Best-Effort Request Labeling and Scheduling on Multicore Servers

Kai Shen and Meng Zhu

The University of Rochester
Computer Science Department
Rochester, NY 14627

Technical Report TR-996
May 2016

Abstract

Multicore servers continue to scale on the CPU count and request execution concurrency. It is desirable to support request quality-of-service such as reducing the latency distribution tail and providing differential services. This paper presents new operating system support that labels CPU executions with requests or their QoS parameters (e.g., request arrival time) and enables online request scheduling. Since OS-level request tracking suffers from inaccuracies in situations like user-space synchronization and batched event processing, we devise effective request tracking/scheduling strategies that consciously tolerate such inaccuracies. In particular, we find that ignoring partially OS-visible synchronizations causes less harm for QoS scheduling than propagating request labels through them. We also augment OS event dispatcher like `epoll_wait()` to return a batch of events with similar (but likely unequal) scheduling labels to user space according to the QoS heuristics. We utilize our best-effort request labeling to schedule for tail latency reduction, request prioritization, and hyperthread-contention throttling. We evaluate our system using an RSA encryption workload, a CPU/cache/memory-stressing server, the Apache Solr search engine, a Google App Engine application, and an event-driven Redis hash store on a 40-CPU machine. Results demonstrate low overhead (no more than 1%) and high scheduling effectiveness (reducing the 99th percentile request response time by 8%, 24%, 35%, 50%, and 43% for the five workloads).

---

This work was supported in part by the National Science Foundation grants CNS-1217372, CNS-1239423, and CCF-1255729, and by a Google Research Award.
1 Introduction

A server system execution is often composed of many concurrent requests, each of which services a user call, like a request for retrieving a web object, or a search of information on a certain topic. The server’s quality-of-service is dependent not only on the overall throughput or average performance, but also on the response latency of individual requests. For instance, the tail (or the worst cases) of the request latency distribution [18] is considered a critical server quality-of-service metric beyond the average or median request latency. Furthermore, the prioritization of certain critical requests or cloud tasks may be desirable due to differential service objectives on shared hosting platforms.

The large scale of modern multicore machines and improving software scalability [11, 12] allow high concurrency of request executions. However, sibling CPUs may share a substantial amount of hardware resources in terms of L1/L2 cache space, the instruction fetch/decode unit, and instruction execution pipeline (between hyperthreads) as well as the last-level cache space and memory bandwidth (between cores). Dynamic resource contention on multicores not only leads to performance degradation (up to $7 \times$ reduction of instructions-per-cycle rate compared to running alone in one of our experimental setups), but also yields more varying, less predictable request responses. Both complicate the quality-of-service on multicore servers.

These motivate the scheduling of server resources according to individual request contexts to realize high quality-of-service. For the purpose of QoS scheduling on request response latency, we use the term request context to encapsulate execution activities that depend (through control and data dependencies) on the request input and must be performed before (or depended upon by) the final response of a request. A request context sometimes coincides with a process (or thread) context. However, a request execution may flow through multiple components of a server system (e.g., a front-end proxy, middleware components, and databases).

A large body of prior work [7,8,10,13,15,16,22,23,34] have presented techniques to track request executions in multi-tier servers, mostly for the purposes of resource accounting and performance analysis. However, online scheduling requires more precise request tracking. In particular, for resource accounting, a short execution segment with an incorrect request context binding would only lead to error that is proportional to the segment’s size. For resource scheduling, however, this may cause substantial delay to a high-priority request (or one that is going to degrade tail latency) if the request completion depends on this mis-identified (as low priority) segment.

OS-level request tracking suffers from several inherent inaccuracies. In particular, some synchronizations between request stages are preformed at the user space (particularly when kernel block-wakeup support is not needed) that are invisible to the OS kernel. Additionally, for event-driven servers, OS event dispatchers may return a batch of events (belonging to multiple request contexts) to the user space for efficiency but the context of subsequent user-space execution becomes ambiguous. This paper presents a new OS-level approach that consciously recognizes and tolerates request tracking inaccuracies in QoS scheduling. For instance, if a short segment of execution is mis-identified as part of a high-priority request, we would only have to pay the cost of executing the segment too early. In addition, mis-identifying one request context to another is not necessarily harmful as long as the mis-identified context carries a similar scheduling parameter (e.g., request arrival time) to the correct one.

We present an application-transparent, OS-level request tracking facility that minimizes the least scheduler-tolerable context mis-identifications. We propagate request contexts and scheduling labels through control and data dependency primitives including process (or thread) forking, signals, Internet and Unix domain sockets, and pipes. To support event-driven servers, we augment
OS event dispatchers to return a batch of events with similar (though likely unequal) scheduling labels to user space. On the other hand, we do not make use of partially OS visible synchronization operations (such as \texttt{futex} waits / wakeups in Linux) which could mis-lead the OS toward substantial mis-identifications of wrong request contexts. Our facility works well for thread-level request multiplexing by letting newly propagated request context binding to overwrite the older one.

We further construct request scheduling for latency distribution tail reduction and critical request prioritization that tolerate the possible mis-identification of a high-priority request as background activity (one that is not depended upon by any request completion). Specifically, it reverts to the default fair share OS scheduling (thus does no harm) at the presence of background activities. Our prioritizing CPU scheduler not only reorders the task executions according to request context priorities, but it also mitigates the hardware resource contention by throttling the sibling CPU(s) of a CPU executing a high-priority request. The latter requires non-work-conserving CPU scheduling that may lead to reduced overall server throughput. However, the throughput reduction is modest if the throttled CPU was subject to high resource contention (particularly in the case of hyperthreading contention).

