Tales of the Tail:
Hardware, OS, and Application-level Sources of Tail Latency

Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble

Department of Computer Science & Engineering, University of Washington

{lijl,naveenks,drkp,gribble}@cs.washington.edu

Abstract

Interactive services often have large-scale parallel implementations. To deliver fast responses, the median and tail latencies of a service's components must be low. In this paper, we explore the hardware, OS, and application-level sources of poor tail latency in high throughput servers executing on multi-core machines.

We model these network services as a queuing system in order to establish the best-achievable latency distribution. Using fine-grained measurements of three different servers (a null RPC service, Memcached, and Nginx) on Linux, we then explore why these servers exhibit significantly worse tail latencies than queuing models alone predict. The underlying causes include interference from background processes, request re-ordering caused by poor scheduling or constrained concurrency models, suboptimal interrupt routing, CPU power saving mechanisms, and NUMA effects.

We systematically eliminate these factors and show that Memcached can achieve a median latency of 11 μs and a 99.9th percentile latency of 32 μs at 80% utilization on a four-core system. In comparison, a naïve deployment of Memcached at the same utilization on a single-core system has a median latency of 100 μs and a 99.9th percentile latency of 5 ms. Finally, we demonstrate that tradeoffs exist between throughput, energy, and tail latency.

Categories and Subject Descriptors   C.4 [*Performance of systems*]

General Terms   Design, Performance, Measurement

Keywords   Tail latency, predictable latency

1. Introduction

Networked services' response times vary substantially across requests. Even for a seemingly simple service, a small fraction of requests can exceed the median latency by orders of magnitude. This *tail latency* presents a challenge for designers, particularly in the case of large-scale, parallel, and interactive applications.

Tail latency is problematic for several reasons. Interactive services can struggle to provide complex functionality under the strict latency budgets required to ensure responsiveness. Under high degrees of parallelism, poor tail latency will impact most user requests. For example, a Facebook web request may access thousands of Memcached servers [15], and a Bing search may access 10,000 index servers [12]. A user's request does not complete until the slowest of these sub-requests has finished. The challenge for developers of individual services, then, is to build systems that have predictably low latency: the one-in-one-thousand case *is* the common case.

What causes some responses to take much longer than normal? Sometimes the answer is application-specific, but even applications specifically designed to have low median latency can have a substantial latency tail. In this paper, we show how hardware, operating system, and application-level design and configuration choices introduce latency variability. To do this, we study the behavior of three simple Linux servers executing on a multi-core computer: a null-RPC server, Memcached, and the Nginx web server.

We begin by using a classical queuing theory model to establish a baseline for the ideal response latency distribution for a particular service and request workload. Variable request inter-arrival times inherently cause tail latency, as bursts of requests that temporarily exceed the server's capacity introduce queuing delays. Queuing models predict that tail latency worsens with increased server utilization, but that it improves as additional processors service a queue.

Next, we measure the latency distributions achieved by our three Linux servers. Somewhat surprisingly, these distributions are substantially worse than predicted by a queuing model. Using fine-grained measurements taken at various levels of the system and stages of request processing, we

systematically identify and quantify the major sources of "excess" tail latency beyond that caused by workload bursts, including:

- interference from other processes, including background processes that run even on a system seemingly dedicated to a single server;

- request re-ordering caused by scheduling policies that are not designed with tail latency in mind;

- application-level design choices involving how transport connections are bound to processes or threads;

- multi-core issues such as how NIC interrupts and server processes are mapped to cores;

- and, CPU power saving mechanisms.

Guided by our measurements, we evaluate techniques for eliminating the excess tail latency exhibited by our servers. Many of these techniques are known in the folklore of systems engineering for their ability to improve scalability and throughput, or reduce latency [20]. Our contribution is to quantitatively evaluate their effects with respect to the "ideal" distribution implied by the queueing model.

These techniques are ultimately effective at explaining and mitigating tail latency. For example, with a default configuration of a four-core computer with a 10 Gb/s NIC, Memcached operating at 80% utilization has a median latency of 33 μs and a 99.9th percentile latency of 14 ms. With our techniques, we were able to improve this to a median latency of 11 μs and a 99.9th percentile latency of 32 μs, an improvement of two orders of magnitude at the tail that closely matches the queuing model's "ideal" distribution. Lastly, we observe that a tradeoff often exists between throughput and tail latency.

2. Queuing Models and Predicted Latency

What is the best possible tail latency achievable by a networked server? If we can understand the answer to this question, we can use it as an "ideal" baseline distribution to gauge how well a particular server implementation and configuration performs. To our knowledge, no previous work has attempted to characterize the ideal latency distribution of a network service.

One might expect this to be a trivial question: shouldn't the ideal latency distribution be a uniform one, where every request has the same response time? We demonstrate that this is unattainable for realistic workloads: there is a latency tail that is *inherent to the workload*. This tail is caused by the queuing delays that are introduced when a burst of requests temporarily exceeds the system's underlying request processing capacity.

To capture this effect, we model the service as a queuing system. Our model allows us to make several observations about the latency characteristics of real servers and workloads. First, even if we could build a server that processes requests in a fixed, deterministic time, there will still be a latency tail

for workloads that have variable request inter-arrival times. Second, the ideal latency distribution depends on the average utilization at which the server is driven. Systems that are run at high utilization have larger latency tails.

Third, adding additional processors to a system can reduce tail latency, even when the workload throughput is scaled up to maintain the same overall server utilization. Fourth, the choice of queuing discipline affects tail latency. FIFO scheduling provides the lowest tail latency, whereas other policies can achieve lower median latency at the cost of worse tail behavior.

Model. We model a server as a single-queue system, in which clients' requests are independent and arrive according to an arrival distribution. One or more workers (processors, threads, or processes) at the server retrieve and process requests from the queue according to a predetermined queuing discipline, such as FIFO. In Kendall's notation, this would be termed an $A/S/c$ queue, where A describes the arrival distribution, S is the service time distribution, and c is the number of independent workers. The average arrival rate (A) must be lower than the average service rate (S), otherwise queuing delays become infinite. We generally model a network service as having a fixed service time, derived from our measurements. For the applications we study, this uniform processing time distribution is appropriate: the fundamental amount of computation involved in processing a request is fixed.
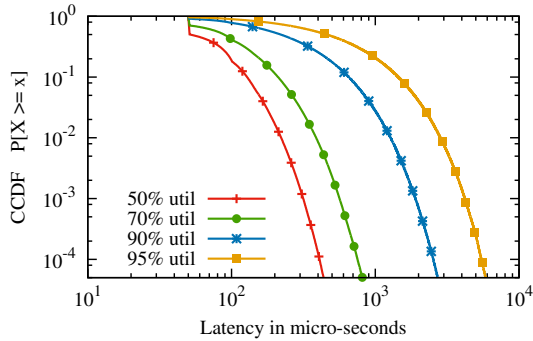
2.1 Arrival distributions

Figure 1 shows the response latency of a single-worker FIFO server with a uniform request processing time of 50 μs when operated at an average utilization of 70%. The graph is a complementary cumulative distribution function, and so the point (x, y) on the graph implies that y is the fraction of requests that experience a latency of at least x μs. This style of graph helps when visualizing latency tails, as Y-axis labels correspond to the 0th, 90th, 99th, 99.9th (and so on) percentile latency.

