CPI^2: CPU performance isolation for shared compute clusters

Xiao Zhang Eric Tune Robert Hagmann Rohit Jnagal Vrigo Gokhale John Wilkes
Google, Inc.
{xiaozhang, etune, rhagmann, jnagal, vrigo, johnwilkes}@google.com

Abstract

Performance isolation is a key challenge in cloud computing. Unfortunately, Linux has few defenses against performance interference in shared resources such as processor caches and memory buses, so applications in a cloud can experience unpredictable performance caused by other programs' behavior.

Our solution, CPI^2, uses cycles-per-instruction (CPI) data obtained by hardware performance counters to identify problems, select the likely perpetrators, and then optionally throttle them so that the victims can return to their expected behavior. It automatically learns normal and anomalous behaviors by aggregating data from multiple tasks in the same job.

We have rolled out CPI^2 to all of Google's shared compute clusters. The paper presents the analysis that lead us to that outcome, including both case studies and a large-scale evaluation of its ability to solve real production issues.

1. Introduction

Google's compute clusters share machines between applications to increase the utilization of our hardware. We provision user-facing, latency-sensitive applications to handle their peak load demands, but since it is rare for all the applications to experience peak load simultaneously, most machines have unused capacity, and we use this capacity to run batch jobs on the same machines. As a result, the vast majority of our machines run multiple tasks (Figure 1). The number of tasks per machine is likely to increase as the number of CPU cores per machine grows.

Unfortunately, interference can occur in any processor resource that is shared between threads of different jobs, such as processor caches and memory access-paths. This interference can negatively affect the performance of latency-sensitive applications: an internal survey elicited examples such as "latency for one task skyrocketed for the period during which a batch job was running on the machine" and "1/66 of user traffic for an application in a cluster had a latency of more than 200 ms rather than 40ms for more than 1 hr". Predictable, low latency is key to end-user satisfaction, so this is a real problem: engineers are paged when an important task or job becomes a victim of such interference.

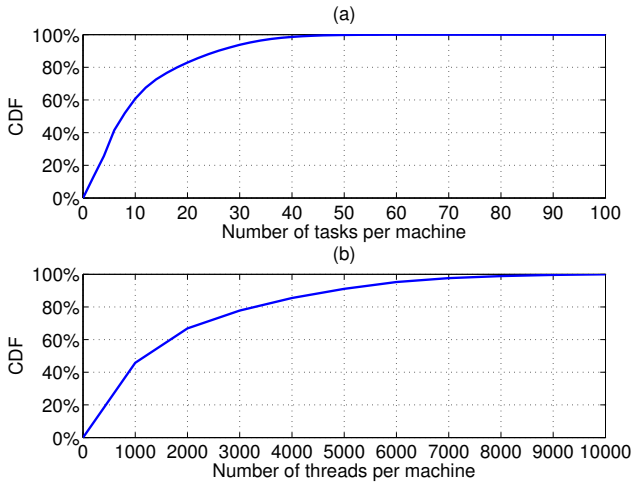

Figure 1: *The number of tasks and threads running on a machine, as cumulative distribution functions (CDFs).*

The current state of performance isolation in commodity Linux operating system kernels gives us limited options: we can tolerate the poor performance (undesirable), grant additional resources to the victim applications (wasteful, and since most Linux kernels do not manage shared processor resources like caches and memory controllers, it might not fix the problem), or grant the victims dedicated, unshared resources (even more wasteful). None of these choices are attractive.

Fortunately, applications designed for large-scale compute clusters often have hundreds to thousands of similar tasks, so it is possible to use a statistical approach to find performance outliers (which we call *victims*), and address them by reducing interference from other tasks (we call such tasks

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

antagonists even though the interference may be accidental). Finding such outliers requires a metric that is relatively stable across well-behaved executions of applications, and is well-correlated with the bad behavior caused by antagonists. An application's cycles per instruction (CPI) is such a metric: most latency-sensitive applications in our compute clusters have fairly consistent CPIs across tasks and time, provided the CPI is averaged over a period that is much longer than the time to perform a single user-facing transaction or query, and the CPI calculations are done separately for each processor type.

This paper describes CPI^2, a system that builds on the useful properties of CPI measures to automate all of the following:

1. observe the run-time performance of hundreds to thousands of tasks belonging to the same job, and learn to distinguish normal performance from outliers;
2. identify performance interference within a few minutes by detecting such outliers;
3. determine which antagonist applications are the likely cause with an online cross-correlation analysis;
4. (if desired) ameliorate the bad behavior by throttling or migrating the antagonists.