2 Related Work

Request and path management: Earlier systems like Resource Containers [8] and Magpie [10] track request contexts in a server system to precisely attribute resource uses and model workload patterns. Other work including Chen et al. [15], Google’s Dapper [34], X-Ray [7], and the Mystery Machine [16] record and process request, path, or transaction traces to perform offline performance analysis and debugging. Aguilera et al. [6] employed signal processing techniques to derive causal relationships among communication events. Whodunit [13] showed that some user-level request context transfers may be observed at the OS by trapping accesses to critical synchronization data structures. The X-Trace framework [23] monitors data flows in network packets. Quanto [22] propagates activity labels across TinyOS devices and tasks. Pivot Tracing [28] proposes happened-before joins to combine distributed path events. Request and path tracking in these systems target resource accounting or offline performance analysis. In comparison, request tracking in this paper is designed to support online scheduling. Our contribution lies in the recognition and tolerance of OS-level request tracking inaccuracies to support effective QoS scheduling.

Server scheduling: Druschel and Banga [20] recognized the need for early network packet demultiplexing and admission control on overloaded servers. Crovella et al. [17] and Schroeder and Harchol-Balter [33] proposed HTTP server scheduling prioritization on short requests or requests with short remaining processing time. However, it is hard to know the remaining processing time for a running request in realistic server applications with complex semantics. The Causeway system [14] presented prioritization scheduling for multi-tier server applications. Our work supports new request scheduling schemes including tail latency reduction and resource-contender throttling. Cohort scheduling [26] groups data-sharing stages in server execution to achieve high caching performance. Such resource schedulers optimize for the overall server efficiency which is orthogonal to our work that targets the quality-of-service of individual request responses, though it is conceivable to construct an integrated server scheduler that balances the tradeoff.

Multicore scalability and performance: The research community has invested heavily in managing the ubiquitous on-chip hardware resource sharing in today’s multiprocessors. Previous proposals include sharing-aware task scheduling [35], fairness-oriented execution timeslice adjustment [21], processor duty-cycle speed reduction [37], reuse distance profiling of shared cache
uses [38], and halting the offending executions [29]. In multicore operating system designs, Corey [12] argued for application-controlled sharing to enhance scalability while Multikernel [11] proposed the separation of per-CPU kernels to minimize sharing and contention. Scalable multicore systems motivate our work of fine-grained, request-level resource scheduling to accomplish high server quality-of-service.

**Tail latency reduction:** Recent research has recognized the importance of reducing the tail of request latency distributions for scalable services [18]. In a distributed system, tail latency reduction can be accomplished through re-launching slow tasks [19] or random sampling based on the power of two choices [30]. Executions with increasing parallelism [24] may also reduce the tail latency at the cost of increased overall resource use. Previous work has supported service-level scheduling or partitioning to reduce the tail latency in the context of mixed latency-sensitive / batch workloads [25, 27]. Our work of request-level scheduling complements these prior techniques with the distinction of fine-grained, request context-aware resource control.

## 3 Best-Effort Request Tracking and Labeling

We present the design and implementation of our OS-level facility that tracks request context propagation and labels scheduling parameters to enable QoS scheduling. Our approach is *best-effort* in that it recognizes the inherent inaccuracies of OS-level request tracking and minimizes the impact of such inaccuracies on QoS scheduling effectiveness.

### 3.1 Principles

On a server, each segment of CPU execution may belong to a certain request context or it may be part of background system/server activities that are not directly depended upon by any request completion. Examples of background execution include server maintenance work between request processing and system daemons like swapper and RCU delayed destructive-to-readers actions. These activities can often be postponed without causing immediate QoS loss. An OS-level request tracking facility binds each CPU segment to a certain request or to the background. Background executions generally account for a small proportion of the overall server CPU usage in practice (no more than two percentages of CPU usage for our workloads in Section 5.1). We discuss how each kind of request context identification inaccuracies affect quality-of-service scheduling and possible scheduler-tolerance to such inaccuracies.

A *background*→*request* context mis-identification indicates an incorrect binding of a background segment to a request context. It is generally harmless if the background segment is mis-identified as a low-priority request context. If it is mis-identified as high priority, it may execute unnecessarily early, causing delays to truly high priority requests. However, the impact is limited given the small amount of background executions.

A *request*→*background* context mis-identification indicates an incorrect binding of a request segment as background execution. This is worrisome if a segment of a high-priority request is bound to the background, which may cause severe delay of the high-priority work, regardless of how small the mis-identified segment is. Given that no prioritization is better than priority inversion, our approach is to revert to the default fair share scheduling whenever a background segment is in the scheduling queue. The lost opportunity for prioritization scheduling is limited as long as the background executions account for a small proportion of the overall server CPU usage.
A request→request context mis-identification indicates an incorrect binding of a request execution segment to a wrong request. In particular, if a high-priority request segment is incorrectly bound to a low-priority request context, the high-priority request may be severely delayed due to the dependency propagation of false priority. On the other hand, mis-identifying one request to another is not necessarily harmful to QoS scheduling as long as the mis-identified context carries a similar scheduling parameter (e.g., request arrival time or priority) to the correct context and therefore would be subject to similar scheduling order.