Each line on the figure corresponds to a different request inter-arrival time distribution. We have plotted several analytically defined arrival processes and the measured inter-arrival distribution of requests to Wikipedia [22], each scaled up to have an average request throughput corresponding to 70% server utilization.

Despite each request taking a deterministic amount of time, the latency tail (i.e., 99th percentile or 99.9th percentile) is high, because of random arrival bursts. For example, if two requests arrive within 50 μs of each other, the second request must be delayed until the first completes and the server becomes available. For each distribution, there is some probability of requests coming together in bursts sufficiently tight as to cause delay. The exception is a uniform arrival distribution, i.e. one where each new request arrives exactly
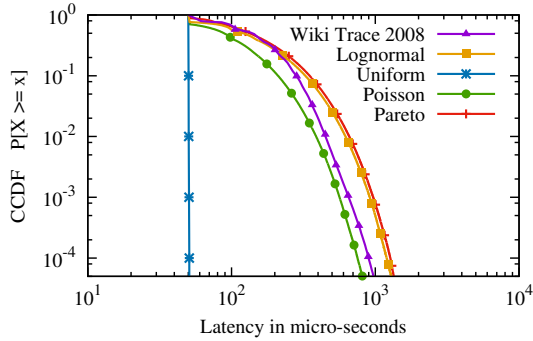
Figure 1. The latency tails exhibited by single-worker, uniform-service-time server, for different request arrival distributions.

Figure 2. The effect of increased utilization on tail latency, for a Poisson request inter-arrival time distribution.

70 μs after the last, which is decidedly unrealistic for network services.

2.2 Utilization

Next, we show how ideal latency distributions vary as we increase the server utilization. Increasing utilization reduces the leeway to handle bursts, which means that commonly occurring small bursts will build up large queues with long delays. Figure 2 shows this effect for a Poisson arrival process. Keeping the number of workers at 1, we increased the average request arrival rate, raising the server's utilization from 50% to 95%. The 99th percentile increases by 10x as we go from 50% utilization to 95%. This suggests a simple (if expensive) way to improve tail latency: running servers at low utilization.

2.3 Parallel servers feeding from one queue

Latency distributions also depend on degree of parallelism at the server. The ideal distribution improves as we change the number of workers (CPUs) at the server, even as we scale up the average request throughput to keep the server utilization constant. Figure 3 shows the effect of adding more workers for Poisson arrivals. Starting with 1 worker and a Poisson arrival process, we doubled the number of workers at each
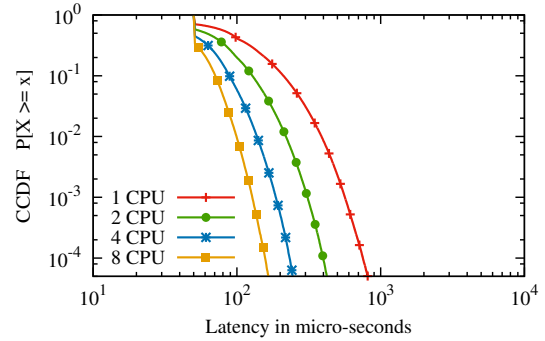
Figure 3. Tail latency improves as we increase the number of workers in a server, while scaling up the request arrival rate to maintain 70% utilization.

step and also doubled the arrival rate, keeping utilization fixed at 70%.

Increasing the number of workers leads to better tail latency. With *n* workers, up to *n* requests can arrive within the same 50 microsecond interval before any requests are queued. For example, the 99th percentile drops by 4x when running 8 workers instead of 1. However, an important caveat is that this improvement depends on all parallel workers pulling requests from a shared queue. If, instead, each worker has a separate queue, the resulting latency distribution would be the same as the single-worker case, as we are just instantiating multiple independent copies of a single-worker system. Throughput would improve, but the latency distribution would not.

2.4 Queuing discipline

The order in which requests are pulled from the queue has a significant impact on both median and tail latency. Figure 4 shows the latency distribution for 4 different queuing disciplines, for a server with 4 workers at 80% utilization and 50 μs of processing delay.

- **FIFO**: Requests are put in a single global FIFO queue at the server and workers pull requests from it. The earliest request is always processed next.

- **LIFO**: The opposite of FIFO; the most recently arriving request is processed next.

- **Random worker**: When requests arrive, they are assigned to a random worker. Each worker has its own FIFO queue.

- **Random request**: All requests arriving at the server are put in a single queue, and workers pull requests from the queue in random order. Each queued request is equally likely to be processed next, regardless of its arrival time.

FIFO queuing has the best tail latency; indeed, it is known to be optimal with respect to worst-case (i.e., tail) latency, as can be seen using a simple exchange argument. In comparison, LIFO has worse tail latency than FIFO, but better *median* latency (50 vs. 64 μs in our example). Both random disciplines have worse tail latency than FIFO, though
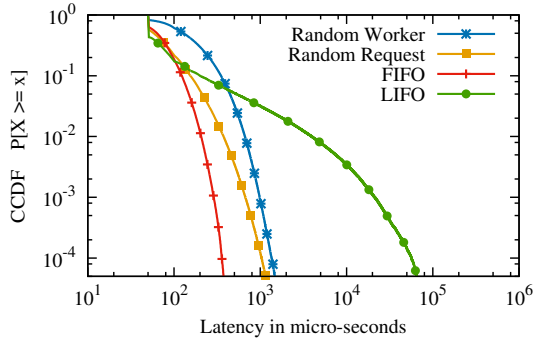
Figure 4. Latency distributions as we change the queuing discipline.

"random request" has better median latency as well, since some requests are able to jump the queue.

3. Measurement Method

In the previous section, we summarized known important results about the "ideal" latency distributions of simple queuing systems for different workloads. In the rest of this paper, we measure the latency tails of real servers and attempt to explain what factors lead them to diverge from the best case predicted by queuing models. Our overall approach is to identify a discrepancy, to analyze the system to explain and mitigate it, and to iterate through this process until we bring the measured latency distribution close to the ideal predicted distribution.

3.1 Applications

We examine three applications in our study; a null RPC server, Memcached, and Nginx. Each of these servers is designed to provide low latency response and high throughput. However, they have different concurrency models (e.g., threads vs. events) and they use different transport protocols (TCP vs. UDP). As we will see, these choices do impact the servers' measured latency tails.

3.1.1 Null RPC server

We begin with our simplest application: a null RPC server that we implemented in C. The server accepts TCP connections, reads 128 byte requests over each TCP connection, and echoes 128 byte responses back to the clients. We chose a classic multi-threaded architecture that somewhat resembles Apache: the server has a main accept thread that spawns new worker threads to handle each arriving client connection. The worker threads enter a blocking loop, reading requests using `read` and immediately writing a response back using `write`. Worker threads do not access any shared data structures or locks, and they do not use any other system calls.

