MittCPU:
Circumventing Millisecond Tail Latency Induced by CPU Contentions in the Cloud

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER’S

DEPARTMENT OF COMPUTER SCIENCE

BY
Meng Wang

CHICAGO, ILLINOIS
2020
Intentionally left blank.
Table of Contents

LIST OF FIGURES ................................................................. V
LIST OF TABLES ................................................................. VI
ABSTRACT ........................................................................ VII
1 INTRODUCTION ................................................................. 1
  1.1 INTRODUCTION .............................................................. 1
2 MITTCPU DESIGN .............................................................. 4
  2.1 MITTCPU Design Overview ............................................ 4
  2.2 Cancellation Mechanism ................................................ 6
     2.2.1 Prompt cancellation ................................................ 6
     2.2.2 Correct request removal ......................................... 7
  2.3 CPU Delay Prediction .................................................... 8
     2.3.1 CFS scheduler ....................................................... 8
     2.3.2 MITTCPU Prediction: from Naive to Complete .......... 10
3 IMPLEMENTATION ............................................................. 14
4 EVALUATION ................................................................. 15
  4.1 Experiment setup ........................................................ 15
  4.2 Performance Evaluation ............................................... 16
     4.2.1 Effect of MITTCPU (3 nodes, dummy CPU-intensive threads) 16
     4.2.2 MITTCPU vs. other techniques
          (6 nodes, Representative CPU benchmarks) ............... 17
     4.2.3 Precision Accuracy ............................................... 21
5 LIMITATIONS & FUTURE WORK .............................................. 23
6 RELATED WORK ............................................................. 25
7 CONCLUSION ................................................................. 27
## List of Figures

1.1 **Tail latency.** *The figure, in CCDF format, shows tail latencies due to CPU contention*  

2.1 **MittCPU Principle Illustration.** *The figure shows how MittCPU works when a request is predicted to be unable to finish within deadline.*  

4.1 **Tail latency.** *The figure shows CCDF graphs of the client-perceived latency distribution with CPU noises. In each graph, we show the “Best” (no contention), MittCPU, and “Raw” (no mitigation) lines.*  

4.2 **MittCPU vs. other techniques.** *The latency CCDF (with \( \log_2 \)-scale y-axis) compares MittCPU versus other popular techniques (please see §4.2.2 for explanation).*  

4.3 **MittCPU prediction precision.** *The figures show CDFs (0.0 to 1.0 in y-axis) of CPU delay prediction errors (\( D_{err} \) in ms in x-axis) across CPU benchmark mixes.*
List of Tables

4.1 Average latencies. The table shows the average latencies of the latency distributions in Figure 4.2. ................................................................. 18
4.2 Benchmark mixes. Benchmark mixes for measuring MittCPU’ prediction accuracy. . . . 21
Abstract

It’s important for distributed storage systems to provide stable and low-latency services, since a slow response can be detrimental to the whole user experience. However, in modern storage deployments, resource sharing is a defacto configuration, and the resulting resource contention can produce long tail latencies for the services.

To address this problem, we present MittCPU which provides operating system support to mitigate millisecond-level tail latencies caused by CPU contentions in the cloud. In MittCPU, we allow users to specify SLOs (e.g. deadline) for their requests. When requests arrive at the server side, we accurately predict request delay based on white-box knowledge of server-side CPU scheduling policies. If the SLO cannot be met, we promptly reject the request such that clients can quickly failover to another less-busy node without waiting.

We have implemented MittCPU rejection in the Linux OS TCP stack, and implemented an accurate request delay predictor based on Linux’ CPU scheduling algorithm. MittCPU exposes simple APIs for applications and we successfully integrated it into Cassandra and MongoDB. Our evaluations show that compared to speculative retry, a popular tail-tolerant technique, MittCPU can reduce the completion time of user requests by 31-66% at 90P and 35-70% at 95P.
Chapter 1

Introduction

1.1 Introduction

It’s important for distributed storage systems to provide stable and low-latency services, since a slow response can be detrimental to the user experience. However, nowadays many data center applications are suffering from the well-known “tail latency problem” [12]: their 99.9 percentile request latency are usually orders of magnitude worse than the median. Tail latency can be a critical problem to modern distributed systems, as in these systems one job usually consists of multiple parallel tasks. In that case, one slow task can cause an unacceptable delay to the whole job.

One dominant factor of tail latency is resource contention. In modern storage deployments, resource sharing is a defacto configuration, and the resulting resource contention can produce long tail latencies for the services. This include IO contentions in disks and cache[17, 26, 8, 22], process/VM contentions in CPUs [14, 18, 19, 20, 24, 25, 30];, and garbage collection in memory [9, 15, 16, 21, 23, 27].

Let’s take CPU contention as an example. Consider a distributed key-value store server which shares CPUs with other applications on the same machine. When a new request comes, in order to process the incoming request, the server application first needs to get a CPU to run on. However, due to the contention from other running processes, the target server application may not get the
CPU immediately. In that case, the request needs to wait and thus lead to unacceptable tail latency.

