- *RUBiS* [3] is a J2EE-based multi-component online service that implements the core functions of an auction site including selling, browsing, and bidding. It uses a three-tier service model, containing a front-end web server, a back-end database, and nine business logic components implemented as Enterprise Java Beans. RUBiS runs on the JBoss 3.2.3 application server with an embedded Tomcat 5.0 servlet container. The back-end is powered by the MySQL 5.0.18 database with a hosted dataset of 1,038 MB.
- *WeBWorK* [7, 33] is a web-based application that allows teachers to post math or physics problems for their students to solve online. In particular, teacher-supplied WeBWorK problems are interpreted by the application server as content-generating scripts. It is unique from traditional web applications in its *collaborative content*—considered by some a distinctive "Web 2.0" feature. Specifically, its request processing is heavily dependent on content supplied by end users—teachers in this case. Our WeBWorK installation runs Apache 2.2.8 web server, a variety of Perl PHP modules, and the Moodle course management system [2]. Our empirical examination is driven by around 3,000 teacher-created problem sets (ranging from pre-calculus to differential equations) and user requests extracted from system logs at the real site.

Our set of applications are good representations of many of today's online services. They include a variety of typical online service components and they possess a wide range of request processing complexities from simple file retrieval in web server to complex multi-stage processing in RUBiS and WeBWorK. Several applications (TPCC, TPCH, and RUBiS) involve significant database operations while the web server and WeBWorK employ little or no database processing.

We performed experiments on a machine with two dual-core (four cores total) Intel Xeon 5160 3.0 GHz "Woodcrest" processors. Two cores on each processor share a single 4 MB L2 cache (16-way set-associative, 64-byte cache line, 14 cycles latency, writeback). The whole machine contains 2 GB memory. Each processor core is equipped with two general-purpose performance event counter registers in addition to two fixed counters [1]. The two fixed counters record the number of CPU cycles (when the CPU is in non-halt state) and the number of retired instructions. Each of the general-purpose counters can be configured to track a variety of hardware events including references and misses to level-1/level-2 caches, memory bus cycles and bus transactions, floating-point operations, branch instructions and mis-predicted branches.

We run the Linux 2.6.18 operating system with kernel instrumentation for statistics collection and request context construction. A request may not execute continuously on a CPU in a concurrent server environment. Further, a request may propagate over multiple server modules in a multi-stage server system. In order to accurately account for each request's CPU execution periods, we instrument the operating system to track request context switches on CPUs and context propagations through inter-process communications (particularly socket operations). Details of our request context maintenance mechanism were presented in a prior paper [27, Sec-

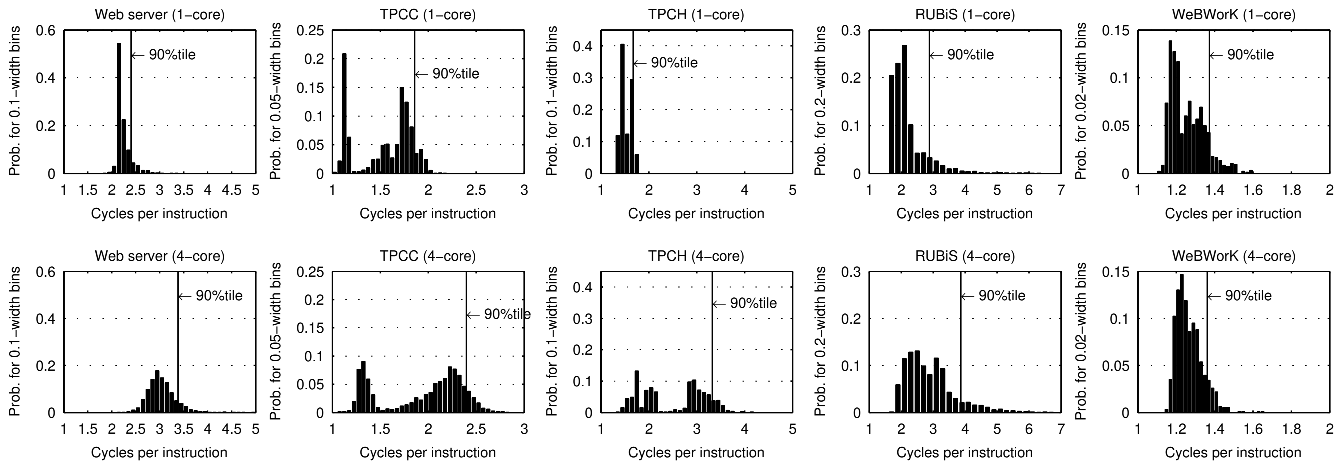

Figure 1. An illustration on multicore performance obfuscation in terms of request CPI distributions. The first row illustrates the 1-core serial execution performance while the second row shows the 4-core concurrent execution performance. Two figures in each column have the same X/Y-axis scales for easy comparison on the same application. On each plot, we mark the 90-percentile value for the request CPI distribution.

tion 4.1]. With collected hardware counter metrics for many execution periods that belong to a request, we finally serialize them into a continuous request execution timeline.

2.2 Inter-Request Request Variations

Figure 1 shows the distribution of per-request CPU cycles per instruction (CPI) for each of the five server applications. The per-request CPI is calculated by dividing a request's total CPU cycles over its total retired instruction count. We show the distributions for both 1-core serial executions and 4-core concurrent executions. Under the serial execution, requests for each application exhibit tightly clustered CPI metric. The TPCC distribution shows multiple clusters due to several distinctive transaction types.

Under the multicore concurrent executions, we observe that the request performance is generally much less clustered while the peak-level request CPI (like the 90-percentile values marked in Figure 1) becomes significantly worse for many applications. This is because inter-core resource sharing on multicores obfuscates the request execution performance, leading to increased behavior variations. We notice that the effect of such multicore performance obfuscation is application-dependent. In particular, it doubles the 90-percentile CPI for TPCH while WeBWorK sees no significant impact.

Note that the experimentation of 1-core serial executions is only used here for the comparison purpose. Experiments for the rest of the paper all employ concurrent executions on the full 4-core machine.

2.3 Intra-Request Behavior Variations

In addition to the behavior variations across different requests, an individual request's execution may also exhibit varying behaviors over its course. We show examples of realistic intra-request metric variation patterns. With one representative request from each of the five server applications, Figure 2 illustrates request behavior variations in terms of CPU cycles per instruction, L2 cache references per instruction (indicating the usage of shared resource), and L2 misses per reference (indicating the performance on shared resource). In general, we see significant metric variations over the course of request executions.

Previous research [13, 17, 30] has shown that applications execute as a series of phases with relative homogeneous behavior within each phase. Specifically they have identified stable phases at the granularity of 10 million instructions or larger for a number of technical computing and workstation workloads. Long stable phases, however, are not common for server applications. For instance, the later portion of the WeBWorK request exhibits unstable CPI variations. This is probably due to the large number of fine-grained Perl PHP modules that the request executes through—a feature commonly seen in today's componentized server architectures. Further, short requests like those of the web server naturally possess very fine-grained variation patterns. Finally, the high request concurrency and frequent context switches in server executions can lead to fluctuating hardware characteristics, particularly on resource-sharing platforms like multicores.

Request behavior variations bring challenges for their characterization and identification, but they also present opportunities for workload modeling and resource scheduling. Our results motivate the development of techniques to track request variation patterns and utilize them in adaptive system management.

3. Identification of Request Behavior Variations

Request behavior variations may support online system management as well as offline request modeling. In both cases, such behavior characteristics need to be captured online in realistic production environments. This section presents operating system-level management to track fine-grained request behavior variations. Our management can only utilize control mechanisms and information available to the OS. Compared to architecture-level approaches, the OS-level technique can be more easily applied in commodity systems. Compared to compiler or language-level approaches, OS management requires no change in application binary or need for re-compilation. However, the operating system management may incur high overhead due to expensive user/kernel domain switches. Our goal is to devise a cost-effective approach suitable for online operations.