For this server, request queuing and scheduling are managed by the OS. When a request arrives on an established TCP connection, the OS places the corresponding worker thread on a ready queue, eventually scheduling it on an avail-

able core. The order in which requests are processed therefore depends on how TCP data flows through the OS and the OS's thread scheduling policy.

3.1.2 Memcached

Memcached is a fast, flexible and lightweight key-value store, primarily used for accelerating dynamic web applications. Memcached is used in environments where low tail latency is important. For example, Facebook's applications may execute thousands of Memcached queries for each user request [15]. Accordingly, Memcached is specifically engineered to have predictable latency, storing all data in an in-memory hash table and using custom memory management with $O(1)$ operations.

Memcached supports both TCP and UDP connections. Memcached servers are often configured to have a number of threads proportional to number of cores in the system, to exploit parallelism. In UDP mode, worker threads simultaneously wait for messages on the UDP socket, retrieving messages from the kernel's receive queue in FIFO order. When running in TCP mode, all incoming TCP connections are statically partitioned among the Memcached worker threads. Processing a request generally requires a lookup into the hash table, acquiring one or more locks, and updating linked list pointers. As we show in our measurements, this application level processing takes only 1-2 μs with very little variance.

In our experiments, we use Memcached version 1.4.15 and store 64-byte keys with 1024-byte values. We generate a workload consisting of 90% reads and 10% writes.

3.1.3 Nginx

The Nginx web server is designed for high throughput and to scale to many cores. Unlike Apache, which uses threads or processes and blocking system calls to process requests, Nginx has an event-driven architecture, dividing HTTP processing into various stages and using non-blocking system calls to perform asynchronous disk and network I/O. To take advantage of multicore and multiprocessor systems, Nginx runs multiple worker threads (typically one per core), and statically assigns each client to a specific worker thread at connection establishment.

In our experiments, clients direct all HTTP requests to the same static file. Because of this, all file reads hit in the file system buffer cache, avoiding any latency variability introduced by storage device I/O. Tail latency effects due to disk I/O are beyond the scope of this paper. Clients issue 85-byte HTTP requests, and the server generates 849-byte responses (including HTTP headers and payload). These experiments use Nginx version 1.4.4.

Nginx worker threads check for completed I/O events using the `epoll` system call. The Nginx worker then iterates over the associated connections and performs the necessary HTTP processing. Importantly, epoll returns a list of file descriptors in the order they became ready, unlike previous

interfaces that returned an unordered list. This allows each Nginx worker to process HTTP requests in FIFO order.

3.2 Deriving the ideal distribution

Given an application, a workload, and a system configuration, how do we determine the ideal latency distribution? We accomplish this by determining a reasonable estimate for the amortized time the application takes to process a single request, by measuring the actual request arrival time series experienced by a server while engaged in an open-loop workload with clients, and then by feeding this time-series and estimated static request processing time into a queuing model.

For the CPU-bound workloads we study, we can estimate the amortized request processing time by running the server on a single core at 100% utilization and measuring the throughput achieved. Inverting the throughput number gives an estimate of the amortized latency of processing a single request. For example, in case of Memcached, we measured peak throughput at 125,000 requests per second per core. Hence, our amortized processing time is 8 μs per request.

In practice, request processing times will vary. For example, batching effects can reduce the processing time of some requests, cache misses can increase the processing time of some requests, and contention for shared locks can introduce delays under load. However, for the specific servers that we are studying, the request processing logic is simple enough that we have found per-request processing variability to be negligible relative to burst-induced queuing delays.

Once we obtain this amortized request processing time (t), we simulate an M/D/c queue as described in Section 2. In our simulation, we assume every request takes a deterministic time of t μs. The input request distribution for the simulator is the sequence of request arrival times measured during our experiment. This allows us to compare a measured latency distribution of an actual server to the latency distribution of an ideal server.

3.3 Testbed

We measure the performance of our three applications on a testbed consisting of Dell PowerEdge R610 servers, each with two Intel Xeon L5640 6 core (12 threads, 2.27 GHz) processors, running Ubuntu Linux 12.04 with kernel version 3.2.0. Each machine has 24GB of DRAM (1333 MHz) divided into two 12GB NUMA nodes. All servers are connected to a single Arista 7150S 24 port 10 Gbps switch using Mellanox ConnectX-3 adapters.

We use one of the servers to run the application under test, and five machines to run clients to generate the workload. The clients generate requests as an open-loop Poisson process [17]; open-loop workloads better represent large scale Internet-driven arrival processes, and they are known to induce higher degrees of burstiness and therefore can more significantly affect tail latency. We adjust the request rate to maintain a target CPU utilization on the server. Except as otherwise noted, for TCP-based workloads, we open all connections at the start of the experiment, outside the measurement interval, so that connection setup overheads do not affect our results.

3.3.1 Timestamping

To precisely understand and pinpoint the sources of latency variation, we need a fine-grained timestamping method to measure how much time a request spends in different parts of the server OS and application. We start timestamping when a request packet first arrives on the host from the server's NIC, and we end timestamping immediately before the OS transfers the response packet back to the NIC. We use the system clock with microsecond precision as the global reference of time for timestamping, and we disable NTP to avoid measurement errors arising from time updates.

To timestamp requests, we append an empty 30-byte buffer to the original request packet. As the request makes its way through various stages of server processing, we write the system clock time into the buffer. To implement this, we modified the Linux kernel source, network drivers, and application protocols to write timestamps into the appended buffer at the right offset. This method is attractive because it allows us to collect multiple timestamps with low overhead, and it avoids the need for the server to log requests: the response packet itself contains all the timestamps, so the clients can maintain the log.

For this study, we timestamp at each of the following events:

T_1: In the network driver, when the NIC has notified the system that a packet is available.

T_2: After TCP/UDP processing, but before the application is scheduled.

T_3: After the application thread has been scheduled onto a core.

T_4: After the application's read system call returns, i.e., after the request data has been copied to user space.

T_5: When the application makes a `write` system call to send the response.

T_6: In the network driver, when the response packet is sent to the NIC.

We report T_6-T_1 as the latency of a given request. The intermediate timestamps are diagnostic, used to identify the source of tail latency. T_1-T_2 is the network stack processing delay. T_2-T_3 is the queuing and wakeup delay. T_3-T_4 is the packet copy and return to user-level delay. T_4-T_5 is the user space application processing time. T_5-T_6 is the packet transmit delay.

