Using Variability as a Guiding Principle to Reduce Latency in Web Applications via OS Profiling

Amoghvarsha Suresh, Anshul Gandhi
PACE Lab, Stony Brook University
{amsuresh,anshul}@cs.stonybrook.edu

ABSTRACT

Request latency is a critical metric in determining the usability of web services. The latency of a request includes service time – the time when the request is being actively serviced – and waiting time – the time when the request is waiting to be served. Most existing works aim to reduce request latency by focusing on reducing the mean service time (that is, shortening the critical path).

In this paper, we explore an alternative approach to reducing latency – *using variability as a guiding principle when designing web services*. By tracking the service time variability of the request as it traverses across software layers within the user and kernel space of the web server, we identify the most critical stages of request processing. We then determine control knobs in the OS and application, such as thread scheduling and request batching, that regulate the variability in these stages, and demonstrate that tuning these specific knobs can significantly improve end-to-end request latency. Our experimental results with Memcached and Apache web server under different request rates, including real-world traces, show that this alternative approach can reduce mean and tail latency by 30–50%.

1 INTRODUCTION

Web applications provide important services, such as online retail, messaging, and search, to end-users on a daily basis [3, 5, 8, 13]. A critical metric for determining the usability of such web applications is the *latency* of user requests; different service providers employ different measures of latency, such as mean, median, 95%ile, and 99%ile [7, 37, 47]. A delay of even a few milliseconds in request latency can lead to significant revenue loss for service providers due to user abandonment [14, 44].

Conceptually, request latency can be broken down into two distinct components, *service time* and *waiting time* [27]. Service time is defined as the time during which the request is being actively serviced. Waiting time is then defined as the remaining time during which the request is waiting to be served. To maintain acceptable latencies, service providers often optimize their web servers to reduce the mean service time of requests, that is, shorten the critical path of request processing. For example, Chronos [24] uses user-level networking with NIC-level request dispatch to reduce lock contention and lower the latency of web applications; Jose et al. [53] make Memcached RDMA-capable to shorten the critical path; Li et al. [31] advocate using a real-time scheduler to reduce request scheduling delay.

An alternative approach to reducing web latencies that we explore in this paper is to minimize the *variability* in request processing. Queuing models show that, in addition to mean service time, the variability of the service time is also important when trying to reduce latency. Surprisingly, there has been very little work on actually reducing the variability in the system [7, 19, 49]. While much of the variability is intrinsic to the workload (e.g., burstiness in customer traffic), some of the variability is due to the *application and software design* (e.g., garbage collection, context switches, CPU scheduling policies, etc.), and can be regulated by making subtle changes to the system.

In this paper, we investigate the following system design question – "*is it worth reducing variability in request processing times at the potential expense of lengthening its critical path?*". While theoretical analysis suggests that this is indeed the case, especially for heavy-tailed distributions (see Section 2), evaluating this idea in practice for web applications is challenging for several reasons:

- Web applications have a *complex processing lifetime*, going through several paths of processing in the kernel, including the TCP stack, parsing of requests, and scheduling of the request on server cores. Identifying the most likely culprit(s) that contributes to service time variability will require low-overhead yet accurate and fine-grained *request tracing* in the user and kernel space.
- There are several control knobs within a web server that can be tuned to reduce service time variability, such as the OS scheduler, page allocation strategy, etc. Prior work has also shown that new application-specific control knobs can be dynamically generated [18]. Given the numerous choices, efficiently finding the right control knob to mitigate variability is challenging, especially since the choice of the optimal control knob may depend on the server and web application configuration. Worse, employing the wrong control knob can *hurt* request latency.
- Most control knobs within the system that can be tuned to reduce service time variability *invariably hurt mean service time*. For example, prior work has shown that admission control can reduce service time variability by preventing server overload [21]. But this reduction comes at the expense of lengthening the critical path (since requests are held back), and possibly hurting latency. It is thus important to carefully tune the control knob to balance the trade-off between variability and mean service time.

We address the above challenges in the context of web services and demonstrate the benefits of *using service time variability as a guiding principle* to improve request latency. We employ light-weight, fine-grained request profiling to track service time variability; this allows

WWW '19, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313406

us to identify components on the critical path of the request-response cycle that exacerbate service time variability. Using variability as our guiding principle, we then determine control knobs in the system and/or application that can be tuned to mitigate service time variability in the identified components. We evaluate our approach by investigating variability in Memcached, a popular in-memory (key-value store) caching service used to speed up web applications [12], and the Apache web server [45], a widely deployed http server application. Specifically, we explore the following use cases:

(1) *Request batching at the Memcached client*: To reduce the network overhead of sending short packets, the Linux kernel at the client batches incoming requests until it gets a response back from the server saying it has received the previous batch (Nagle's algorithm [40]). However, since network conditions can be variable, this default batching behavior leads to bursts of processing at the server, followed by idle times, resulting in significant service time variability. We modify this default batching behavior to closely regulate the time between batches, lowering the variability (by as much as 76%) and improving tail latency by up to 40%.

(2) *Redesigning the LRU management on the Memcached server*: To maintain the Least-Recently-Used (LRU) ordering of items in the cache, the Memcached server needs to constantly reorder items. To shorten the critical path, Memcached moves the LRU maintenance off the request processing path and delegates this responsibility to an LRU management thread that is run periodically. When this thread is active, it interferes with the Memcached request processing, resulting in high service time variability. We redesign the LRU management functionality and include it on the critical path of Memcached request processing in the form of fine-grained slices of LRU work. This counter-intuitive redesign reduces variability and improves latency by about 30%.

(3) *Application thread pinning for the Apache web server*: The Apache web server schedules its various worker threads and processes on any available idle CPU core opportunistically to start serving customer requests as soon as possible. Consequently, threads may move between cores, resulting in context switch overheads and state migration. We explore thread pinning to reduce this overhead and associated variability, though at the expense of possibly delaying request processing when the pinned core is busy. We show that, at high load, this approach can reduce latency by 50%.

We experimentally evaluate the performance improvements for the above three use cases under various workload scenarios, including different levels of load, different inter-arrival time distributions, and different (time-varying) arrival traces. While existing approaches, such as exclusively focusing on reducing mean service time or employing ad-hoc control knobs, can end up *hurting* request latency, we use our alternative approach of focusing on service time variability to substantially reduce mean and tail latency (by up to 30–50%) in all three cases. Importantly, we do so by simply changing the metric that we use to identify the critical stage of request processing and determine the appropriate control knob.

2 MOTIVATION AND SCOPE OF OUR WORK

To motivate our approach of focusing on service time variability, we leverage queueing theory to analyze request latency for a server as a function of variability. Although models are only approximations of today's complex applications, the resulting analysis is instructive [16, 33, 36, 48], and guides our system design in later sections.

2.1 Request latency versus variability

Recent studies at Bing [15, 19], Google [23, 24, 43], and Facebook [3], suggest that modern web applications often experience high variability in inter-arrival time (IAT) and service time (ST). Variability in IAT represents workload variability, such as bursty arrivals. ST variability represents variability in processing times due to differences in work requirements (e.g., reads vs. writes) or differences in the request path (e.g., due to batching or TLB misses), or due to misconfigurations at the server that lead to anomalous behavior or "jitters" [31]. Note that ST is the amount of service required by a request to complete processing; alternatively, ST is the minimum possible request latency, assuming no delays.