Figure 1.1 illustrates the impact of CPU contention on tail latency. It shows the latency CCDF (complementary/reverse CDF) of Cassandra requests when the server process is competing over the same CPUs with other processes. The “+Noise” line shows that around 15% of the requests are experiencing tail latencies (vs. the stable “Best” line without CPU noises).

Many approaches have been proposed during the last decade to address the tail latency problem, but they all have some limitations. One existing approach is redundancy [28], where they clone every request and send it to multiple replica servers, and they use the first one that completes. This technique can effectively reduce tail latency, but it doubles system utilization. A more conservative alternative approach is speculative retry [7, 13, 32], where the duplicate request is sent only after the first request is outstanding for more than a certain amount of time (a deadline). However, slow requests need to wait before being retried, and the wait time cannot be ignored. This approach causes extra load as well: speculative retry after waiting for the Pth-percentile latency will lead to (100-P)% backup requests [12].

One another approach is Replica Selection [10, 29, 31]. It predicts which replica can serve the requests faster, and direct requests to that replica. The prediction is usually based on the historical latency data monitored at the client side, and is refreshed sparsely (often every few minutes). Therefore, it’s unable to reactively handle bursy requests which appear and disappear in second-
level intervals.

None of the techniques discussed above uses the information of underlying resource busyness. However, the OS layer does have such information, though it may be unexposed. If we can make use of the resource busyness information, we will be able to better address the tail latency problem.

Therefore, in this paper we propose MittCPU: a fast-rejecting approach which addresses tails based on white-box knowledge of server-side CPU scheduling policies. In MittCPU, we allow clients to specify SLOs (e.g. deadlines) for their requests. When requests arrive at the server side, we accurately predict request latency based on white-box knowledge of server-side CPU scheduling policies. If SLO’s cannot be met, we promptly return an EBUSY rejection to the client. Then client can then quickly failover to another less-busy node without waiting.

We have implemented this fast-rejecting interface on a Linux machine. We have also designed an accurate predictor of request latency based on Linux’s CPU scheduling policy. These capabilities are written in 2,300 LOC in the Linux kernel. MittCPU exposed simple APIs for applications to benefit from our approach, e.g. we modified Cassandra and MongoDB only in 70 and 50 lines respectively. Our evaluations show that compared to speculative retry, a popular tail-tolerant technique, MittCPU can reduce the completion time of user requests by 31-66% at 90P and 35-70% at 95P.
Chapter 2

MittCPU Design

2.1 MittCPU Design Overview

We now present MittCPU design.

MittCPU is a Linux extension with request cancellation and prediction capabilities, focusing on CPU delays. Due to CPU contention, an arriving request might not be directly processable by the server application until the OS allocates a CPU for the application. In that case, the request will be delayed and thus generates long tail latency. MittCPU is designed to deal with this kind of tail latency.

When circumventing CPU contention by prediction and cancellation, the challenge is how to detect that a request is delayed in serving the arriving request while the application itself needs the CPU. We cannot do it in the application layer because the application first needs to get CPU in order to send the rejection. Before the application gets the CPU, the CPU contention may already have happened and the request may already have been delayed. Fortunately, today’s latency-sensitive services form a client-server communication where requests pass through the OS, hence allowing MittCPU to carry the burden of prediction and cancellation.

The mechanism of MittCPU can be illustrated by Figure 2.1. MittCPU adds CPU delay prediction within the TCP stack, before the arriving request is delivered to the application’s buffer.
If MittCPU predicts that the request’s remaining deadline cannot be met given the current level of CPU contention, MittCPU automatically sends a cancellation notification to the sender of the request (via the same socket) and remove the request from the receiving queue.

There are two main challenges in building MittCPU:

1. how to cancel the request promptly and properly
2. how to predict the request latency accurately

We implemented both these two capabilities in Linux kernel, in 750 LOC for prompt request cancellation and 1550 LOC for request prediction. Since both rejection and prediction are implemented in the OS, there is no need to modify the server application code. On client side, we provide users with two simple OS APIs to employ our approach:

1. reqsend() is a non-blocking routine that sends a request. It accepts SLO (e.g. deadline), request ID set by application, and other arguments such as socket descriptor as in the sendto() system call

2. notifyrecv() is for retrieving a cancellation notification (on the same socket descriptor), which includes the ID of the request being cancelled by the server.

We modified Cassandra and MongoDB in 70 LOC and 50 LOC respectively to employ MittCPU by using these two APIs.
We next describe the design and implementation of request cancellation and latency prediction in detail.

2.2 Cancellation Mechanism

There are two issues to tackle in cancelling requests: when to promptly cancel requests within the TCP stack and how to correctly remove requests from the TCP byte stream without breaking TCP semantics.

2.2.1 Prompt Cancellation

As mentioned earlier, we cannot do request cancellation in the application layer, because the CPU contention may already have caused delay before the application gets CPU to reject the request. Therefore, we must do the cancellation in the OS layer.