This paper focuses on tail latency in network servers. Our experiments measure only the NIC-to-NIC processing delay on the server, and exclude any latency caused by the network fabric or client. The network can also be an important source of tail latency in datacenters, as has been studied extensively
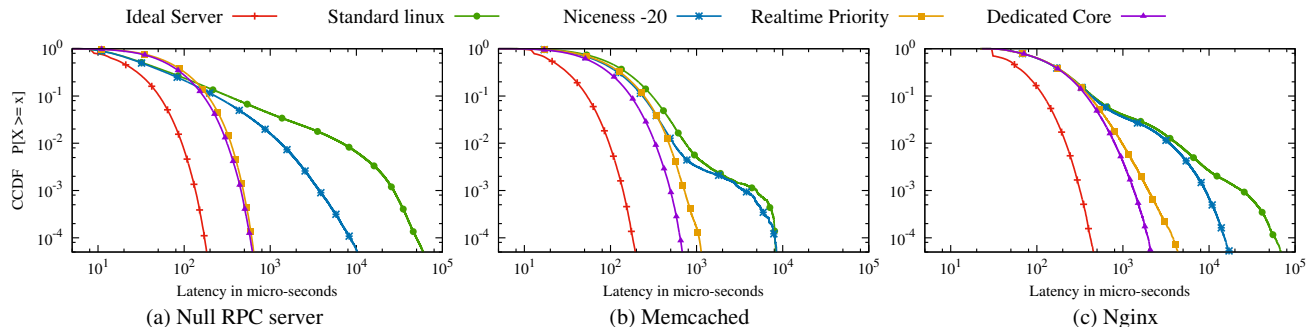
Figure 5. The effect of background processes on tail latency. For this experiment, we disabled all but a single core on a single CPU, and we loaded the server at 80% utilization.

by previous work [1, 23, 28], but this is outside the scope of our study.

Our measurements begin once the network driver is notified that a packet is available. It is possible that the NIC itself is a source of tail latency, so a more accurate measurement would use a timestamp taken in the hardware itself. We were not able to use such a measurement for most of our experiments, but Appendix A demonstrates that these effects are negligible.

4. Sources of Tail Latency

We now turn our attention to the measured latency behavior of our three servers and the factors that affect it. We first use the method previously described in Section 3.2 to derive the "ideal" latency distribution of each server. Next, we measure their actual latency distributions in a default configuration on Linux, showing that each performs significantly worse than its ideal.

We study these applications in increasingly complex configurations, beginning with a single-core system and later moving toward multi-core and multi-processor configurations. In the process, we identify the causes of deviation from the ideal latency distribution. By ameliorating each cause, we show that our server can achieve a latency tail that is close to ideal.

4.1 Background Processes

We start with the simplest configuration: a single CPU, single core system running a single server at a time. For this configuration, we disable all but one CPU core on our server machine, and we also disable HyperThreading. For each server application, we adjust the clients' workload to achieve a target server utilization of 80%.

In Figure 5, we plot the servers' ideal latency distributions and their actual measured latency distributions. Although the ideal and measured median latencies are similar (e.g., 29 μs for the measured null RPC server versus 21 μs for the ideal distribution), the measured 99th percentile latencies are 10–1000× the ideal latencies.

With only a single core available on the machine, the server application has to contend with other running background processes for the core. Although our test machine is not running any other servers or compute-intensive tasks, it still runs a standard complement of Linux daemons (e.g., sshd, NetworkManager) and cluster management software (e.g., Ganglia).

By default, all user level processes have the same priority. As a result, when the kernel schedules a background process, our application has to wait for the core to become available. The scheduler used by our Linux kernel assigns processes time-slices on the order of milliseconds, which explains why some requests in Figure 5 take more than a millisecond. There is a tail amplification effect that increases impact of these long delays: if the core is blocked by a background task for a long period of time, this greatly increases the latency of not just a single request but all requests that arrive in that interval.

The natural approach to mitigating this problem is to assign the server application a higher priority than the background tasks. However, Linux's normal priority mechanism (niceness) is not powerful enough to be effective for reducing tail latency. Figure 5 shows the effect of increasing the priority to its maximum normal value (niceness −20). This causes the server process to be scheduled for longer time-slices than other processes on the system, providing a slight improvement in tail latency, but still remaining far away from the ideal latency. The difference remains because the server cannot preempt other processes. It must wait for them to finish their time-slice, which is still much larger than an individual request's processing time.

Linux's realtime scheduler allows us to raise the priority of our server application strictly higher than all other processes. Making the server process a realtime process allows it to preempt any normal-priority process. This improves the tail latency dramatically, nearly eliminating the effects of background processes. For comparison, Figure 5 also shows a "Dedicated Core" line, where we move all other processes to a second CPU core, leaving the first entirely dedicated to the server application. The slight difference between the dedicated-core and realtime-priority configurations can
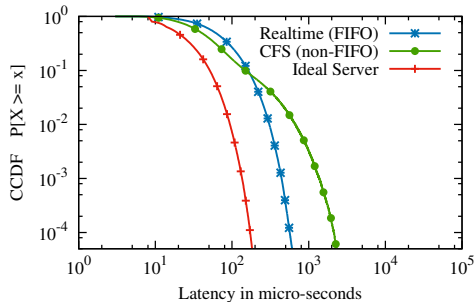
Figure 6. Linux CFS scheduler (non-FIFO) running at strictly higher priority versus realtime (FIFO) scheduler for our Null RPC server.

be attributed to increased context switching overhead and occasional cases where another running process is in a non-preemptable section of kernel code.

The realtime scheduler has a second effect. In addition to running processes at strictly higher priority, it also schedules threads of the same priority in FIFO order, which is not the case for normal-priority processes. We break down the impact of this change in the following section.

4.2 Non-FIFO Scheduling

As we established via theoretical analysis in Section 2, a FIFO scheduling discipline has better tail latency (but potentially higher median latency) than other policies. The default Linux scheduler, CFS (Completely Fair Scheduler [3]) favors fairness over order, resulting in a non-FIFO scheduling policy. This has a measurable effect on some of our experiments.

The null RPC server is the only one of our applications impacted by FIFO vs non-FIFO scheduling, because it is the only multithreaded application. The null RPC server relies on the kernel's scheduler to determine which thread to run next, and therefore which request to process first. The CFS scheduler chooses which thread to run based on how much CPU time each thread has received in the past, rather than which one became runnable earliest, so requests will not be processed in the order that they are received. This scheduling policy does not affect the two other applications, because they use event-driven architectures with only a single thread per core.