3.1 Online Event Sampling

During the course of a request execution, our system samples cumulative processor hardware event counters including elapsed CPU cycles, retired instructions, L2 cache reference counts, and L2 misses. It samples at multiple moments and calculates the counter

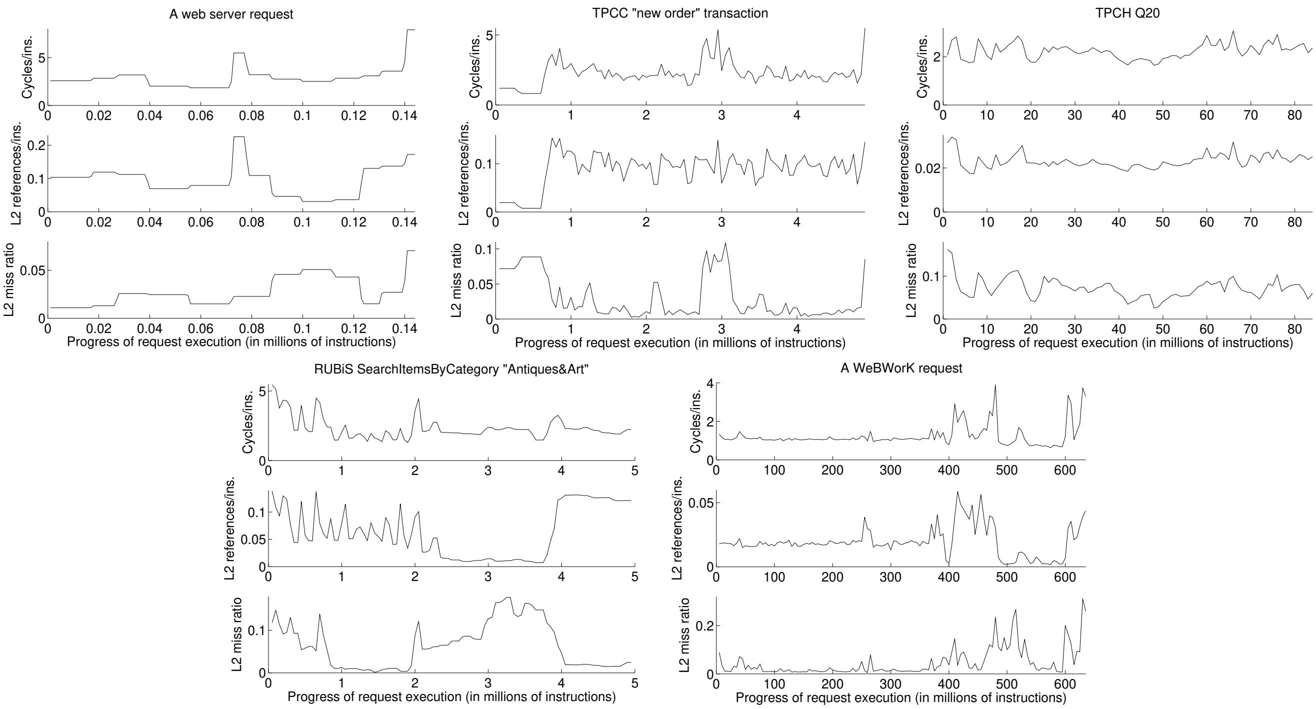

Figure 2. Examples of behavior variations within a single request execution (one example for each of the five applications). Variations on CPU cycles per instruction, L2 cache references per instruction, and L2 miss ratio (L2 misses per reference) are shown for each request. Note that the request lengths of different applications are at different scales. In particular, a web server request typically executes a few hundred thousand instructions while a WeBWorK request may execute as many as 600 million instructions.

metric for each period between consecutive sampling. To maintain per-request event metrics, we sample the counter values at the request context switch time to properly attribute the before-switch and after-switch event counts to the respective requests. Request context switches may occur at CPU context switches between different threads/processes. It also occurs when a request's host thread/process changes (*e.g.*, when it moves from the web server to the database server in a multi-stage processing) [27].

In addition to sampling at request context switches, further sampling is needed to capture fine-grained behavior variations. In particular, the OS can sample counter values at periodic interrupts. On our Intel processors, the CPU-local APIC allows event counter overflow interrupts on an arbitrary overflow threshold of CPU cycle count. We can generate periodic interrupts at very fine granularities—theoretically only limited by the system's ability of processing the interrupts, and practically up to once per ten microseconds in our experiments.

Overhead and Observer Effect Hardware counter sampling incurs overhead to the system. At the same time, the sampling operation produces additional processor events that do not belong to the inherent application behaviors. This effect, called the *observer effect*, leads to perturbation to collected counter metrics. Therefore it is important to understand the event sampling overhead and mitigate the observer effect. In our system, the sampling cost is mostly due to the tasks of reading the counter values and updating cumulative statistics (on per-CPU and per-request basis) in memory. Compared to in-kernel counter sampling (*e.g.*, during request context switches), interrupt-driven counter sampling incurs higher cost in additional user/kernel domain switching.

The exact cost and event count impact of hardware counter sampling depend on the dynamic cache state pollution by the running workload. We assess the range of such effects using two microbenchmarks. The first, called *Mbench-Spin*, spins the CPU with almost no data access. The second, called *Mbench-Data*, repeats a procedure of sequentially accessing 16 MB data in memory. Mbench-Spin exhibits minimum cache state pollution while Mbench-Data very quickly replaces the entire cache state. Table 1 shows the measured per-sampling average cost and additional counter events under different situations. Note that the L2 cache reference metric is an indirect indication of L1 cache misses.

To mitigate the observer effect in our statistics collection, we subtract the measured event count of each sampling period by the sampling-induced additional event count. One challenge is that the sampling cost and effect on cache performance events can be workload-dependent (see the different results on the two microbenchmarks in Table 1). While it is difficult to pinpoint the exact sampling observer effect for a given running workload, we follow a "do no harm" principle by subtracting the minimum observer effect (that of running Mbench-Spin in our case). Since this approach never over-compensates, it should always lead to better or equal accuracy than the originally collected statistics.

Results on Captured Variations We use the metric of *coefficient of variation* to quantitatively assess captured request behavior variations. Specifically, consider a set of n execution periods of lengths $t_1, \cdots, t_n$. The measured metric values for these periods are $x_1, \cdots, x_n$, respectively. Also let $\bar{x}$ be the overall metric value for the whole execution. Then:

$$\text{Coefficient of variation} = \frac{\sqrt{\frac{\sum_{i=1}^{n} t_i \cdot (x_i - \bar{x})^2}{\sum_{i=1}^{n} t_i}}}{\bar{x}} \qquad (1)$$

Here the inter-request coefficient of variation is calculated when we assume each request exhibits a uniform metric value over its

Hardware counter sampling at an in-kernel context

Workload	Time cost	Additional hardware event count			
		Cycles	Inst.	L2 ref.	L2 miss
Mbench-Spin	0.42 μs	1,270	649	N/M	N/M
Mbench-Data	0.46 μs	1,374	649	13	N/M

Hardware counter sampling at an interrupt

Workload	Time cost	Additional hardware event count			
		Cycles	Inst.	L2 ref.	L2 miss
Mbench-Spin	0.76 μs	2,276	724	N/M	N/M
Mbench-Data	0.80 μs	2,388	734	12	N/M

Table 1. Per-sampling average cost and additional event counts. We show results for sampling at an in-kernel context (like during a context switch) and sampling at an APIC interrupt. We also show results for two running workloads with different cache pollution effects. "N/M" means we see no measurable effect. Measurements were done in the Linux 2.6.18 on an Intel Xeon 5160 3.0 GHz "Woodcrest" processor.

execution (*i.e.*, a full request execution counts as a unit period in the above definition). To also consider the intra-request metric variation, we sample the metric values at multiple execution periods during each request execution. In order to acquire sufficient request-level statistics, we sample more frequently for applications with finer-grained requests. Specifically, we sample once per millisecond for long-request applications including TPCH and WeBWorK. For TPCC and RUBiS, we sample more frequently at once per 100 microseconds. For the web server with shortest requests, we sample once per 10 microseconds.

Figure 3 illustrates the captured inter-request and intra-request variations on three processor metrics. We observe that the consideration of intra-request behavior fluctuations leads to much stronger metric variations for most applications except TPCH. In TPCH, a request typically applies a specific SQL query on a long sequence of data, resulting in uniform behaviors over the course of its execution. In the other four applications, however, a request's behavior may vary significantly during execution. Capturing such intra-request variations is essential for request modeling and adaptive system management.

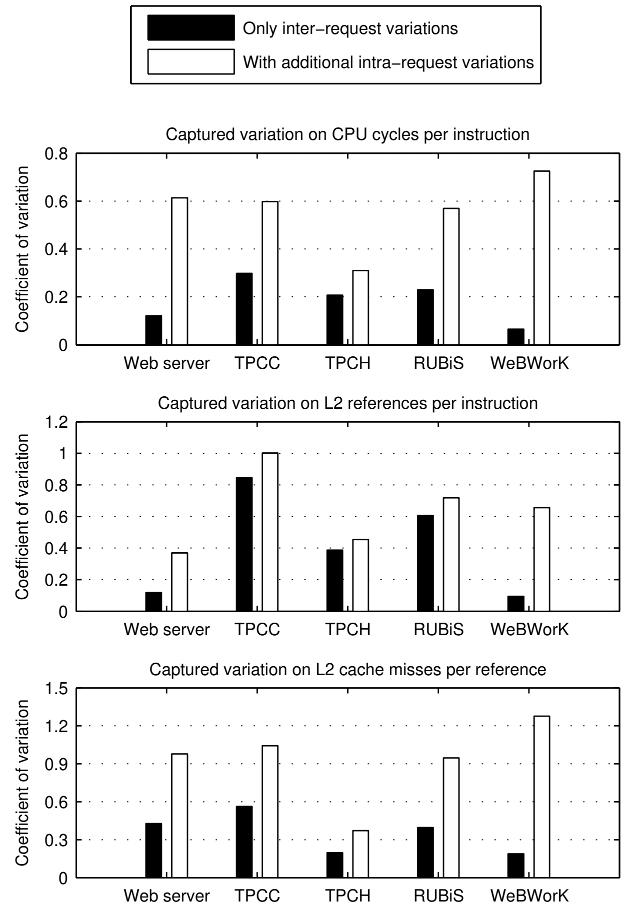

Figure 3. Captured request behavior variations on three metrics—CPU cycles per retired instruction, L2 cache references per instruction, and the L2 misses per reference. For each metric, we show the inter-request metric variations and the metric variations when intra-request behavior fluctuations are also considered.

