

Hardware Counter Driven On-the-Fly Request Signatures^{*}

Kai Shen Ming Zhong[†] Sandhya Dwarkadas Chuanpeng Li Christopher Stewart Xiao Zhang

Department of Computer Science, University of Rochester
{kshen, zhong, sandhya, cli, stewart, xiao}@cs.rochester.edu

Abstract

Today’s processors provide a rich source of statistical information on application execution through hardware counters. In this paper, we explore the utilization of these statistics as request signatures in server applications for identifying requests and inferring high-level request properties (e.g., CPU and I/O resource needs). Our key finding is that effective request signatures may be constructed using a small amount of hardware statistics while the request is still in an early stage of its execution. Such *on-the-fly* request identification and property inference allow guided operating system adaptation at request granularity (e.g., resource-aware request scheduling and on-the-fly request classification). We address the challenges of selecting hardware counter metrics for signature construction and providing necessary operating system support for per-request statistics management. Our implementation in the Linux 2.6.10 kernel suggests that our approach requires low overhead suitable for runtime deployment. Our on-the-fly request resource consumption inference (averaging 7%, 3%, 20%, and 41% prediction errors for four server workloads, TPC-C, TPC-H, J2EE-based RUBiS, and a trace-driven index search, respectively) is much more accurate than the online running-average based prediction (73–82% errors). Its use for resource-aware request scheduling results in a 15–70% response time reduction for three CPU-bound applications. Its use for on-the-fly request classification and anomaly detection exhibits high accuracy for the TPC-H workload with synthetically generated anomalous requests following a typical SQL-injection attack pattern.

Categories and Subject Descriptors D.4.7 [Operating Systems]: Organization and Design

General Terms Measurement, Performance, Design, Reliability, Experimentation

Keywords Operating system adaptation, Hardware counter, Server system, Request classification, Anomaly detection

^{*}This work was supported in part by the National Science Foundation (NSF) grants CCR-0306473, ITR/IIS-0312925, CNS-0411127, CAREER Award CCF-0448413, CNS-0509270, CNS-0615045, CNS-0615139, CCF-0621472, and CCF-0702505; by NIH grant 1 R21 GM079259-01A1; and by several IBM Faculty Partnership Awards.

[†]Zhong is currently affiliated with Google (mzhong@google.com).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLoS’08, March 1–5, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-958-6/08/0003...\$5.00

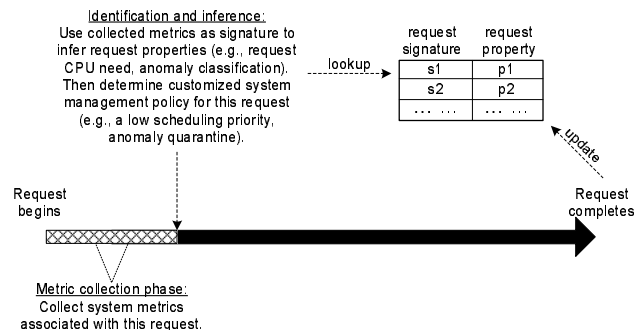


Figure 1. A single request’s view of our on-the-fly request signature identification and request property inference.

1. Introduction

Many operating system (OS) management functions benefit from the knowledge of runtime workload properties. For instance, server requests can be scheduled for better performance or quality-of-service if each request’s resource needs are known at scheduling time [28, 19]. As another example, components of a distributed network service can be better composed to save communication costs if the inter-component communication patterns are known when the service composition policy is determined [24]. Conventional profiling approaches such as offline profiling or history-based online prediction rely on past request statistics to predict future workload properties. However, as input parameters vary and runtime conditions change, the properties of individual request executions may deviate significantly from general patterns of past requests (even from very recent ones).

This paper explores a new approach for server request identification and property inference (illustrated in Figure 1). By collecting available system-level metrics and attributing them to specific request contexts, we can use such metrics as signatures to identify requests and infer request properties based on known signature-to-property mappings. We construct and utilize request signatures while a request executes (or *on-the-fly*). Such on-the-fly request signatures facilitate request-granularity OS adaptations. Although our goals of request identification and property inference may be realized through direct application assistance or manipulation, our approach functions *transparently* at the OS (*i.e.*, requiring no change of or assistance from applications or middleware software running above the OS). Transparent system management provides more general benefits and it is essential for third-party management environments such as service hosting platforms.

Many types of system-level metrics can be transparently collected in today’s computing systems. For example, modern processors, through a set of counter registers, provide detailed hardware information such as instruction mix, rate of execution (instructions per cycle), branch (control flow) prediction accuracy, and memory

access behavior (including miss rates at each level of the memory hierarchy as well as bus activity). At the software level, the OS processing also leaves a trail of statistics such as task context switch rate, system call frequency, and I/O operation pattern. In this paper, we focus on hardware counter driven request signatures. We believe this is an appropriate first-step effort due to two advantages of hardware counter metrics: 1) event counter maintenance in hardware requires no runtime overhead; 2) OS processing statistics may be scarce in applications with few system calls and I/O activities while processor hardware metrics are consistently available during execution.

We address two key challenges in supporting hardware counter driven on-the-fly request signatures. First, we derive general principles by which to guide the selection of hardware counter metrics used in the construction of request signatures. Our investigation focuses on several factors that affect the metric effectiveness as a request signature: the metric normalization base, environmental dynamics, and application variations. One notable factor unique to server applications is the *concurrent* execution of multiple requests and associated frequent context switches. Second, we propose OS mechanisms for transparent online collection and management of per-request counter metrics. This is challenging due to the continually changing request-to-process-binding in multi-component server systems. We employ a transparent message tagging scheme so that request contexts can be tracked across multiple server components with no application assistance.

Based on our on-the-fly request signature, we demonstrate its effectiveness or potential in assisting request-granularity system adaptations through case studies. First, the online knowledge of each request’s resource needs makes it possible to realize shortest-remaining-time-first scheduling [21], which is known to achieve minimal average request response time. Further, on-the-fly request signature identification presents the opportunity for early request classification and anomaly detection. By classifying a request early, an online workload tracker may save the overhead for further tracing and event logging on the classified request. By detecting anomalous requests early, the system may apply targeted monitoring or even online request quarantine.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 investigates several factors associated with the use of hardware counter metrics as request signatures. Section 4 describes the OS mechanisms necessary for on-the-fly request context tracking, per-request counter metric collection, as well as request identification and property inference. Section 5 provides evaluation results using several server applications and benchmarks. Section 6 illustrates possible OS adaptations that can benefit from on-the-fly request identification and property inference. Section 7 concludes the paper with a summary of findings.

2. Related Work

Continuous profiling (DCPI [2]) is a sampling-based profiling system that uses hardware counters to characterize activities within the entire system. More recently, Barham *et al.* (Magpie [8]), Chen *et al.* (Pinpoint [12]), and Aguilera *et al.* [1] presented techniques to capture the resource demands and other properties of application requests as they are serviced across components and machines. Magpie uses clustering to classify requests and summarize the behavior of a workload. The tools resulting from these studies are excellent for offline (or online history-based) performance analysis and debugging. However, they do not provide on-the-fly identification and behavior prediction with respect to individual requests before their executions complete. Such on-the-fly prediction is essential for request-granularity online system adaptation.

For the purposes of architectural and program adaptation, prior work has utilized hardware counter metrics to dynamically iden-

tify execution phases [14, 23], and to predict other system properties [15]. Hardware counter metrics were also employed to identify appropriate simulation points for desired workloads [16], to predict CPU scheduling performance [10, 29], and to detect anomalies [25]. Our use of hardware counters as request signatures presents unique challenges associated with our target server environments. In particular, the effectiveness of hardware counters as request signatures is substantially affected by concurrent request execution and frequent context switches. Further, it is challenging to attribute collected counter metrics to appropriate requests on-the-fly.

Cohen *et al.* [13] showed that a set of system metrics (mostly in software) can serve as signatures that cluster system failures of similar types. Gniady *et al.* [17] used program counter-based classification to estimate application I/O access patterns at a particular point of execution. Although their studies do not address request-granularity workload identification, their choices of system metrics (other than hardware counters) can be incorporated into our request signatures. Additional system metrics can potentially improve the effectiveness of our request signature while possibly incurring additional runtime overhead. Further investigation would be necessary to evaluate the benefit and cost of such extensions.