As a result, switching to the realtime scheduler has two effects on the null RPC server. It reduces interference from background processes, leading to lower tail latency, *and* it processes requests in FIFO order, which further reduces tail latency but also increases median latency. We separate these two effects by installing a custom scheduler into the Linux kernel, which chooses which of the null RPC server's threads to run using exactly the same policy as CFS, but gives them strictly higher priority over any other process on the system. We compare this non-FIFO scheduler to Linux's realtime scheduler, which has both strictly higher priority and FIFO

ordering, in Figure 6. The results are consistent with our theoretical analysis (Figure 4).
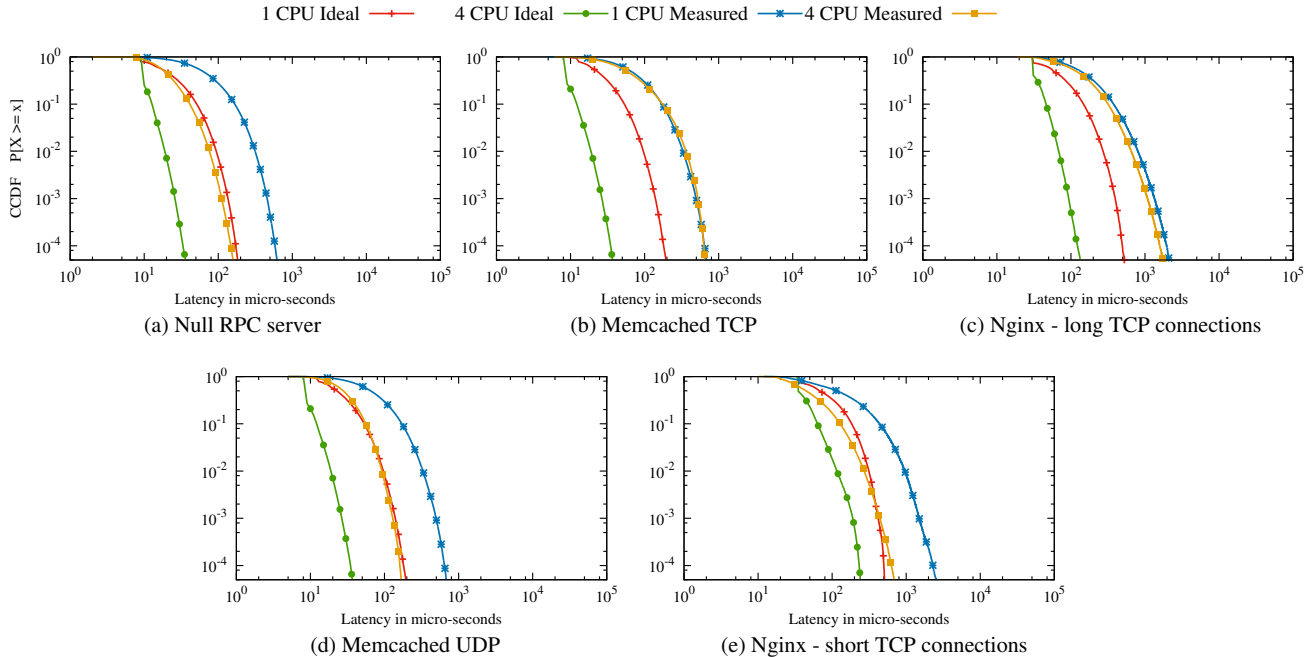
4.3 Multicore

Next, we move from a single-core configuration to a multi-core one. Our theoretical model predicts that increasing concurrency in this way will inherently reduce tail latency (Section 2.3). Does this effect occur in practice? To test this, we run our applications on four cores. For this experiment, we activated four cores on the same physical CPU to avoid (for the moment) NUMA and cache coherence effects. When scaling up the server to four cores, we also scale up the workload by a factor of 4 to maintain the same overall server utilization.

Figure 7(a-c) shows the ideal latency distribution for both the single core and multi-core setup, as well as the actual measurements. For null RPC server, moving to a multi-core server improves tail latency, as predicted by the theoretical model. The latency is better than even the ideal for a single-CPU server, although a gap still remains between the measured performance and the multi-CPU ideal. The results for Memcached and Nginx, however, remain essentially unchanged from the single-core case.

Recall the caveat from Section 2.3: in order for increasing the number of processors to benefit tail latency, the system must follow a single-queue model where any processor can process any request. This is the case for the null RPC server, but not Memcached and Nginx. The null RPC server resembles a single-queue system because CPUs pull threads from the pool of all runnable threads. However, both Memcached and Nginx statically assign incoming TCP connections to specific workers. Our experiments send multiple requests over a persistent HTTP connection to avoid the connection setup cost, so every request can only be handled by the worker thread responsible for its connection. Memcached and Nginx thus resemble a multi-queue system where each worker has its own request queue. Section 2.4 predicts that such a system will have the same latency distribution as a single-CPU setup, and our experiments bear this out.

One simple way for converting Memcached into a single-queue system is by switching the transport layer from TCP to UDP. As a result, all worker threads pull messages from a single UDP socket and this naturally follows the single-queue model as seen in Figure 7(d).

However this technique cannot be applied to Nginx, as it uses HTTP over TCP. The underlying issue here is that all TCP connections are created at the start of the experiment and closed at the end, so each individual request can be handled only by a specific worker process. Also, this is not a realistic workload for a websever facing the outside world. Hence, we tested Nginx with a slightly more complex workload in which open-loop clients with poisson arrivals create a TCP connection, send 20-40 GET requests and then close the connection. This way, each *client-arrival* can be assigned to any Nginx worker process and the system

(a) Null RPC server

(b) Memcached TCP

(c) Nginx - long TCP connections

(d) Memcached UDP

(e) Nginx - short TCP connections

Figure 7. Tail latency improvement as we scale to multiple cores. 4 cores at 80% utilization.

behaves approximately as a single-queue server. With this new workload, we do see improvement in latency as we move from single core to multiple cores.

4.4 Interrupt Processing

Even after negating the effects of background processes and non-FIFO scheduling, we still notice a gap of 2-3× between the ideal and measured 99th percentile latency for each application. Even though the server applications have strict priority over (and can preempt) background processes, requests are still delayed. Our analysis identified kernel interrupt processing as the cause. When packets arrive at the NIC, it interrupts a host CPU to initiate packet processing, triggering the kernel to receive the packet and process it through the TCP or UDP stack. By default, the "irqbalance" daemon is enabled by Linux. At a lower load, "irqbalance" operates in *power-save* mode and assigns interrupts to one centralized CPU core. However, as the load goes up, "irqbalance" switches to *performance* mode and spreads interrupts to all CPUs to keep them at similar utilization. Hence, our application threads were interrupted frequently by incoming packets on all cores.

This behavior causes the system to deviate from the ideal model in two ways. First, each request no longer takes a fixed amount of time to process: the interrupt introduces both context-switching overhead and cache pollution [19]. Second, processing is no longer done in a FIFO manner. Some part of a later request (network stack processing) takes place before the application-level processing of an earlier request is finished.

In order to fix these issues, we configured the system to dedicate a single core for interrupt processing and used the

remaining three cores to run application threads. This ensures that the application threads are not preempted by interrupts, and we maintain FIFO-ness in the whole system. Figure 8 shows the improvement we obtain by making this change. We are now very close to the ideal latency distribution and off by only a few microseconds. (Note that the ideal server line for Nginx in Figure 8(c) reflects the optimal latency for the multi-queue configuration, as discussed in Section 4.3.)

Employing this approach means that we must carefully balance the number of cores dedicated to interrupt processing and to the application. Otherwise, we might end up wasting resources because we do not end up fully saturating the dedicated interrupt core, and hence achieve lower throughput. For Memcached and our null RPC server, this ratio was roughly 1:3. A single core can process around 350,000 packet interrupts per second and each application thread can process 120,000 requests per second. For applications that perform more application-level processing, a higher ratio may be necessary. For large multicore systems, multiple cores may need to be dedicated to processing interrupts.