We found two choices to do cancellation in OS: when the packet arrives in the interrupt context (but before the TCP protocol processes the packet) or in the TCP receiving/processing context (which usually happens when the destination process gets a CPU). The former is ideal for prompt cancellation and notification but it is not safe to interfere with the packet stream outside the regular TCP procedure (TCP packet processing is important for checking corruption, out of order delivery, and many other purposes). On the other hand, the latter is safer but slower, due to the nature of Linux TCP “prequeue“. In Linux, when a TCP request arrives, the regular TCP packet processing may not happen immediately. The packet may be put into the TCP “prequeue“ and later be processed when the destination process gets CPU. However, at that time CPU contentions may already have happened and thus the rejection will be slow.

A more ideal scenario is to get both of the benefits—process the packet during interrupt context such that cancellation can be done without any delay, which is the choice we made. This did not just involve moving Linux tcp_recv() function to the interrupt context, but we had to reroute the
stream from TCP “prequeue” to the main TCP receive queue. One concern was that the interrupt context becomes more heavyweight, however we did not see the implication in our tests. Another fortunate news came nine months later when Linux developers also removed TCP prequeueing, but for a different reason; TCP prequeueing is optimized for “single process with blocking read” design, which is no longer a common style (polling-based calls are more frequent).

For cancellation notification, server-side MittCPU sends back ACK messages that contain the IDs of cancelled requests (in general, request information is embedded in the TCP header fields in both directions). For prompt delivery, we disable TCP delayed acknowledgement for cancellation ACKs, or otherwise they could be delayed by more than 40ms. Upon seeing the notice, the client-side MittCPU passes up the message to the application via the notifyrecv() API.

### 2.2.2 Correct request removal

Cancelled requests should be treated differently from dropped (missing) packets. The former case implies that the packets have been received successfully, but not yet delivered to the application. Thus, cancelled requests should not alter the sequence numbers. However at the same time, we need to give the illusion that the cancelled requests have been read by the application such that the packet read ordering is not broken (specifically, Linux TCP’s copied_seq variable should be updated carefully). To do this, we check the state of the application’s receiving queue. If it is not empty, copied_seq should not be modified. We increase copied_seq accordingly until all packets in the receiving queue are read by the application. In more detail, we find the tail packet of the receiving queue and record the length of the cancelled requests. When the application completes reading the tail packet, we perform copied_seq+=\text{(tailPacket.len+cancelledPacket.len)}.
2.3 **CPU Delay Prediction**

This section describes how we build a high precision prediction capability for measuring CPU delay on every latency-sensitive request. Linux implements three schedulers: deadline, real-time, and fair schedulers. Although each of the schedulers tries to optimize its own QoS metric (fairness, deadline, etc.), request deadline can still be missed in shared servers. The fair scheduler one, cooperative fair sharing (CFS), is the default and most complex scheduler (5270 LOC). To take the biggest challenge, we decided to build a predictor for CFS. Unlike other works that modify thread/core management [14, 18, 19, 20, 24], our predictor does not modify CFS because we want to adhere to resource manager simplicity and don’t want to interfere with the resource layer’s QoS policies. Our predictor only needs to change when CFS evolves, which rarely happens, e.g. within the span of Linux v4.0 to v4.20 (3.5 years), CFS only changes by 700 lines annually. For high precision, the predictor must consider many types of process/thread characteristics.

Below, we first briefly describe CFS scheduling policy, and then describe our predictor from a naive version to a complete one.

### 2.3.1 CFS Scheduler

**CFS Overview**

The Completely Fair Scheduler (CFS) is Linux’s default scheduler for scheduling regular tasks. The CFS’s design can be summed up in a single sentence: CFS basically models an "ideal, precise multi-tasking CPU" on real hardware.

In an "ideal multi-tasking CPU", with 100% physical power and textitn tasks running, each task should run at precise equal speed, in parallel, at $1/n$ speed. This is obviously impractical. On real hardware, a CPU can only run one task at once.

CFS tries to mimic the "ideal multi-tasking CPU" by time-slicing the CPU among the runnable tasks. The time slice is not fixed, but calculated based on the number of runnable tasks.
defines a fixed time interval, and divides the interval among tasks proportionally to their weights. During this interval, each task should get a chance to run. Here textit"weight" is determined by the task’s priority. A task with higher priority would have a larger weight value, thus get more time to run during the interval.

At each timer interrupt, CFS scheduler checks how long the current running task has been running. If the running time has exceeded the time slice, the running task will be preempted and CFS will pick some other task to run.

While picking the next task to run, in order to be fair, CFS wants to the task which executed least so far. To achieve this goal, CFS introduces a concept named "virtual runtime". The value of virtual runtime is calculated based on the process’s real runtime, but is also affected by some other factors (e.g. process’s priority). In Linux kernel, this value is tracked by the per-task variable p.se->vruntime.

The runqueue of the tasks is implemented as a rbtree, which is ordered by the tasks’ vruntime value. When choosing the next task to run, CFS basically picks the left-most node (i.e. the task with the smallest vruntime).

As the system progresses forwards, the executed processes’ vruntime increase, so they are put into the tree more and more to the right, giving a chance for every process to become the left-most one.

The process’s vruntime is also affected by process’s priority. The lower the priority, the quicker the vruntime increases. Thus, a low priority process runs for less real time.