3.2 Design

We first bind the process (or thread) that receives a new request to the request context. A request arrival may occur at the acceptance of a new socket connection at the server’s port number. In the case of server pooling or persistent connections, a new request may also start on an existing connection and we need to recognize the protocol (e.g., HTTP) syntax to capture new request arrival messages. Besides the request arrival identification, our request tracking facility requires no additional application-specific information or assistance.

After initial binding, request contexts may propagate through additional processes belonging to different server stages. We capture such propagations through OS-visible data and control flows. Specifically we identified the following OS primitives to propagate request contexts—process (or thread) forking, signals, Internet and Unix domain sockets, and pipes. In the case of forking, a newly forked process or thread inherits the context binding of its parent. Signal-based request context propagation can be easily supported as well. For socket and pipe-based data flows, the data receiver, at the time of data receipt, should inherit the context binding of the sender at the send time. In the case of persistent connections, client and server processes may reuse a single socket connection for multiple request executions over time. Because send and receive operations may occur asynchronously (in particular, the send may occur before the receipt), the request context of the sender at the send time should be tagged with the buffered data. The context binding tag is then inherited by the receiver at the message receipt.

OS-level request tracking suffers from several inherent inaccuracies including invisible user-space synchronization, batched event processing, and late request unbinding. We manage them following the principles in Section 3.1.

**Invisible User-space Synchronization**  Request contexts may propagate through synchronization operations. For instance, a thread may signal another waiting thread through a monitor (including a mutex lock and condition variables). Much of a synchronization operation can be implemented in the user space except when the waiter has to block (e.g., yielding the CPU through a futex wait in Linux) while waiting for the synchronization signal. However, if such blocking is unnecessary (e.g., when the waiter can proceed immediately since the waited-for signal has already occurred), the entire synchronization may present no visibility to the OS kernel. Incomplete OS visibility leads to lost request context propagation binding. Such loss not only leads to request→background context mis-identification, it could also produce substantial request→request context mis-identification when a stale, incorrect context binding (due to the invisibility to a user-space synchronization) propagates through visible synchronizations and other propagation events like sockets. Figure 1(A/B) illustrate this problem for a realistic request in a Google App Engine application.

To minimize request→request context mis-identifications, our design choice is not to make use of partial OS visibility to synchronization operations. Specifically in Linux, we do not propagate
Figure 1: Request context propagations of a Google App Engine request between a proxy server thread and four Java threads. The proxy communicates with Java Thread 1 through a socket and Java Thread 1 interacts with other Java threads through synchronizations. Darkened portions of a thread timeline shows the CPU execution. Arrows indicate context propagation events observed at the OS. (A) shows the correct propagation of context binding A to all request execution segments. (B) shows that when one user-space synchronization becomes invisible, the wrong binding D ripples through many segments of the execution, eventually reaching the proxy thread through a socket message. The invisible synchronization and incorrect context bindings are marked in red. (C) shows the request context binding when propagating only on non-synchronization events.

request contexts through futex waits/wakeups, and therefore minimize the propagation of false request bindings. While this design choice leads to an increase of request→background context mis-identifications (e.g., losing the context binding for Java Threads 2/3/4 in Figure 1(C)), it is a worthwhile tradeoff given our recognition that request→background context mis-identifications are more tolerable than request→request mis-identifications by request schedulers. Furthermore, synchronization operations are usually short critical section executions. For the example request in Figure 1(C), not reaching Java Threads 2/3/4 leads to the mis-identification of about 0.5 mSec of request execution as background, which is a tiny fraction of the total request CPU execution time of 179.4 mSecs.

**Batched Event Processing** Event-driven servers including earlier Flash [31] and SEDA [36] as well as today’s Redis data store [2] divide the server work into non-blocking stages triggered by
certain events. Event polling and dispatch are usually handled by OS primitives such as \texttt{select()} and \texttt{epoll_wait()}. In a highly loaded server, an event dispatch call may return a batch of ready events for user space processing. However, different events in a returned batch usually belong to different request contexts and thus the context of subsequent user-space execution becomes ambiguous.

One solution to the problem is to return at most one ready event in each event dispatch call and then bind the subsequent execution to the request context that generated the returned event. This approach, however, leads to increased event dispatch calls and low efficiency given the high cost of event polling [9]. To maintain the batched efficiency, our approach is to return a batch of events with similar scheduling parameter (e.g., request arrival time or priority) that would be subject to similar scheduling priority. We will explore the design of such an event scheduler in Section 4.2.

Late Request Unbinding  There is no explicit context unbinding operation in our scheme. An inherited request context expires when the process (or thread) receives a new context propagation (e.g., through a socket message from a new request) or when the process exits. The former occurs when a process is reused to execute many requests over its lifetime (as in process / thread pooling servers). Strictly speaking, any execution after sending the last byte of data to the requesting client is not part of the request context since it is not depended upon by the request response. However, it is difficult for the OS to recognize such ending points without the ability to look into future executions or receive a notification from the application. Our inability of promptly closing a request binding produces background→request context mis-identification for after-final-response executions. Fortunately those executions are often very short and our discussion in Section 3.1 showed that a moderate amount of background→request context mis-identifications are well tolerated by QoS schedulers.

3.3 Implementation

We have implemented our request tracking facility in the Linux 3.12.13 kernel. We encapsulate the state of an active request context in a 672-byte data structure that includes a creation timestamp, control flags, locks, scheduling-specific data (such as priority), and accumulated software statistics (like context propagation counts) and hardware metrics (including retired instructions and wall / non-halt CPU cycles). To properly maintain the hardware metrics, we read relevant processor hardware performance counters whenever a request switches on a CPU (including process switches on a CPU and request context binding changes of a process).