Our on-the-fly request signature attempts to identify requests and infer high-level request properties. Although application-level information can infer request properties for applications with simple semantics (*e.g.*, request resource consumption inference through the requested file size in a static-content web server [19]), such application-level inference is difficult for server applications with more complex semantics. Further, direct application involvement compromises system transparency. Consequently, its benefit is restricted to specific applications and it is difficult to deploy when application changes are not allowed.

3. Hardware Metrics As Request Signatures

We provide a simple example to motivate the use of hardware metrics as request signatures. Figure 2 shows the cumulative values of one hardware counter metric for four different requests (running four different TPC-H [27] queries). In this example, the hardware metric (floating point operations per CPU cycle) serves as a good signature to differentiate TPC-H Q4 and Q3. This is the case even when statistics are collected for only a few milliseconds after requests begin execution. However, this metric does a relatively poor job in differentiating Q13 from Q17 — even though, these requests have very different CPU needs. Additional hardware metrics may help differentiate them.

In this section, we derive an understanding of the effectiveness of the use of individual hardware metrics as request signatures. Such analysis is essential to selecting an appropriate set of metrics in request signature construction. Metric selection is necessary because the processor usually has a limited number of physical counter registers to which the hardware metrics must map. Additionally, the configurations of some counter metrics are in conflict with each other and thus they cannot be observed simultaneously. Although multiple sets of metrics may be mapped to limited counter registers through time-division multiplexing [5, 29], they provide inaccurate event count statistics that are inappropriate for our on-the-fly request identification. In addition to accommodating the limited counter registers, metric selection is also desirable in order to screen out hardware metrics that provide little assistance in identifying and differentiating requests with different behavior.

The challenge of selecting the right set of hardware metrics lies in the various factors that may affect each metric’s effectiveness as a component of the request signature. Specifically, we investigate several such factors: 1) time-based normalization (per-cycle metric) vs. progress-based normalization (per-instruction metric); 2)

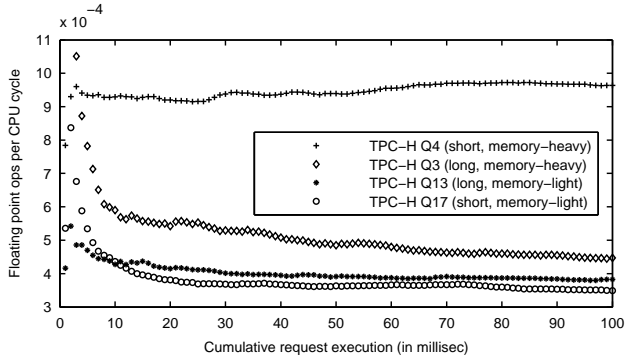


Figure 2. The cumulative floating-point-ops-per-CPU-cycle (up to 100 ms) for four requests (running four different TPC-H [27] queries) with different CPU needs and memory usage intensity. Here we identify a request’s memory usage intensity through the hardware metric of memory-bus-event-count-per-CPU-cycle.

environmental dynamics such as concurrent request execution and processor hardware resource sharing; 3) application-specific characteristics.

To facilitate our study, we define an effectiveness measure for a hardware metric to serve as request signature. The intuition behind our measure (called *metric-request-correlation*) is as follows — if two requests with similar hardware metric values are likely to be inherently similar requests, then the metric serves as a good request signature. In the context of our study, we assess the inherent similarity of two requests using the difference of their resource consumption (e.g., CPU usage for CPU-bound requests). Specifically, given n request-pair samples, let m_i be the difference of hardware metric values for the i -th request pair. Let r_i be the difference of their resource consumption. Further, let m_i ’s and r_i ’s have expected means \bar{m} , \bar{r} , and non-zero standard deviations σ_m , σ_r . We calculate their correlation coefficient as:

$$p_{m,r} = \frac{\text{Covariance}(m,r)}{\sigma_m \cdot \sigma_r} = \frac{\sum_{i=1}^n (m_i - \bar{m})(r_i - \bar{r})}{n \cdot \sigma_m \cdot \sigma_r} \quad (1)$$

A larger coefficient indicates a stronger positive correlation between hardware metric similarity and inherent request similarity. Note that a correlation coefficient cannot exceed 1.0.

3.1 Impact of Normalization Bases

As a request’s execution progresses, we can acquire stable hardware counter-based metrics by normalizing the hardware event count with the elapsed time (available in CPU cycles). However, in a concurrent server environment, the same request execution (represented by a unique sequence of instructions) may make non-deterministic progress within the same number of CPU cycles. Since many hardware event counts are linearly correlated with instruction executions, the unstable execution progress introduces noise in these hardware event counts within a given time period. This motivates a progress-based normalization, or per-instruction hardware event count metric, with the goal of reducing dependence on environmental variations.

To assess the impact of the two alternative normalization bases, we show experimental results of some hardware metrics on Intel Xeon processors. We examine 22 hardware metrics that we are able to configure for counter observation (listed in Table 1). Among these 22 metrics, three represent instruction execution progress: INSTRCTN_RTD, UOPS_RETIRE, and UOPQ_W. We choose UOPQ_W — the number of μ -instructions — as the base for progress-based normalization since it causes least conflict in

Hardware metric	Description
NONHALT_TICKS	Num. of ticks that CPU is in non-halt state
INSTRCTN_RTD	Num. of retired instructions
UOPS_RETIRE	Num. of retired μ ops
L1_MISS_RTD	Num. of L1 cache misses due to retired accesses
L2_MISS_RTD	Num. of L2 cache misses due to retired accesses
L2_MISS	Num. of L2 cache misses
L2_REFERENCE	Num. of L2 cache references
DTLB_MISS_RTD	Num. of data TLB misses due to retired accesses
PGWKMISSTLTLB	Num. of page walks that page miss handler performs due to data TLB misses
DELIVER_MODE	Num. of cycles in trace cache deliver/build modes
TRACECACHE_MS	Num. of trace cache lookup misses
PGWKMISSTLTLB	Num. of page walks that page miss handler performs due to instruction TLB misses
FSB_DATAREADY	Num. of Data Ready and Data Busy events that occur on the front side bus
BUSACCES_CHIP	Num. of transactions on the bus issued by chip
X87_FP_UOP	Num. of X87 float point μ ops
MEM_CANCEL	Num. of canceled requests in the Data Cache Address Control Unit
UOPQ_W	Num. of valid μ ops written to the μ op queue
RES_STALL	Num. of stalls in the Allocator
MISPRED_BRANCH	Num. of mis-predicted branches
RTD_MISPRED_BRANCH	Num. of retired mis-predicted branches
BRANCH	Num. of branches
FRONT_END_EVENT	Num. of load/store μ ops

Table 1. 22 counter-observable hardware metrics on the Intel Xeon processors.

group counter setups. Figure 3 illustrates the comparison between time-normalized metrics and progress-normalized metrics for the TPC-H workload.

Results in Figure 3 suggest that progress-based normalization exhibits stronger or similar metric-request-correlation for most hardware metrics with one clear exception — DELIVER_MODE. This result is also consistent for several other applications we experimented with. A closer look uncovers that the DELIVER_MODE metric is not really an event count but rather it indicates the time duration for a particular processor state (the number of execution cycles during which the trace cache is in deliver mode). Unlike event counts, it is affected more by the length of execution rather than by execution progress.

In summary, our finding on the normalization base is that “event count”-style metrics should be normalized with the request execution progress while “time duration”-style metrics (along with instruction count metrics) should be normalized with the elapsed time. For the remainder of this paper, we use per-cycle values for INSTRCTN_RTD, UOPS_RETIRE, UOPQ_W, and DELIVER_MODE. We use per-instruction values for other hardware metrics.

3.2 Impact of Environmental Dynamics

In a server environment, the hardware execution behavior (reflected through counter metrics) of a request may vary as a result of dynamic environmental effects. In particular, the presence of other requests results in potentially frequent context switches and consequently processor cache behavior may vary. Further, concurrent request execution on hardware resource-sharing processors (multi-core or hardware multi-threading) yields unstable behavior due to resource contention and conflicts. Unstable metrics in dynamic execution environments introduce noise into the request signature.