3.2 System Call-Triggered Sampling

An interrupt processing involves an expensive kernel domain trap. Table 1 shows that this leads to more than 1,000 additional CPU cycles on our experimental platform. We can avoid the additional domain switch cost if we sample the processor hardware event counter values when the system already executes in the kernel domain. In particular, cheaper in-kernel sampling can be performed during system calls. This idea is reminiscent of the Soft Timers technique [9] that utilizes system call-driven device polling to replace expensive network interface interrupts.

System Call Occurrences In Server Applications Unlike the periodic interrupts, the OS does not have direct control of the occurrence patterns of system calls. A low system call frequency in a server workload could limit the effectiveness of system call-context event sampling. To understand this, we measure the cumulative probability for the distribution of next system call distances in our applications. Specifically, the cumulative probability at the distance D is defined as the probability that from an arbitrary instant in a request execution, the next system call would occur in no more than distance D. A low probability would indicate long periods of execution without system calls.

Figure 4 shows the distribution of next system call distances in both time and the instruction count. We observe that system calls are very frequent in the web server, TPCH, and RUBiS, due to a large number of network and storage I/O operations. The componentized server architecture in RUBiS also results in additional system calls. Specifically, the probabilities for the next system call to occur within 16 microseconds from an arbitrary instant are 97%, 83%, and 72% for the three applications respectively.

On the other hand, Figure 4 shows that WeBWorK exhibits relatively long system-call-free executions due to its CPU-intensive content generations (math computation and graphics rendering) that make few system calls. TPCC also exhibits long system-call-free executions as the result of its compute-intensive query processing. But even for these two applications, there are high chances to see a system call within one millisecond from an arbitrary instant of execution. The specific probabilities are 82% and 81% for TPCC and WeBWorK respectively.

Approach and Overhead Evaluation Frequent system calls in many server applications indicate that they can support very fine-grained in-kernel event sampling. On the other hand, it is unnecessary to sample at every system call which could lead to excessive overhead. Further, since applications sometimes exhibit long

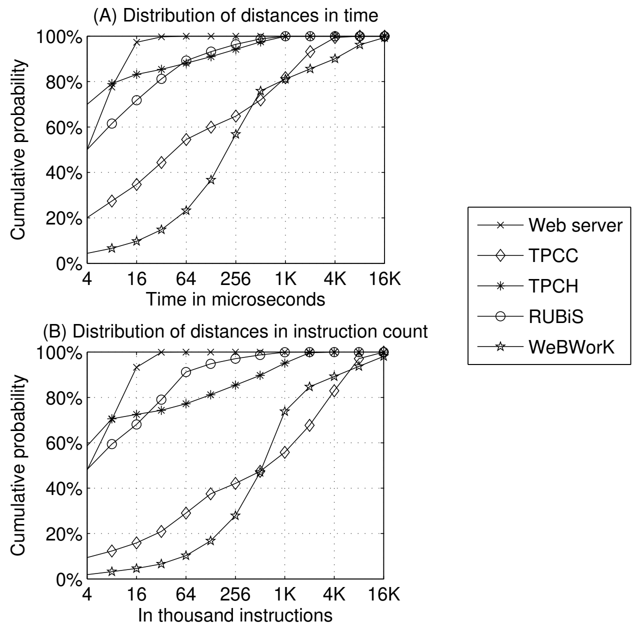

Figure 4. Cumulative probability plot for the distribution of next system call distances in time (A) and in instruction count (B). Note that the X-axis is in the logarithmic scale.

system-call-free executions, it is important to cover these execution periods with a backup interrupt-based sampling scheme.

Specifically, our system call-triggered sampling works as follows. When a request is switched in for execution, we sample the processor event counters and set a timer for an interrupt-based sampling at $T_{\text{backup\_int}}$ from now. At the kernel entrance of each system call, we check whether the elapsed time from the last sampling is at least $T_{\text{syscall\_min}}$. If so, we perform a new sampling and reset the interrupt timer at $T_{\text{backup\_int}}$ from now. In our approach, the backup interrupt delay $T_{\text{backup\_int}}$ is substantially larger than the minimum system call delay $T_{\text{syscall\_min}}$ so that no interrupts actually occur when system calls are frequent.

We compare the overhead of our system call-triggered processor counter sampling with the interrupt-based sampling approach described in Section 3.1. For fair comparison, we set $T_{\text{backup\_int}}$ and $T_{\text{syscall\_min}}$ carefully for each application such that our system call-triggered sampling generates similar overall sampling frequencies as the interrupt-based approach does. We also verify that the two approaches capture similar levels of request behavior variations. We estimate the sampling overhead for each experiment in the following way. We first count the in-kernel and interrupt-based counter samples that are needed for capturing request behavior variations during experimentation. We then compute the total overhead using the measured per-sample costs in Table 1 (that of Mbench-Spin). Figure 5 shows the overhead comparison results on the five applications. Compared to the interrupt-based sampling, the system call-triggered processor counter sampling saves 18–38% overhead for these applications.

Behavior Transition Signals For the purpose of capturing request behavior variations, it is more cost-effective to perform sampling at targeted opportunities that are more correlated with request behavior transitions (*e.g.*, a change from low CPI to high CPI). Application-issued system calls are an integral part of the application semantics and it is intuitive to expect that some system calls may serve as signals for impending behavior transitions. We study

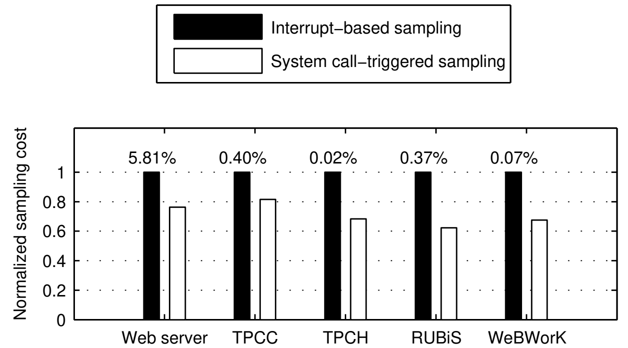

Figure 5. The overhead comparison of the system call-triggered processor counter sampling (Section 3.2) and the interrupt-based sampling (Section 3.1). The shown overhead is normalized to that of interrupt-based sampling approach for the respective application. We also mark the base costs (in the percentage of CPU consumption) of the interrupt-based sampling on top of corresponding bars. The base costs vary substantially across the five applications since they have different request granularities and employ different sampling frequencies (from once per 10 microsecond to once per millisecond) as described in Section 3.1.

Web server

System call name	*CPI change*
`writev`	Increase 3.66±2.27
`lseek`	Decrease 1.99±2.42
`stat`	Decrease 1.39±1.57
`poll`	Increase 1.22±2.17
`shutdown`	Increase 0.82±2.35
`read`	Increase 0.61±2.30
`open`	Decrease 0.14±1.38
`write`	Decrease 0.11±2.06

Table 2. Example mappings from system call names to the change of CPI over the 10-microsecond periods before and after the system call. The CPI changes are shown in average ± standard deviation. These examples are for Apache web server.

an enhancement of our approach that only samples processor counters at system calls that are likely to signal request behavior transitions.

During an online training process, at each moment of a system call, we map the system call name to the change of target execution metric over certain periods before and after the moment. While system calls of a specific name may be observed many times during execution, we continuously maintain the average and standard deviation of metric changes for it. The metric change average indicates the significance of subsequent behavior transitions while the standard deviation indicates their uniformity. As examples, Table 2 lists some mappings from system call names to subsequent changes in CPU cycles per instruction (CPI) for Apache web server. We can understand some of the system call-signaled CPI changes through the application semantics. For instance, the `writev` system call entrance signals the start of writing HTTP headers, which apparently exhibits high CPIs (probably due to its fragmented piecemeal accesses to memory). Consequently, the `writev` system call entrance typically signals a substantial increase of the CPI metric in Apache web server.

Based on these results, we choose a subset of the system calls that are most correlated with request behavior transitions and only use them as triggers for processor counter sampling. Specifically for the web server example, we select the following system call triggers: `writev`, `lseek`, `stat`, and `poll`. We compare the captured CPI variations using the enhanced sampling approach and the original system call-triggered sampling. Since the enhanced approach uses a subset of targeted system calls, for fair comparison, we set a smaller $T_{\text{syscall\_min}}$ for this approach so that the two approaches generate similar overall sampling frequencies (and therefore incurring similar sampling costs). Results show that our enhanced sampling approach using behavior transition signals improves the captured request behavior variation—specifically, the coefficient of variation for the produced samples (defined in Equation 1) increases from 0.60 to 0.65.