Aside from all the running request contexts, we also maintain a special background context that executions without a request context binding are bound to. It is used to collect statistics on background executions and enable a scheduling context for such executions.

To tag the propagated request context in a socket message, we create a new option field in the socket message header. Our header option is compatible with the Internet protocol standard so our mechanism is safe to use in a distributed system where a peer node may or may not run our request context tracking kernel. In the latter case, the new option field is not recognized and therefore discarded by a vanilla operating system, but the socket communication still proceeds correctly. Inter-machine request context propagation is not important for our work of request scheduling within a single multicore machine, though it has been shown to be essential in distributed system performance analysis [6,16,23,32,34]. We will discuss this further in Section 6.

We manage active request contexts as inodes in a new pseudo file system. This enables us to implement a request context binding to a process (or thread) as an open file. Therefore we
can reuse the existing OS mechanisms for open file reference counting and resource reclamation (e.g., automatically closing and releasing all active request context bindings when a process exits). However, since a server process may run for a very long time and handle many requests over its lifetime, it is still important to close the request context file when the current context binding is overwritten by a new request. A request context is freed as part of the associated inode destruction when it is not used by any open file.

4 Quality-of-Service Request Scheduling

Built on our best-effort request tracking, we develop new OS schedulers that improve the quality-of-service of multicore servers. Our schedulers recognize request tracking inaccuracies described in Section 3 and mitigate their impact on the scheduling QoS. Below we present several specific request schedulers that reduce the tail of the request latency distribution (through CPU scheduling and event dispatching) and prioritize the execution of certain critical requests.

4.1 Tail Latency Reduction CPU Scheduling

The tail (or the worst cases) of the request latency distribution is critically important for a server’s quality-of-service [18]. This is because very fast responses below a human perception threshold makes little difference in the quality-of-service while an abnormally long delay causes dissatisfaction. Furthermore, less variance of each server’s response latency makes it easy to compose distributed operations with high performance when the final completion is dependent on the worst latency from any of the component servers.

A scheduler can reduce the latency variance and tail latency by giving higher priorities to requests who tend to finish with longer latencies. At a given scheduling moment, such longer-latency tendency can be recognized by two heuristics—an earlier request arrival time and higher amount of remaining work. The latter requires a workload prediction that would complicate the scheduler design. We focus on the former heuristic for a simple demonstration of request-context scheduling. Specifically, at each CPU scheduling opportunity, we examine the arrival timestamp of each task’s binding request context among all tasks in the CPU’s run queue and pick the one with the earliest arrival time.

When a background task is present in the CPU run queue, we face two design choices. An aggressive scheduler assumes the perfect accuracy of request context bindings of tasks. A background-context execution, by definition, is not depended upon by any request response and therefore should receive the lowest priority behind all tasks with request context bindings. This approach, however, may cause priority inversion and long tail latency if a segment of early-arriving request is mis-identified as belonging to the background.

In an alternative design, we recognize the possible context mis-identifications and conservatively revert to the default fair share scheduling when a background task is present in the run queue. This approach may lose some opportunities for request prioritization but it does not suffer from priority inversion.

Our scheduler change does not hurt the CPU utilization, nor does it degrade the task-CPU caching performance. The latter is true because the arrival timestamp is a static property for each request and prioritization based on such a static request property results in a stable scheduling order. Consequently each request tends to reside on a CPU continuously for a substantial amount of time. We expect no loss of the overall server efficiency.
4.2 Tail Latency Reduction Event Dispatching

On event-driven servers [2,31,36], tail latency reduction cannot be accomplished by CPU scheduling alone. An event-driven server process may handle many events on behalf of different requests and the event handling order within the process is not controlled by CPU scheduling. We enhance the event dispatching for tail latency reduction in the following fashion. First we tag each file descriptor triggering event with the request context at the trigger time. We then augment OS event dispatcher like `epoll_wait()` to prioritize the return of events belonging to requests with earlier arrival timestamps (those that more likely affect the tail latency). Earlier returns lead to their earlier processing and tail latency reduction.

Following Section 3.2, one approach (we call it Strict Ordering) returns only one ready event with the earliest request arrival timestamp in each `epoll_wait()`. It results in strict event dispatch ordering but incurs high overhead due to increased event dispatcher calls. To maintain the event batching efficiency, we can relax the event dispatch ordering by returning a group of events whose request arrival timestamps fall within a short distance (proximity threshold) from the earliest timestamp. We call this Proximity Batching. The proximity threshold presents a tradeoff between scheduling precision and event batching degree that affects the server efficiency.

Assuming work-conserving event scheduling without scheduler overhead, the proximity threshold of $T$ should not lead to a tail latency increase of more than $T$ compared to Strict Ordering. Specifically, let $x$ be a request exhibiting tail latency $L$ under Proximity Batching. We will identify one request that has at least $L - T$ latency under Strict Ordering—

- We first consider the case that no request that arrived after $x$ is scheduled to execute before $x$ under Proximity Batching. In this case no order relaxation has harmed $x$'s execution under Proximity Batching. Therefore its completion under Proximity Batching should be at least as fast as that under Strict Ordering. In other words, $x$'s latency under Strict Ordering should be no less than $L$.