To investigate the impact of variability on request latency, we consider a web server with a given inter-arrival time (IAT) distribution and a given service time (ST) distribution. To parameterize variability, we use the squared coefficient of variation (C^2), defined as the ratio of variance and square of the mean. Then, we have:

$$C^2_{IAT} = Var(IAT)/E^2[IAT], \text{ and} \quad (1)$$

$$C^2_{ST} = Var(ST)/E^2[ST], \quad (2)$$

where $E[]$ is the mean and $Var()$ is the variance of the random variable. To examine the full range of variability ($[0, \infty)$), we consider the following distributions:

- D (Deterministic), with $C^2 = 0$, is the ideal case of no variability.
- M (Exponential), with $C^2 = 1$, represents nominal variability.
- H_2 (Hyper-exponential), with $C^2 > 1$ (customizable), represents the case of high variability.

The $M/G/1$ queueing model, with Exponential IAT and generic ST distributions, allows us to analyze mean request latency, $E[T]$, as a function of ST variability, $Var(ST)$, and mean ST, $E[ST]$, via the Pollaczek-Khinchin (P-K) formula [27]:

$$E[T] = Var(ST) \cdot \frac{\lambda}{2(1-\rho)} + E[ST] \cdot \frac{2-\rho}{2(1-\rho)}, \quad (3)$$

where $\lambda = 1/E[IAT]$ is the mean request arrival rate and $\rho = \lambda \cdot E[ST]$ is the normalized system load [27]. Clearly, both $E[T]$ *and* $Var(ST)$ impact request latency.

Consider the $M/G/1$ model where all parameters, including $E[ST]$, are fixed, and only $Var(ST)$ is varied. Let the workload have a mean ST, $E[ST]$, of 1 millisecond, and system load of, say, 60%; thus, request rate, $\lambda = 0.6 \cdot E[ST] = 600$ req/s. Using Eq. (3), we find that $E[T] = 1.7ms$ under Deterministic ST ($C^2_{ST} = 0$). For comparison, for a system with Deterministic IAT and ST, we get $E[T] = 1ms$. Thus, the added variability due to an Exponential IAT already increases mean latency by 70%. For the $M/G/1$ system, if we now consider Exponential ST ($C^2_{ST} = 1$), we get $E[T] = 2.5ms$, a 250% increase over the baseline. If we further increase the variability in ST to $C^2_{ST} = 2$ using a H_2 distribution, we get $E[T] = 3.3ms$, a 330% increase! The increase in request latency with variability is even more pronounced at higher loads.

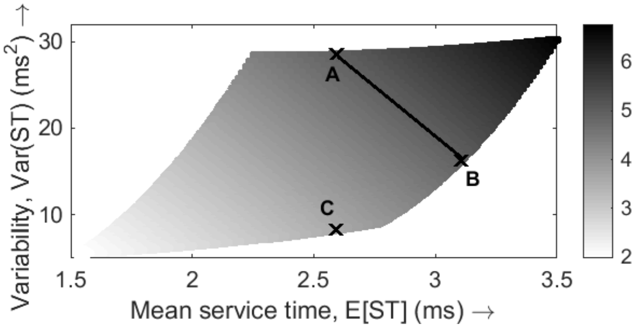


Figure 1: Heatmap of mean latency as a function of mean service time ($E[ST]$) and variability in service time ($Var(ST)$). The black diagonal line denotes the equi-latency line of $5ms$.

While the above results highlight the importance of reducing variability in ST, we note that making changes to a system to reduce variability may lead to other performance overheads, resulting in higher $E[ST]$. For example, the OS scheduler may opportunistically move threads between cores to leverage idle cores. However, this introduces ST variability due to the associated state migration and context switches. Disabling this opportunistic behavior can mitigate variability, but may lead to scenarios where threads are waiting on a busy core even though other cores are idle. There is thus a trade-off between reducing $Var(ST)$ and increasing $E[ST]$, which is captured by Eq. (3).

Figure 1 illustrates the impact on mean request latency of the trade-off between $E[ST]$ and $Var(ST)$ for the $M/G/1$ model. We set $\lambda = 100$ req/s, and use an H_2 distributed ST whose variability can be controlled. To highlight the trade-off, we show the equi-latency line of $5ms$ in black. Point B on this line has a *higher* $E[ST]$ but *lower* $Var(ST)$ compared to point A; nonetheless, both have the same $5ms$ latency. Thus, we can afford some overhead in $E[ST]$ when reducing $Var(ST)$ of a system. Finally, point C has the same $E[ST]$ as A, but has much lower $Var(ST)$; as a result, the mean request latency for C is almost 30% lower than that for A.

Key takeaways: *(i) Reducing service time variability can significantly lower request latency (by 2× or higher), and (ii) It is beneficial to reduce variability even if doing so increases the mean service time.*

2.2 Objective and scope of this work

The design of today's web server systems is primarily driven by the intent to shorten the critical path of requests, that is, reducing the mean ST. However, our analytical results above suggest an alternative solution to improving request latencies in systems – *reducing variability*. Variability is often assumed to be intrinsic to the hardware and the workload, and thus not easily controllable. However, this is not always true. While variability in IAT depends on customer request behavior, variability in ST can be regulated to some extent by modifying the application and software stack. The objective of our work is to demonstrate that this alternative viewpoint of "focusing on ST variability for designing web servers" can reveal viable solutions to reduce latency. Note, however, that we are *not* arguing against solutions that focus on reducing ST.

3 SOLUTION OVERVIEW

Given a web application, our goal is to improve its latency by targeting a reduction in service time (ST) variability. However, web applications can be complex, consisting of several processes and threads that work asynchronously. Further, web requests typically pass through several software layers before completing service. Thus, we must first identify the potential software layers or components that significantly contribute to ST variability, and then determine control knobs that can be tuned to reduce ST variability in the identified software layers. Some control knobs are readily available, such as OS thread scheduler, TCP congestion control, etc. However, in some software layers, effective control knobs may not be available, necessitating modifications to existing software.

We use the following methodology to achieve our goal:

(1) *Fine-grained, unobtrusive request tracing*: We employ low-overhead tracing for the web requests. Our tracing encompasses all layers of the software stack that a request goes through during its processing lifetime, including the OS and network stack.
(2) *Identifying the source(s) of service time variability*: We aggregate the tracing information across all requests using low-overhead histograms to identify the software layers with the highest variability that are amenable to modification.
(3) *Modifying the system software to mitigate variability and reduce request latency*: We explore available control knobs in the system that can reduce the observed ST variability in the target software layer, even if this reduction comes at the expense of an increase in mean ST (see Section 2.1). Once the knob is determined, we investigate its optimal setting to minimize end-to-end request latency.

Our choice of "variability of service time" as the guiding principle in our methodology is important. We demonstrate, in Sections 6 – 8, that our methodology allows us to uncover the critical stages of request processing for two popular web applications and guides our selection of control knob. We show later, in Section 9, that naïvely employing request tracing (for example, by using mean ST as the metric instead of variability) or selecting control knobs in an ad-hoc manner can actually *hurt* application performance.

4 REQUEST TRACING

Our request tracing works by timestamping the request through several layers starting from when it arrives at the host from the server's NIC until immediately before the application transfers the response packet back to the OS. Given our focus on variability in service times, we only trace the request at the host server, and not the client.

To store the timestamps, we append an empty 64-byte buffer to the original request packet, similar to prior works [31]. Then, as the request goes through different stages of processing on the client and the server, timestamps are recorded at appropriate offsets for the stage of processing by writing the system clock time into the buffer; we use the system clock with nanosecond precision to record timestamps within the server. By appending the small buffer to the request, we can record multiple timestamps and track the request to which they correspond without requiring any additional post-processing. The above tracing implementation required modification to the Linux kernel source, network drivers, and the application protocols to write timestamps into the appended buffer at the right offset.