The result is that troublesome performance interference can be detected and acted on, which makes it possible to continue to share resources between applications, and maintain high utilization levels. A prototype of CPI^2 has already been deployed in Google's compute clusters.

The contributions of this paper are to demonstrate the viability of CPI as an appropriate measure in this environment; describe the structure of the CPI^2 system; and show that it works by studies from production systems.

2. Background

In Google's cluster management system, both latency-sensitive and batch jobs are comprised of multiple tasks, each of which is mapped to a Linux process tree on a machine. All the threads of a task run inside the same resource-management container (a cgroup [27]), which provides limits on the amount of CPU and memory the task can use. Jobs with many tasks are the norm: 96% of the tasks we run are part of a job with at least 10 tasks, and 87% of the tasks are part of a job with 100 or more tasks. Tasks in the same job are similar: they run the same binary, and typically process similar data.

A typical web-search query involves thousands of machines working in parallel [6, 19, 29, 25], each one contributing some portion of the final result. An end-user response time beyond a couple of hundred milliseconds can adversely affect user experience [33], so replies from leaves that take too long to arrive are simply discarded, lowering the quality of the search result and wasting the resources spent to generate them. Reducing the performance variation that results from imperfect isolation is one way to minimize this problem.

Even MapReduce [12, 18] applications can benefit: a typical MapReduce job doesn't finish until all its processing has been completed, so slow shards will delay the delivery of results. Although identifying laggards and starting up replacements for them in a timely fashion [39, 3] often improves performance, it typically does so at the cost of additional resources. And it doesn't always help: consider the case of a slow storage server, where adding another map task that reads from it will make things worse. Better would be to eliminate the original slowdown.

Each of our clusters runs a central scheduler and admission controller that ensures that resources are not oversubscribed among the latency-sensitive jobs, although it speculatively over-commits resources allocated to batch ones. Overcommitting resources is a form of statistical multiplexing, and works because most jobs do not use their maximum required resources all the time. If the scheduler guesses wrong, it may need to preempt a batch task and move it to another machine; this is not a big deal – it's simply another source of the failures that need to be handled anyway for correct, reliable operation.

Jobs are classified and prioritized into "production" and "non-production" by users or the framework that runs them (e.g., MapReduce jobs are batch by default). In one typical cluster, 7% of jobs run at production priority and use about 30% of the available CPUs, while non-production priority jobs consume about another 10% CPU [30].

Although severe resource interference between tasks is relatively rare, the scale of the compute load at Google means that it does happen, and sometimes causes bad performance. Tracking down the root cause of such problems consumes a great deal of effort, since poor performance isolation is just one possible cause amongst many.

CPI^2 improves behavior of latency-sensitive jobs when they experience interference by: detecting CPU performance isolation incidents, automatically identifying which jobs are causing the problem, and (optionally) shielding victim jobs by throttling the antagonists. CPI^2 is one of many techniques to reduce or compensate for variability in response time, which becomes increasingly important at scale [11].

Our goal is to identify inter-job CPU interference so that it can be addressed by throttling. We do not attempt to determine which processor resources or features are the point of contention; that typically requires low-level hardware event profiling as well as human analysis, and is beyond the scope of this work. Nor do we attempt to address interference on other shared resources such as network and disk. We focus on CPU interference because we find enough examples where this is a problem to make it worthwhile.

The remainder of the paper is structured as follows. It starts by showing that CPI is a well-behaved metric with

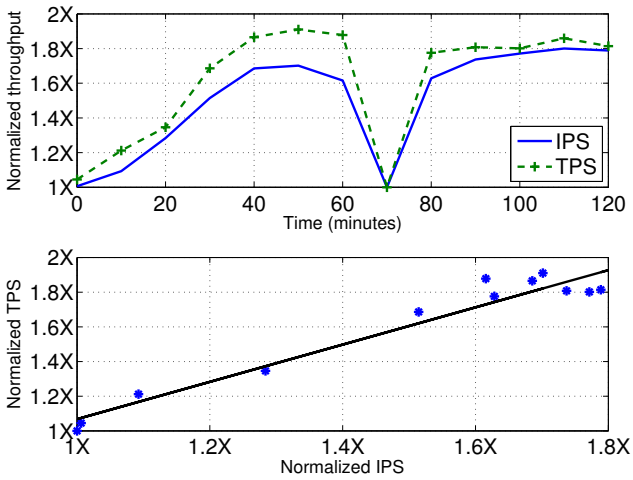

Figure 2: *Normalized application transactions per second (TPS) and instructions per second (IPS) for a representative batch job: (a) normalized rates against running time; (b) scatter plot of the two rates, which have a correlation coefficient of 0.97. Each data point is the mean across a few thousand machines over a 10 minute window. The data is normalized to the minimum value observed in the 2-hour collection period.*