Looking to the future, we suspect that many server systems on large multicore systems will benefit from spatial allocation of cores to processes and threads, rather than the temporal multiplexing of multiple applications on each core. Over long time scales (minutes), the number of cores allocated to each application can change. Over short time scales (microseconds to seconds), this allocation would remain fixed, helping to prevent latency tail effects caused by interference and context switching between applications.
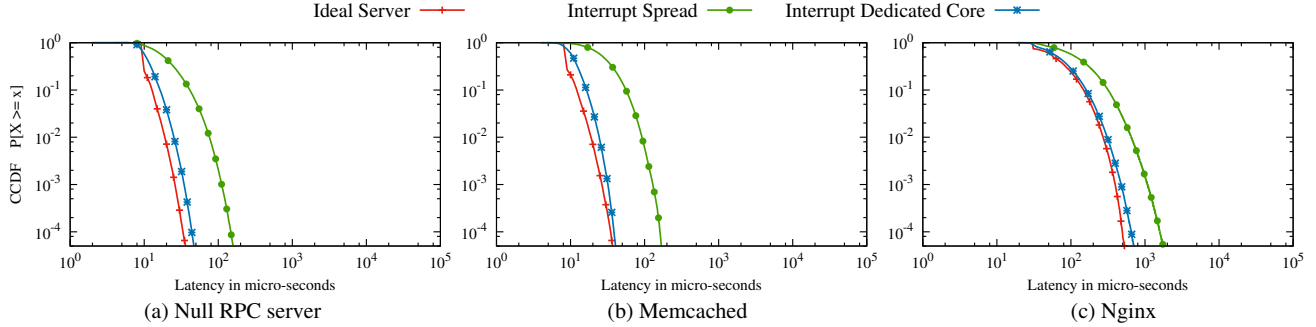
Figure 8. Impact of directing interrupts to a specific core versus spreading across all cores. In the first case, interrupts are spread across all cores, whereas in the second case interrupts go to a dedicated core and application threads run on remaining cores.
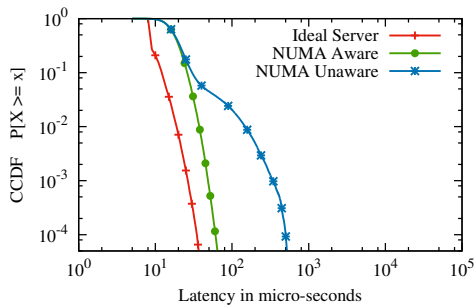
Figure 9. Impact of NUMA aware memory allocation in Memcached running on 8 cores (across NUMA nodes) at 80% utilization.

4.5 NUMA Effects

As we scale our system beyond multiple cores on a single CPU to multiple CPUs, new issues related to non-uniform memory access latency (NUMA) arise. We investigate these issues by running Memcached with 8 threads spread across the two processors in our system, with dedicated interrupt cores on each processor. Our theoretical model anticipates that this will improve tail latency because the number of workers has doubled. Figure 9 shows that the opposite is true: the tail latency is in fact *worse* than running on a single core.

Our investigation revealed that the cause of this increase is increased memory access latency. This problem is caused in part by Linux's default NUMA memory allocation policy. By default, Linux allocates memory from a single NUMA node to a process, until no more memory is available on that node. As a result, half of the Memcached threads must make cross-NUMA-node memory accesses, which have higher latency. We do not observe the same issue in our null RPC server or Nginx, because they are less memory intensive than Memcached.

As an alternative, we run two instances of Memcached, one on each processor. We force each instance to allocate only memory from the same NUMA node using numactl. Figure 9 shows the improvement we achieve.

4.6 Power Saving Optimizations

All of our previous experiments have considered high-utilization systems (80% CPU utilization). At lower utilization levels, hardware power saving optimizations come into play. Our server, like nearly all machines today, incorporates several CPU power saving optimizations, such as idle power states and frequency scaling. These optimizations save precious energy resources, but they inflate the tail latency at low utilizations. Indeed, they cause a counter-intuitive effect: while our theoretical model would predict lower tail latency for low-utilization systems, power saving mechanisms can cause these systems to have *higher* tail latency. We see this effect when running our servers at 10% utilization (Figure 10).

When CPUs become idle, they can be placed in an energy-saving state referred to as "C-state". There are multiple C-states, each causing more components of the CPU subsystems to be shut down. For example, on our Intel CPU, the C1 state stops the main internal clock, while the C3 state stops all internal and external clocks. Linux chooses which C-state to use based on CPU utilization and other factors. However, a higher C-state requires a longer wake up time, and the actual wake up time varies for different systems. Requests that are executed on a processor in power-saving mode will experience a higher latency than usual because they are delayed while the CPU is reactivated. Our testing machine has a wake up time of 200 microseconds for state C3 (the highest C-state in our machine), and we observe some requests delayed by this amount for all applications in Figure 10. By disabling power-saving states, forcing the CPU to stay in state C0 (running), we eliminate the highest-latency part of the tail, at the cost of increasing the CPU's power draw.

A second power conserving feature is dynamic frequency scaling, where the operating system dynamically changes the processor clock frequency. Frequency scaling has previously been identified as a source of tail latency [25]. When the CPU is mostly idle, the operating system reduces the clock frequency to save power. If a CPU-bound request executes
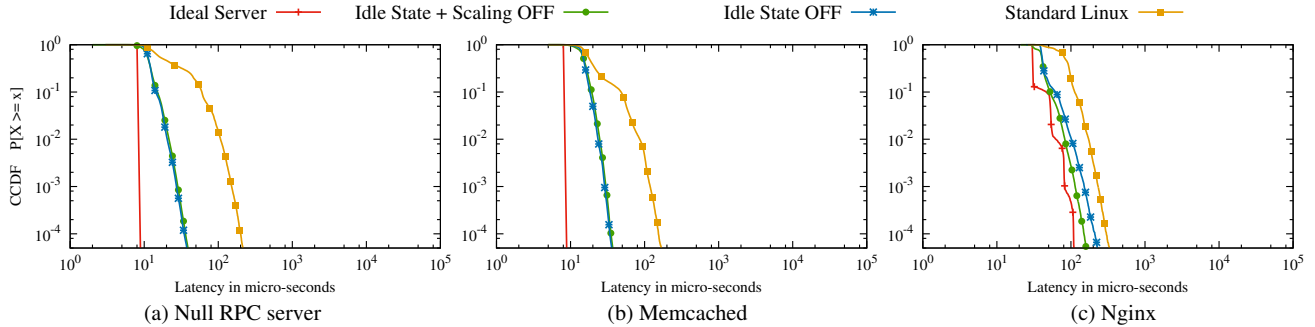
Figure 10. Impact of power saving optimizations on tail latency at low (10%) utilization.

while the CPU frequency is reduced, it can take longer time than usual. Figure 10 shows that this does not noticeably affect our null RPC server and Memcached, which require little computation for each request. However, Nginx, which is more CPU-intensive, shows a slight improvement when the CPU is forced to always run at maximum frequency.