[redacted]

The overhead on request latency through all layers is low, about 5%; this overhead is incurred by both the baseline and our approach.

To choose the timestamping locations, we consider all possible components within the host server that may contribute to service time variability; these are locations where significant request processing may occur. For this study, we timestamp at the following locations/events at the host server:

e1: In the host network driver when the packet arrives at the NIC.
e2: At the end of TCP processing.
e3: When the application drains the requests from the socket.
e4: When the application starts processing an individual request.
e5: When the application hands off the response to the kernel.
e6: When the response is dispatched from the host server's NIC.

Timestamping at these different event boundaries provides us an opportunity to analyze the time spent by the requests at different stages, T_{ei}, for $i = 1,2,...6$, as shown in Figure 2. However, other fine grained events are used when required, see Section 7.3.

- **driver-to-tcp**: $(T_{e2} - T_{e1})$ is the time spent by the request from driver to TCP layer, representing the network stack processing delay.
- **tcp-to-socket**: $(T_{e3} - T_{e2})$ is the time spent between the TCP layers and the socket, and represents the wakeup/scheduling delay.
- **socket-to-parse**: $(T_{e4} - T_{e3})$ is the time between application processing and socket, and represents the queuing delay at the application level caused by batching of requests.
- **parse-to-response**: $(T_{e5} - T_{e4})$ is the user-space application processing time.
- **response-to-send**: $(T_{e6} - T_{e5})$ is the network stack processing time to dispatch the response from the host server.

To efficiently compute variability at each stage, we maintain a running sum (across requests) of X and X^2, where X is the time spent in a stage. At the end of an experiment, we compute the sample moments $E[X]$ and $E[X^2]$ by dividing the running sums by the number of requests. Finally, we compute $Var(X) = E[X^2] - (E[X])^2$.

In our evaluation, we find that the most significant stages, in terms of variability, are the tcp-to-socket, socket-to-parse, and parse-to-response. We will discuss their role in request processing in detail in the evaluation sections. We also traced the request on the outgoing send path (server socket to driver), but the variability on this path is much smaller, only about 20% of the variability on the incoming path from client to server. Unless otherwise noted, we report $(T_{e6} - T_{e1})$ as the latency of a given request. This definition only includes the server-side latency, which is the focus of our current work.

5 EXPERIMENTAL SETUP

Testbed: We employ two servers in our experiments, one as the client to generate the workload and the other as the server hosting the web application. Each server is equipped with two sockets of Intel Xeon E5-2620 6-core (12 threads, 2.4 GHz) processors, running Ubuntu Linux 14.04 with kernel version 3.16.7. The servers each have 64GB of DRAM (1333 MHz) divided into two 32GB NUMA nodes. The servers each have an Intel I210 1Gb NIC, and are connected via a Quanta LB4M switch. Hyper-threading and power saving mechanisms such as DVFS and sleep states have been disabled in both machines as they are known to cause variability [31].

Web applications: We employ two popular web applications, Memcached [12] and Apache web server [45], for our evaluation.

- *Memcached* is a lightweight and scalable in-memory key-value store, primarily used for accelerating dynamic web applications by acting as a cache for the back-end database. Memcached is employed by several web services, including Facebook [3, 42], Twitter [46], Wikipedia [35], and YouTube [9]. Memcached maintains a list of least-recently-used (LRU) active items in memory to aid the memory reclamation process when an item expires or needs to be evicted. Memcached, from version 1.5.0, has overhauled its LRU maintenance mechanism by using a *dedicated LRU maintainer thread* that speeds up the release of expired items and the allocation of new items [38]. While the dedicated LRU thread is not on the critical path of request processing, it may interference with requests by requiring a core to run at frequent intervals.

  Our Memcached server hosts six million key-value pairs, with the key and value size for each pair set to 20 bytes and 50 bytes, respectively. The workload consists of 90% get and 10% set requests. We use Memcached version 1.5.1 and test two different throughput configurations (Sections 6 and 7).
- *Apache* is a modular, process based web server used by more than 43% of the websites [34]. The Apache web server forks identical processes and can serve a request independent of others. Apache in its newer versions from v2.0 can run in a hybrid multi-process, multi-threaded mode, which improves scalability. We use a newer version (v2.3.4) in our evaluation (Section 8).

Request rate traces: We employ request rate traces in many of our evaluations to drive the application load. Specifically, we use three digitized Memcached traces from Facebook [3], appropriately scaled for our setup, to mimic the load variation experienced by real-world web services. The VAR trace has a gradual drop in request rate, followed by a gradual rise. The APP trace has a steep rise, followed by a relatively constant request rate, and ends with a steep drop. The ETC trace is similar to APP, but is much more bursty. The peak-to-min request rate ratio in all three traces is about 2.

6 EVALUATION 1: REQUEST BATCHING AT THE MEMCACHED CLIENT

Our first evaluation focuses on the Memcached application. In our experiments, we find that Memcached has different sources of variability under high and low throughput configurations. We thus evaluate these configurations separately, starting with high throughput. All the results we report are averaged over 5 experimental runs each.

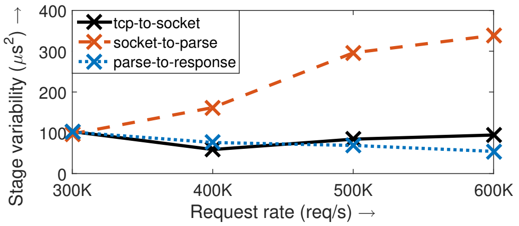


Figure 3: Variability at various request processing stages for Memcached under high throughput setting.

6.1 Application setup and tracing results

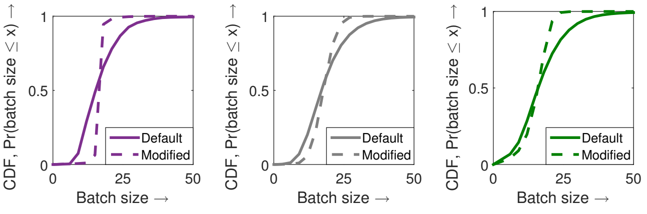
In the high throughput configuration, we run Memcached on the server with six worker threads on six cores of a socket. We use a custom client load generator that sends read/write requests at controllable request rates. In addition to the request rate traces described in Section 5, we also experiment with different inter-arrival time (IAT) distributions, motivated by prior studies on web IATs [2–4, 25]: Deterministic ($C^2_{IAT} = 0$), Exponential ($C^2_{IAT} = 1$), and Bounded Pareto ($C^2_{IAT} = 20$, IAT range of (5μs, 100ms)). The load generator emulates multiple clients by opening several connections sequentially for each worker thread, and sends multiple requests on each connection with the configured IAT. The load generator also supports different popularity distributions for the requested data (keys); unless stated otherwise, we use the random (discrete uniform) distribution.

Figure 3 shows the results of our request tracing for the three stages with the highest variability under Memcached. We see that the *socket-to-parse* stage has the highest variability, by far, compared to others. The socket-to-parse stage at the server involves parsing the drained batch of requests from the socket one request at a time (see Section 4). The time taken in this stage is thus proportional to the size of the batch being served by the application. Consequently, the variability in this stage suggests that there is variability in the request batch sizes as observed by the application at the host server.