Our approach is simplistic in that it only uses the system call name as behavior transition signals. For a long request in which system calls of the same name may occur many times in different semantic contexts, it is unlikely for a system call name to consistently signal strong request behavior transitions. Consequently this approach is less effective for the other four applications (beyond the web server) with much longer requests. Possible improvements to our approach include employing more complex signals like a sequence of two or more recent system call names and system call caller addresses within the application. We do not investigate these improvements in this paper. Nevertheless, our work on the web server case study demonstrates the promises of system calls as behavior transition signals.

4. Variation-Driven Request Modeling

Identified request behavior variations can enable new and improve existing server system management functions. We provide case studies on practical variation-driven server system management. This section presents our results of fine-grained request behavior modeling. Section 5 describes our study of adaptive scheduling that eases resource usage contention on multicores.

In a server system, online continuous collection of per-request information can help construct workload models, classify workload patterns, and support performance projections. For instance, grouping similar requests into clusters helps understand the proportion of requests with different levels of resource consumption, which consequently enables offline performance projection on new processor/memory platforms. Capturing different request execution patterns may also support the detection of anomalous behaviors. Further, identifying a request online can help predict its property such as CPU consumption, and consequently enable adaptive request management.

4.1 Request Differencing

A key foundation for request modeling is the ability to quantify the difference between two requests. In Magpie [10], requests are identified using software event sequences (such as system calls) while the difference of two requests is quantified as Levenshtein's string edit distance [21] between their respective event sequences. More specifically, the difference is quantified as the minimum number of insertion, deletion, or substitution operations needed to transform one event sequence into the other.

In our past work [27], we construct request signatures using average metric values over request executions. Then the difference of two requests is simply quantified as the difference of their average metric values. Fine-grained request variation patterns (in the form of time-ordered varying metric values) represent a more precise form of request signatures. To effectively utilize such information, we need a measure that quantifies the difference between two ordered sequences of varying metric values.

A simple way to quantify the difference of two sequences of values is to compute their L1 distance. Consider a time-ordered

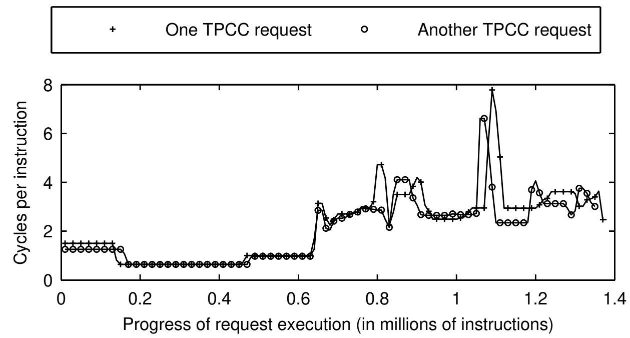

Figure 6. An example of two inherently similar TPCC requests whose executions drift apart slightly (with shifted peak points) after about 800,000 instructions.

sequence of measured metric values for request $\mathcal{X}$ to be $x_1, \cdots, x_m$. Each value in the sequence is measured for a fixed-length period (the length can be in time or in the number of executed instructions). Let the sequence for another request $\mathcal{Y}$ be $y_1, \cdots, y_n$. We define the two requests' L1 distance to be:

$$\text{L1\_distance}(\mathcal{X}, \mathcal{Y}) = \left[\sum_{i=1}^{\min\{m,n\}} |x_i - y_i| \right] + |m - n| \cdot p \quad (2)$$

Here p is the penalty for unequal request lengths. We set the penalty using a peak-level metric difference for the application. Specifically, we set it as the 99-percentile value of the distribution of metric differences at two arbitrary points of application execution.

However, the L1 distance may over-estimate the difference of two requests whose execution progresses slightly drift apart from each other. This may happen due to differing runtime environments like cache or lock contention. Further, inaccuracies in the per-request metric collection (*e.g.*, failure of maintaining proper request context in some cases [27, Section 4.1]) can lead to the appearance of time shifting between two requests' metric value sequences. Figure 6 illustrates an example of two inherently similar TPCC requests whose executions drift apart slightly after about 800,000 instructions.

The *dynamic time warping* is an approach to address the time shifting problems in time series comparisons. It was first proposed in automatic speech recognition [23, 25] and was more recently used in the alignment of computer system execution traces [16]. In our problem context, the dynamic time warping distance between two requests' metric value sequences can be understood as follows. For each request, we maintain a pointer that points to an element in the request's metric value sequence. A warp path w is defined as a series of steps of pointer moves. Formally, after the i-th warp step, let $w(\mathcal{X}, i)$ be the index location of the request $\mathcal{X}$ pointer and let $w(\mathcal{Y}, i)$ be that of the request $\mathcal{Y}$ pointer. Also let $w(\mathcal{X}, 0)$ and $w(\mathcal{Y}, 0)$ be the initial index locations of the two pointers. A valid s-step warp path must satisfy the following conditions:

- The two pointers initially point to the beginning of respective requests. In other words, $w(\mathcal{X}, 0) = 1$ and $w(\mathcal{Y}, 0) = 1$.
- The two pointers finally move to the end of respective requests. In other words, $w(\mathcal{X}, s) = m$ and $w(\mathcal{Y}, s) = n$.
- Two kinds of warp steps are allowed. First, both pointers can move to the next element in respective request sequences—$w(\mathcal{X}, i+1) = w(\mathcal{X}, i) + 1$ and $w(\mathcal{Y}, i+1) = w(\mathcal{Y}, i) + 1$. We call this a synchronous warp step. Alternatively, one pointer may move to the next element in its request sequence while the

other pointer remains unmoved. We call this an asynchronous warp step.

The distance of a warp path is defined as the sum of the metric differences at the two pointer locations over all warp steps, or:

$$\text{Distance of warp path } w = \sum_{i=0}^{s} |x_{w(\mathcal{X},i)} - y_{w(\mathcal{Y},i)}|. \quad (3)$$

Finally, the overall dynamic time warping distance is the minimum distance of all valid warp paths for the two requests. This can be solved using dynamic programming. The complexity of its computation is $O(m \cdot n)$, which is far higher than the $O(\max\{m, n\})$ complexity for the L1 distance computation.

The asynchronous warp steps allow the time shifting (like the example illustrated in Figure 6) without the cost of adding to the distance measure. This can address the over-estimation of the L1 distance. However, we find that such asynchronous warp steps can significantly under-estimate request differences through no-cost time shifting. Therefore we also consider a variant of the dynamic warping distance by applying a penalty for each asynchronous warp step. In practice, we set this penalty the same as the penalty for unequal request lengths in the L1 distance (or p in Equation 2).

So far we have described several alternative approaches for quantifying the difference between requests. We will compare their effectiveness in classifying similar requests into groups, presented in the next subsection.

4.2 Request Classification

The request metric variation patterns can serve as a form of signatures which allow us to classify requests into groups with similar patterns. Such classification can help workload characterization and performance prediction. In addition, the classification may uncover a small number of requests that do not share common execution behaviors.

A frequently used clustering algorithm is the k-means algorithm—it iteratively classifies requests into groups so that each group member is closer to the *group mean* than to the mean of any other group. However, the mean of a set of request variation patterns is not well defined. Therefore we employ a modified algorithm called k-medoids [18]. In this approach, a cluster centroid request replaces the cluster mean at each round of the k-means processing. We define the cluster centroid as the request whose sum of distances to all other cluster members is the minimum.

Following our algorithm, we created request clusters for all five applications. In this case study, we set the cluster number k at 10. We used a number of request differencing measures described earlier in Section 4.1, including Levenshtein's string edit distance of request system call sequences, the difference of average request metric values, the L1 distance of request metric value sequences, the dynamic time warping distance, and the dynamic time warping distance with enhanced penalties for asynchronous warping steps. Under each approach, we measure the request classification quality in terms of how closely cluster members resemble the respective cluster centroids.

Specifically, below we define cluster members' divergence from centroids in the request CPU execution time. Consider a request with CPU time $\mathcal{C}_r$ and its cluster centroid with CPU time $\mathcal{C}_c$. The request's divergence from its centroid is defined as $\frac{|\mathcal{C}_r - \mathcal{C}_c|}{\mathcal{C}_c} \times 100\%$. Such divergence from centroid, averaged over all requests, are shown in Figure 7(A) for all comparison cases. Similarly, Figure 7(B) shows the comparison results for another request property—the request peak (90-percentile) CPI. These results allow us to make following observations:

- Compared to other approaches, the dynamic time warping with enhanced penalties for asynchronous warp steps achieves high classification qualities for all cases. In particular, the asynchrony penalty is very important for achieving such results. Without it, the original dynamic time warping may produce very poor classifications, due to the under-estimation of request differences through no-cost time shifting.