**CFS Group Scheduling**

CFS implements group scheduling. In CFS, the scheduling entity might not be a real thread; instead, they may represent a specific group of threads. In the latter case, the scheduling entity will have its own run queue within it. This turn scheduling entities into a hierarchical structure.

When the scheduler picks the next task to run, it looks at all of the top-level scheduling entities
and takes the one which is considered most deserving of the CPU. If that entity is not a real thread (but a higher-level scheduling entity), then the scheduler looks at the run queue contained within that entity and starts over again. Things continue down the hierarchy until an actual thread is found, at which point it is run. As the thread runs, its runtime statistics are collected as usual, but they are also propagated up the hierarchy so that its CPU usage is properly reflected at each level.

The group scheduling mechanism makes the threads more organizable. For example, suppose we have two users Alice and Bob. Alice has 1 process to run, and Bob has 9 processes to run. Without group scheduling, Alice will get 10% CPU time, and Bob will get 90%. However in this case, we want to ensure fairness by user, rather than by thread. With group scheduling, we can put Alice and Bob’s threads into different groups, so they both get 50% CPU time.

2.3.2 MITT CPU Prediction: from Naive to Complete

Based on the knowledge of CFS scheduling logic, we now describe our predictor from a naive version to a complete one.

**Linear Prediction**

In a naive scenario where all process threads are CPU bound and long running on one CPU core and exists in the same user and priority group, the prediction can be based on a simple equation. For every thread $T$ in the waiting (ready) queue, the future time slice when $T$ will get the CPU is:

$$
\left( \frac{T.vruntime - U.vruntime}{\text{timeSlice}} + 1 \right) \ast \text{timeSlice}
$$

where $vruntime$ is the weighted time a thread has run on the CPU [3, 5], $\text{timeSlice}$ is 4ms, and $U$ is the next thread to be scheduled after $T$. Thus, to measure the CPU delay of a request designated to a thread $X$, we find all the threads supposed to be scheduled before $X$ (the threads on the left side of $X$’s position in CFS rbtree), then calculate how much time each must wait, and finally sum
all the wait time.

**Hierarchical Prediction**

However as mentioned earlier, CFS implements a complex hierarchical scheduling. When the scheduler picks a task to run next, it first searches from the top-level scheduling “entities” and takes the one with the lowest \(v_{\text{runtime}}\). If the chosen entity is not a real thread but rather another high-level scheduling entity (i.e. a nested hierarchy), the scheduler dives into the entity, searches through its runqueue, and repeats the procedure again until an actual thread is found. The chosen thread will be given a time slice to run (4ms) before being preempted. After this, the thread’s runtime statistics are updated and also propagated up the hierarchy so that its new \(v_{\text{runtime}}\) is properly reflected to the \(v_{\text{runtime}}\) of its parent entities.

The implication of this hierarchy is that linear prediction no longer suffices. We must “simulate” what likely will happen, but at the same time not tamper with the actual accounting values. Thus, our predictor maintains a *shadow copy* of the entire hierarchy. When a prediction is needed, the shadow copy is first refreshed from the original values, after which the delay prediction is run on the shadow copy.

**Precise Timeslice Adjustment**

CFS performs scheduling on every timer interrupt or when a thread relinquishes CPU (e.g. when calling a blocking operation). Thus a thread does not necessarily run at a time slice boundary. Upon a timer interrupt, if the last execution time window of the currently running thread has not exceeded its assigned slice, CFS will skip scheduling on the current timer interrupt. Due to this imperfect time alignment, our predictor is occasionally off by roughly 4ms. Theoretically, if thread \(A\) starts at a timer interrupt \(t\), then at the next interrupt \(t+1\) we would intuitively assume that \(A\) has run for exactly 4ms. Upon further investigation, we found that the accounted execution time is slightly shorter than 4ms (e.g. 3.99ms). The reason is that the time taken for CFS to find the next
running thread (e.g., 0.01ms) is not accounted into A’s execution time. This causes imprecision when a thread’s assigned slice is exactly (or multiple of) a time slice (4ms). With this observation, our predictor must slightly underestimate vruntime (e.g., by 0.01ms).

**Dependent and I/O-bound threads**

So far we assumed all threads are long running and independent, but this assumption does not hold for all types of workload. For example in the vips benchmark (an image processing system [4]), the threads are dependent on each other via synchronization primitives such as futex. For example, a thread A occasionally wakes up another mostly-idle thread B to execute some operation. Since B’s CPU consumption is very small, it is favored by CFS to run next and B only runs in a short burst and then sleeps again. Our prediction is imprecise because of this nature of dependency and short burst that does not consume a full time slice.

To incorporate this behavior, MittCPU marks dependent threads (via futex tracing) and estimates how long a dependent thread must wait before being wakened up by another thread. We record every dependent thread’s vruntime and idle duration and use an exponential moving average to make the estimation. Let $z_n$ denotes our estimation of a thread’s idle duration at the $n^{th}$ time it is waiting and $t_n$ the real idle duration, then by using an exponential moving average, our estimation for the next $(n+1)^{th}$ wait time is:

$$z_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot z_n$$

Here $\alpha$ represents a decreasing weight, a constant smoothing factor between 0 and 1; a higher $\alpha$ will value more recent observations. We use 0.05 for $\alpha$.