- We then consider the case that some requests that arrived after $x$ has been scheduled to execute before $x$ under Proximity Batching. Let these requests be $y_1, ..., y_k$, ordered increasingly by their arrival time. By the definition of proximity threshold $T$, $y_k$'s arrival time is at most $T$ later than $x$'s arrival time. Under Strict Ordering, $y_k$ should run after $y_1, ..., y_{k-1}, x$, and all requests arrived before $x$ following the arrival time ordering. In other words, $y_k$ under Strict Ordering must follow all executions preceding $x$'s completion under Proximity Batching. Therefore $y_k$ under Strict Ordering should complete no earlier than $x$ under Proximity Batching. Since $y_k$'s arrival time is at most $T$ later than $x$'s arrival time, $y_k$'s latency under Strict Ordering should be at least $L - T$.

In practice, the tail latency increase of Proximity Batching is much less than the analytical bound, and in fact it often exhibits lower tail latency than Strict Ordering due to much improved server efficiency. Our implementation and experimentation recommend two settings of the proximity threshold for Proximity Batching—one or two times the average request latency. The former tends to produce highest tail latency reduction at slight throughput degradation while the latter can reduce tail latency at almost no loss of server throughput.

4.3 Prioritization and Contender Throttling

A server or cloud system may desire differential services to different requests and particularly prioritization of certain critical requests over others. Given our request context tracking facility,
a request prioritization scheduler can be easily constructed by preferentially running tasks whose binding request contexts carry high priority.

Beyond the scheduler reordering, we also recognize that concurrent request executions on a large-scale multicore machine may suffer from contention on shared hardware resources. For instance, on a 40-CPU (20 cores, each equipped with two hyperthreads) Intel "IvyBridge" machine, the instructions-per-cycle rate may degrade by up to 86% from single-request standalone runs to all-CPU highly concurrent request runs in our experimentation. We further observe that a large portion of the degradation occurs between 20-CPU executions (when sibling hyperthreads do not run at the same time) and all-CPU executions, which suggests intensive hyperthreading resource contention (on shared L1/L2 caches, instruction fetch/decode unit, and instruction execution pipeline).

Contention mitigation on resource-sharing multiprocessors has long been recognized [21,29,35,37,38]. Our request context tracking facility enables us to apply resource contention reduction at the request granularity. In particular, when a CPU finds that its own run queue only contains low-priority requests while a resource-contending sibling CPU is running high-priority work, it will temporarily suspend execution. Our resource-contender throttling leads to non-work-conserving CPU scheduling—resources may be idled while there is still pending work to do—which degrades the overall server efficiency. However, the throughput reduction is modest if the throttled CPU was subject to high resource contention (and therefore was not contributing much to the overall throughput) before the throttling.

Cross-CPU interrupt and synchronization is expensive in a high-throughput server. While our resource-contender throttling requires coordination between sibling CPUs, we implement such coordination asynchronously without any cross-CPU interrupts. In particular, this concerns the design point that a CPU with only low-priority work needs to check the current running status of its sibling CPU to determine whether to throttle. In our interrupt-free implementation, each CPU actively maintains the priority of the currently running request context in a global, per-CPU data structure. The per-CPU data is cacheline-aligned to avoid false sharing in cache. A CPU directly reads the running priority status of a sibling CPU without any synchronization. Though the status read may return stale information under certain race conditions, the mistake will likely be corrected at the next scheduling opportunity.

Since request→background context mis-identification is possible, we adopt a similar doing-no-harm approach as in Section 4.1 to avoid priority inversion. Specifically, when a background task is in the run queue, we revert to the default fair share scheduling to order tasks. Furthermore, we do not throttle a CPU even if only low-priority and background tasks are present in its run queue and a resource-contending sibling CPU is running a high-priority request.

5 Evaluation

We carry out experimental evaluation using a variety of server-style workloads described in Section 5.1. We will evaluate the effectiveness and overhead of our OS-level request tracking facility (Section 5.2). We will also evaluate the performance of our CPU and event scheduling for tail latency reduction (Section 5.3) as well as request prioritization and resource-contender throttling (Section 5.4).

We conducted experiments on a dual-socket machine where each socket contains an Intel Xeon E5-2660 v2 "IvyBridge" processor (10 cores, 2 hyperthreads per core, 2.20 GHz, 25 MB of shared L3 cache). The entire machine has 20 cores and 40 hyperthread CPUs.
5.1 Evaluation Workloads and Characteristics

Our evaluation uses five server and cloud computing workloads. OpenSSL RSA-crypto is a synthetic security processing server workload. Each request runs the RSA encryption and decryption procedures from OpenSSL 1.0.0d. It contains three types of requests: each uses one of the three encryption keys provided as examples in OpenSSL. The encryption server contains a pool of worker processes, each of which executes many requests over its lifetime.

StressAppTest, or Stressful Application Test [3], is an open-source benchmark that runs the Adler-32 checksum algorithm over 1 MBytes of memory with additional memory copying operations. It stresses the CPU core units, cache space, and memory access bandwidth simultaneously. We build a server workload in which each request invokes the StressAppTest processing on a separate memory region.

Apache Solr [1] is an open-source search engine platform from the Apache project. It uses the Lucene Java search library as the core for full-text indexing and searching. The search server runs within a Servlet container such as Tomcat. Our deployment uses Solr 3.6.1 and Tomcat 6.0.35 software. We construct a search workload of 1,414,444 indexed documents from the Wikipedia data dumps [5]. The indexed data fits into the memory of our server machine. Client queries in our workload are generated by randomly selecting and sequencing article titles in the Wikipedia data dump.