4.7 Summary

We identified several ways in which hardware, operating system policies, and application-level design choices can impact measured tail latency. To summarize our observations and recommendations for mitigation:

- Interference from background processes has a large effect on tail latency. Linux's normal priority mechanism (niceness) isn't sufficient to prevent this, but realtime priority is effective.

- For multithreaded applications, a thread scheduler that maintains FIFO ordering reduces tail latency.

- The increased concurrency of a multicore system can help tail latency, but common concurrency architectures can negate this effect by requiring certain requests to be processed by specific workers.

- Dedicating certain cores to processing interrupts and others to application processes is beneficial.

- Poor placement of threads and memory on NUMA systems can lead to tail latency problems.

- At low utilization levels, there is a tradeoff between power saving and tail latency.

Table 1 summarizes the various sources of tail latency we observed and how they affect the latency distribution. We suggest possible mitigations and list the tradeoffs involved.

Our techniques for mitigation are effective. We are able to reduce the 99.9th percentile latency for each of the three applications we studied to within a few percent of optimal. In contrast, an untuned Linux system has 99.9th percentile latency that exceeds the optimal value by two to three *orders of magnitude*.

5. Related Work

Data center services are now routinely evaluated in terms of their 99th or 99.9th percentile latency. Dean and Barroso described their efforts to tame tail latency in Google's interactive applications; their goal is to allow parallel systems to tolerate latency variability in individual components, to "create a predictably responsive whole out of less-predictable parts [5]." The techniques they used include re-issuing slow requests to a different host [6, 30], issuing redundant requests across replicas to improve latency at the cost of lowered throughput [21], replicating data using quorum protocols that do not require every replica to answer [14], or accepting slightly incomplete results in information retrieval systems. More recent work takes an end-to-end view of all stages of a workload to choose which of these techniques to apply to the different stages of a request [12]. Related techniques have been used to avoid the "straggler" problem in data-parallel computation frameworks like MapReduce and Spark [2, 6, 9, 30, 31].

This body of work is complementary to our work on understanding and eliminating the sources of tail latency on an individual servers. Their approaches focus on higher-level techniques such as request replication in a distributed system or using pre-calculated statistics about job dependencies to dynamically adjust resource allocations, whereas we studied and ameliorated hardware, OS, and concurrency-model induced causes of tail latency on a single server node. Our efforts to improving the tail behavior of lower-level components can benefit higher-level complex distributed systems, but the techniques explored in others' work may still be necessary.

Chronos [13] also analyzed the sources of tail latency, concluding that the majority of tail latency was caused by kernel sources. They proposed avoiding this overhead with kernel-bypass network APIs and packet classification on the NIC. Our study takes a deeper look at the sources of tail latency within the kernel, and in contrast to their approach, we were able to achieve low median latency and good tail behavior using conventional network stacks.

In multi-tenant cluster environments, VMs for different tasks can compete for shared resources, causing severe latency variation. Bobtail [29] is a scheduler that places latency-

Source	Cause of deviation from ideal	Potential way to fix	Trade-offs involved
Background Processes	Scheduling delay caused by interference from background processes.	Raise priority to realtime or assign to a dedicated core.	Realtime priority may starve other tasks. Dedicated core may lower system utilization due to idleness when no requests are pending.
Non-FIFO Scheduling	Threads are scheduled out of order by Linux scheduler	Use a FIFO scheduler such as the realtime scheduler	Changing scheduler also affects priorities, etc.
Concurrency Architecture	Static partitioning of TCP connections effectively creates a queue per worker, violating the single-queue property.	Allow all threads to process all requests: use a UDP event-driven architecture or a thread-per-connection TCP architecture.	UDP loses reliability and congestion control properties of TCP; thread-per-connection architecture may reduce throughput.
Interrupt Processing	Increased processing time due to context switching and loss of FIFO ordering.	Dedicated core for interrupt processing.	Potentially lower throughput if interrupt core runs at low utilization.
NUMA Effects	Increased processing time due to memory accesses across NUMA nodes and cache coherency protocol.	Run an instance of Memcached per NUMA node with partitioned keyset.	Creates multiple queues, thus forfeiting some tail latency benefit. May cause load-balancing issues among instances.
Power Saving	Increased processing time required to wake up a CPU from idle state.	Turn off idle states and CPU frequency scaling.	Higher energy usage

Table 1. Summary of sources of tail latencies along with ways to mitigate them and the trade-offs involved.

sensitive and compute-intensive VMs on different hosts, and DeepDive [16] uses a more sophisticated approach to detect interference. Our work has some similarity, in that we have found it beneficial to pin the threads or processes of servers on separate cores in order to isolate them from interference caused by background processes.

The datacenter network can also be a source of tail latency because of long queuing delays [1]. Deadline-aware congestion control mechanism [23, 28] can help alleviate this problem.

Real-time and time-sensitive OSs address related problems [11]. Similar to us, their goal is to respond to incoming data within a predictable latency bound. To accomplish this, these systems often need to perform admission control to avoid overcommitting resources. As well, they need carefully designed schedulers to bound worst-case scheduling delays, and they must avoid long critical sections to prevent head-of-line blocking and priority inversion. Typically, these systems do not attempt to provide high throughput. In contrast, we are focused on systems that run at relatively high utilization. We recognize that some latency jitter or variation is inevitable due to bursting arrivals, and our goal is to identify and eliminate other sources of latency variation in the system.

The trade-off between energy and performance has been explored in several previous systems. In DRAS, the authors explored policies for load balancing requests across servers and deciding when to power idle servers off, assuming computationally intensive workloads [10], demonstrating that a reduction in average power can be achieved with a slight penalty in average latency. Similar observations were made for the Salsa web server [8], though they also demonstrated that request batching could further save energy. More recently,

Powertail demonstrated how to distribute load across servers that support DVFS in such a way as to minimize energy while guaranteeing a specified 99.9th percentile latency SLA [26]: the load of individual servers should be driven up to a critical utilization before spilling load to additional servers. These projects' results mirror our measurements of the tradeoff between tail latency and CPU idle states on a single host, and their proposed mechanisms for sculpting load balancing policies to selectively enable hosts within a cluster should be applicable in the context of routing requests to cores on a single host.

Prior work has also explored the relationship between the operating system, the network stack, and request latencies. In IsoStack [18], the authors show the benefit of offloading network stack processing onto a dedicated core: the improved cache behavior and reduced number of processor context switches leads to higher throughput at lower utilization, which presumably also can improve latency. LRP [7] demonstrated the importance of queue management, the early drop of load under high utilization, and performing network stack work at the priority of the receiving application; these techniques avoid livelock and improve latency under load. Our work is complementary, in that we quantify the degree to which similar mechanisms (such as using a dedicated core for interrupt handling) has benefits for tail latency in modern multicore systems.