The same observation and technique can be made for I/O bound threads that often run short CPU bursts and wait for I/Os. However, note that when the application makes an I/O, it is not blocked when the I/O is served by the buffer cache or destined to a memory file system. Thus, we need to trace the actual device I/O time for recording $t_n$. 

12
In summary, our predictor can make every prediction in less than 10µs. Future optimizations are possible. For example, refreshing the shadow copy can be made faster by having CFS updates the shadow copy on the fly (which we have not done, in order to not modify CFS code). For high precision, we believe we have covered common workload characteristics, but acknowledge that our predictor can still be imprecise in several cases such as work stealing. For example, when an arriving request is designated for thread $T$ on CPU core $C$, our predictor only scans through the threads queued on core $C$ (CFS maintains per-core hierarchical scheduling). Accuracy will drop when there is a workload imbalance that makes a CPU core steal threads from another core. This is solvable, but we rarely see such cases except on complex CPU benchmarks on hyperthreaded CPUs.
Chapter 3

Implementation

We implemented MittCPU in 2300 LOC in Linux kernel, with 750 LOC for prompt request cancellation and 1550 LOC for request prediction. Since both rejection and prediction are implemented in the OS, there is no need to modify the server application code. On client side, we provide users with two simple OS APIs to employ our approach:

- `reqsend()` is a non-blocking routine that sends a request. It accepts SLO (e.g. deadline), request ID set by application, and other arguments such as socket descriptor as in the `sendto()` system call.

- `reqrecv()` is for receiving requests akin to the `recvfrom()` system call.

- `notifyrecv()` is for retrieving a cancellation notification (on the same socket descriptor), which includes the ID of the request being cancelled by the server.

We modified Cassandra and MongoDB in 70 LOC and 50 LOC respectively to employ MittCPU by using these two APIs, demonstrating the non-intrusiveness of our approach.
Chapter 4

Evaluation

4.1 EXPERIMENT SETUP

We use Emulab d430 machine that has 16 cores (32 logical), 64 GB DRAM, and 1 Gbps network. The retry overhead (machine-to-machine ping-pong) is only 120μs. We deploy MittCPU on 3-node and 6-node clusters respectively, half used as clients and half used as servers.

We mainly use Cassandra as our target storage server application, as Cassandra is popular and well-known for suffering from long latencies due to resource contentions. We warm up the cluster to have the user’s data reside in memory. Every key-value has three copies (the default configuration). For the client workload, we use a microbenchmark where every client node sends 10,000 requests per second and a macrobenchmark with various load, noise, and read/write distributions. The experiments are performed several times to ensure reproducibility.

For CPU noises, the Cassandra servers are colocated with CPU-intensive jobs. We use 25 CPU-intensive applications from three different suites including Minebench (PLSA, ScalParC, HOP), PARSEC (blackscholes, swaptions, fluidanimate, canneal, vips, dedup, bodytrack, faceSim, freqMine, raytrace, x264, ferret), SPLASH 2 (barnes, fft, fmm, lu-cb, ocean-cp, radiosity, radix, raytrace, volrend, water-nquared). These benchmarks test a range of important modern applications, both compute-intensive and memory-intensive. In addition, all workloads are
long running, taking at least 2 minutes to complete when running individually on a single CPU. This duration gives us plenty of time to evaluate the performance and prediction accuracy of MittCPU.

We evaluate MittCPU against popular practices such as Cassandra’s replica selection [1] and speculative execution [2] with various timeout values. For example, “95P speculative retry” is often suggested [6, 12] where a backup request is sent after the 95\textsuperscript{th}-percentile latency value (the timeout value) has elapsed.

### 4.2 Performance Evaluation

For performance evaluation, we first evaluate MittCPU on a 3-node cluster to show the effectiveness of our approach. We then evaluate MittCPU on a larger 6-node cluster setup and compare it with other approaches.

#### 4.2.1 Effect of MittCPU (3 nodes, dummy CPU-intensive threads)

We first compare the “Raw” setup (no tail mitigation) with MittCPU to show the effect of our tail-tolerant approach. In this configuration, we use 1 client node and 2 server nodes where one
of them experiences CPU contention. For CPU noises, we insert 5 dummy CPU-intensive threads per core. We make the client choose the contended node first. The purpose of this experiment is to show a “best-case” scenario that MittCPU can obtain.

First, the “Best” line in Figure 4.1 shows the latency CCDF of the client requests when there is no contention in the three resource layers. The line is vertically straight around x=0.7ms, the best-case scenario we should target.

Second, the “Raw” lines (with noise) in Figure 4.1 show that the noises inflict long tail latencies to the user requests roughly 10% of the time compared to the “Best” line.

Finally, the MittCPU line show that MittCPU can quickly react to the CPU contention. MittCPU successfully eliminates the tail latencies caused by the contentions, as we can see the large gap between the MittCPU and “Raw” line. Compared to the “Best” line, MittCPU is only 28% and 81% slower than the best case at 5% and 1% of the time (y=5 and y=1), respectively.