GoogleAppEngine, or GAE, provides a Platform-as-a-Service cloud computing infrastructure that enables users to build, maintain, and scale web applications. We install the GAE Java software development kit including a Java web server that supports the GAE runtime environment with a local datastore. On our GAE setup, we deploy the Vosao content management application [4], which supports collaborative building of dynamic web sites. We also produce a synthetic GAE-based Servlet application that performs CPU processing upon request. Our workload contains requests for both GAE applications.

Redis [2] is an event-driven server that supports a range of NoSQL data structure operations. We installed Redis 3.0.7 as a hash store in our experiments. Each request performs a number of hash GET or SET operations (9:1 ratio on GET vs. SET requests). The number of operations in each request follow an exponential distribution (averaging 100 operations per request).

These workloads contain the common server features of process/thread pooling (in our RSA-Crypto server, the Apache Tomcat web server, and the GoogleAppEngine platform), event-driven control (in Redis), and multi-stage execution (in all workloads). The Apache Tomcat web server and the GoogleAppEngine software platform run on a Java virtual machine that involves complex synchronization among Java threads.

Figure 2 illustrates the scalability and execution characteristics of these server workloads at increasing server concurrency (or the number of concurrently executing requests). These tests are conducted by a scalable event-driven client load generator that can issue requests at the desired concurrency level with high efficiency. The load generator runs on a different machine.

Figures 2(A) and (B) present the overall server throughput and average request response time respectively. We observe that GoogleAppEngine exhibits the best scalability while OpenSSL RSA-crypto and Apache Solr are also able to achieve a throughput speedup of 22 or higher at the server concurrency of 40. Redis does not scale well since its core event-driven server process cannot use more than one CPU. StressAppTest also manifests poor scalability due to intense resource contention on the multicore, demonstrated by severe degradation of instructions per cycle at increasing server concurrency in Figure 2(C). Other workloads (except Redis) also exhibit decreasing
Figure 2: Server workload scalability and execution characteristics on a dual-socket 40-CPU machine. (A) and (B) illustrate the overall server performance in terms of throughput and average request response time. (C) shows the instructions per cycle at increasing server concurrency.

instructions per cycle when the server concurrency increases. Redis suffers no multicore resource contention, again, because its core event-driven server process runs on one CPU.

### 5.2 Evaluation of Request Tracking

We evaluate how effective our request context tracking facility captures request context bindings in realistic server executions. We also assess its runtime overhead.

**Coverage** We are first interested in the coverage of our request context binding—how much of the total server execution time has been identified to be parts of some request contexts. Note that a low coverage does not necessarily mean context identification errors. However, a high degree of background work labeling in a server workload would hurt our scheduling effectiveness because, as a measure to prevent priority inversion, our schedulers conservatively revert to the default fair share scheduling when some background work is present in the run queue.

Although our request context tracking facility already collects per-task and background statistics, we use an independent data collection and analysis process for higher trustworthiness. Specifically, we employ Linux’s `debugfs`-based kernel tracing utility to collect relevant kernel-level events. We then perform offline trace analysis to acquire the desired metrics.

We show the request context coverage under two server concurrency levels (40 and 80) and two request context propagation policies (whether to use all data, control flow, and synchronization events or to exclude the partially visible synchronization events in `futex` operations)—

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Concurrency 40</th>
<th>Concurrency 80</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>-futex</td>
</tr>
<tr>
<td>RSA-crypto</td>
<td>99.8%</td>
<td>99.8%</td>
</tr>
<tr>
<td>StressAppTest</td>
<td>99.7%</td>
<td>99.7%</td>
</tr>
<tr>
<td>Apache Solr</td>
<td>99.9%</td>
<td>99.5%</td>
</tr>
<tr>
<td>GAE</td>
<td>99.8%</td>
<td>98.9%</td>
</tr>
<tr>
<td>Redis</td>
<td>98.8%</td>
<td>98.8%</td>
</tr>
</tbody>
</table>

Results show high coverage of our request context binding in all cases—at most 2.2% of the CPU execution is marked as background activities. The increasing concurrency only slightly affects the request context coverage. The removal of `futex`-based context propagation leads to at most 2% loss of request context coverage.
**Error**  A complete evaluation of request context binding errors is challenging due to the lack of the ground truth. Our evaluation focuses on a particular portion of binding errors that can be identified—the binding of a request context to any execution segment after the request’s final response to client. This evaluation is not applicable to event-driven Redis since our Proximity Batching event dispatch intentionally produces imprecise request bindings. We also use our independent kernel trace collection and analysis process in this evaluation.

We show the request context post-final-response binding errors under two server concurrency levels (40 and 80) and two request context propagation policies (whether to exclude the use of `futex` events). The error metric is the proportion of the total CPU execution time that is incorrectly bound—

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Concurrency 40</th>
<th>Concurrency 80</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>−futex</td>
</tr>
<tr>
<td>RSA-crypto</td>
<td>0.3%</td>
<td>0.3%</td>
</tr>
<tr>
<td>StressAppTest</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Apache Solr</td>
<td>0.6%</td>
<td>0.3%</td>
</tr>
<tr>
<td>GAE</td>
<td>94.7%</td>
<td>1.2%</td>
</tr>
</tbody>
</table>

Results show potentially very large context binding errors when the partially visible synchronization events in `futex` operations are utilized. This validates our design decision of excluding the use of `futex` events in Section 3.2. Our system produces at most 3.0% post-final-response context binding errors in all cases. Higher server concurrency produces a moderate error increase due to higher chances of interleaved request executions.