6.2 Determining the control knob

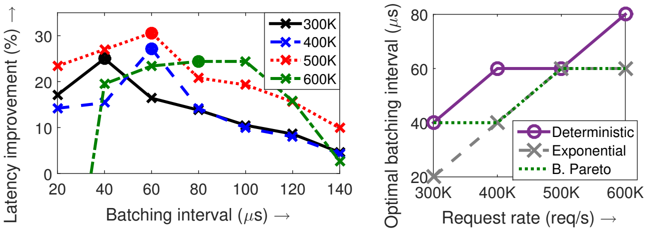
To reduce variability in socket-to-parse stage due to request batching, we need to consider the *source* of batching. Requests at the client are batched by the Linux kernel before being sent out so as to avoid the network processing overhead required for each request that is sent ("small-packet problem"). This batching is regulated by Nagle's algorithm [40]. While there are various conditions in the algorithm that trigger the sending of the next batch, we find that it is the condition of getting a response back from the server (kernel) saying it has received the previous batch that triggers the sending of a new batch in our experiments with Memcached. Since network conditions can be variable, the request batch sizes may be variable as well.

The solid lines in Figure 4 show our empirical observations for batch sizes under 600K request rate for different IAT distributions. We see that the batch sizes vary from 5 to about 50, with the distribution being more variable for Bounded Pareto IAT. This variance in batch sizes leads to instances where the application is overwhelmed by a large batch of requests, resulting in high tail latencies. Note that once the application finishes serving a large batch of requests and returns to the socket, multiple client-side batches of requests may be waiting in the socket to be drained, resulting in the application finding (server-side) batches as large as 50 requests.



(a) Deterministic IAT. (b) Exponential IAT. (c) Bounded Pareto IAT.

Figure 4: CDF of batch sizes for default and modified Nagle's algorithm under different IAT distributions for 600K req/s.



(a) Impact of batch size on request latency improvement under Deterministic IAT. (b) Optimal batch size for different IAT distributions.

Figure 5: Optimizing the batch size for modified Nagle's.

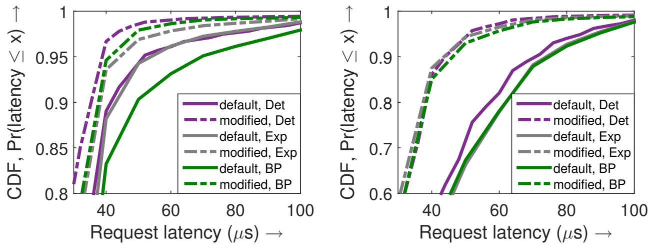
We *modify Nagle's algorithm on the client OS to force batches to be sent out at regular intervals.* This will result in less variable units of work received by the server, indirectly benefiting request latency. Rather than controlling the batch size, we instead control the time between batches, which is easier to implement. When a new request arrives at the client TCP, we append it to the existing batch and check whether the specified batching interval time has passed since the previous batch was sent. If yes, then we dispatch the current batch; else, we wait for the next request. This modification to Nagle's algorithm reduces the variability in batch sizes, as shown by the dashed lines in Figure 4 that represent the CDF of batch sizes after applying our modification and setting specific batching intervals for 600K req/s. For less variable IAT distributions, such as Deterministic, our modification eliminates almost all variability in batch sizes, as represented by the near-vertical CDF (dashed line) in Figure 4(a). Note that we still batch requests, and so preserve the Nagle's algorithm's functionality of preventing small packets from overwhelming the TCP/IP stack [40].

The "batching interval" length that we specify for sending out batches is a parameter that can be tuned. A large batching interval will result in numerous requests being sent simultaneously in a batch, overwhelming the server. On the other hand, a small interval can overwhelm the network stack with frequent packets. The batching interval must thus be carefully chosen, as we discuss next.

6.3 Evaluation results

Figure 5(a) shows the improvement in mean latency (over default Nagle's batching) for different batching intervals in our experiments under Deterministic IAT. We see that the batching interval does have a significant impact on latency. The optimal batching interval, indicated by the solid circles on each line, improves mean request latency by about 27%, on average, compared to the default Nagle's algorithm.

Figure 5(b) shows the optimal batching interval as a function of request rate for different IAT distributions. In general, the optimal batching interval length increases with request rate. An increase in batching interval leads to larger batch sizes being dispatched to the



(a) Request rate: 300K req/s.

(b) Request rate: 600K req/s.

Figure 6: CDF of request latency for default and modified Nagle's algorithm for different IAT distributions.

server, but also results in the server having longer to service each batch. With increasing load, it is likely that the server benefits more from having a longer duration of time to ensure that all requests in the batch are served before taking on a new batch. We find that, compared to the default Nagle's algorithm, our optimal batching leads to more frequent but smaller batches of requests being sent out at predictable intervals.

Figure 6 shows the CDF of request latency under the default and our modified Nagle's algorithm (using the optimal batching interval) for fixed request rates, but with different IAT distributions. Solid lines show the latency for default Nagle's and dashed lines show the latency for our modified algorithm. Given our focus on tail latencies, we intentionally show higher latencies in the CDF figures. For illustration, we show results for the smallest and largest request rate settings in our experiments; results are qualitatively similar for other request rates. We see that the latency reduction is higher for larger request rates. This is because the variable batch sizes overwhelm the server more under high load, and thus our modified batching algorithm has greater opportunity for improvement. The reduction in tail latency (80^{th}–99^{th} percentile) for all distributions ranges from 34-40% for 600K req/s, and from 14-35% for 300K req/s. The improvement in mean request latency likewise ranges from 24-26% for 600K req/s, and from 17-25% for 300K req/s. In general, the improvement is largest for Bounded Pareto IAT (which also has the largest IAT variability), followed by Exponential and Deterministic. This ordering is likely because of the relative variability in batch sizes (dictated by the IAT distribution) for the default case under these distributions.

To validate the efficacy of our chosen control knob in reducing variability, we analyze the socket-to-parse stage variability for the default and modified Nagle's algorithm in Figure 7, for Deterministic IAT; results are similar for other IAT distributions. We see that our modified batching algorithm maintains low variability throughout the request rate range, while the default Nagle's has much higher variability. The reduction in variability in Figure 7 ranges from 34-76%. We find a similar reduction in batch size variability as a function of request rate as well. Note that the reduction in stage variability in Figure 7 is correlated with the latency improvement in Figure 6.

Content popularity: The above results are for random content (key) popularity distribution. We also experiment with the Generalized Pareto distribution using similar parameters as reported by Facebook [3]. In this case, our modified batching reduces tail latency (80^{th}–99^{th} %ile) by about 20-24% and mean latency by about 17%.

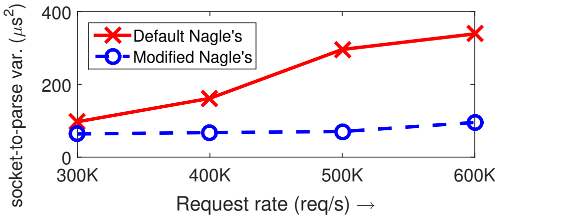


Figure 7: Variability of socket-to-parse stage under the default and modified Nagle's algorithm for Deterministic IAT.

Multiple clients: We also experiment with multiple clients, with each new client sending a fixed number of requests over a new connection. Note that only requests within a connection get batched. We vary the number of (sequential) clients per thread from 1 to 50, and the number of requests per client from 100 to 1000. Under our modified batching, the reduction in tail latency (80^{th}–99^{th} %ile) ranges from 13-21% and that for mean latency ranges from 15-17%, across all cases.