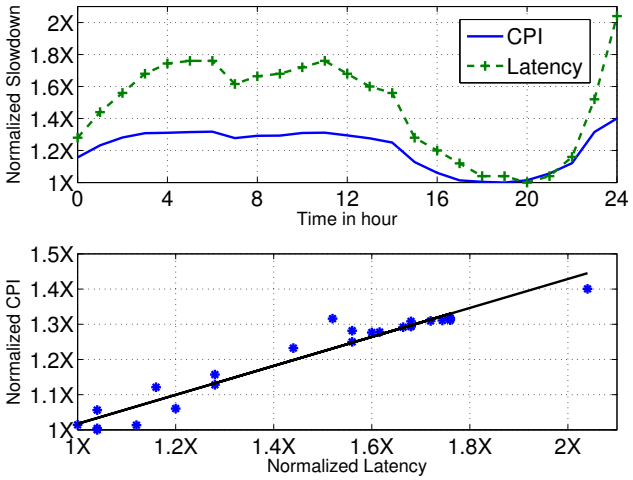

Figure 3: *Normalized application request latency and CPI for a leaf node in a user-facing web-search job: (a) request latency and CPI versus time; (b) request latency versus CPI; the correlation coefficient is 0.97. The latency is reported by the search job; the CPI is measured by CPI^2. The results are normalized to the minimum value observed in the 24-hour sample period.*

good predictive powers (section 3) and that it is useful for finding antagonists (section 4) before describing how we handle those antagonists (section 5). Section 6 describes our experience of deploying CPI^2 in Google's production clusters and section 7 evaluates its effectiveness via a large-scale experiment. We discuss related work (section 8) and future work (section 9 before summarizing our conclusions in section 10.

3. CPI as a metric

Our system relies on cycles per instruction (CPI) measures as a performance indicator for detecting interference. In this section, we explain that choice, and argue that CPI correlates well with observed application behavior.

CPI can be measured easily and cheaply on existing hardware and doesn't require application-level input. But is it useful for performance-interference detection with the kinds of applications that run in cloud computing clusters? There are a few potential concerns:

- CPI might not be well correlated with application-level behavior. We show that it is.
- The number and mix of instructions required to accomplish a fixed amount of work may vary between tasks of the same job, or over time in one task (e.g. due to just-in-time compilation or synchronization overheads). In practice, we have not found this to be an issue.
- CPI only shows a symptom, not the root cause. Yes, but it doesn't really matter: treating the symptom can restore good performance.
- CPI doesn't measure network or disk interference effects. True. Other techniques are needed to detect and handle I/O-interference. But there are enough examples of CPU interference to make it worth addressing.

In support of the first point, consider Figure 2, which compares an application-specific measure of throughput (transactions per second, TPS) and CPU instructions per second (IPS) for a 2600-task batch job. The transaction rate is reported by the job; the CPU instruction rate is calculated by dividing the observed CPU cycle speed by the observed CPI. The rates track one another well, with a coefficient of correlation of 0.97.

Figure 3 shows data for average CPI and request latency in a latency-sensitive application (a web-search leaf node), with similar results. Again, we see a coefficient of correlation of 0.97.

The CPI is a function of the hardware platform (CPU type). Figure 4 shows data for CPI and request-latency of individual tasks in three web-search jobs on two different platforms; the CPU clock speed was not changed in these experiments. Two of the jobs are fairly computation-intensive and show high correlation coefficients (0.68–0.75), but the third job exhibits poor correlation because CPI does not capture I/O behavior: it is a web-search root node, whose request latency is largely determined by the response time of other nodes, not the root node itself.