6. Discussion

Our study reaffirmed the common belief that tail latency is highly sensitive to background processes and daemons. This suggests that we explore other ways of sharing CPU resources, particularly in shared environments where we

would like to co-locate multiple latency-sensitive services on the same host. Spatial scheduling – partitioning CPU cores among applications – seems an attractive alternative to conventional time-sharing. Others have proposed it as a way to manage forthcoming large multicore systems [27], and we believe that predictable latency provides another strong motivation. We have already seen an example of its benefits, in that dedicating certain cores to interrupt processing can reduce tail latency.

There has been a longstanding debate in the systems community over the superiority of threads or events for managing concurrency. Prior work has dissected their relative merits in terms of performance, scalability, and programmability [4, 24]. Our study adds a new angle to this old debate: certain architectural choices affect the tail latency of a network server. Thread-based architectures require a thread scheduler that ensures FIFO ordering to achieve optimal tail latency (and such a scheduler is not used by default in Linux). Event-driven architectures must ensure that any request can be handled by any worker; otherwise, as with Nginx, they will negate the benefits of parallelism for reducing tail latency.

7. Conclusion

This paper explored hardware, OS, and application-level causes of tail latency in multi-core servers. Known results from queuing theory explain why the natural burstiness in request arrival processes will introduce an unavoidable baseline of variable queuing delay. As well, the theory shows how increased utilization worsens the latency tail, how parallelism can improve the latency tail, and the importance of choosing an appropriate queuing discipline.

There are many additional complicating factors that affect servers' latency tails. To understand them, we instrumented three applications: a multithreaded null RPC server, Memcached, and the Nginx web server. Next, we measured these servers' response time distributions, demonstrating that their latency tails are significantly worse than what queuing models would predict. Our instrumentation helped us to identify the causes of the tail inflation, including interference from background processes, request re-ordering by the OS scheduler or application concurrency framework, poor interrupt routing, CPU power saving mechanisms, and NUMA effects.

We implemented several mechanisms and configuration changes to fix these problems. To isolate the server from background processes, we either use real-time scheduling priorities or we isolate server threads/processes on dedicated cores. To remedy non-FIFO OS scheduling, we modified Linux's scheduler. We showed that dedicating a core to interrupt processing improves tail latency at a potential cost to maximum throughput, and we showed that under low utilization, CPU power savings mechanisms hurt tail latency. Finally, to combat NUMA issues, we pin server processes or threads to cores and force them to allocate memory from the same NUMA node.
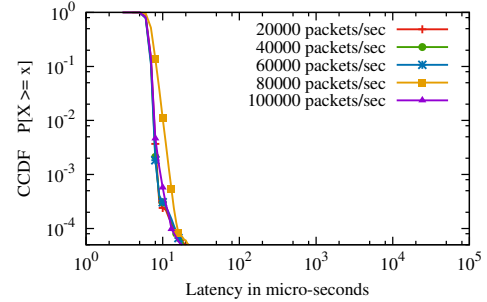
Figure 11. Network fabric latency distribution as measured using switch-generated timestamps.

Our modifications substantially improved the tail latency of all three servers, to the point where they behave close to the ideal distributions predicted by queuing models. For example, we improved the 99.9th percentile latency of Memcached at 75% utilization from 5 ms to 32 μs, an improvement of more than two orders of magnitude.

A. NIC-Induced Latency

As mentioned in Section 3, our latency measurements begin and end in the network driver, when an incoming packet arrival notification is received and when an outgoing packet is placed in the NIC's send queue respectively. This excludes any tail latency caused by the network hardware itself. Here, we quantify this latency and demonstrate that it is negligible relative to the other factors we studied.

Our Arista 7150 switch can timestamp packets at nanosecond granularity. While we were not able to use this functionality in most of our experiments because of the complexity of synchronizing the switch and host timestamps, we use it here to demonstrate that network fabric delay is consistent. We collect additional timestamps T_0 and T_7 when request and response packets traverse the switch. In this experiment, we use the Memcached workload at various request rates. Figure 11 shows the distribution of $T_7 - T_0 - (T_6 - T_1)$, the difference between our switch-measured and software-measured latency, i.e., the latency introduced by network hardware. Because the switch takes timestamps on packet ingress, this includes any transmission delay and interrupt latency on the request packet as well as NIC queuing delay in both directions. As the figure demonstrates, the latency variance is substantially lower than any of the other factors we studied, including workload-inherent queuing delay.

Acknowledgements

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. DCTCP: Efficient packet transport for the commoditized data center. In *Proceedings of ACM SIGCOMM 2010*, New Delhi, India, Aug. 2010.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, Oct. 2010.

[3] CFS. Completely fair scheduler. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`.

[4] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, St. Emilion, France, Sept. 2002.

[5] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, USA, Dec. 2004.

[7] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Oct. 1996.

[8] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *Proceedings of USITS 03: 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, Mar. 2003.

[9] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM SIGOPS EuroSys (EuroSys '12)*, Bern, Switzerland, Apr. 2012.

[10] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Distributed, robust auto-scaling policies for power management in compute intensive server farms. In *Proceedings of the Sixth Open Cirrus Summit (OCS '11)*, Atlanta, Georgia, Oct. 2011.

[11] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on a commodity OS. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, USA, Dec. 2002.

[12] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013.

[13] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the 3rd Symposium on Cloud Computing (SOCC '12)*, San Jose, CA, USA, Oct. 2012.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[15] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2013.

[16] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Annual Technical Conference*, San Jose, CA, USA, June 2013.

[17] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.

[18] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack – highly efficient network processing on dedicated cores. In *Proceedings of the 2010 USENIX Annual Technical Conference*, Boston, MA, June 2010.

[19] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, Oct. 2010.

[20] SolarFlare. Filling the pipe: A guide to optimising memcache performance on SolarFlare hardware. `http://goo.gl/FqwtkN`.

[21] C. Stewart, A. Chakrabarti, and R. Grifth. Zoolander: Efficiently meeting very strict, low-latency SLOs. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA, June 2013.

[22] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. `http://www.globule.org/publi/WWADH_comnet2009.html`.

[23] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-aware datacenter TCP (D^2TCP). In *Proceedings of ACM SIGCOMM 2012*, Helsinki, Finland, Aug. 2012.

[24] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS '03)*, Lihue, HI, USA, May 2003.

[25] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS '13)*, Philadelphia, PA, USA, June 2013.

[26] S. Wang, W. Munawar, J. Liu, J.-J. Chen, and X. Liu. Power-saving design for server farms with response time percentile guarantees. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012)*, Beijing, China, Apr. 2012.

[27] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *Operating Systems Review*, 42(2):76–85, Apr. 2009.

[28] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of ACM SIGCOMM 2011*, Toronto, ON, Canada, Aug. 2011.

[29] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, USA, Apr. 2013.

[30] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, USA, Dec. 2008.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mc-Cauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, USA, Apr. 2012.