We note that both in the “Best” and MittCPU lines, we still observe a small <1% latency tail, caused by “unknown” cases not covered by MittCPU. For example, in the Emulab testbed, we always observe 0.3–0.5% long latency tail in a simple ping-pong workload, probably caused by network contention. This small residual tail can be amended by combining MittCPU with a small number of timeout-based speculative retries, as we show later.

4.2.2 MittCPU vs. Other Techniques
(6 Nodes, Representative CPU Benchmarks)

We now thoroughly compare MittCPU with other popular practices. We use three client nodes and three servers and every key-value is replicated three times. All the server nodes are shared between Cassandra and other CPU-intensive jobs. Here we use real representative CPU noises—for every core, we repeatedly run 2 threads of random CPU benchmarks (from the 25 benchmarks) in short bursts and short pauses in between every two runs.

Figure 4.2 shows the same CCDF plot as in the prior figure, but now we have a $log_2 y$-axis
Figure 4.2: **MittCPU vs. other techniques**. The latency CCDF (with log\(_2\)-scale y-axis) compares MittCPU versus other popular techniques (please see §4.2.2 for explanation).

![CCDF Latency Graph](image)

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Average Latency (ms)</th>
<th>Slowdown vs. Best (in %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw</td>
<td>2.17</td>
<td>234</td>
</tr>
<tr>
<td>SR-95</td>
<td>1.39</td>
<td>114</td>
</tr>
<tr>
<td>SR-90</td>
<td>1.05</td>
<td>62</td>
</tr>
<tr>
<td>SR-85</td>
<td>0.75</td>
<td>15</td>
</tr>
<tr>
<td>MittCPU-95</td>
<td>1.14</td>
<td>76</td>
</tr>
<tr>
<td>MittCPU-90</td>
<td>0.81</td>
<td>24</td>
</tr>
<tr>
<td>MittCPU-85</td>
<td>0.72</td>
<td>10</td>
</tr>
<tr>
<td>M85+SR98</td>
<td>0.67</td>
<td>near best → 3</td>
</tr>
<tr>
<td>Best</td>
<td>0.65</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 4.1: **Average latencies**. The table shows the average latencies of the latency distributions in Figure 4.2.

and 9 lines representing the best and raw scenarios, speculative retry and MittCPU with 3 different deadline/timeout configurations (85P, 90P and 95P values), and using MittCPU with 98P deadline speculative retry.

**BEST AND RAW SCENARIOS**

We first compare the best and raw scenarios (left-most vs. right-most lines). Here we can see that the CPU noises introduce tail latencies roughly 15% of the time (y=15 in Figure 4.2).
TIMEOUT-BASED SPECULATIVE RETRY ("SR-85" TO "SR-95")

To recap, timeout-based speculative retry implies that if a request has not received a response after a certain duration (the timeout value), this method will send a backup request to another replica.

In this technique, setting a proper timeout value is difficult. Too short means too pessimistic (leads to more inefficiency from sending too many backup requests). Too long means too optimistic and it will reduce application reactivity in retrying promptly. For fairness, we tried three timeout values, 1.5ms at 85P, 3ms at 90P, and 8ms at 95P (based on the percentile values in the Raw distribution).

Figure 4.2 shows that speculation is effective in cutting tail latencies, but not as reactive as MittCPU, mainly because speculation does not know what is currently happening in the servers. For example, in the “SR-90” case, the speculation only sends backup requests after waiting for 3 ms (the 90P Raw value). On the other hand, MittCPU will quickly send a cancellation notice when it knows the request deadline cannot be met. Hence, our MittCPU-supported application does not have to wait for any timeout; cancellation notices were sent promptly and the application reacts much faster.

Figure 4.2 also shows that speculation with a pessimistic timeout (e.g. at 80P) leads to many backup requests (20%) that cause more inefficiencies.

MITTCPU

We can see that overall MittCPU is more efficient. Just like timeout-based retry, MittCPU depends on the application to provide the deadline value for the requests. Thus, we have three MittCPU lines in the figure with 85P, 90P and 95P deadline values for “apple-to-apple” comparison to speculative retries. MittCPU-85 and MittCPU-90 exhibit shorter tail latencies while MittCPU-95 does not trigger many retries (due to the relaxed 8ms deadline).
MittCPU with 98P (2%) speculation

MittCPU only covers contention in the resource layers that participate in providing cancellation and prediction capabilities. It does not eliminate all sources of tails. As mentioned before, even the best-case lines always show almost 1% of tail latencies (due to network contention or other unknown factors). For this reason, it is wise to combine MittCPU with speculative retry but with an optimistic timeout. For example, in the “M85+SR98” line, the application sets an 85P deadline value for MittCPU and still sends backup requests when a 98P-value timeout has elapsed, sending only 2% backup requests.