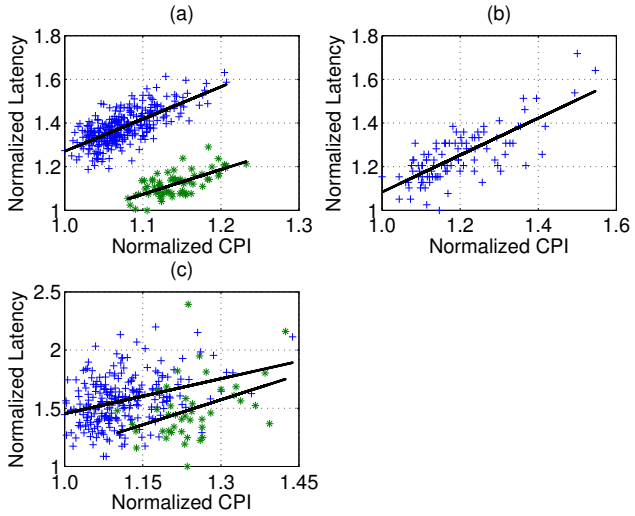

Figure 4: *Normalized request latency and CPI of tasks in three web-search jobs: (a) a leaf node; (b) an intermediate node; (c) a root node. Each point represents a 5-minute sample of a task's execution. Different colors indicate different hardware platforms.*

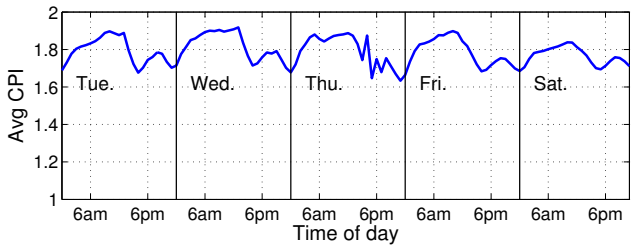

Figure 5: *Average CPI across thousands of web-search leaf tasks over time. The first day is 2011.11.01.*

CPI changes slowly over time as the instruction mix that gets executed changes. Figure 5 plots the mean CPI of the tasks of a web-search job over 5 days. It demonstrates a diurnal pattern, with about a 4% coefficient of variation (standard deviation divided by mean). Similar effects have been observed before (e.g., [25]).

We conclude that there is a positive correlation between changes in CPI and changes in compute-intensive application behavior, and that CPI is a reasonably stable measure over time.

3.1 Collecting CPI data

In order to use CPI as a predictive metric, we need to collect it from all the machines in a cluster. This section describes how we do that.

Figure 6 shows our CPI pipeline: CPI data is gathered for every task on a machine, then sent off-machine to a service where data from related tasks is aggregated. The per-job, per-platform aggregated CPI values are then sent back

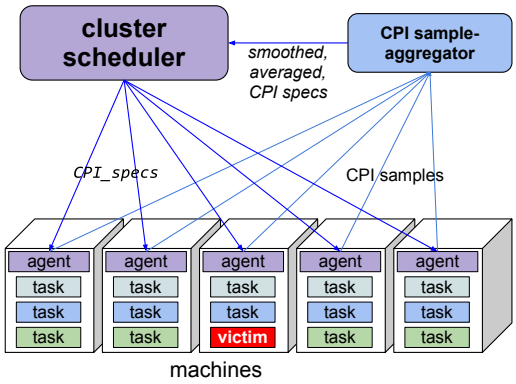

Figure 6: *The CPI^2 data pipeline.*

to each machine that is running a task from that job. Anomalies are detected locally, which enables rapid responses and increases scalability.

CPI sampling CPI data is derived from hardware counters, and is defined as the value of the `CPU_CLK_UNHALTED.REF` counter divided by the `INSTRUCTIONS_RETIRED` counter. These are counted simultaneously, and collected on a per-cgroup basis. (Per-CPU counting wouldn't work because several unrelated tasks frequently timeshare a single CPU (hardware context). Per-thread counting would require too much memory: running thousands of threads on a machine is not uncommon (figure 1b).)

The CPI data is sampled periodically by a system daemon using the `perf_event` tool [13] in counting mode (rather than sampling mode) to keep overhead to a minimum. We gather CPI data for a 10 second period once a minute; we picked this fraction to give other measurement tools time to use the counters. The counters are saved/restored when a context switch changes to a thread from a different cgroup, which costs a couple of microseconds. Total CPU overhead is less than 0.1% and incurs no visible latency impact to our users.