Trace-driven: Finally, we also experiment with dynamic trace-driven request rates (see Section 5), with the request rate ranging from 260K–610K req/s. Our modified batching reduces tail latency by about 26-39% and mean latency by about 14-20%, across all traces. The improvement is more pronounced for the ETC trace, likely because it has the highest mean request rate (500K req/s) among all traces.

Our focus on variability of service time in this section led us to request batching as the right control knob for Memcached under high throughput. We show, in Section 9, that request batching is *not* effective for the Apache web server; this is because the socket-to-parse stage variability for Apache is low (see Section 8). It is thus important to consider variability when deciding the right control knob.

7 EVALUATION 2: REDESIGNING THE LRU MANAGEMENT THREAD IN MEMCACHED

7.1 Application setup and tracing results

In the low throughput configuration, we run Memcached with different worker thread and core counts, starting with two worker threads on two cores of a socket. This configuration is representative of typical VM sizes requested by customers in public clouds [6]. We again employ our custom Memcached load generator (see Section 6.1) to vary the request rate (including via traces) and IAT distribution. Unless otherwise stated, we report results for the Deterministic IAT.

Figure 8 shows the results of our request tracing; note the much higher magnitude of variability values here (two core) when compared to the six core configuration in Figure 3. We see that the *tcp-to-socket* stage now has much higher variability compared to the other stages, and the socket-to-parse stage has much lower variability, in contrast to Figure 3. We do not employ the modified batching algorithm here so as to preserve the default application behavior and identify the most significant source of variability under the low throughput configuration. We evaluate the impact of modified batching for the low throughput configuration in Section 9.

The tcp-to-socket stage denotes the time between the TCP processing and the application picking up requests from the socket. Specifically, at the start of this stage, the TCP has enqueued the request(s) in socket, and the kernel now attempts to wake up/schedule

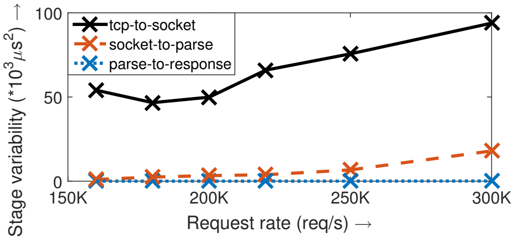


Figure 8: Variability at various request processing stages for Memcached under low throughput setting.

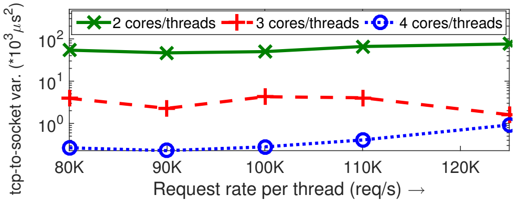


Figure 9: tcp-to-socket variability for different Memcached server configurations. Note the log scale on y-axis.

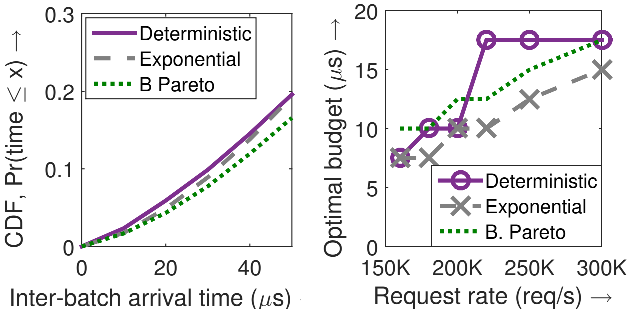
the application's worker thread on a core so it can pick up the enqueued requests from the socket. The high variability in this stage indicates that there is variability in the time it takes the application to be scheduled, suggesting that the application is either busy elsewhere or has been scheduled out for other processes. Possible sources of this variability include the application scheduling overhead and competing background threads, such as the dedicated Memcached LRU thread (see Section 5).

To precisely determine the most likely source of variability, we study the change in tcp-to-socket variability as we change the number of cores and threads assigned to Memcached, as shown in Figure 9. Note that the x-axis denotes the request rate *per thread*. We see that the tcp-to-socket variability goes *down* as the number of cores/Memcached threads increases, suggesting that the source of variability is mitigated by increasing the number of cores. This suggests that the dedicated LRU thread interfering with the Memcached worker threads is the likely cause for the increased tcp-to-socket variability. If the source of variability was instead the application scheduling overhead, then the tcp-to-socket variability should not have decreased with the number of threads (since we scale request rate with number of threads), which is contrary to the findings in Figure 9.

7.2 Creating a new control knob

The dedicated LRU thread is an application-specific function and does not expose any tunable knobs. While we can modify the application to do less LRU maintenance work, this would impact the application's functionality. Instead, we wish to preserve the LRU maintenance functionality while reducing variability by mitigating the interference between the LRU thread and the Memcached worker threads.

Our key idea is to *offload the LRU maintenance work to the Memcached worker threads and disable the dedicated LRU thread entirely* so it does not cause high variability for the tcp-to-socket stage. Specifically, at the end of each request processing phase and before being scheduled out, we leverage the worker thread itself do some controllable amount of LRU maintenance work. In other words, we *amortize* the LRU maintenance work over all worker thread invocations. We refer to our LRU maintenance approach as the *amortized LRU.*



(a) CDF of inter-batch arrival times at the server for different IAT distributions.

(b) Optimal budget for amortized LRU as a function of request rate.

Figure 10: Optimizing the budget for our amortized LRU.

While it may appear that the LRU maintenance work is now on the critical path, this is not entirely true. By scheduling LRU maintenance only *after* a batch of requests is processed, we are leveraging the idle time between batches to do LRU work. Of course, it is possible that a new batch of requests arrives before our LRU work is completed. To minimize this overstepping, we *regulate the amount of LRU work* done by each worker thread before being scheduled out. We refer to this controllable amount of work as the "budget" of our amortized LRU.

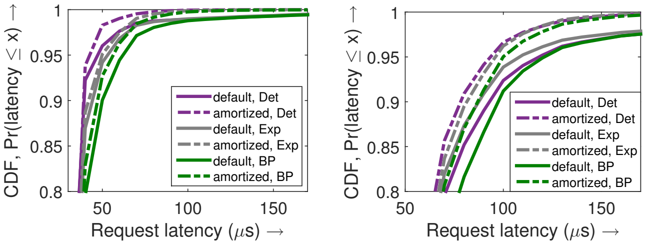
The budget for our amortized LRU must be carefully chosen so as to (i) ensure that the same amount of LRU maintenance work is being done as the dedicated LRU thread, and (ii) minimize the overlap with future request arrivals. While the former constraint implies a larger budget, the latter concern suggests a smaller budget.

7.3 Evaluation results

To determine the optimal budget for our amortized LRU, we analyze the idle time at the server between processing of successive request batches by the application. We timestamp two additional events– e7: When a batch of requests has finished processing (or event e5 for the last request in a batch, see Section 4), and e8: When a new batch of requests has completed its TCP processing and is ready to be serviced (or event e2 for a new batch). Figure 10(a) shows the CDF of inter-batch arrival times, ($T_{e8} - T_{e7}$), for 300K req/s request rate under different IAT distributions; we intentionally focus on a small region of the CDF. We see that almost 90% of the batches have an inter-batch arrival time of at least 30μs. The percentage is even higher for smaller request rates. This suggests that as long as the LRU work is scheduled for no more than 30μs, the negative impact on latency should be small. We thus consider budget values smaller than 30μs to investigate the benefits of amortizing LRU.