Important to note that the 98P timeout value in M85+SR98 is the 98P value (y=2) of the MittCPU-85 performance, not the “Raw” performance. In general, applications can adjust the speculation timeout value accordingly depending on the level of unknown noises that they observe. This also suggests that MittCPU does not require all layers to have cancellation and prediction capabilities; we only advocate that major resource layers participate in the MittCPU ecosystem.

Overall results

In addition to CCDF graphs, Table 4.1 shows the average latencies. As shown, cutting long latency tail by implication reduces the average latency. MittCPU with 85P deadline and with 98P speculative retry (“M85+SR98”) is the most optimum because in the original distribution without tail mitigation (the Raw line in Figure 4.2), the latency tail starts appearing around 85P, and in the MittCPU-85 line the residual latency tail is roughly 2%. We also observe that the deadline P value can be decided with a simple algorithm that finds the latency point at 45° angle in the original distribution (the Raw line).

Overall, compared to speculative retries with the same P values, MittCPU reduces the completion time of user requests by 31-66% at 90P and 35-70% at 95P (see Figure 4.2), and 5-23% on average (derived from Table 4.1). Compared to the best-case scenario, our most optimum setup, “S85+SR98”, is only 3% slower on average (see Table 4.1) and 13% and 22% slower at 90P and
### Table 4.2: Benchmark mixes. Benchmark mixes for measuring MittCPU’ prediction accuracy.

<table>
<thead>
<tr>
<th>Mix</th>
<th>Description</th>
</tr>
</thead>
</table>
| A   | User 1: blackscholes, swaptions, fluidanimate  
+ User 2: faceSim, freqMine, raytrace         |
| B   | barnes, fft, fmm, lu-cb, ocean-cp, radiosity, vips |
| C   | radix, raytrace, volrend, water-nsquare, barnes, fft, dedup |

95P, respectively (see Figure 4.2).

### 4.2.3 Precision Accuracy

As discussed earlier, we implemented a predictor to estimate how long an arriving request must wait to get the CPU. We now evaluate the accuracy of our predictor.

To measure MittCPU’ prediction precision, we use applications from the 3 CPU benchmark suites mentioned earlier. We start with a 1-core evaluation where we run a mix of 3 to 7 applications, as listed in Table 4.2. We performed many mixes and repeated several times, and here we present three representative mixes of benchmarks with unique results that show the importance of MittCPU prediction features.

We measure MittCPU imprecision by instrumenting Linux and adding information about when threads are running and preempted. In every 100 ms, we pick a random thread \( T \) in the wait queue and let MittCPU predicts how long \( T \) has to wait before obtaining a CPU, *i.e.*, the estimated delay \( D_{est} \). At the same time, our instrumentation also monitors the actual delay \( D_{real} \). We measure the error \( D_{err} = D_{est} - D_{real} \) and collect 1000 data points for every mix.

Figures 4.3a-c show the CDF of the \( D_{err} \) data points in the three mixes. In every figure, we show the naive feature to the most complete feature: linear prediction (LIN), plus hierarchical prediction (+HIER), plus timeslice adjustment (+ADJ), and plus dependent thread awareness (+DEP).

In Mix-A (Figure 4.3a) with CPU-bound benchmarks, our hierarchical feature (+HIER) is better than the linear prediction (LIN), but not too precise, until the timeslice adjustment is added (+ADJ). In Mix-B (4.3b), with the presence of vips, an image processing system with dependent
Figure 4.3: **MittCPU prediction precision.** *The figures show CDFs (0.0 to 1.0 in y-axis) of CPU delay prediction errors ($D_{err}$ in ms in x-axis) across CPU benchmark mixes.*

threads, the first three features are not enough, but with dependent thread awareness (+DEP), the predictor becomes more precise. Mix-C is a different mix but with dedup which also contains dependent threads, hence +DEP shows more precision. Overall, Figures 4.3 a-c show that MittCPU is highly precise; the +DEP lines hover around x=0 (i.e. 0ms errors) and are only off by mostly +/- 8ms in 5% of the time.

Finally, we show an experiment on 32 cores (Figure 4.3d) where we ran 2-3 random benchmarks per core with a mix of 70% real CPU and 30% synthetic benchmarks (the memory space is not enough to run all real CPU benchmarks). The solid line in Figure 4.3d shows that MittCPU exhibits a 92% precision. The dashed “worst-cpu” line represents the CPU core where MittCPU predicts accurately only 74% of the time. However, on the “best-cpu” core, MittCPU is precise up to 98%. While being not fully precise, MittCPU only mispredicts by 1-2 timeslices (+/- 4-8ms across the x-axis).
Chapter 5

Limitations & Future work

Currently MittCPU only predicts how much time a request needs to wait to be processed. It doesn’t take into account how long the application needs to spend processing the request. It’s possible that a request takes long time to be processed, which cannot be ignored and should be taken into account when doing delay prediction.

Also, it’s possible that the request doesn’t need to wait for long time to be read by the application, but when the application is processing the request, CPU contentions can happen and some other thread may steal the CPU from the server application. MittCPU doesn’t consider this scenario and leaves it for future work.