A cluster typically contains several thousand machines; the CPI samples are collected from all the machines by a per-cluster data-gathering system, and sent back to a centralized component for further data aggregation. The data gathered for each task consists of the following fields:

```
string jobname;
string platforminfo; // e.g., CPU type
int64 timestamp;     // microsec since epoch
float cpu_usage;     // CPU-sec/sec
float cpi;
```

CPI data aggregation Most jobs are structured as a set of identical tasks, and their CPIs are similar (see Table 1 for some examples). Although individual latency-sensitive requests may have noticeably different CPIs, these variations are smoothed out over the 10s sample period.

Many of our clusters contain multiple different hardware platforms (CPU types) which will typically have different

Job	CPI	tasks
Job A	0.88 ± 0.09	312
Job B	1.36 ± 0.26	1040
Job C	2.03 ± 0.20	1250

Table 1: *CPI values (mean and standard deviation) of a few representative latency-sensitive jobs, and the number of tasks they contain.*

CPIs for the same workload, so CPI^2 does separate CPI calculations for each platform a job runs on.

Many production jobs run for a long time, so it is straightforward to acquire CPI samples from them to build a model of their behavior. Other jobs run repeatedly, and have similar behavior on each invocation, so historical CPI data has significant value: if we have seen a previous run of a job, we don't have to build a new model of its CPI behavior from scratch.

The data aggregation component of CPI^2 calculates the mean and standard deviation of CPI for each job, which is called its *CPI spec*. This information is updated every 24 hours (we plan to increase the frequency to hourly). The result is the following data for each job/hardware-platform combination:

```
string jobname;
string platforminfo;   // e.g., CPU type
int64 num_samples;
float cpu_usage_mean;  // CPU-sec/sec
float cpi_mean;
float cpi_stddev;
```

Since the CPI changes only slowly with time (see Figure 5), the CPI spec also acts as a predicted CPI for the normal behavior of a job. Significant deviations from that behavior suggest an outlier, which may be worth investigating.

Because the important latency-sensitive applications typically have hundreds to thousands of tasks and run for many days or weeks, it is easy to generate tens of thousands of samples within a few hours, which helps make the CPI spec statistically robust. Historical data about prior runs is incorporated using age-weighting, by multiplying the CPI value from the previous day by about 0.9 before averaging it with the most recent day's data. We do not perform CPI management for applications with fewer than 5 tasks or fewer than 100 CPI samples per task.

4. Identifying antagonists

The process of determining the likely cause of a performance problem proceeds in stages. CPI data for every task in a job is gathered once a minute and compared against the job's predicted CPI. If the observed CPI is significantly larger than the prediction, it is flagged; if this happens often enough for a task, we look for possible correlations with an antagonist. If that succeeds we report an incident and initiate actions to address the situation. The rest of this section describes the details of this process.

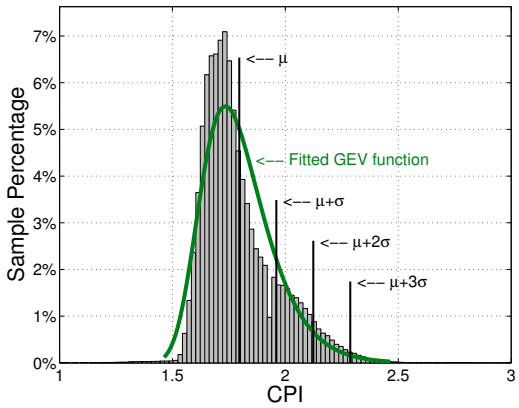

Figure 7: *CPI distribution for a web-search job in a cluster running on thousands of machines of the same type over a 2-day period. The graph includes more than 450k CPI samples and has mean $\mu = 1.8$ and standard deviation $\sigma = 0.16$. We also show the best-fit generalized extreme value curve $GEV(1.73, 0.133, -0.0534)$.*

4.1 Detecting performance anomalies

To avoid a central bottleneck, CPI values are measured and analyzed locally by a management agent that runs in every machine. We send this agent a predicted CPI distribution for all jobs it is running tasks for, as soon as a robust CPI prediction is available, and update it as needed.

Figure 7 shows a measured CPI distribution from a web-search job. The shape has a skewed distribution: the rightmost tail is longer than the leftmost one since bad performance is relatively more common than exceptionally good performance. We fitted the data against normal, log-normal, Gamma, and generalized extreme value (GEV [15]) distributions; the last one fit the best.