We find that the exact value of the budget does not significantly impact request latency. As a result, we seek the *smallest* budget that can still provide an equivalent amount of LRU maintenance as the default dedicated LRU thread. Figure 10(b) shows the optimal budget for different request rates under different IAT distributions. We see that the optimal budget increases with load, as expected, since the amount of LRU maintenance required for items (even in case of the default LRU thread) is proportional to the rate at which they are accessed.

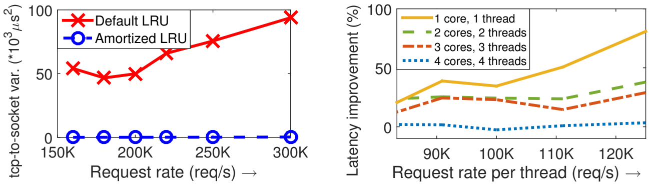
Figure 11 shows the CDF of request latency for the default LRU and our amortized LRU mechanism (using the optimal budget for each request rate) for different IAT distributions and for the smallest and largest request rate setting. Solid lines show the latency for default LRU and dashed lines show the latency for our amortized



(a) Request rate: 160K req/s.

(b) Request rate: 300K req/s.

Figure 11: CDF of request latency for the default and our amortized LRU mechanisms for different IAT distributions.



(a) Variability of tcp-to-socket stage under the default LRU and our amortized LRU.

(b) Improvement in mean latency under amortized LRU for different configurations.

Figure 12: Reduction in variability and sensitivity analysis.

LRU. The difference in latencies is more pronounced for the higher percentiles, and so we focus on this region in the figures. We see that the latency improvement afforded by our amortized LRU is slightly higher for larger request rates. This is because the default LRU thread interferes with more requests per second when the request rate is higher, and also because the LRU thread has to run for a longer time (to do more maintenance) when the request rate is high; thus, there is more opportunity for improvement at high request rates.

The reduction in tail latency (80^{th}–99^{th} percentile) for all IAT distributions ranges from 4-32% for 300K req/s, and from 6-31% for 160K req/s, with the improvement being slightly higher for Bounded Pareto IAT in general. The improvement in mean request latency across all experiments is about 28%. We also experiment with the Generalized Pareto content popularity distribution. In this case, our amortized LRU reduces tail latency by about 6-23% and mean latency by about 7%. Finally, we also experiment with request rate traces, with the request rate ranging from 90K-200K req/s. Under our amortized LRU, the reduction in tail and mean latency ranges from 7-42% and 22-31%, respectively, across all traces.

Figure 12(a) shows the variability of the target tcp-to-socket stage for the default and amortized LRU mechanisms, for Deterministic IAT. Our amortized LRU significantly reduces the stage variability, with the reduction being more prominent at higher request rates. This is in agreement with the latency improvement trends in Figure 11.

The above results are for the Memcached configuration of 2 cores and 2 threads. We now investigate the results under different configurations of cores and threads, since the impact of interference by the default LRU thread depends on these settings. Figure 12(b) shows the mean latency improvement under Deterministic IAT using our amortized LRU for n cores and n threads, where $n = 1,2,3,4$. The improvement decreases with the number of cores/worker threads; this is because the interference caused by the default LRU thread is shared among all cores, so higher the number of cores, lesser is the impact.

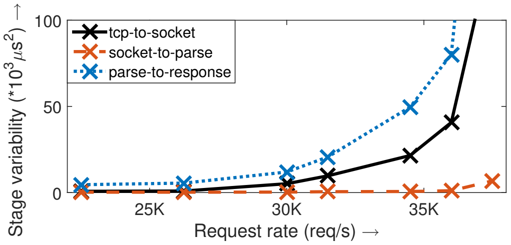


Figure 13: Variability at various stages for Apache web server.

In this section, we demonstrated how non-trivial control knobs can be developed to reduce variability in the critical stages of request processing. By only adding tens of lines of code to change LRU maintenance, we are able to reduce Memcached request latency by about 30%, a significant performance benefit. The evaluations in this and the previous section also highlight the need for request profiling, as different configurations of an application can have different sources of variability and thus different control knobs and solution strategies.

8 EVALUATION 3: APPLICATION THREAD PINNING FOR THE APACHE WEB SERVER

8.1 Application setup and tracing results

We use Apache v2.3.4 with mpm-event module configured to use 6 child processes, running on 6 CPU cores of a socket. We vary the number of threads per process from 5 to 25; unless otherwise stated, we use 25 threads/process. The server hosts a static web page, resulting in a response of 280 bytes. We use httperf [39] as our client driver to experiment with different request rates, including traces. We vary the load by changing the rate at which connections are opened; each connection issues 75 requests using Deterministic IAT. We report load in terms of the total request rate (across all connections).

Figure 13 shows the results of our request tracing for the three stages with the highest variability. We see that the parse-to-response and tcp-to-socket stages have the highest variability; further, the variability increases rapidly with request rate. The parse-to-response stage represents the time taken by the application to process the request, and the tcp-to-socket stage represents the application thread scheduling latency. The high variability in these stages suggests that the worker threads are experiencing unpredictable delays which is affecting their ability to complete the application processing on time (thus the high variability in parse-to-response), and is also delaying their return to pick up requests from the socket. Given that Apache does not have a dedicated maintenance thread like the LRU thread in Memcached, we instead focus on *application thread scheduling* as a possible source of variability. The increase in stage variability with request rate in Figure 13 strengthens our hypothesis since we expect scheduling overhead to increase with load.

8.2 Determining the control knob

By default, the Apache worker threads are not pinned to the cores, and can thus be migrated between cores by the OS scheduler. The OS scheduler (CFS, in our case) decides on thread migration based on core availability, cache placement, NUMA affinity, etc. While not pinning the threads has the advantage of leveraging idle cores to reduce mean service time, there is the potential overhead of context switching and associated state migration when migrating threads across cores. This overhead may hinder the progress of worker threads,

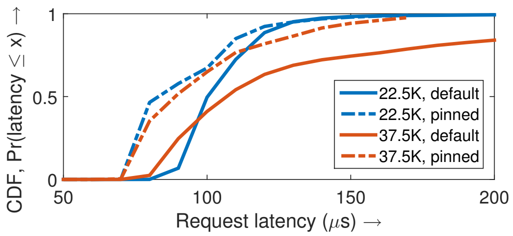
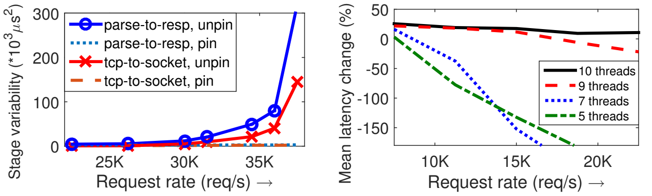


Figure 14: CDF of request latency when threads are unpinned (default) and when threads are pinned.

resulting in unpredictable delays. To reduce variability, we thus *pin the Apache application threads to cores* to prevent thread migration. Specifically, we pin all threads of a process on one core each; thus, each of the six cores handles threads from a specific process.

8.3 Evaluation results

Figure 14 shows the CDF of request latency under the default unpinned case and under the pinned case for the smallest and largest request rate settings in our experiments. We see that pinning threads significantly improves latency over the default case when threads are unpinned (i.e., can run on any of the 6 cores of the socket). The reduction in median and 90%ile latency for 37.5K request rate is 19% and 52%, respectively; the corresponding reduction for 22.5K request rate is 20% and 9%, respectively. The mean latency improvement varies from 15% at low request rates to 50% at high request rates. The higher improvement at high request rates stems from the larger potential in reducing variability (as discussed later). We also experiment with request rate traces, with the request rate ranging from 18K-37K req/s. Pinning threads improves tail latency by about 36-62% and mean latency by about 27-49%; the improvement is highest for the ETC trace.