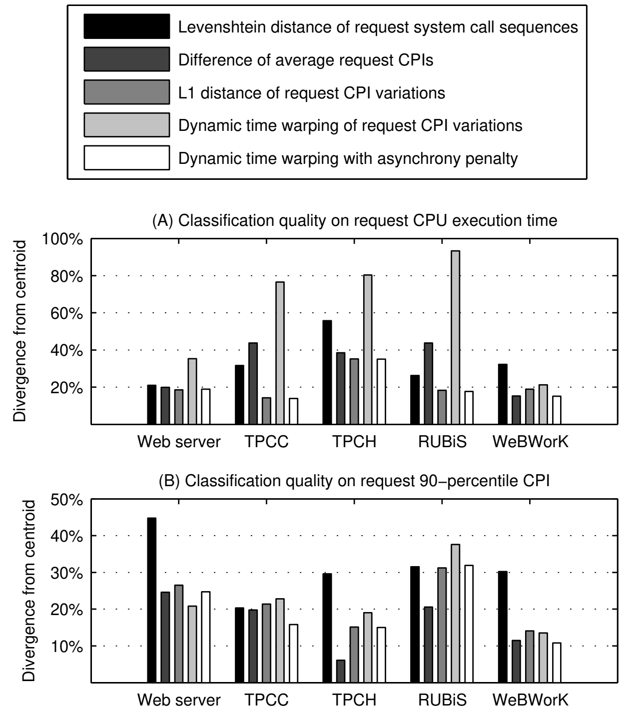

Figure 7. Request classification effectiveness when using different request differencing measures. The classification quality is measured as cluster members' divergence from respective cluster centroids. Such divergence can be defined on different request properties. This figure includes comparison results on two request properties—the request CPU time and the request peak (90-percentile) CPI.

- The software metric-only request differencing (Levenshtein distance of system call sequences) exhibits relatively poor classification qualities. This is because it fails to consider dynamic execution effects on resource-sharing processors like multicores.
- The difference on average request CPIs achieves high classification qualities on the peak request CPI (Figure 7(B)) due to the strong correlation between the classification factor (average CPI) and the target (peak CPI). However, it achieves poor classification qualities on the request CPU time (Figure 7(A)) compared to approaches that consider fine-grained request behavior variation patterns. This is because the latter represents a more precise form of request signatures.
- The L1 distance produces slightly worse classification qualities than the dynamic time warping with asynchrony penalty. This is due to its over-estimation of request differences for the time shifting cases explained in Section 4.1.

Our overall findings are that the dynamic time warping with asynchrony penalty is most effective in quantifying request differences. However, the L1 distance measure is also quite effective and its computation cost is significantly lower. It can be the more at-

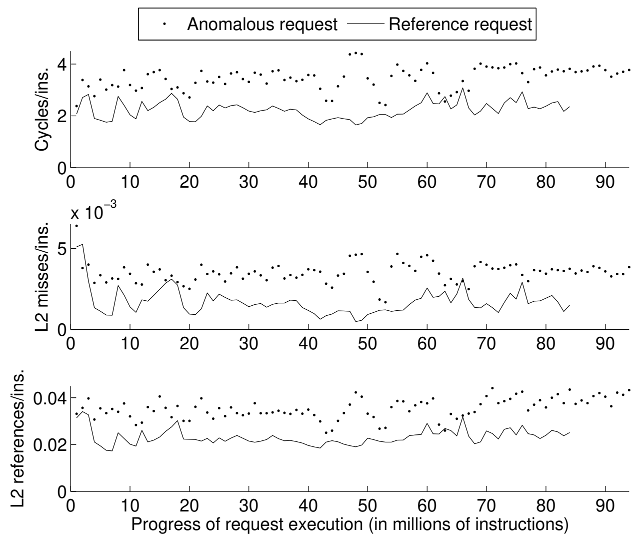

Figure 8. Comparison of metric variation patterns between an anomalous request in TPCH and the centroid of the group of requests processing the same TPCH query (Q20 in this case). We show the variation patterns on CPU cycles per instruction (CPI), L2 cache misses per instruction, and the L2 references per instruction.

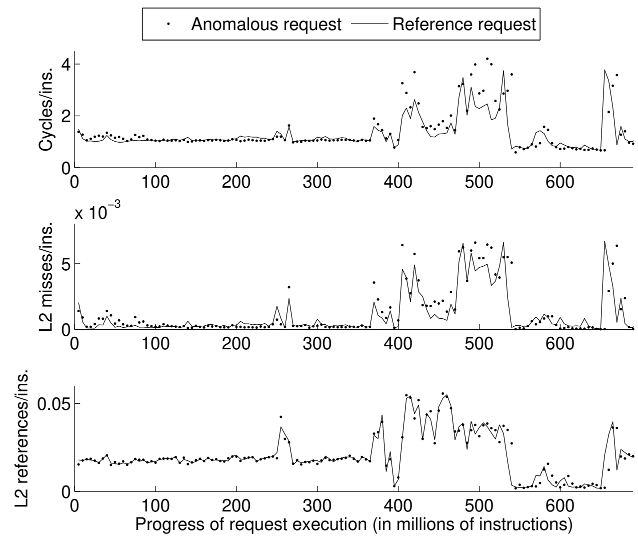

Figure 9. Comparison of metric variation patterns between an anomalous request in WeBWorK and a reference. We show the variation patterns on CPU cycles per instruction (CPI), L2 cache misses per instruction, and the L2 references per instruction. Both requests process the problem identifier of 954 in WeBWorK.

tractive approach when the cost of computing request differences must be kept low (particularly for online request modeling).

4.3 Anomaly Detection and Analysis

Informally, anomalous requests are those whose behaviors deviate from the expectation. Anomalies are oftentimes linked to worst-case performance that is of interests to system dependability and quality-of-service management. In our study, we are particularly concerned with the rare adverse effects of dynamic concurrent executions on hardware resource-sharing processors. For both anomaly detection and analysis, it is helpful to identify a reference request to the anomaly [28]. Specifically, we say that a request exhibits anomalous symptoms if its behavior deviates from that of a reference against the expected similarity between them.

With the identification of fine-grained request behaviors, we can detect potential anomalies in several ways. First, we consider a group of requests with similar application-level semantics and instruction streams (*e.g.*, those processing the same SQL query in TPCH). Requests with highest distances to the group centroid share least common execution behaviors with typical request patterns, and therefore we identify them as suspected anomalies. We can use the group centroid as the reference request in our anomaly analysis. As an example, Figure 8 compares the metric variation patterns between a suspected anomaly and its reference in TPCH. The comparison in CPI shows that the anomalous request exhibits poor performance (higher CPI) for much of its execution.

We can also detect potential anomalous requests through multi-metric differences. In this case study, our particular goal is to identify anomalies due to adverse effects of dynamic concurrent executions on L2 cache-sharing multicores. Specifically, we search for anomaly-reference request pairs in which the anomaly and its reference share very similar patterns on L2 references per instruction (*i.e.*, similar reference streams to the shared resource) but exhibit different performance on CPU cycles per instruction. In this offline analysis, we employ the dynamic time warping with asynchrony penalty (described in Section 4.1) as the request differencing measure. An an example, Figure 9 compares the metric variation patterns between a potential anomaly and its reference in WeBWorK. The comparison in CPI shows that the anomalous request exhibits higher CPIs in certain regions of execution (*e.g.*, about 500 million instructions since the request start).

We perform some analysis of the identified anomaly cases. In both cases, the anomalous patterns of CPI increases match very well with the anomalous patterns on the L2 cache misses per instruction. This is intuitive in that the poor performance on the shared resource (the L2 cache) is the primary reason for anomalous CPI performance. One important implication of this result is that the monitoring and prediction of the L2 misses per instruction can help understand and manage the request performance (including the worst-case behavior) on multicores.

For an anomaly-reference pair with similar application-level semantics and instruction streams, we expect that their L2 cache references per instruction patterns to be very similar. While this is true for the WeBWorK case, we observe some increases of the L2 reference rate during anomalous executions for the TPCH case result. We can think of two possible explanations for this. First, software-level contention (like a lock contention in the database) across multiple TPCH requests may lead to additional reference instructions, and consequently a higher data reference rate. Second, coherence misses at the L1 cache can cause additional L2 references during concurrent request executions. A supporting evidence for the first explanation is that the anomalous request executes more instructions than the reference does. If true, the first explanation warns that the poor request performance on multicores may be not only due to the competition on shared hardware resources, but also to potential software-level contention.