To assess the impact of concurrent request execution and resource-sharing hardware, we experimentally examine the metric-request-correlation in three different execution environments: 1) requests run one-by-one with no concurrency in the server; 2)

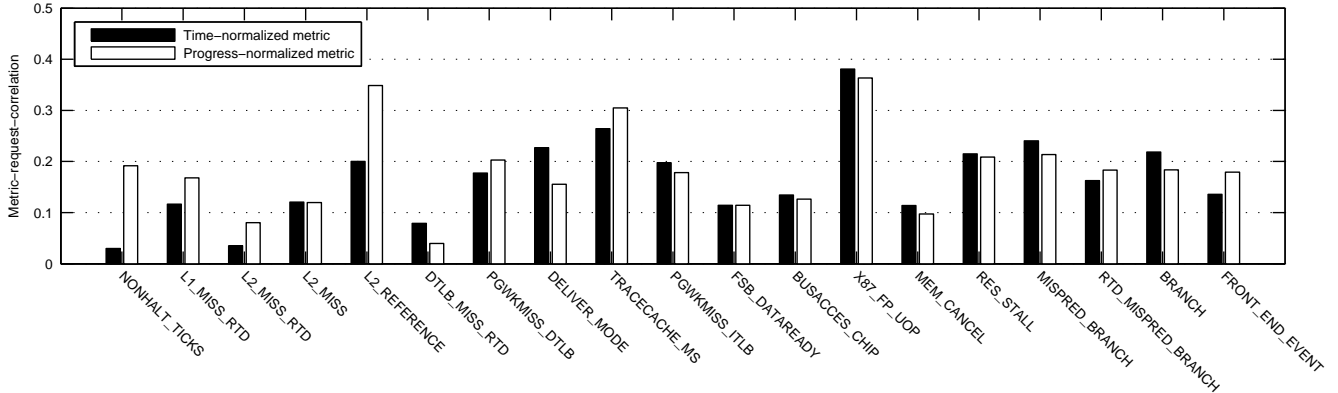


Figure 3. Comparison of time-normalized (per-cycle) metrics and progress-normalized (per-instruction) metrics for the TPC-H workload. We do not show results for the three instruction count metrics (INSTRCTN_RTD, UOPS_RETIRED, and UOPQ_W) since their choice of normalization base is obvious — per-instruction normalization of instruction counts would yield total information loss.

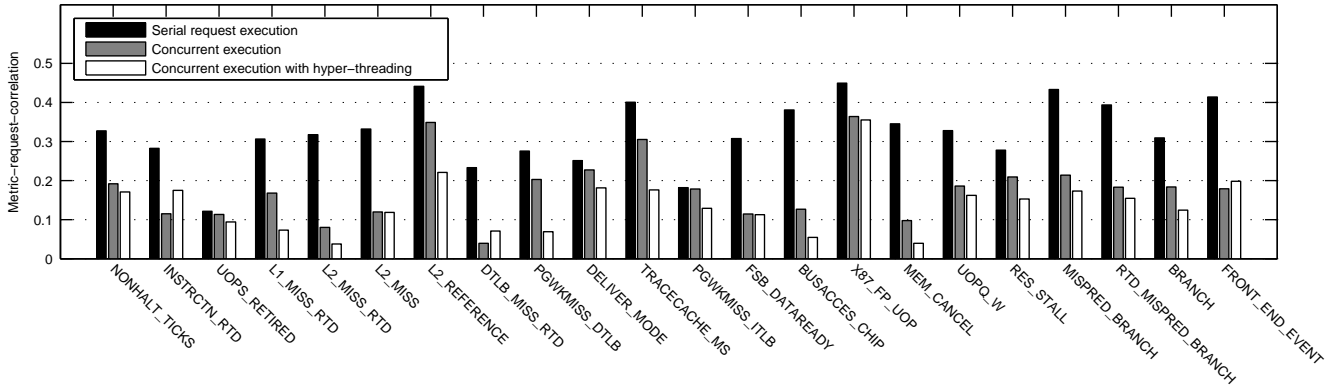


Figure 4. Impact of environmental dynamics (including concurrent request execution and hardware resource-sharing in hyper-threading) for the TPC-H workload.

requests run concurrently on a two-processor SMP machine; 3) requests run concurrently on a two-processor SMP machine where each processor supports two hardware threads (Intel hyper-threading). The three environments provide increasing levels of dynamic effects on hardware counter metrics. Figure 4 shows the metric-request-correlation for the TPC-H workload.

Comparing serial and concurrent request executions, our results show that the execution concurrency degrades the metric-request-correlation for almost all metrics. Among the most significantly affected are metrics related to memory or L2 caching behavior — L2_MISS_RTD, L2_MISS, FSB_DATAREADY, BUSACCES_CHIP, MEM_CANCEL, and FRONT_END_EVENT. This is intuitive since L2 cache misses and memory accesses are heavily influenced by frequent request context switches. L1 cache misses are not as affected due to the L1’s fast warmup time. Note that the L2 cache reference count reflects received workload at the L2 cache, which is related to the L1 caching behavior.

Figure 4 also shows that the processor-level hyper-threading significantly degrades the metric-request-correlation for L1 cache related events (L1_MISS_RTD, L2_REFERENCE), trace cache event (TRACECACHE_MS), and TLB event (PGWKMISS_DTLB). This is also intuitive since hyper-threads share these functional

units within a processor and contention for these units causes instability of the related hardware metrics.

In summary, our finding on environmental dynamics is that concurrent request execution in server environments substantially degrades the effectiveness of memory and L2 cache miss related hardware metrics as request signatures. Processor-level hardware resource-sharing can cause further degradation for metrics related to shared resources. Finally, it is important to note that these trends may not warrant absolute metric exclusion. Some affected metrics may still exhibit strong correlation for request identification in dynamic environments — one particular example is L2_REFERENCE.

3.3 Impact of Application Variations

We are also interested in whether the effectiveness of a metric as a request signature is consistent across different server applications. Specifically, we examine four applications: TPC-H, TPC-C, J2EE-based RUBiS, and index search (details about these applications can be found in Section 5). We find large differences in metric effectiveness across applications. For instance, metric X87_FP_UOP appears to be the best request signature for TPC-H but it is almost useless for others (likely because other applications perform very few floating point operations).

Application	Selected hardware metrics to form request signature
TPC-H	L2_REFERENCE, TRACECACHE_MS, X87_FP_UOP, RES_STALL, MISPRED_BRANCH, FRONT_END_EVENT
TPC-C	UOPS_RETIRED, L2_REFERENCE, TRACECACHE_MS, MISPRED_BRANCH, FRONT_END_EVENT
RUBiS	UOPS_RETIRED, L2_REFERENCE, PGWKMISS_DTLB, MISPRED_BRANCH, FRONT_END_EVENT
Retriever	L2_REFERENCE, DTLB_MISS_RTD, PGWKMISS_ITLB, RTD_MISPRED_BRANCH, BRANCH

Table 2. Request signature composition for four server applications. In addition, UOPQ_W is always selected as the base for calculating progress-normalized metrics.

The lack of consistent metric effectiveness across applications makes it unlikely that one can construct a universally effective set of hardware metrics as a request signature. Instead, calibration would be beneficial in order to arrive at an appropriate request signature setup according to application-specific metric-to-request correlations. The final selection must also consider physical constraints for metric setup on the target processors. Further, some metrics are inherently redundant and selecting one representative from each redundant group is sufficient. A simple correlation analysis among metric pairs uncovers the following redundant groups: (L1_MISS_RTD, L2_REFERENCE) and (MISPRED_BRANCH, RTD_MISPRED_BRANCH). Table 2 lists the hardware counter metrics (on Intel Xeon processors) selected as request signatures for the four server applications.

4. Operating System Mechanisms

We collect per-request hardware counter metrics and synthesize them on-the-fly. Constructed request signatures are then used to identify requests or to infer desired high-level request properties. This section presents the OS mechanisms necessary for transparent management of hardware counter driven on-the-fly request signatures.