**Overhead**  We show the server throughput changes between original Linux and the enhanced OS with our request context maintenance.

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Original Linux</th>
<th>Request context</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA-crypto</td>
<td>1,138 reqs/sec</td>
<td>1,142 reqs/sec</td>
</tr>
<tr>
<td>StressAppTest</td>
<td>965 reqs/sec</td>
<td>967 reqs/sec</td>
</tr>
<tr>
<td>Apache Solr</td>
<td>611 reqs/sec</td>
<td>611 reqs/sec</td>
</tr>
<tr>
<td>GAE</td>
<td>672 reqs/sec</td>
<td>670 reqs/sec</td>
</tr>
<tr>
<td>Redis</td>
<td>900 reqs/sec</td>
<td>891 reqs/sec</td>
</tr>
</tbody>
</table>

Experiments were run at the server concurrency of 40. In each setting, we perform at least five rounds of experiments, each for 50 seconds (executing more than 30,000 requests for each workload). The reported throughput is the average of three median results from the five-round tests. Results show that the overhead is negligible for all five workloads (measured at no more than 1%). The very slight throughput increase for RSA-crypto and StressAppTest is due to the fact that the small overhead sometimes fall inside the measurement errors.

5.3 Evaluation of Tail Latency Reduction

We evaluate the performance of our request tail latency reduction mechanisms including the CPU scheduler presented in Section 4.1 and event dispatching presented in Section 4.2. We compare against the Linux fair share scheduler that provides fair resource allocation to processes (or threads). We also compare between two variations of our tail latency reduction schedulers—one assuming perfect request context and the other reverting to the default fair scheduler when any background
Figure 3: 99th percentile tail (first row) and average (second row) request latency under the fair share scheduler in Linux and two of our request-context tail latency reduction schedulers—one assuming perfect request context and the other reverting to the default fair scheduler when any background task is present in the CPU run queue.

Figure 4: The request latency distribution (in probability density) under the fair share scheduler in Linux (first row) and our request-context tail latency reduction scheduler (second row). The shown results are collected from executions at the server concurrency of 160. The Y axis (probability density) on a linear scale. The absolute probability density values are trivially determined by the histogram bin widths in each plot and therefore we do not show them.

task is present in the CPU run queue. The second approach can avoid the priority inversion problem in the event that request→background context mis-identifications occur.

Figure 3 shows the tail (99th percentile) and average request latency under the three schedulers with increasing server concurrency. Particularly at the server concurrency of 160, compared to the Linux fair scheduler, our tail latency reduction scheduler that assumes perfect request context reduces the 99th percentile request response time by 10%, 32%, 9%, and 59% for RSA-crypto, StressAppTest, Apache Solr search, and GoogeAppEngine. But it increases the tail latency of Redis hash store by 27%. Our priority-inversion-avoiding scheduler reduces the 99th percentile request response time by 8%, 24%, 35%, 50%, and 43% for the five workloads respectively. Though the tail latency reductions are slightly less for StressAppTest and GoogleAppEngine, the improvement is
far greater for the Apache Solr search and Redis hash store.

The second row of Figure 3 shows very similar average latency results between the three schedulers in most cases, suggesting that our request-context tail latency reduction scheduler does not degrade the overall server efficiency. We also notice that in one exceptional case (StressAppTest at the server concurrency of 160) our tail reduction schedulers actually improve the average request latency by about 19% compared to the Linux fair scheduler. We believe this is due to our tail reduction scheduler’s better caching efficiency at high server concurrency. Specifically, our arrival timestamp-based prioritization results in a stable scheduling order and tends to keep a request resident on a CPU continuously for a substantial amount of time.

Figure 4 provides some comparative illustrations of the request latency distribution under the Linux fair share scheduler and our request-context tail latency reduction scheduler (the second variation that avoids priority inversion). We can see that the distribution tails are reduced for all five workloads and the reductions are substantial for StressAppTest, Apache Solr, GoogleAppEngine, and Redis.

Figure 5 provides some comparative illustrations of the request latency distribution under the Linux fair share scheduler and our request-context tail latency reduction scheduler (the second variation that avoids priority inversion). We can see that the distribution tails are reduced for all five workloads and the reductions are substantial for StressAppTest, Apache Solr, GoogleAppEngine, and Redis.

Figure 5: Tail latency and throughput tradeoff of the Redis hash store under different event dispatching approaches.

**Comparison of Event Dispatching Approaches** Section 4.2 presented two tail latency reduction event dispatching approaches—Strict Ordering and Proximity Batching. It further explains a potential tradeoff between tail latency and throughput on the proximity threshold of Proximity Batching. We recommend two settings of the proximity threshold—one or two times the average request latency. Earlier evaluation in this section used Proximity Batching with the threshold of two times the average request latency. Here we compare the tail latency and throughput of Redis under different event dispatching approaches.

Figure 5 shows that while Strict Ordering can reduce tail latency at low load, it actually produces worse tail latency than the original Linux at high load (160 server concurrency). This is because its one-event-per-call approach substantially increases the number of OS event dispatcher calls and incurs high overhead. Proximity Batching with the threshold of average request latency produces
substantial tail latency reduction (57% at 160 server concurrency) with slight (4%) degradation of overall server throughput. Alternatively, the threshold of two times average request latency reduces tail latency (43% at 160 server concurrency) with almost no loss of server throughput.