4.4 Online Request Signature Identification

Fine-grained request variation patterns represent a more precise form of request signature than the average metric value-based request signatures [27]. We can utilize it to identify a request on the fly. Specifically, shortly after a request begins its execution, we can match its partial request variation patterns against those in a bank of representative request signatures maintained by the system. The one with smallest difference is considered the match. By assum-

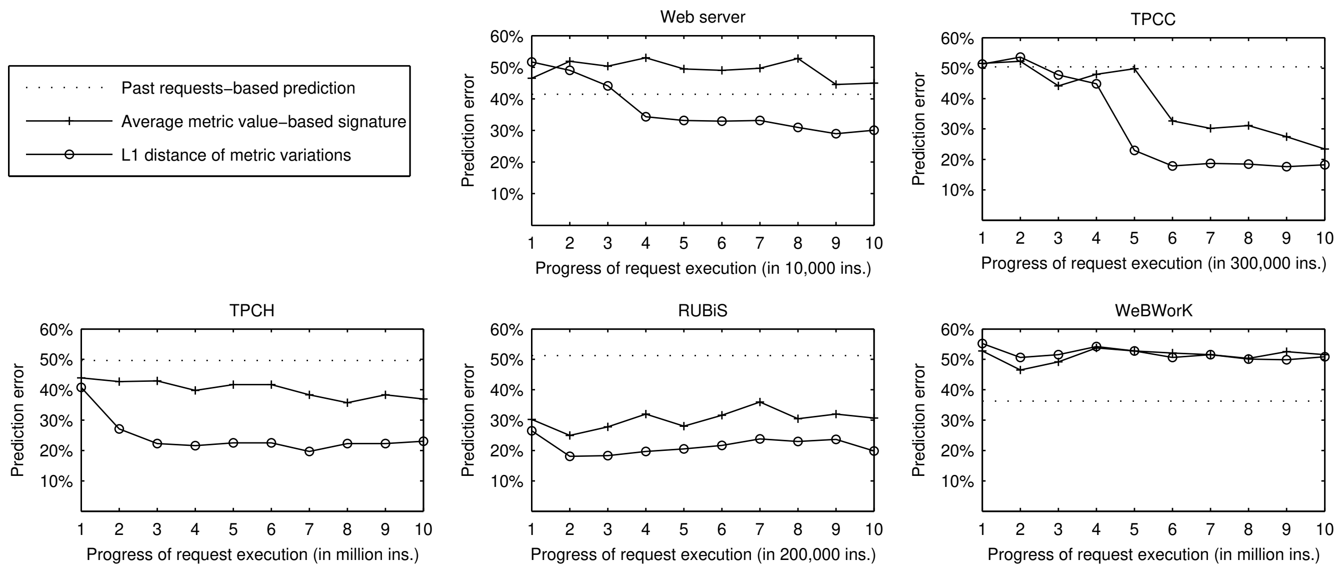

Figure 10. Effectiveness of the online request signature identification and CPU usage prediction. For each request, we show the prediction accuracy by utilizing request variation patterns for incremental amount of cumulative request executions—up to 10 million instructions for long-request applications TPCH and WeBWorK, and up to the median request length for the other three applications. The accuracy is defined as the percentage of requests with correct prediction (higher or lower than the median CPU usage).

ing the current request shares similar properties with the matched request in the bank, we can then predict the current request's properties well before it completes execution. For online request differencing, we use the L1 distance (defined in Section 4.1) for its low computation cost.

We evaluate the effectiveness of request variation pattern-driven online signature identification. In our evaluation, we collect a bank of 500 representative request signatures for each application. While the request signature can be the variation pattern of any hardware counter metric, we choose one that reflects the inherent request behavior free of dynamic effects on the shared L2 contention. Specifically, we employ the variations of L2 references per instruction as the signature. We utilize the request signature identification to predict whether the request CPU consumption is going to be higher or lower than a threshold. For each server application, the threshold is set as the median CPU usage of all requests in the workload.

We compare the effectiveness of the variation pattern-driven request signatures against the request signatures constructed from average metric values (used in our past work [27]). In both cases, the online identification utilizes the partial request execution (since the beginning of request execution) to construct its signature. For an additional comparison base, we look for a representative conventional approach that is also transparent to server applications (*i.e.*, requiring no application instrumentation or assistance). Fundamentally, without online information about an incoming request, there is little other choice but to use recent past workloads as the basis to predict incoming workloads. Specifically, we estimate the CPU usage of each request as the average CPU consumption of 10 recent past requests.

We compare the prediction accuracy under these approaches. The predication accuracy is the percentage of correctly identified request CPU usage (higher or lower than the threshold). For online identifications, we show the results on using incremental amount of cumulative request executions. Results in Figure 10 demonstrate that the variation-driven request signatures lead to significantly better predication accuracy than average metric value-based signatures for the web server, TPCC, TPCH, and RUBiS. Specifically, the prediction errors are reduced by around 10% or more for these applications. However, we observe poor effectiveness of both forms of request signatures for WeBWorK. We discover that all WeBWorK requests follow almost identical processing semantics for the early part of request executions. Signatures constructed for the first 10 million instructions (out of the total several hundred million instructions in a typical WeBWorK request) cannot effectively identify the requests.

5. Variation-Driven Request Scheduling

Online identification of request behavior variations can also enable adaptive CPU scheduling on multicore platforms. The basic idea is that by matching appropriate request execution periods in co-execution, we may improve efficiency by easing the resource contention. In a more limited scenario, the worst-case performance often arises from coincidental co-execution of peak-resource-usage request periods. The avoidance of such cases would improve the system performance dependability.

Our study is naturally related to previous multicore adaptive scheduling research for non-server applications [14, 15, 38]. Resource-aware request scheduling in server applications, however, requires the identification of request-level workload behavior variations. It also introduces significant challenges when the variation patterns are substantially more fine-grained than typical CPU scheduling quanta.

5.1 Online Behavior Prediction

To support online system adaptation like the resource-aware CPU scheduling, it is important to predict request behavior variations on the fly. Specifically, at each sampling moment, we need to estimate the target metric value for the coming execution period (until the next counter metric sampling). Our choices of online predictors are limited by our OS-only management that does not leverage any compiler assistance or special hardware support. For instance, we do not possess the program-level statistics like the basic block vector [30].

We are particularly interested in the exponentially weighted moving average (EWMA) filters due to their low-cost maintenance in an online continuous fashion. The basic EWMA filter, like the one used for predicting the network round-trip time in TCP congestion control, can be continuously maintained in the following way:

$$E_k = \alpha \cdot E_{k-1} + (1 - \alpha) \cdot O_k. \qquad (4)$$

Here E_k is the new estimate of the target metric while E_{k-1} is the last estimate. O_k is the current observation. The gain parameter, α, adjusts the balance between the filter's stability and agility.

While the EWMA filter exponentially weighs the past samples, it assumes each new sample leads to an equal amount of aging for previous samples. In our processor counter sampling approach, only periodic interrupts produce fixed-length samples. Samples collected at request context switches and system calls may have widely varying time durations. Here we introduce a variable-aging EWMA filter, called *vaEWMA*, that considers this factor. Specifically, let t_i be the length of observation i and the unit length be $\hat{t}$. Then our vaEWMA filter can be incrementally maintained as follows:

$$E_k = \alpha^{t_k/\hat{t}} \cdot E_{k-1} + (1 - \alpha^{t_k/\hat{t}}) \cdot O_k. \qquad (5)$$

Expressed only using observed samples, the prediction is:

$$E_k = \sum_{i=0} \alpha^{\sum_{j=k-i+1}^{k} t_j/\hat{t}} \cdot (1 - \alpha^{t_{k-i}/\hat{t}}) \cdot O_{k-i}. \qquad (6)$$

Note that the purpose of Equation 6 is to show the actual weight for each observed sample. The computation in practice follows the much simpler Equation 5.

We evaluate the effectiveness of the online request behavior prediction. We introduce two additional predictors for the comparison purpose. The first assumes the request behavior does not vary so it employs the request average metric value (using cumulative data from the request beginning to the prediction point) to predict the metric value at each point. The second assumes short-term stable behaviors so it predicts the metric value for the next period using that of the last.

Our experimental work in this section focuses on the long-request applications TPCH and WeBWorK. This is because sub-request-granularity scheduling makes little sense for short requests that are comparable to or more fine-grained than a typical CPU scheduling quantum. We compare the request behavior prediction accuracy (on predicting L2 cache misses per instruction) of the these approaches. For the vaEWMA filter, we show prediction results with multiple settings for the gain parameter α. The unit observation length $\hat{t}$ is 1 millisecond.

Figure 11 presents the prediction accuracy in the root mean square errors. Specifically, let the request execution consist of n sample periods of lengths $t_1, \cdots, t_n$. The actual target metric values for these samples are $x_1, \cdots, x_n$ respectively. Predicted values under a particular approach are $\hat{x_1}, \cdots, \hat{x_n}$. Then we have:

$$\text{Root mean square error} = \sqrt{\frac{\sum_{i=1}^{n} t_i \cdot (x_i - \hat{x_i})^2}{\sum_{i=1}^{n} t_i}} \qquad (7)$$

Results in Figure 11 show that our EWMA filters (with appropriate settings of the gain parameter α) achieve better prediction than alternative approaches for our test cases. This is because they adapt to request behavior changes while avoiding instability due to short-term fluctuations. For the rest of our case study in this paper, we set the gain parameter $\alpha = 0.6$. However, we do not claim the general applicability of this setting. Application-specific calibration of the gain parameter may be necessary.