4.1 On-the-Fly Request Context Binding

We attribute collected hardware counter metrics to corresponding server requests by maintaining on-the-fly request context binding in the system. We use the *request context* to encapsulate runtime activities belonging to a single request execution. The maintenance of on-the-fly request context binding can also support request-granularity OS adaptation, where a customized set of OS policies and configurations are used for each request.

A request context mostly coincides with a thread/process context in many cases (aside from the proper attribution of kernel activities such as interrupt handlers). Therefore, the system can bind a thread or process to the context of the request it executes. The currently active request context is the one that the active thread/process is bound to. However, the request context binding must be propagated when a request execution flows through multiple threads/processes. For instance, a request may include activities in an application server process and a database thread. The application server itself may also contain multiple components (*e.g.*, Enterprise Java Beans in J2EE services) that a request traverses through.

The issue of request context binding in multi-component servers was addressed in several previous studies. In resource containers [6] and Pinpoint [12], applications or the component middleware must explicitly pass request context bindings across multiple threads/processes in the system. In Magpie [8], system events are logged regardless of their request contexts and they are attributed to specific requests after request completion (online or offline) according to application-specific schema. Though Magpie provides a high level of flexibility in request modeling, it does not support on-the-fly request context binding.

We propose an OS-level approach to transparently track each request across multiple server components. The high-level guiding principle for our transparent request tracking is that component activities reachable through control or data flows are semantically connected, and therefore are very likely parts of one request context. Specifically, we consider two such control/data flows: process/thread forking and message passing. In the first case, we let the newly forked process or thread inherit the context binding of its parent. This is easy to implement in the OS and it has already been supported in past work [6]. For message passing, we intend to realize the following simple propagation rule when thread/process \mathcal{S} sends a message to thread/process \mathcal{R} : “If \mathcal{S} ’s request context binding is \mathcal{C} at the message send time, then \mathcal{R} inherits \mathcal{C} at the message receipt time.” Below we describe how to implement this rule for TCP/IP socket messages using a transparent message tagging mechanism.

We tag each socket message header with the identifier of the request context for the sending thread/process \mathcal{S} . To maintain compatibility with the Internet protocol standard, we store the tag in a new option field of the TCP message header. In this way, communicating peers that do not understand the tagging will simply ignore it but still receive the message properly. At the receiving side, we would like to bind the receiving thread/process \mathcal{R} to the tagged request context. Since \mathcal{R} might not have initiated the receive operation when the message arrives, we record the association of propagated request context with the buffered message at the socket. \mathcal{R} will inherit the context when initiating the receive operation. When component interactions employ connection pooling, a single socket connection may be used for propagating multiple request contexts. Therefore, the socket to request context binding may change dynamically at runtime. Figure 5 illustrates request context propagations for a single request in a J2EE-based service.

There is no explicit context unbinding operation in our scheme. An inherited request context expires when the thread/process receives a new context propagation or when it exits. A thread/process in some server components may be used repeatedly to execute requests (as in thread pooling and event-driven servers). Our scheme automatically handles this situation by performing a request context switch whenever a new context propagation is received in a socket message.

Our message-tagging based request context propagation is applicable for both intra-machine or cross-machine messages. For multi-machine server systems, sub-instances of a request context may exist on multiple machines and an on-the-fly aggregation of these sub-instances may incur significant overhead. A proper handling of such cross-machine aggregation falls beyond the scope of this paper and all experiments in this paper utilize single-machine servers.

Our current scheme is sufficient to support many multi-component server applications. However, we acknowledge two limitations that need to be addressed in future work. First, our scheme targets server applications in which request contexts propagate through process/thread forking or message passing. However, some applications contain context propagations over other means (*e.g.*, thread synchronization via shared memory). Second, there can be am-

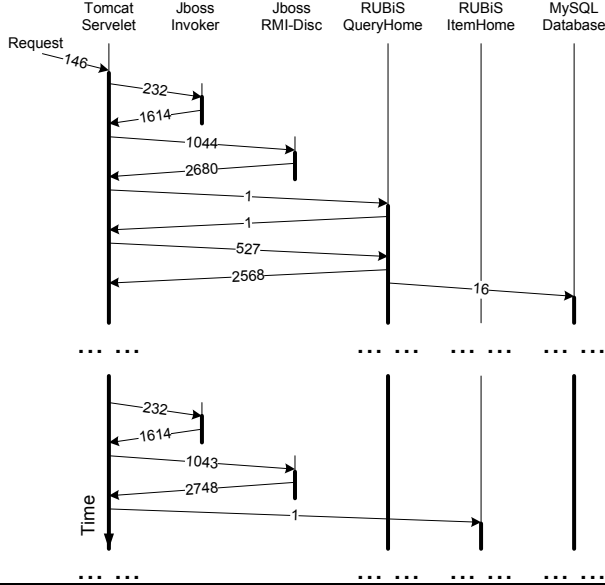


Figure 5. Illustrated request context propagations using socket communications for a RUBiS [20] request. RUBiS in this example is supported by the JBoss Application server and MySQL database. Darkened lines indicate portions of component executions attributed to the request context. The number on each message indicates the application-level message size in bytes. Our request propagation tag contains a small request context identifier and it only consumes an additional 12 bytes per message.

biguous request context propagation when a single receive operation reads data across the boundary of multiple messages that bear different context bindings. These messages are typically demultiplexed later at application level, which is beyond OS detection. Such a scenario may occur in event-driven servers.

4.2 Metric Collection and Synthesis

We maintain raw system statistics in the form of cumulative event counters per processor. To retrieve the event counts for a duration of continuous execution on a processor, we only need to sample the counter values at the start and end of the duration and then calculate the difference. When there is a request context switch on a processor, we must sample the counter values at the 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. They also occur when the request context binding of a running thread/process changes. In addition to sampling at request context switches, it may also be desirable to sample the counter values periodically to acquire fine-grained execution statistics. On our experimental platform (Linux 2.6.10), kernel timers allow us to sample at a frequency as high as once every millisecond.

The collected counter metric trace for each request consists of a series of metric samples in chronological order. Each sample contains the elapsed CPU cycles and incremental hardware metrics since the last sampling point. Periodic sampling at once every millisecond results in a 1 ms upperbound on the duration of each metric sample. However, the metric samples for each request typically do not follow synchronized steps due to non-deterministic request context switches in concurrent server environments. When calculating cumulative event counts up to a specified execution point (e.g., 3 ms since the beginning of the request), there often exists one metric sample whose duration crosses the desired execution

point. In this case, we approximate by discounting part of the event counts in this metric sample with the assumption of constant event occurrence frequency over the sample duration.

Processor-level multi-threading technologies such as Intel’s hyper-threading allow concurrent thread execution on a single processor. This feature slightly complicates hardware counter metric collection because multiple hardware threads on one physical processor share a single set of counter registers. In a typical setup, some counter registers may be exclusive to one of the hardware threads. A single request may utilize different sets of counter registers when it migrates over different hardware threads.

4.3 Request Identification and Property Inference

Our on-the-fly request identification is realized by matching request signatures (as a composite of several hardware counter metrics) against those in a bank of representative requests maintained by the system. The matching request in the bank can simply be the one with the most similar signature to the on-the-fly request. To infer high-level request properties (e.g., CPU and I/O resource usage), the request bank also contains a signature-to-property mapping, which returns the property of the matching request. The bank of representative requests can be constructed either offline or through online self-learning. In online self-learning, signatures and additional properties of interest for just-completed requests are added to the request bank as replacements of older requests. This is particularly useful for automatically adapting to gradual system changes in long-running servers.

Our approach is simplistic in that it does not need any prior knowledge of the semantic relationships between the collected hardware metrics and the desired request properties. Such knowledge would be required for parametric prediction techniques such as neural networks, hidden Markov models, and Bayesian networks.