A CPI measurement is flagged as an *outlier* if it is larger than the 2σ point on the predicted CPI distribution; this corresponds to about 5% of the measurements. We ignore CPI measurements from tasks that use less than 0.25 CPU-sec/sec since CPI sometimes increases significantly if CPU usage drops to near zero (see case 3 in section 6.1).

To reduce occasional false alarms from noisy data, a task is considered to be suffering *anomalous behavior* only if it is flagged as an outlier at least 3 times in a 5 minute window.

4.2 Identifying antagonists

Once an anomaly is detected on a machine, an attempt is made to identify an antagonist that is causing the performance problem. To prevent the analysis itself from disturbing the system, at most one of these attempts is performed each second.

An active scheme might rank-order a list of suspects based on heuristics like CPU usage and cache miss rate, and temporarily throttle them back one by one to see if the CPI of the victim task improves. Unfortunately, this simple approach may disrupt many innocent tasks. (We'd rather the antagonist-detection system were not the worst antagonist in the system!) Instead, we use a passive method to identify likely culprits by looking for correlations between the victim's CPI values and the CPU usage of the suspects; a good correlation means the suspect is highly likely to be a real antagonist rather than an innocent bystander.

The antagonist correlation is calculated as follows. Suppose we have a time window $[T_1, T_n]$ (we typically use a 10-minute window). Let $\{c_1, c_2, ..., c_n\}$ be CPI samples for the victim V and $c_{threshold}$ be the abnormal CPI threshold for V. Let $\{u_1, u_2, ..., u_n\}$ be the CPU usage for a suspected antagonist A, normalized such that $\sum_1^n u_i = 1$. Set *correlation(V,A)* $= 0$ and then, for each time-aligned pair of samples u_i and c_i:

if $(c_i > c_{threshold})$ then

$$correlation(V,A) \quad += \quad u_i * (1 - \frac{c_{threshold}}{c_i})$$

else if $(c_i < c_{threshold})$ then

$$correlation(V,A) \quad += \quad u_i * (\frac{c_i}{c_{threshold}} - 1).$$

The final correlation value will be in the range [-1, 1]. Intuitively, correlation increases if a spike of the antagonist's CPU usage coincides with high victim CPI, and decreases if high CPU usage by the antagonist coincides with low victim CPI. A single correlation-analysis typically takes about 100μs to perform.

The higher the correlation value, the greater the accuracy in identifying an antagonist (section 7). In practice, requiring a correlation value of at least 0.35 works well.

This algorithm is deliberately simple. It would fare less well if faced with a group of antagonists that together cause significant performance interference, but which individually did not have much effect (e.g., a set of tasks that took turns filling the cache). In future work, we hope to explore other ways of decreasing the number of indeterminate cases, such as by looking at groups of antagonists as a unit, or by combining active measures with passive ones.

5. Dealing with antagonists

What should we do once we have identified one or more antagonists for a victim task? The first thing to note is that the antagonist's performance may also be impacted – i.e., it probably experiences interference from the "victim", which could itself be classified as an antagonist. Our policy is simple: we give preference to latency-sensitive jobs over batch ones.

If the suspected antagonist is a batch job and the victim is a latency-sensitive one, then we forcibly reduce the antagonist's CPU usage by applying CPU hard-capping [37]. This bounds the amount of CPU a task can use over a short period of time (e.g., 25 ms in each 250 ms window, which corresponds to a cap of 0.1 CPU-sec/sec). Performance caps are currently applied for 5 minutes at a time, and we limit the antagonist to 0.01 CPU-sec/sec for low-importance ("best effort") batch jobs and 0.1 CPU-sec/sec for other job types. Google's batch frameworks (e.g., MapReduce) have built-in mechanisms to handle stragglers, so they are already designed to handle this case. The values chosen for the caps and the duration are easily-changed parameters.

CPI^2 will do hard-capping automatically if it is confident in its antagonist selection and the victim job is eligible for protection (e.g., because it is latency-sensitive, or because it is explicitly marked as eligible). When this happens, we expect the victim's performance to improve (i.e., see a lower CPI), but if the victim's CPI remains high, then we return for another round of analysis – presumably we picked poorly the first time. Since throttling the antagonist's CPU reduces it's correlation with the victim's CPI, it is not likely to get picked in a later round of antagonist-identification.