Furthermore, in event-driven storage systems like Cassandra, a request will be passed among different stages, and each stage runs a different thread. CPU contentions can happen when the request is transferred between these stage threads and cause delays. For example, when the request arrives at the server machine, it will be read by stage A. Stage A may be able to quickly get CPU and read the request, in which case the request won’t be rejected by MittCPU. But later when stage A passes the request to stage B, it’s possible that stage B cannot get the CPU immediately, which may cause tail latencies. In the future, we may need to apply MittCPU principle to all the stages, and do multi-stage delay prediction and request rejection in order to provide better tail mitigation.

Additionally, CPU contentions not only happen between applications, but also can happen
between Virtual Machines (VMs). Consider a Cassandra server deployed on a VM. When the request arrives at the hypervisor (the physical machine), the request first needs be directed to the target VM by the hypervisor OS, and then be directed to the Cassandra server by the VM OS. However, the hypervisor may run multiple VMs on the same CPU. When the request comes, the target VM may not be able to immediately get CPU due to contentions from other VMs, which thus leads to tail latencies.

We may apply MittCPU principle to deal with tail latencies caused by CPU contentions between VMs. The rejection needs to be done in the hypervisor layer, before the CPU contentions happen. Note that MittCPU rejection implementation cannot apply here, because the hypervisor serves as router, which can only deal with IP layer, but can do nothing in the TCP layer (reminder that we implemented MittCPU rejection in the OS TCP stack). One suggestion is to implement the rejection in the ICMP layer, since ICMP messages are commonly used by the router to generate error messages to the source node.

We can also apply MittCPU principle to deal with contentions in other resource layers, e.g. runtime memory, network, lock, etc. We can integrate all of them to mitigate tail latencies caused by different resource contentions.
Chapter 6

Related Work

There are many papers that discuss the issue of tail latencies due to CPU contention and modify applications’ or resource layers’ QoS policies to mitigate this issue. MittCPU does not modify any QoS policies.

Bobtail [30] says that in Amazon EC2 the tail of round trip latency can be up to 30ms at p99.9 due to Xen’s VM scheduling delays. Hence it designs a instance selection algorithm to find the best N instances with lowest possibility of producing long latency tails. Zygos [25] says that in-memory data services using kernel-bypass dataplane approaches can suffer from long tail latencies due to temporary load imbalance between cores. Hence, it modifies IX kernel in 2K LOC and the Dune framework in 200 LOC to reduce the tail. Shenango [24] says that tail in Memcached can be up to 2ms at p99.9 with batch work running, because Linux rebalances tasks across cores in response to millisecond-scale timer ticks, which is too slow to handle microsecond-scale requests efficiently. Hence it designs an I/O Kernel in 2K LOC and a runtime in 6K LOC to reduce the tail. Shinjuku [19] says that Linux thread management can produce millisecond-scale tail latencies because Linux is not designed for microsecond-scale tasks and frequently produces long and unpredictable scheduling delays. Hence it implements a preemptive scheduler at the microsecond-scale for the operating system to reduce the tail. It modifies Dune in 1K LOC and designs its dispatcher and worker in 2K LOC. RPCValet [11] says that imbalanced distribution of
RPC load on CPU cores can result in high tail latency. Hence it implements a co-designed hardware and software system to achieve dynamic load-balancing across CPU cores, which delivers up to 4X lower tail latency before saturation. Prophet [10] says that latency-sensitive applications on accelerators suffer from tail latencies due to contention among processing elements on accelerator memory bandwidth and PCIe bandwidth. Hence it designs a methodology to precisely predict the tail latencies due to co-locations on non-preemptive accelerators. With that, it enables safe co-locations and improves utilization without violating the QoS requirement. FM [18] says that in web search systems, tail latencies induced by ineffective software parallelism can surge up to 1500 ms at p99. To mitigate that, it modifies the process management layer in 1100 LOC. Bolt [14] says that the characteristics of applications sharing a cloud platform can be accurately detected. As a result, extracting this information enables previously-impractical DoS attacks that can increase tail latency by 140x. Hence, it prohibits core sharing, which, however, leads to significant inefficiencies and performance penalties. SubMS QoS analysis [20] says that Memcached can suffer from long tails when co-located with other workloads due to increases queuing delay, thread load imbalance, and scheduling delay. Hence, it applies interference-aware provisioning on latency-critical services to reduce queuing delay, pins threads to distinct cores to mitigate thread load imbalance, and modifies Linux CFS scheduler in 150 LOC to reduce scheduling delay.
Chapter 7

Conclusion

We have shown how CPU contentions on the cloud can result in tail latencies. We have shown MittCPU, an operating system support which aims mitigating tail latencies induced by CPU contentions.

MittCPU accurately predicts the delay latency caused by CPU contentions, and promptly rejects requests that cannot meet their deadline, enabling the client to quickly failover to another client without waiting. We have shown how MittCPU builds its request cancellation and delay prediction capabilities, and have successfully demonstrated MittCPU can help applications achieve stable latencies.

We have also shown how integrating MittCPU with other techniques, e.g. Speculative Retry, can further improve latencies. In the future, people can apply the fast-rejection principle to other resource layers (e.g., network, VM, etc) and integrate them to provide better performance and reliability.
References