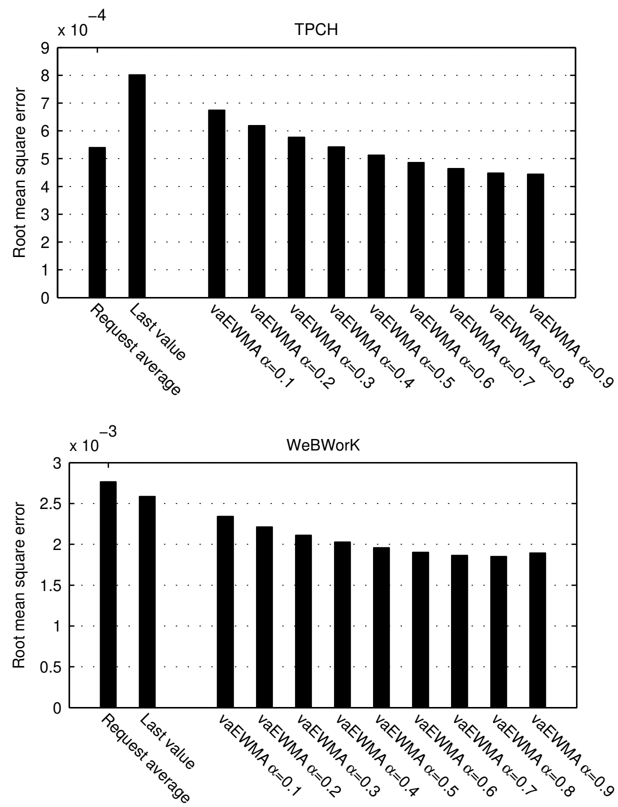

Figure 11. Accuracy of predicting L2 cache misses per instruction for TPCH and WeBWorK using several online predictors.

5.2 Contention-Easing CPU Scheduling

We then study on the potential effectiveness of request behavior variation-driven scheduling to ease multicore resource contention. At the high level, our scheduling policy is that requests during their high resource usage periods should avoid co-execution as much as possible. We implement such a policy in the Linux 2.6.18 kernel. At each scheduling opportunity, the scheduler performs the following actions:

1. It checks whether any other CPU core is currently executing a request during a high resource usage period. If not, the scheduler chooses the request at the head of its local `runqueue` in the normal fashion.
2. Otherwise, it searches its local `runqueue` for a request that is *not* in a high resource usage period. If multiple such requests exist, it picks the one closest to the `runqueue` head for execution. If no such request exists, it gives up by scheduling in the normal fashion. Note that our current implementation does not migrate requests between different CPU core `runqueues` for simplicity.

Two issues are worth further discussions. First, general-purpose operating systems often employ large CPU scheduling quanta to avoid frequent cache pollution across context switches. In Linux, a scheduling quantum can be as long as 100 milliseconds. Such large quanta leave little room for fine-grained adaptive scheduling. To overcome this, we modify the scheduler to attempt request re-scheduling at no more than 5 millisecond intervals. With frequent re-scheduling, the cache pollution costs at context switches can significantly negate the benefits of adaptive scheduling. Consider an

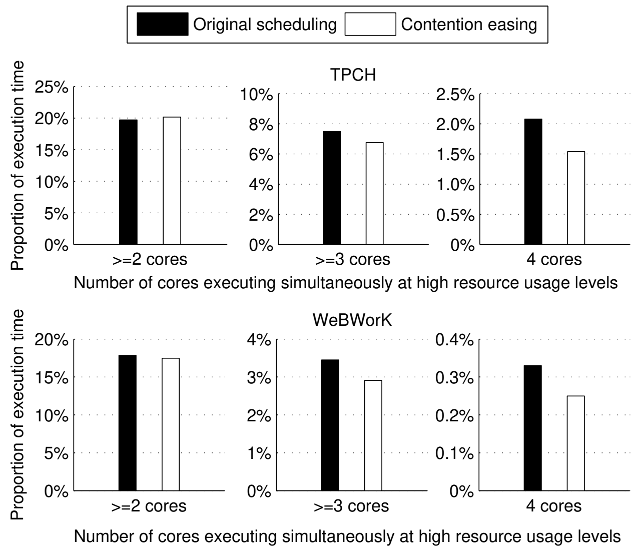

Figure 12. Effectiveness of contention-easing request scheduling for TPCH and WeBWorK. For each scheduling approach, we show the proportion of execution time when multiple CPU cores are simultaneously executing at high resource usage levels.

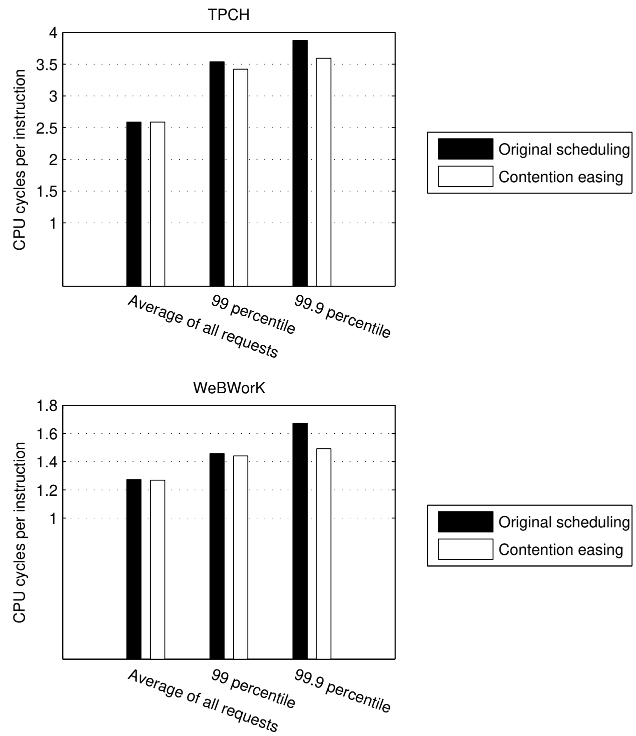

Figure 13. The request performance of CPU cycles per instruction (CPI) under contention-easing CPU scheduling for TPCH and WeBWorK. Lower CPI indicates better performance. We show both average and worst-case (in high-percentile CPI) request performance.

extreme (and unlikely) worst case, we have devised a microbenchmark to record a context switch cache pollution cost at more than 12 milliseconds on our experimental platform. To minimize unnecessary re-scheduling, we keep the current request at the head of the local `runqueue` before each attempt of adaptive scheduling. The current request would resume execution (without cache pollution) if no contention-easing re-scheduling opportunity emerges.

Second, the OS needs to monitor an appropriate hardware counter metric that indicates the resource usage intensity of the request executions. In our experiments, we use the metric of L2 cache misses per retired instruction. It not only reflects the performance of the shared on-chip L2 cache, but also provides an indication of the memory bandwidth usage. The latter is particularly important for fine-grained requests that do not have large working sets. Consequently their performance is more constrained by the memory bandwidth than by the L2 cache space. Finally, our anomaly analysis in Section 4.3 showed that high L2 misses per instruction are good indicators of worst-case poor performance. For each application, our experiments use the 80-percentile value of L2 cache misses per instruction as the threshold between high and low resource usage.

Figure 12 illustrates the effectiveness of contention-easing request scheduling for TPCH and WeBWorK. The shown results are averaged over three 1000-request test runs. Our scheduler tries to avoid co-execution of high-resource-usage cores in the system. We observe a reduction of high resource contention under the contention-easing request scheduling. In particular, the most intensive contention periods (when all four cores execute at high resource usage levels) are reduced by around 25% for the two applications. On the other hand, our contention-easing scheduling is unable to completely eliminate all high-contention executions. One reason is that our online behavior prediction (described in Section 5.1) still exhibits significant errors such that many high contention periods cannot be predicted for avoidance. Further, the high resource usage periods in some applications (particularly WeBWorK, as shown in Figure 2) are unstable and their granularities can be far shorter than the CPU scheduling quantum.

While our scheduler can reduce the resource usage contention on the multicore platform, we are interested to see whether such reduction translates into request-level performance enhancements. Specifically, Figure 13 shows the request CPI performance of contention-easing CPU scheduling. Results show that the contention-easing scheduling can reduce the worst-case request CPI (around 10% reduction for the two applications) but it does little to improve the average request performance. This mixed result is largely due to our scheduler's focus on the worst-case resource contention. An improvement to our scheduling approach is possible but challenging. In particular, a scheduler that attempts to resolve mild resource contention may frequently fail to find suitable scheduling targets. On the other hand, relieving the worst-case resource contention still brings a significant benefit in the system performance dependability. For instance, service-level agreements may specify requirements in high-percentile worst-case request performance.

6. Related Work

Application behavior variations on hardware execution metrics have long been recognized. In particular, there is a large body of work on application phase identification and recognition in the literature [12, 13, 17, 29, 30]. Compared to these studies, our work is unique in terms of supporting request-oriented server workloads and operating system-level management. Specifically, server applications possess different workload features from those of technical computing and workstation applications. For instance, frequent network and storage I/O operations, along with the increasingly componentized server architectures, result in behavioral fluctuations that may not form long stable phases. Server systems also introduce additional concerns such as high execution concurrency and new management functions like online request classification. Finally, our work targets transparent OS-level system management requiring no change in software applications or special hardware assistance. We only utilize control mechanisms and information available to the OS.