We provide an interface to system operators so they can hard-cap suspects, and turn CPI protection on or off for an entire cluster. Since our applications are written to tolerate failures, an operator may choose to kill an antagonist task and restart it somewhere else if it is a persistent offender – our version of task migration. We don't automatically do this because it would cause additional work (the moved task would have to recompute data since its last checkpoint), and it can take tens of seconds to load a task's binary onto a new machine.

To allow offline analysis, we log and store data about CPIs and suspected antagonists. Job owners and administrators can issue SQL-like queries against this data using Dremel [26] to conduct performance forensics, e.g., to find the most aggressive antagonists for a job in a particular time window. They can use this information to ask the cluster scheduler to avoid co-locating their job and these antagonists in the future. Although we don't do this today, the data could be used to reschedule antagonists to different machines, confine the most offensive ones to a subset of machines, and automatically populate the scheduler's list of cross-job interference patterns.

Table 2 summarizes the parameters of our system. Some are chosen for design convenience (e.g., hard-capping parameters) or because they represent a deliberately conservative policy (e.g., the values we picked for the sampling rates), others (e.g., the antagonist correlation threshold) are based on the experimental evaluation described in section 7.

Parameter	*Value*
Collection granularity	task
Sampling duration	10 seconds
Sampling frequency	every 1 minute
Aggregation granularity	job × CPU type
Predicted CPI recalculated	every 24 hours (goal: 1 hour)
Required CPU usage	≥ 0.25 CPU-sec/sec
Outlier threshold 1	2σ (σ: standard deviation)
Outlier threshold 2	3 violations in 5 minutes
Antagonist correlation threshold	0.35
Hard-capping quota	0.1 CPU-sec/sec
Hard-capping duration	5 mins

Table 2: *CPI^2 parameters and their default values.*

6. Case studies

Because of the way our cluster scheduler places tasks onto machines, each machine is likely to be running a different mix of jobs. This created a natural testbed when we deployed CPI^2 in Google's production clusters. In this section, we present some case studies of the results. At the time of these experiments, most throttling was done manually by our system operators in response to outputs from CPI^2.

6.1 Effectiveness of the antagonist identification algorithm

We present four representative case studies to demonstrate the effectiveness of our antagonist identification algorithm.

Case 1 On 16 May 2011 it was reported that the performance of a latency-sensitive task on a machine was significantly worse than that of 37 other similar tasks on other machines with the same platform type (outlier CPI threshold 2.0). Meanwhile, our system detected a CPI increase for that particular job that began at 2am on the same day, reaching 5.0 at 2:30am. The affected machine had 57 tenants running on it; our antagonist-selection algorithm identified the 5 suspects shown in figure 8(a).

In this example, CPI^2 identified the video-processing antagonist as the one to suppress, because it has the highest correlation and is the only non-latency-sensitive task among the top 5 suspects. In support of this analysis, figure 8(b) shows the CPI of the victim task and the CPU usage of the video-processing task. The two curves match well. In this case, we were early in the deployment of CPI^2, and a system administrator took its advice and killed the video-processing task, after which the victim's performance returned to normal.

Case 2 On 26 Sep 2011 our system detected that the CPI of one of 354 latency-sensitive tasks in a job was consistently exceeding its CPI-outlier threshold (1.7). The victim task was running on a machine with 42 other tenants; the top 5 suspects had CPI correlations of 0.31–0.34, and CPI^2 again picked a best-effort batch job as the one to throttle.

Job	*Type*	*Correlation*
video processing	batch	0.46
content digitizing	latency-sensitive	0.44
image front-end	latency-sensitive	0.43
BigTable tablet	latency-sensitive	0.39
storage server	latency-sensitive	0.39

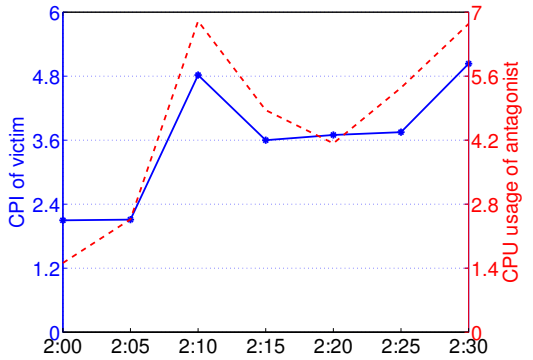

Figure 8: *Case 1: (a) The top 5 antagonist suspects. (b) The CPI of the victim and the CPU usage of the top antagonist.*