(a) Variability of parse-to-response and tcp-to-socket under unpinned vs pinned.

(b) Change in mean latency under pinning for different number of threads/process.

Figure 15: Reduction in variability and sensitivity analysis.

Figure 15(a) shows the variability of the target parse-to-response and tcp-to-socket stages for the unpinned and pinned cases. We see that pinning threads significantly reduces the stage variabilities. This is because pinning reduces the context switching and thread migration penalty, resulting in more predictable delays for the worker threads. For example, pinning reduces the dTLB and L1-dcache load misses by about 87% and 22%, respectively, compared to the default unpinned case. The reduction is more pronounced at higher request rates. This is likely because the impact of context switching and thread migration penalty under the default unpinned case increases with load. Note the correlation between stage variability reduction in Figure 15(a) and improvement in latency in Figure 14.

The above results for Apache are for the high-throughput case with 25 worker threads/process. We now study the sensitivity of the configuration – number of threads/process – to the improvement in mean latency. Figure 15(b) shows our evaluation results as a function of request rate; we intentionally focus on lower number of threads. We clearly see that the thread configuration has a significant impact on the benefits of pinning over not pinning. For the case of 5 threads/process, pinning does not help for any of the request rates we experiment with, and can lead to a significant increase in latency; this is in contrast to the case of 25 threads/process where pinning is always beneficial. As we increase the number of worker threads, pinning starts outperforming the unpinned case for low request rates. The inflection point occurs at 10 threads, where pinning always helps. This is because the penalty of thread migration under unpinning is higher when there are more threads being scheduled in and out across cores, resulting in cache contents being constantly disrupted; pinning avoids this penalty.

The results in this section highlight the importance of analyzing the sensitivity of the control knob to the application configuration to understand *when* the control knob should be employed, since the benefits (50% latency reduction) and consequences (150% latency increase) of the choice can be significant, as shown in Figures 14 and 15(b). In the next section, we discuss the consequences of selecting the *wrong* control knob for a given application.

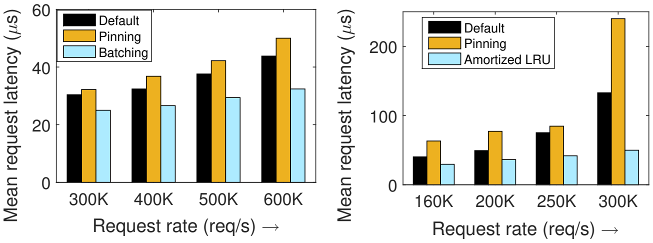
9 SIGNIFICANCE OF USING VARIABILITY IN SERVICE TIME AS GUIDING PRINCIPLE

The two key components of our approach, identifying the critical stage and determining the control knob, both rely on using "variability in service time (ST)" as the guiding principle. We now discuss the importance of this metric by comparing with alternative approaches.

Using variability to identify the critical stage:
A natural question to consider in our request profiling approach is – what if we employ "mean ST" as the metric when identifying the target stage instead of "variability of ST"? Consider the Memcached application under low throughput configuration, from Section 7. When employing mean ST as the metric for request profiling, we find that the socket-to-parse stage has the highest ST. Using the arguments from Section 6, we then employ batching as our control knob. However, the impact on request latency when using batching is inconsistent. For low request rates, the optimal batching does improve mean latency by about 28%. However, for higher request rates, even the optimal batching ends up *hurting* mean latency by as much as 32%. By contrast, when using variability of ST as the guiding metric, we consistently reduce mean latency by 26-61% by employing amortized LRU to mitigate variability in the tcp-to-socket stage. This shows that exclusively focusing on reducing mean ST may not always improve latency; a more robust approach is to also consider ST variability.

Using variability to determine the control knob:
Selecting control knobs in an ad-hoc manner is an alternative and sometimes easier approach. Thus, an obvious question is how important is it to select the right control knob?

Consider the Memcached application; an alternative control knob is to pin application threads. Figure 16 shows the resulting mean latency when using the incorrect knob (pinning) and our chosen control knob under Deterministic IAT. For the high throughput configuration, the incorrect knob *hurts* mean latency by about 11.5%, whereas our chosen control knob (batching) improves latency by about 20.9%. Likewise, for the low throughput configuration, the



(a) Memcached, high throughput config.

(b) Memcached, low throughput config.

Figure 16: Using the wrong control knob can hurt latency.

incorrect knob *hurts* latency by about 51.1%, whereas our chosen control knob (amortized LRU) improves latency by about 35.4%.

Similarly, for the Apache web server, another option is to batch requests by modifying the Nagle's algorithm, as in Section 6. However, even under the optimal batching interval for Apache, the improvement in mean request latency is only about 2.7%, whereas that under our chosen control knob (pinning) is about 21.1%.

10 RELATED WORK

Reducing the variability of end-to-end request latency

Dean et al. [7] analyze tail latencies in a production data center that supports their Google search system. The authors acknowledge that latency variability is a problem, and propose application-specific solutions, such as differentiating service classes and reducing head-of-line blocking. While this seminal work established the importance of variability and tail latency in production systems, it focuses on variability in request latency as a whole, and not on specific stages.

Li et al. [31] study sources of tail latency in interactive services by instrumenting applications and the kernel. While our request profiling approach is inspired by the authors' work, we focus on variability at individual stages of request processing whereas the above work focuses on end-to-end latency. Focusing on per-stage variability can reveal system design opportunities that may otherwise be overlooked. In particular, while the above work also studies the Memcached application, they do not uncover request batching or LRU maintenance as potential optimizations.

Jalaparti et al. [19] focus on the tail latency of Bing web search, and propose application-specific strategies such as reissuing stragglers and speeding up stragglers. Each of these strategies applies to a different "stage", where the concept of stage here refers to a step in the application's workflow and may span thousands of servers; this is in contrast to our definition of "stage" which refers to a fine-grained section of the request's processing path within a single server.

Reducing the latency variability for specific components

Hocko et al. [17] show that making the physical page allocation cache-aware reduces performance non-determinism without significantly increasing mean latency. Pusukuri et al. [30] show that changing the scheduling and migration policies to be aware of cache misses and context switches simultaneously reduces performance variation and improves performance. In addition, several techniques have been developed for promoting fairness in multi-threaded processor cores, shared caches [20, 26, 52], and memory controllers [10, 41], for improving performance by minimizing resource interference.

The above works focus on reducing variability in specific components (such as the OS scheduler), which are known a priori to be the performance bottlenecks. In our work, we profile the request processing path to first determine the potential bottlenecks. Thus, the above works can be employed as control knobs by our approach to reduce the variability at specific stages, *after* those stages have been identified as the potential source of variability.

Shortening the critical path of request processing

Kanev et al. [23] perform a detailed analysis of Google's data center jobs and find that CPU stalls and cache misses are the main causes of low core utilization. To improve utilization, the work calls for shortening the critical path of request processing by employing hardware specialization. Kapoor et al. [24] analyze data center applications and find that kernel overheads contribute significantly to request latency. The authors propose Chronos, a communication framework that bypasses the kernel by employing user-level, zero-copy network functionality, thus shortening the critical path.

The above works are classic examples of approaches that improve performance by reducing the mean ST (shortening the critical path). Our work presents a complementary approach that improves performance by reducing the *variability* of ST.