To determine matching requests, we need a measure of distance between two request signatures, each of which is represented as a vector of hardware metrics $(\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_k)$. We have tried three distance measures: $L1$ distance, $L2$ distance, and cosine distance (the cosine of the angle between two vectors). We find that different distance measures yield very small deviations in identification accuracy. This is because similar requests tend to exhibit small distance under any reasonable distance measure. For simplicity, we currently employ the normalized $L1$ distance. Specifically, the distance between two request signature vectors $(\mathcal{M}_1[x], \mathcal{M}_2[x], \dots, \mathcal{M}_k[x])$ and $(\mathcal{M}_1[y], \mathcal{M}_2[y], \dots, \mathcal{M}_k[y])$ is:

$$\sum_{i=1}^k \frac{|\mathcal{M}_i[x] - \mathcal{M}_i[y]|}{\bar{\mathcal{M}}_i} \quad (2)$$

where the weight $\bar{\mathcal{M}}_i$ is the expected mean of metric \mathcal{M}_i for all requests.

For each request, the amount of time spent collecting metrics before querying the request bank for request identification must be carefully assessed. Too short a metric collection phase may provide insufficient information for identifying a request. On the other hand, too long a metric collection phase requires large collection overhead and most importantly a late-stage inference may not allow effective system adaptation for this request. Although it is simple to make request identifications at a deterministic point of a request execution, different requests in a server application may require different time periods for metric collection in order to achieve accurate identification. We therefore consider two approaches to determining the request identification time:

1. *Fixed-point identification.* This approach attempts to identify all requests by querying the request bank for the closest match at a fixed time point (e.g., after the request runs for 3 ms). The

point is determined by offline calibration to achieve satisfactory request identification and property inference accuracy while still being sufficiently early for effective OS adaptation using the inferred request property.

2. *Confidence-driven incremental identification.* For each request, this approach queries the request bank at incremental stages (e.g., 1 ms, 2 ms, 3 ms, ... after the request begins) using up-to-date cumulative metrics. At each stage, it assesses a confidence metric in the current request identification result and stops further identification if a high-enough confidence has been attained. In our current design, a high confidence is indicated by the agreement of the identification results made during the most recent stages. For example, we can finalize a request identification if the property of the closest matching request (in the request bank) at the most recent stage is within a δ difference from that at the second most recent stage.

5. Implementation and Evaluation

We have developed a prototype implementation of the proposed request signature management in the Linux 2.6.10 kernel. Our implementation supports the online self-learning based request bank construction described in Section 4.3. Hardware counters are typically accessed through privileged instructions (for concerns such as information leaking [29]), so counter value sampling must be performed in the OS kernel. To avoid domain crossing overhead, all our hardware counter management is performed in the kernel. It is also possible to employ a microkernel-style implementation, which would retain only the basic counter metric collection and request context maintenance in the kernel while leaving the signature construction, request identification, and property inference at the user level. This alternative architecture would allow easier policy changes at the cost of additional domain crossing overheads.

Using our prototype, this section evaluates the overhead and request property inference accuracy of our system. We also performed several system adaptation case studies using our request signatures, which we present in the next sections. The machines in our experimental platform each contain dual 2 GHz Intel Xeon processors and 2 GB memory. We configure the hardware counter registers to report the application-specific hardware metrics listed in Table 2. Each experiment in our application studies involves a server and a load generation client. The client generates input workloads according to traces or synthetic setups in application/benchmark specifications.

Our evaluation employs four server applications:

- *TPC-C* [26] 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 30%, 20%, 20%, 10%, and 20% of all requests, respectively. TPC-C runs on the MySQL 5.0.18 database.
- *TPC-H* [27] is a database-driven decision support benchmark. The TPC-H 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. TPC-H runs on the MySQL 5.0.18 database.
- *RUBiS* [20] 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,

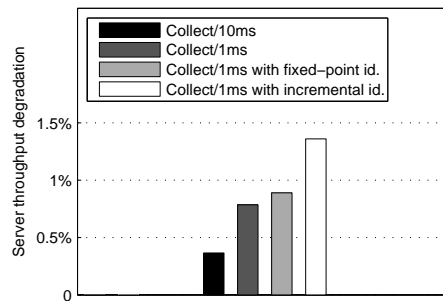


Figure 6. System overhead for TPC-H at two different metric collection frequencies (once per 10 ms and once per 1 ms) as well as two different request identification approaches (fixed-point and incremental).

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. In our application setup, most of the server CPU consumption (around 84%) is in the JBoss J2EE application server and its hosted application components.

- *Index search:* The above workloads are all fully CPU-bound. To enhance the workload variety, we include one data-intensive server application that provides full-text search on a web keyword index dataset. The dataset, acquired from the Ask.com search engine [4], contains 1.58 million indexed web pages. Its size — 2.37 GB — is slightly larger than the server memory size. The search queries in our test workload are based on a real trace recorded at Ask.com in summer 2004.

5.1 System Overhead

We assess the overhead of our per-request system metric collection and on-the-fly request identification. We consider two different metric collection frequencies (once every 10 ms and once every 1 ms). We also consider two request identification approaches (fixed-point and incremental). For the incremental request identification, we assume all requests require 10 stages to complete. This is an over-estimation to provide us an overhead upper bound.

Figure 6 illustrates the overhead assessment result for TPC-H. We find that the hardware metric collection incurs 0.4% and 0.8% overhead at the frequencies of once per 10 ms and once per 1 ms respectively. Note that the overhead does not scale linearly with the collection frequency because the per-collection cache warmup cost is less when the collection routine runs more often. The two request identification approaches yield additional 0.1% and 0.6% throughput reductions respectively. We believe this overhead is sufficiently low for runtime deployment.

5.2 Request Property Inference Accuracy

We evaluate the effectiveness of hardware counter metric driven on-the-fly request property inference. Although our framework can support many inference targets, our evaluation here focuses on predicting request resource consumption, which is particularly useful for some online system adaptations (as illustrated later in Section 6.1). For CPU-bound applications (TPC-C, TPC-H, and RUBiS), our prediction target is the request CPU usage. For data-intensive index search, our prediction target is the request I/O size.

As a comparison basis to our hardware counter driven request property inference, we look for a representative conventional approach that is also transparent to server applications (*i.e.*, requir-