Barroso *et al.* [11] and Keeton *et al.* [19] examined architectural hardware execution characteristics for commercial server workloads. In particular, they discovered that processor features optimized for technical workloads may not be as effective for commercial server workloads. Besides separating user and kernel-level execution statistics, these studies were limited to reporting aggregate architectural characteristics for the whole server applications. However, request-level behavior characterization is necessary for fine-grained system management such as adaptive CPU scheduling and request classification.

Previous research has proposed OS-level techniques for online workload characterization. For instance, Anderson *et al.* presented a continuous profiling infrastructure that can accurately account low-level architectural events for individual instructions [8]. Barham *et al.*'s Magpie system addressed the challenge of attributing collected system metrics to individual requests in concurrent server environments [10]. Shen *et al.* further presented per-request characterization of hardware execution metrics [27]. Built on top of the previous results, this paper characterizes workload behavior variations within individual requests. Consequently, our work can enhance the precision of characterized workload behaviors and enable new fine-grained system management functions.

To achieve high performance and fairness, the system resource management cannot ignore the increasingly ubiquitous on-chip hardware resource sharing in today's multiprocessors. Recent research proposals include directly partitioning the shared on-chip cache [22, 34, 39] as well as indirectly managing resource contention through contention-easing scheduling [14, 38], fairness-oriented execution timeslice adjustment [15], and targeted execution throttling [40]. For instance, Fedorova *et al.* proposed that the task CPI can be a good indicator of CPU pipeline usage and simultaneously scheduling high-CPI tasks with low-CPI tasks on a multi-threading processor can reduce contention on the pipeline resource [14]. Existing results were mostly applied to non-server applications or they treat the whole server application as a scheduling unit. To support adaptive resource management within a single server application, it is essential to exploit fine-grained workload behavior variations at the request level.

As a form of sub-request granularity resource management, previous research investigated staged server processing architectures. Specifically, SEDA divides each request execution into a number of event-driven stages and applies staged resource allocation and control [37]. Cohort scheduling advocates the aggregate execution of similar stages across multiple requests to achieve high cache utilization efficiency [20]. Capriccio employs resource-aware task scheduling to prioritize request stages according to the availability of their needed resources [36]. Most of these approaches require manual programmer specification (*e.g.*, in the form of event-driven programming) to mark request stages. Capriccio adds stage boundary annotations automatically through compiler support. In contrast, our characterization of request behavior variations may transparently identify potential stage transitions at the OS and annotate each stage with its unique hardware execution characteristics.

7. Conclusion

This paper provides a characterization of request behavior variations using several realistic server applications. We find that the inter-core resource sharing on multicore platforms obfuscates the request execution performance, leading to increased behavior variation. Further, our characterization shows that individual request executions exhibit fine-grained behavior variation patterns that are often stronger than the inter-request differences. This paper also presents operating system management to track request behavior variations online. Our OS management utilizes the frequent system calls in server requests to perform low-cost in-kernel event sampling. Further, the semantic implications of system call events allow some to act as signals for impending behavior transitions which can be exploited to improve the cost-effectiveness of online variation tracking.

Identified request behavior variation patterns represent a strong request signature that is more descriptive than the average metric value. Such signatures can help classify requests into groups with common variation patterns, and at the same time detect anomalous patterns that are often linked to worst-case performance. It can also support better online request identification and resource usage prediction.

Finally, our research produced mixed results for variation-driven request scheduling to ease multicore resource contention. In particular, we find that our contention-easing scheduling is unable to significantly improve the average request performance. This is in part due to the difficulty of accurately predicting online request behaviors in order to perform contention-easing scheduling. Another reason is that many request variation stages in realistic server applications are finer-grained than the typical operating system scheduling quantum. On the positive side, the contention avoidance tends to be more effective when it focuses on the rare, most intensive resource contention. Despite the lack of improvement on the average request performance, we see the encouraging result that variation-driven request scheduling can help alleviate worst-case performance. This enhances system dependability and better satisfies service-level agreements that specify requirements in high-percentile worst-case request performance.

For future work, our characterized request workload may serve as input to server system performance models to predict performance or its bounds under different system configurations. In particular, fine-grained behavior variation patterns can help project request resource consumption on a new hardware platform. In addition, our current prototype system only supports server applications running on a single machine. This is only due to the limitation of our request tracking and statistics maintenance infrastructure. The online management of request behavior variations across a distributed server architecture can expose both local and inter-machine variations. This would present a new dimension for request behavior classification. It may also guide additional distributed system resource management such as component placement.

Acknowledgments

We received help from Xiao Zhang (University of Rochester) on configuring and reading processor hardware counter registers on our experimental platform. We thank the anonymous ASPLOS reviewers for comments that helped improve this paper. We also thank our shepherd Richard Draves (Microsoft Research) for assistance in the final revision.

References

[1] Intel 64 and IA-32 architectures software developer's manual volume 3B: System programming guide, part 2, table B-7. http://download.intel.com/design/processor/manuals/253669.pdf.

[2] Moodle course management system. http://moodle.org/.

[3] RUBiS: Rice University Bidding System. http://rubis.objectweb.org.

[4] SPECweb99 benchmark. http://www.specbench.org/osg/web99.

[5] TPC-C benchmark. http://www.tpc.org/tpcc.

[6] TPC-H benchmark. http://www.tpc.org/tpch.

[7] WeBWorK: Online homework for math and science. http://webwork.maa.org/moodle/.

[8] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. on Computer Systems*, 15(4):357–390, November 1997.

[9] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Trans. on Computer Systems*, 18(3):197–228, August 2000.

[10] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modeling. In *6th USENIX Symp. on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[11] L.A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *25th Int'l Symp. on Computer Architecture*, pages 3–14, Barcelona, Spain, July 1998.

[12] A.P. Batson and A.W. Madison. Measurements of major locality phases in symbolic reference strings. In *ACM SIGMETRICS*, pages 75–84, Cambridge, MA, March 1976.

[13] A.S. Dhodapkar and J.E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Int'l Symp. on Computer Architecture*, pages 233–244, Anchorage, AL, May 2002.

[14] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. In *SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[15] A. Fedorova, M. Seltzer, and M.D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 25–38, Brasov, Romania, September 2007.

[16] M. Hauswirth, A. Diwan, P.F. Sweeney, and M.C. Mozer. Automating vertical profiling. In *20th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–296, San Diego, CA, October 2005.

[17] C. Isci and M. Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *12th Int'l Symp. on High-Performance Computer Architecture*, pages 121–132, Austin, TX, February 2006.

[18] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: An introduction to cluster analysis*. Wiley, New York, 1990.

[19] K. Keeton, D.A. Patterson, Y.Q. He, R.C. Raphael, and W.E. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. In *25th Int'l Symp. on Computer Architecture*, pages 15–26, Barcelona, Spain, July 1998.

[20] J.R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *USENIX Annual Technical Conf.*, Monterey, CA, June 2002.

[21] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10, 1966.

[22] 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 *14th Int'l Symp. on High-Performance Computer Architecture*, Salt Lake City, UT, February 2008.

[23] C. Myers, L.R. Rabiner, and A.E. Rosenberg. Performance tradeoffs in dynamic time warping algorithms for isolated word recognition. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 28(6):623–635, December 1980.

[24] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical report, Department of Computer Science and Engineering, University of Washington, May 2000.

[25] H. Sakoe and S. Chiba. Dynamic programming optimization for spoken word recognition. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 26(1):43–49, February 1978.

[26] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based internet services. In *5th USENIX Symp. on Operating Systems Design and Implementation*, pages 225–238, Boston, MA, December 2002.

[27] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. In *13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 189–200, Seattle, WA, March 2008.

[28] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS*, pages 85–96, Seattle, WA, June 2009.

[29] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, Boston, MA, October 2004.

[30] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Int'l Symp. on Computer Architecture*, pages 336–349, San Diego, CA, June 2003.

[31] A. Snavely and D. Tullsen. Symbiotic job scheduling for a simultaneous multithreading processor. In *9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, Cambridge, MA, November 2000.

[32] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *Second USENIX Symp. on Networked Systems Design and Implementation*, pages 71–84, Boston, MA, May 2005.

[33] C. Stewart, M. Leventi, and K. Shen. Empirical examination of a collaborative web application. In *IEEE Int'l Symp. on Workload Characterization*, Seattle, WA, September 2008.

[34] 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, June 2007.

[35] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *5th USENIX Symp. on Operating Systems Design and Implementation*, pages 239–254, Boston, MA, December 2002.

[36] R. von Behren, J. Condit, F. Zhou, G.C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *19th ACM Symp. on Operating Systems Principles*, pages 268–281, Bolton Landing, NY, October 2003.

[37] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *18th ACM Symp. on Operating Systems Principles*, pages 230–243, Banff, Canada, October 2001.

[38] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, May 2007.

[39] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multi-core cache management. In *4th European Systems Conf.*, pages 89–102, Nuremberg, Germany, April 2009.

[40] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conf.*, San Deigo, CA, June 2009.