Request batching

SEDA [51] proposes a staged event-driven architecture for web applications with event batching and thread pool sizing. Yaksha [21] is a control-theoretic admission control proxy for internet services that prevents overload while maintaining high throughput. Elnikety et al. [11] propose black-box admission control for internet services by measuring execution costs and differentiating between request types. While the above works employ batching or admission control to avoid server overload, we precisely control when new requests start being processed by the server to minimize ST variability.

Thread pinning

Kumar et al. [28, 29] consider thread migration to exploit power-performance-area tradeoffs in multi-core machines. Li et al. [32] propose OS scheduling algorithms based on predicting the thread migration overheads. Wang et al. [50] study the interaction between various microarchitectural resources and thread characteristics to efficiently map threads. Thread migration has also been explored for changing thread-level parallelism [1] and for software data spreading [22]. Our work employs thread pinning to specifically reduce ST variability by avoiding state migration across cores.

11 CONCLUSION

Latency is a critical metric for user-facing web services. Existing work typically focuses on shortening the critical path (or mean service time) to improve performance. This paper takes an alternative approach to improving application performance - reducing the variability in request processing. By using "reduction in service time variability" as the guiding principle in web server design, we reveal control knobs that improve request latency (both mean and tail) by up to 28-50% across different scenarios.

Our end-goal is to make the case for using service time variability as a metric to guide system design. The three use cases presented in this paper demonstrate the validity of this approach.

ACKNOWLEDGMENT

This work was supported by NSF grants 1617046, 1717588, & 1750109.

REFERENCES

[1] Murali Annavaram, Ed Grochowski, and John Shen. 2005. Mitigating Amdahl's Law Through EPI Throttling. *SIGARCH Comput. Archit. News* 33, 2 (2005), 298–309.

[2] M. F. Arlitt and C. L. Williamson. 1997. Internet Web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking* 5, 5 (1997), 631–645.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. London, England, UK, 53–64.

[4] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. Melbourne, Australia, 267–280.

[5] S. Brin and L. Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proceedings of the 7th International World-Wide Web Conference*.

[6] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China, 153–167.

[7] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Communications of ACM* 56, 2 (2013), 74–80.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. Stevenson, Washington, USA, 205–220.

[9] Cuong Do. 2007. YouTube Scalability. (2007).

[10] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. Pittsburgh, PA, USA, 335–346.

[11] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. 2004. A Method for Transparent Admission Control and Request Scheduling in e-Commerce Web Sites. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. New York, NY, USA, 276–286.

[12] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux Journal* 2004, 124 (2004).

[13] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. 2007. Youtube Traffic Characterization: A View from the Edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC '07)*. San Diego, CA, USA, 15–28.

[14] Tabb Group. 2008. The Value of a Millisecond: Finding the Optimal Speed of a Trading Infrastructure. http://www.tabbgroup.com/PublicationDetail.aspx?PublicationID=346. (2008).

[15] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. *SIGPLAN Not.* 50, 4 (March 2015), 161–175. https://doi.org/10.1145/2775054.2694384

[16] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. 2003. Size-based Scheduling to Improve Web Performance. *ACM Transactions on Computer Systems* 21, 2 (2003), 207–233.

[17] Michal Hocko and Tomas Kalibera. 2010. Reducing Performance Non-determinism via Cache-aware Page Allocation Strategies. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10)*. San Jose, CA, USA, 223–234.

[18] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. Newport Beach, CA, USA, 199–212.

[19] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding Up Distributed Request-response Workflows. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 219–230. https://doi.org/10.1145/2534169.2486028

[20] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 208–219. https://doi.org/10.1145/1454115.1454145

[21] A. Kamra, V. Misra, and E. M. Nahum. 2004. Yaksha: a self-tuning controller for managing the performance of 3-tiered Web sites. In *Proceedings of the Twelfth IEEE International Workshop on Quality of Service*. Montreal, Canada, 47–56.

[22] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2010. Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. Toronto, Canada, 460–470.

[23] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 158–169. https://doi.org/10.1145/2872887.2750392

[24] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2391229.2391238

[25] T. Karagiannis, M. Molle, M. Faloutsos, and A. Broido. 2004. A nonstationary Poisson view of Internet traffic. In *Proceedings of IEEE INFOCOM 2004*, Vol. 3. 1558–1569.

[26] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. Antibes Juan-les-Pins, France, 111–122.

[27] Leonard Kleinrock. 1975. *Queueing Systems, Volume I: Theory*. Wiley-Interscience.

[28] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. 2003. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. San Diego, CA, USA, 81–92.

[29] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *SIGARCH Comput. Archit. News* 32, 2 (2004), 64–75.

[30] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2013. ADAPT: A Framework for Coscheduling Multithreaded Programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 45 (2013), 45:1–45:24 pages.

[31] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2670979.2670988

[32] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. Reno, NV, USA, 1–11.

[33] Zhenhua Liu, Minghong Lin, Adam Wierman, Steven H. Low, and Lachlan L.H. Andrew. 2011. Greening Geographical Load Balancing. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '11)*. San Jose, CA, USA, 233–244.

[34] Netcraft Ltd. 2018 (accessed July 30, 2018). *March 2018 Web Server Survey*. https://news.netcraft.com/archives/2018/03/27/march-2018-web-server-survey.html

[35] mediawiki.org. 2014. memcached. http://www.mediawiki.org/wiki/Memcached. (2014).

[36] David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. PowerNap: eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. Washington, DC, USA, 205–216.

[37] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. San Jose, CA, USA, 319–330.

[38] memcached 2018 (accessed July 30, 2018). *Memcached documentation new lru*. https://github.com/memcached/memcached/blob/master/doc/new_lru.txt

[39] David Mosberger and Tai Jin. 1998. httperf—A Tool for Measuring Web Server Performance. *ACM Sigmetrics: Performance Evaluation Review* 26 (1998), 31–37.

[40] John Nagle. 1984. Congestion Control in IP/TCP Internetworks. *SIGCOMM Computer Communication Review* 14, 4 (1984), 11–17.

[41] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. 2006. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. Orlando, FL, USA, 208–222.

[42] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398.

[43] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC '12)*. San Jose, CA, USA, Article 7.

[44] Eric Schurman and Jake Brutlag. 2009. The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search. http://velocityconf.com/velocity2009/public/schedule/detail/8523. (2009).

[45] The Apache Software Foundation. [n. d.]. Apache HTTP Server Project. https://httpd.apache.org. ([n. d.]).

[46] Twitter. [n. d.]. Twemcache: Twitter Memcached. ([n. d.]). https://github.com/twitter/twemcache.

[47] B. Urgaonkar and A. Chandra. 2005. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC '05)*. Seattle, WA, USA, 217–228.

[48] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. 2005. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*. Banff, Alberta, Canada, 291–302.

[49] Q. Wang, C. Lai, Y. Kanemasa, S. Zhang, and C. Pu. 2017. A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 207–217. https://doi.org/10.1109/ICDCS.2017.32

[50] W. Wang, T. Dey, J. Mars, L. Tang, J. W. Davidson, and M. L. Soffa. 2012. Performance analysis of thread mappings with a holistic view of the hardware resources. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems Software*. New Brunswick, NJ, USA, 156–167.

[51] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35, 5 (2001), 230–243.

[52] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. *SIGARCH Comput. Archit. News* 37, 3 (2009), 174–183.

[53] M. Zhang, N. S. Islam, S. Sur, H. Subramoni, D. K. Panda, J. Huang, M. Wasi ur Rahman, M. Luo, X. Ouyang, J. Jose, and H. Wang. 2011. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*. Taipei, Taiwan, 743–752.