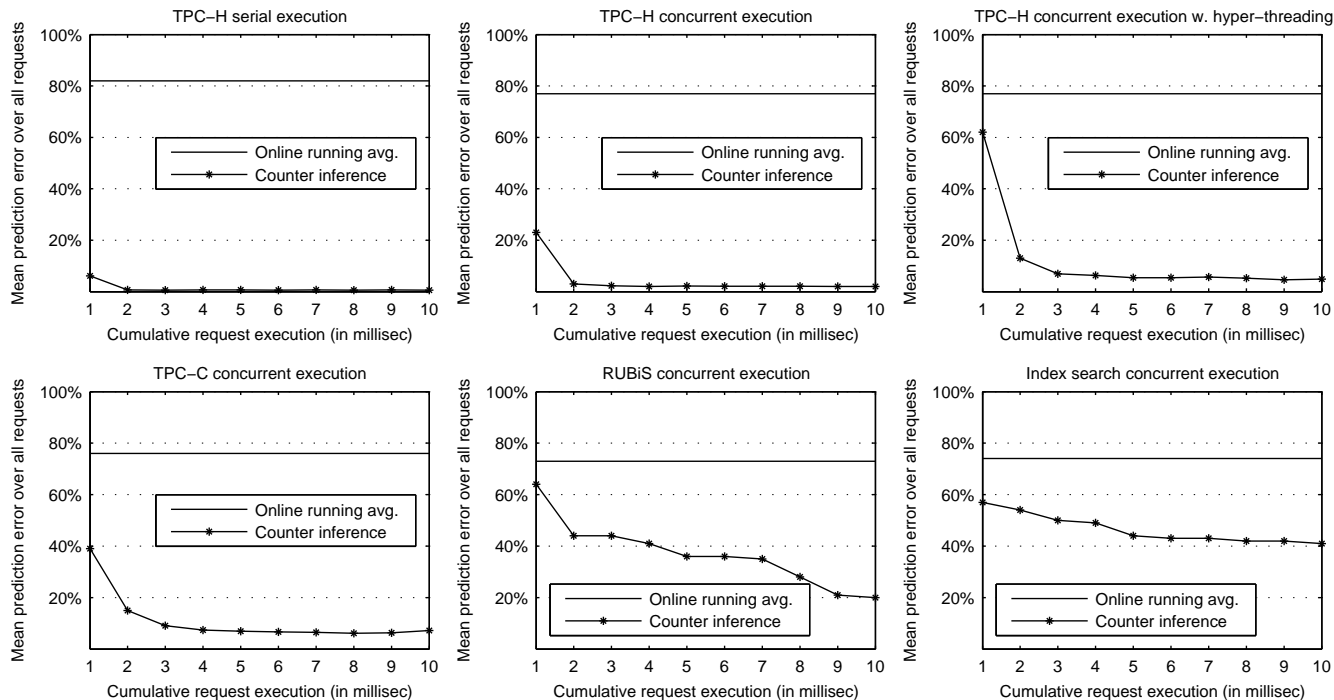


Figure 7. The accuracy of predicting request CPU usage (I/O size for the data-intensive index search) using our hardware counter driven inference and using an online running average. For our hardware counter driven method, we show the prediction accuracy using up to 10 ms of execution statistics for each request. The mean full request execution time (mean CPU time for each request to complete) is 600.8 ms, 25.9 ms, 29.3 ms, and 16.3 ms for TPC-H, TPC-C, RUBiS, and index search respectively. The prediction error for a particular request is defined as $\frac{\text{prediction} - \text{actual}}{\text{actual}}$. Note that this definition of error may exceed 100% and we count it as 100% in such cases.

ing no application instrumentation or assistance). Fundamentally, without on-the-fly information about an incoming request, there is little other choice but to use recent past workloads as the basis to predict incoming workloads [11, 22, 9]. Specifically, we employ a transparent workload property prediction method — online running average — as our comparison basis. In this method, the property of the next runtime request is estimated as the average of N recent past requests. We find that the prediction accuracy is not very sensitive to the parameter N and our reported results were produced using $N = 10$.

Figure 7 illustrates the inference accuracy for our four server applications (we also show the serial execution and hyper-threading-enabled results for TPC-H). With 10 ms execution statistics for each request, the prediction errors for TPC-C, TPC-H, RUBiS, and index search are 7%, 3%, 20%, and 41% respectively. They are all substantially lower than the online running average-based prediction (73–82% errors).

Comparing across the four applications, the prediction accuracy of TPC-C and TPC-H is much better than that of RUBiS and index search. Further, TPC-C and TPC-H requests can reach high prediction accuracy with no more than 3 ms request execution statistics. Both RUBiS and index search require more statistics. Our application studies suggest the following explanation. For TPC-C and TPC-H, different requests exhibit clearly differentiated execution behaviors early in their executions. In contrast, all index search requests follow similar code paths, which makes them very difficult to differentiate. Finally, RUBiS requests start with almost identical code paths due to common processing for Enterprise Java Beans, but they deviate later with processing behaviors unique to the respective request functions.

5.3 Request Identification Timing

Determining the time at which request identification is performed is critical since the need to achieve reasonable prediction accuracy must be balanced by the need to ensure that the identification is early enough to guide request-granularity system adaptation. In Section 4.3, we described two approaches to determining this time: fixed-point request identification and confidence-driven incremental identification. Results in Figure 7 can directly guide the choice of the fixed inference time point (e.g., 2 ms for TPC-H requests and 9 ms for RUBiS requests).

In the confidence-driven incremental approach, request identifications are performed incrementally (e.g., every millisecond) and only those with high confidence are finalized at each stage. We evaluate this approach using our simple confidence measure described in Section 4.3. Here, we focus on RUBiS and index search since their request identification accuracy is more dependent on the identification timing. Figure 8 shows that by making request property predictions only when the confidence is high, higher prediction accuracy can be achieved for the requests identified (compared to the prediction error when using a fixed cumulative window, as specified on the X-axis, for all requests). However, this comes at the cost of incomplete request identifications, as shown in Figure 9.

6. Operating System Adaptations

Our hardware counter driven request signature supports on-the-fly request identification and inference of high-level request properties. This makes it possible to adapt system management on a per-request basis using the request identification or inferred properties. This section explores several such adaptations: resource-aware re-

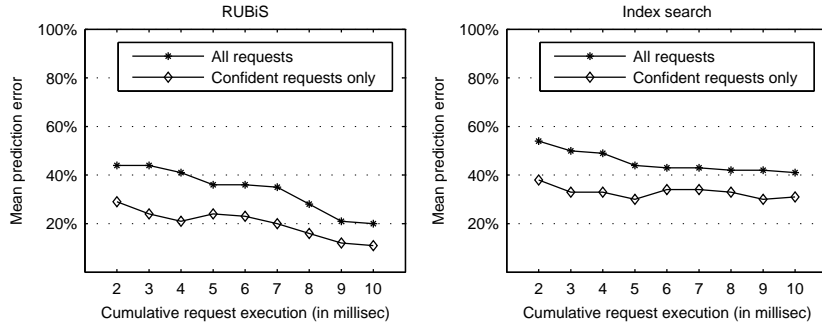


Figure 8. Request property inference accuracy when identification is made only for requests demonstrating a high confidence.

quest scheduling, on-the-fly request classification, and anomaly detection.

6.1 Resource-Aware Request Scheduling

It is well known that user request rates for server systems can fluctuate dramatically over time. Consequently, it is important to manage requests efficiently under high load conditions. Knowledge of request resource usage at scheduling time is essential or at least helpful in realizing several request management schemes:

- *Shortest-remaining-processing-time (SRPT) scheduling* is known to achieve minimal average request response time [21]. Bansal and Harchol-Balter [7] further showed that concern over its unfairness to long-running tasks is unwarranted.
- *Deadline-driven scheduling*: Interactive users often desire service responses within a certain time limit. Deadline-driven scheduling can benefit from advance knowledge of whether a request can be completed before its deadline.
- *Resource-aware admission control*: Scenarios in which a relatively few resource-hungry requests in a server consume a disproportionately large amount of resources are not uncommon. An overloaded system may want to drop these resource-hungry requests in order to achieve higher request throughput.

Despite these benefits, acquiring request resource usage information before a request completes is challenging. Although application-level information can infer request resource consumption for applications with simple semantics (*e.g.*, inferred through file size in a static-content web server [19]), such inference is difficult for applications with more complex semantics. Further, the involvement of application information compromises system transparency. As a different design point, Capriccio [28] acquires application resource consumption information through extensive compiler and language-level runtime support. However, such methods are not applicable to many existing applications or new applications written in unsupported programming languages. In contrast, our OS-level inference of request resource usage provides a high level of transparency that requires no application assistance or change.

An Empirical Evaluation of SRPT Scheduling We measure the effectiveness of our hardware counter driven request resource usage inference in supporting the SRPT request scheduling. To facilitate this study, we implemented a simple SRPT scheduling scheme in the Linux kernel. Our implementation mainly involves an augmentation each time the Linux CPU scheduler is about to pick a task from the head of the ready task queue. Specifically, at this time, we search for the ready task whose request binding has the shortest re-

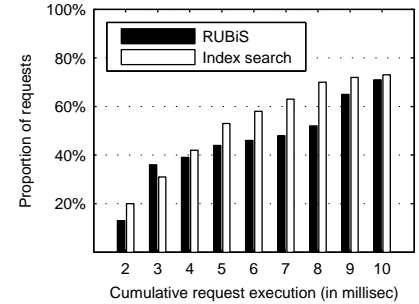


Figure 9. The cumulative proportion of request identifications made under high confidence.

maining processing time, and move this task to the head of the task queue so that it will be chosen at the subsequent scheduling point. Note that our simple implementation is not perfect SRPT since we do not interfere with the Linux task quantum management, which may force round-robin scheduling when the task quanta are used up.

Our on-the-fly request resource usage inference can enable SRPT request scheduling. Note that a running request does not have inferred resource usage during its metric collection phase before our signature-based identification is made. Our scheduler always gives such unidentified requests higher priority over those that are already identified. We compare the performance of our request scheduling scheme against three alternative approaches: 1) *Default Linux* scheduling; 2) SRPT scheduling using *online running average*-based request resource usage estimation; and 3) a hypothetical *oracle* SRPT scheduling that has perfect knowledge of request resource usage before execution.