5.4 Evaluation of Request Prioritization and Contender Throttling

We evaluate the performance of our request prioritization and resource-contender throttling schedulers presented in Section 4.3. For the OpenSSL RSA-crypto workload, we divide its requests into three groups according to the encryption key used in processing. We call them Type-1, Type-2, and Type-3 requests. We then run experiments with schedulers that prioritize for each type of requests. We compare between the original Linux scheduler, our request prioritization scheduler that reorders tasks in the scheduler run queue according to the priority of bound request context, and our enhanced request prioritization scheduler that also throttles the resource-contending CPUs to high-priority executions. Since the RSA-crypto workload is particularly susceptible to contention between sibling hyperthread CPUs, we throttle a CPU with only low-priority requests in its run queue while its hyperthread sibling is executing high-priority work.

Figure 6 illustrates the scheduling performance of experiments that prioritize for each type of RSA-crypto requests at the concurrency levels of 80 and 160. In each test case, we observe that our request prioritization scheduler can significantly lower the latency of the high-priority requests compared to the default scheduler. At the server concurrency of 160, the speedup for high-priority requests are $1.7 \times$, $1.5 \times$, and $2.4 \times$ for the three prioritization request groups respectively. Our resource-contender throttling mechanism can further reduce the latency of high-priority requests. The resulted speedups to the default scheduler are $1.8 \times$, $1.7 \times$, and $3.0 \times$ for the three prioritization request groups respectively.

Among the three types of requests, our prioritization schedulers are less effective for Type-1 and Type-2 requests. This is because they are relatively small (less than 10 mSecs of CPU time in a standalone run) and present fewer opportunities for scheduling optimization. Type-3 requests (around 40 mSecs of CPU time in a standalone run) present more scheduling optimization opportunities and we are able to achieve high performance in request prioritization. The request slowdowns compared to standalone executions are only $1.7 \times$ and $2.2 \times$ at the server concurrency levels of 80 and 160 respectively.

Our basic request prioritization scheme does not affect the overall server throughput. Our resource-contender throttling mechanism, however, results in non-work-conserving scheduling and possibly degraded server throughput. Fortunately, throughput reduction is modest when the throttled CPU was subject to high resource contention (and therefore was not contributing much to the overall throughput) before the throttling. In all cases of our experiments, we observe no more than 4% reduction in the overall server throughput resulted from our resource-contender throttling mechanism.

6 Conclusion

This paper demonstrates the feasibility and benefits of operating system support for application-transparent, request-context resource scheduling on multicore servers. We face the challenge that the OS-level request tracking suffers from inherent inaccuracies of invisible user-space synchronizations, batched event processing, and late request unbinding. Our contribution is to devise best-effort
Figure 6: Scheduling performance of request prioritization for Type-1 requests (first row), Type-2 requests (second row), and Type-3 requests (third row) in the OpenSSL RSA-crypto workload. We show results at the server concurrency of 80 (first column) and 160 (second column). In each case we compare between the original Linux scheduler, our request reordering scheduler, and our request reordering scheduler augmented with resource-contender throttling. The performance metric is the latency of each type of requests normalized to the latency when the given type of requests run alone on the machine with no concurrency.

request tracking that minimizes the least scheduler-tolerable context binding inaccuracies. In particular, background→request context mis-identifications present no significant priority inversion concerns when the system contains a small amount of background work. Request→background mis-identifications are more worrisome but can be addressed by reverting to the default fair share scheduling (doing no harm) when any background work is present in the CPU run queue. Finally, while request→request mis-identifications may lead to priority inversion, it presents little harm as long as the mis-identified context carries a similar scheduling parameter to the correct one.

We constructed request schedulers for latency distribution tail reduction and critical request prioritization upon our best-effort request context tracking. Our tail latency reduction mechanisms
manage both CPU scheduling and event dispatching orders (the latter is critical for event-driven servers). Our prioritizing CPU scheduler not only reorders the task executions according to request context priorities, but it also mitigates the hardware resource contention by throttling the sibling CPU(s) of a CPU executing a high-priority request. For ease-of-use and broad applicability, our request context tracking and scheduling mechanisms reside in the OS kernel with full transparency to applications (except for the identification of a request arrival).

We performed experiments with an OpenSSL-RSA encryption workload, a CPU/cache/memory-stressing synthetic server [3], the Apache Solr search engine [1], a Google App Engine cloud workload, and an event-driven Redis hash store [2] on a 40-CPU multicore machine. Results show that our request context tracking facility produces high coverage and low errors for realistic server workloads. The overhead is negligible (measured at no more than 1%) for all five workloads. Compared to the Linux fair share scheduler, our tail latency reduction scheduler and event dispatcher reduces the 99th percentile request response time by 8%, 24%, 35%, 50%, and 43% for the five workloads respectively. Our request prioritization and resource-contender throttling scheduler can speed up high-priority requests by 1.7–3.0× during highly concurrent server executions.

This paper investigates request-context resource scheduling within a single multicore server machine. In a networked, cloud environment, the full scope and latency of a request may cover executions over multiple machines and wide-area network delays. Previous studies [6,16,23,28,32,34] have shown the great importance of networked (or distributed) systems request or path analysis. We believe it is possible to extend our machine-level request-context CPU scheduling into a distributed environment as long as request contexts, start timestamps, and priorities can be maintained throughout the system. Our principles on request scheduler’s tolerance to certain context binding inaccuracies should still apply in a distributed system. Its design and implementation is out of the scope of this paper.

References