Figure 10 illustrates the mean request response time under different request scheduling schemes when the workload request rates approach server saturation load levels. Since only CPU-bound applications are affected by CPU scheduling, here we only show results for RUBiS, TPC-C, and TPC-H. The results demonstrate that our counter-driven scheduling yields 15–27% less request response time compared to the online running average based scheduling. Its improvement over default Linux is greater (up to 70% response time reduction). More importantly, the performance of our approach is within 5% that of the oracle scheduler, indicating its ability to realize the full benefit of resource-aware scheduling.

6.2 On-the-Fly Request Classification and Anomaly Detection

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 performance projection on new processor/memory platforms. As far as we know, existing online request modeling techniques (and Magpie [8] in particular) classify each request into a request cluster after it completes. Our on-the-fly request signature makes it possible to classify a request shortly after it begins execution. By classifying a request early, an online workload tracker may save the overhead for further tracing and event logging on the classified request. Further, early request classification allows on-the-fly flagging of potentially anomalous requests — those that do not fall into any existing request cluster. The sys-

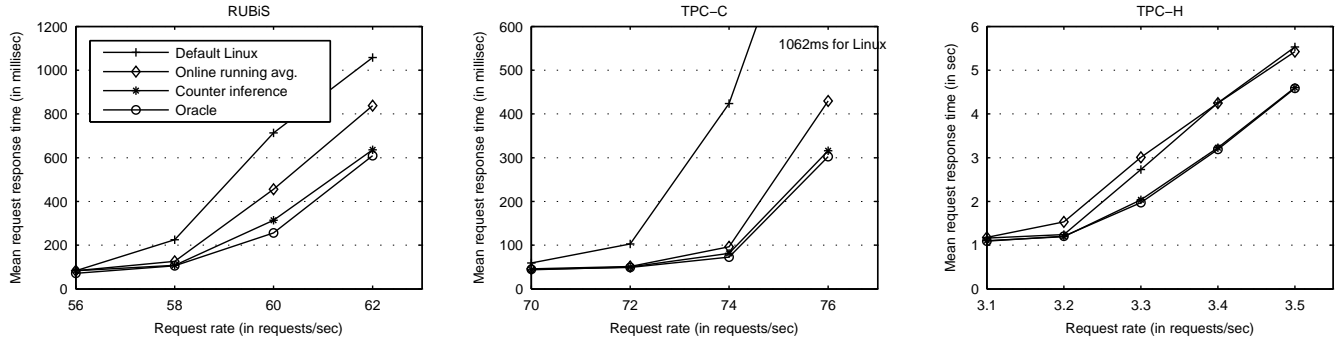


Figure 10. Performance of shortest-remaining-processing-time scheduling when workload request rates approach server saturation load levels.

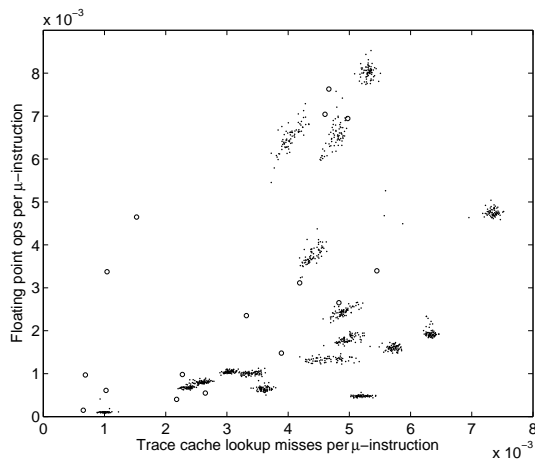


Figure 11. Per-request 10 ms-cumulative values of two hardware counter metrics (floating point operations and trace cache lookup misses per μ -instruction) for an execution of TPC-H requests. The plot includes 1227 normal TPC-H requests (marked as dots) and 16 anomalous requests (marked as circles).

tem may then apply targeted monitoring or even online quarantine on those requests.

We provide a simple illustration to motivate the use of hardware metrics for request classification and anomaly detection. Figure 11 shows the per-request 10 ms-cumulative values of two hardware counter metrics for an execution of over 1000 TPC-H requests. This execution includes some anomalous requests generated synthetically following the pattern of SQL-injection attacks, which may enable the attackers to bypass authentication and extract sensitive data [3, 18]. In particular, a tautology-based attack is to inject code in one or more SQL conditional statements so that they always evaluate to true (or they are effectively removed from conjunctive conditions). For each of the 17 TPC-H query types (except Q13, which does not contain a conditional *where* clause), we generated one anomalous request by removing one or two conditional statements at the end of the *where* clause. Results in Figure 11 indicate that normal requests do tend to form clusters according to early-execution hardware metrics and anomalous requests are typically distant from these clusters. Note that this plot only considers two hardware metrics due to illustrative limitation while our full request signature includes more metrics.

An Empirical Study of Request Classification Using a simple empirical study, we assess the potential of our request signature enabled on-the-fly request classification. To facilitate this study, we implemented a simple request classifier. First, we define the *cluster signature* as the centroid of all signatures (hardware metric vectors) of requests in the cluster. For each new request, we match its on-the-fly signature with the cluster signatures of all existing request clusters. If the closest match exhibits a small enough signature difference, the request is then classified into the corresponding request cluster. Otherwise, it initiates a new cluster.

Figure 12 illustrates the accuracy of our on-the-fly request classification for TPC-H (without anomalous requests). We show results using different windows for per-request metric collection (up to 100 ms since request begins execution). We define *perfect classification* as the one that groups requests according to similar CPU usage. We then define the error of an on-the-fly classification as its deviation from the perfect classification. More specifically, a *misclassified* request is one that is put into a cluster of requests with predominantly different CPU usage. An *over-classified* request is not mis-classified but it (along with other similar requests in the same cluster) should be merged into another (larger) cluster. Misclassification is much more worrisome since it would yield erroneous information in the produced workload model. In comparison, over-classification only incurs some additional cluster management overhead.

Results in Figure 12 show that our on-the-fly request classification can achieve zero mis-classification with as low as a 2 ms window of cumulative request execution statistics. The over-classification rate is less than 25% with a 4 ms or higher window of cumulative statistics. Since the mean TPC-H request execution time is around 600 ms, on-the-fly request classification with 4 ms cumulative statistics can potentially save up to 99% of request monitoring and tracing overhead.

An Empirical Study of Anomaly Detection We perform another simple empirical study to assess the potential of our request signature enabled on-the-fly anomaly detection. Building on our request classifier, we consider requests in unusually small clusters as anomalous requests. We use two measures to evaluate the effectiveness of our anomaly detection:

$$\text{anomaly recall} = \frac{\# \text{ of correctly detected anomalies}}{\# \text{ of actual anomalies}}$$

$$\text{normal request recall} = \frac{\# \text{ of correctly identified normal requests}}{\# \text{ of actual normal requests}}$$

A high anomaly recall indicates that most anomalies are properly detected while a high normal request recall indicates that most normal requests are not misclassified as anomalies. Note that the

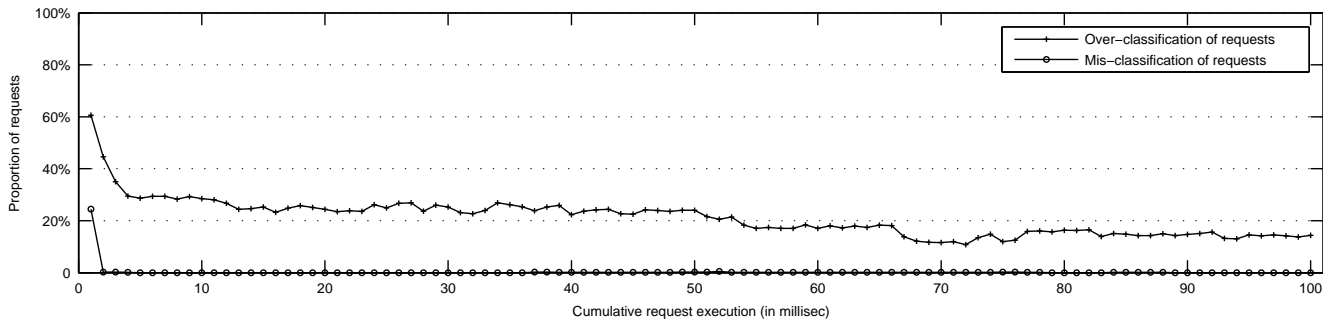


Figure 12. Classification accuracy of hardware counter driven on-the-fly request classification for TPC-H (without anomalous requests). We show results using different windows for per-request metric collection (up to 100 ms since request begins execution).

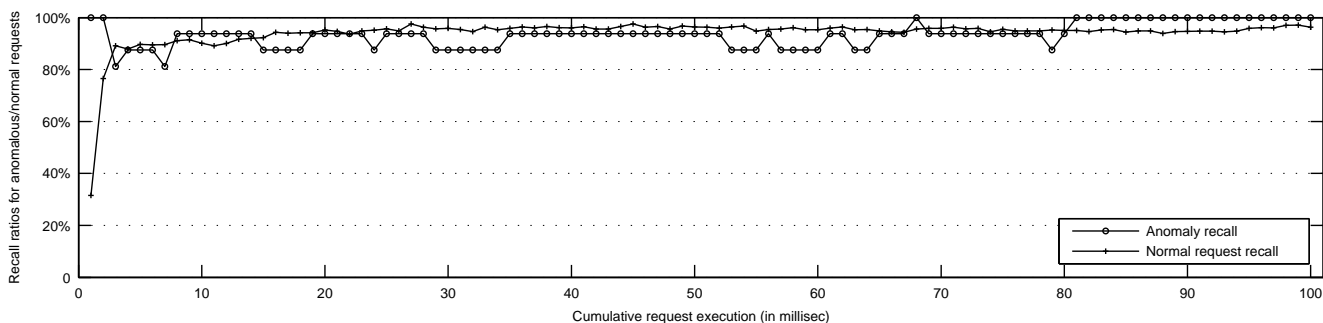


Figure 13. Detection accuracy of hardware counter driven on-the-fly anomaly detection for TPC-H (including 1227 normal requests and 16 SQL-injection-style anomalous requests).

second measure is necessary since a trivial detector that flags all requests as anomalous would achieve the perfect anomaly recall.

Figure 13 shows the accuracy of our on-the-fly anomaly detection for TPC-H (with some SQL-injection-style anomalous requests). For this experiment, small clusters containing less than 1% of all requests are considered as anomalous. Results suggest that high recall ratios (around 90% or higher) for both anomalies and normal requests can be achieved with only a 4 ms or higher window of cumulative statistics for each request.

7. Conclusion

This paper makes the case for constructing on-the-fly request signatures using hardware counter metrics available on modern processors. The signature enables on-the-fly request identification and inference of high-level request properties, which subsequently allow request-granularity system adaptations that are otherwise impossible (or difficult). We address two key challenges in realizing such on-the-fly request signatures: deriving general principles to guide the selection of hardware counter metrics, and proposing OS mechanisms for transparent online management of per-request counter metrics. Our experiments using four server applications demonstrate the high accuracy of our on-the-fly request identification and request resource usage inference. We also illustrate the effectiveness of request signature-enabled OS adaptations including resource-aware request scheduling, on-the-fly request classification, and anomaly detection.

Our experience allows us to make several important conclusions that can guide the practical deployment of our technique. First, a number of factors may influence the effectiveness of hardware met-

rics as request signatures in a concurrent server environment. In particular, metrics that are most susceptible to concurrency-related environmental perturbations tend to be less effective. Second, dependence on application characteristics makes it unlikely to find a small but universally useful set of hardware metrics as a request signature. Instead, application-specific calibration is needed to derive the appropriate request signature composition for each server application. Third, quick and accurate request identification is more likely for those server applications whose requests possess a variety of different semantics and tend to exhibit differentiating patterns of execution early. Identification is more difficult for applications whose requests only bifurcate in behavior later in their execution path (such as RUBiS in our study) but they may still benefit from our technique.

Today’s hardware counter interfaces are non-standard across processor versions and are not “architected” for general software utilization. While our experiments in this paper employ a single x86 processor platform, almost all of the metrics we collect (*e.g.*, floating point operations, L1 misses, number of loads, stores, and branches, CPI) are available on every processor platform we are familiar with (including Power and other x86 versions). More importantly, along with several other recent studies [25, 10, 29], we demonstrate the potential importance of their use in improving software system performance and dependability. This may help influence the standardization of processor hardware metrics and their broad exploitation in computer systems.

References

- [1] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, October 2003.
- [2] 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.
- [3] C. Anley. Advanced SQL Injection in SQL Server Applications. Technical report, Next Generation Security Software Ltd., 2002.
- [4] Ask.com Search Engine (formerly Ask Jeeves). <http://www.ask.com>.
- [5] R. Azimi, M. Stumm, and R. W. Wisniewski. Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters. In *Proc. of the 19th ACM Conf. on Supercomputing*, pages 101–110, Cambridge, MA, June 2005.
- [6] G. Banga, P. Druschel, and J.C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of the Third USENIX Symp. on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.
- [7] N. Bansal and M. Harchol-Balter. Analysis of SRPT Scheduling: Investigating Unfairness. In *Proc. of the ACM SIGMETRICS*, pages 279–290, Cambridge, MA, June 2001.
- [8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modeling. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.
- [9] J.M. Blanquer, A. Batchelli, K. Schausser, and R. Wolski. Quorum: Flexible Quality of Service for Internet Services. In *Proc. of the Second USENIX Symp. on Networked Systems Design and Implementation*, pages 159–174, Boston, MA, May 2005.
- [10] J.B. Bulpin and I.A. Pratt. Hyper-Threading Aware Process Scheduling Heuristics. In *Proc. of the USENIX Annual Technical Conf.*, pages 103–106, Anaheim, CA, April 2005.
- [11] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, and R.P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 103–116, Banff, Canada, October 2001.
- [12] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proc. of the First USENIX Symp. on Networked Systems Design and Implementation*, pages 309–322, San Francisco, CA, March 2004.
- [13] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, pages 105–118, Brighton, United Kingdom, October 2005.
- [14] A.S. Dhodapkar and J.E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proc. of the 29th Int'l Symp. on Computer Arch.*, pages 233–244, Anchorage, AL, May 2002.
- [15] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *Proc. of the Int'l Conf. on Parallel Arch. and Compilation Tech.*, pages 220–231, New Orleans, LA, September 2003.
- [16] L. Eeckhout, H. Vandierendonck, and K.D. Bosschere. Workload Design: Selecting Representative Program-Input Pairs. In *Proc. of Int'l Conf. on Parallel Arch. and Compilation Tech.*, pages 83–94, Charlottesville, VA, September 2002.
- [17] C. Gniady, A.R. Butt, and Y.C. Hu. Program-Counter-Based Pattern Classification in Buffer Caching. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, pages 395–408, San Francisco, CA, December 2004.
- [18] W.G.J. Halfond, J. Viegas, and A. Orso. A Classification of SQL Injection Attacks and Countermeasures. In *Int'l Symp. on Secure Software Engineering*, Arlington, VA, March 2006.
- [19] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-Based Scheduling to Improve Web Performance. *ACM Trans. on Computer Systems*, 21(2):207–233, May 2003.
- [20] RUBiS: Rice University Bidding System. <http://rubis.objectweb.org>.
- [21] L.E. Schrage and L.W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research*, 14(4):670–684, 1966.
- [22] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation*, pages 225–238, Boston, MA, December 2002.
- [23] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proc. of the 30th Int'l Symp. on Computer Arch.*, pages 336–349, San Diego, CA, June 2003.
- [24] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proc. of the Second USENIX Symp. on Networked Systems Design and Implementation*, pages 71–84, Boston, MA, May 2005.
- [25] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using Hardware Performance Monitors to Understand the Behaviors of Java Applications. In *Proc. of the Third USENIX Virtual Machine Research and Technology Symp.*, pages 57–72, San Jose, CA, May 2004.
- [26] TPC Benchmark C. <http://www.tpc.org/tpcc>.
- [27] TPC Benchmark H. <http://www.tpc.org/tpch>.
- [28] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 268–281, Bolton Landing, NY, October 2003.
- [29] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor Hardware Counter Statistics As A First-Class System Resource. In *Proc. of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, May 2007.