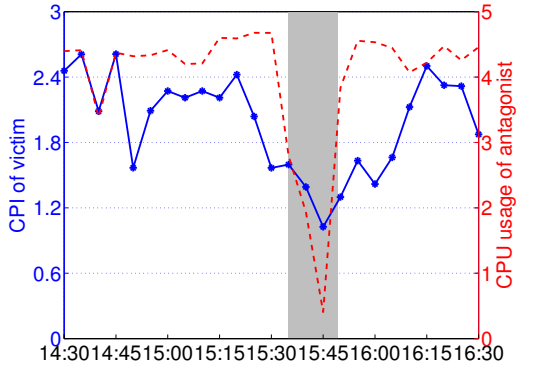

Figure 9: *Case 2: CPI of the victim and CPU usage of the prime suspect antagonist (a best-effort batch job). CPU hard-capping was applied from 15:35 to 15:49 (indicated by the shaded area).*

This time we applied CPU hard-capping to the antagonist for about 15 minutes. As shown in figure 9, the CPU usage of the antagonist was drastically reduced while it was hard-capped, and the victim's CPI improved from about 2.0 to about 1.0. Once the hard-capping stopped and the antagonist was allowed to run normally, the victim's CPI rose again. We conclude that hard-capping is a useful tool, and that CPI^2 had correctly identified the problem's cause.

Case 3 On 26 May 2011 our system detected that the CPI of a front-end web service task was fluctuating from about 3 to about 10. It was running on a machine with 28 other tenants, but the highest correlation value produced by our algorithm was only 0.07, so CPI^2 took no action.

Further investigation revealed that the varying CPI was due to a bimodal CPU usage by the "victim". Figure 10 shows that high CPI corresponds to periods of low CPU usage, and vice versa. This pattern turns out to be normal for this application. The minimum CPU usage threshold described in section 4.1 was developed to filter out this kind of false alarm.

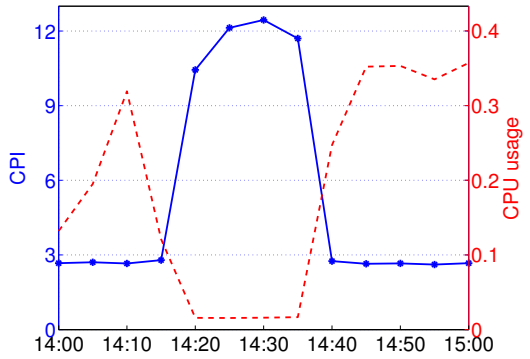

Figure 10: *Case 3: the CPI and CPU usage of the "victim": the CPI changes are self-inflicted.*

Case 4 On 4 Aug 2011 CPI^2 spotted a user-facing service task that consistently crossed its CPI threshold of 1.05 and identified the 9 suspected antagonists shown in figure 11(a).

In this case, only one antagonist was eligible for throttling (scientific simulation), since it was the only non-latency-sensitive task. As shown in figure 11(b), our first attempt to throttle the batch job had barely any effect on the victim's CPI. A second try produced a modest improvement: the victim's CPI dropped from 1.6 to 1.3. The correct response in a case like this would be to migrate the victim to another machine.

In summary, we believe that our correlation technique is a good way to quantify the likelihood of a suspected antagonist being the real culprit, and that hard-capping is an effective way of addressing the underlying problem.

6.2 Antagonists' tolerance to CPU hard-capping

Many best-effort jobs are quite robust when their tasks experience CPU hard-capping: the tasks enter a "lame-duck" mode and offload work to others. Once hard-capping expires, they resume normal execution. Figure 12(a) shows the victim's CPI dropping while the antagonist is throttled, and for a while afterwards; in this case we throttled the antagonist twice. Figure 12(b) shows the behavior of the antagonist. During normal execution, it has about 8 active threads. When it is hard-capped, the number of threads rapidly grows to around 80. After the hard-capping stops, the thread count drops to 2 (a self-induced "lame-duck mode") for tens of minutes before reverting to its normal 8 threads.

On the other hand, some tasks don't tolerate CPU hard-capping, preferring to terminate themselves if their performance drops too far, for too long, in the hope that they will be rescheduled onto a different machine with better performance. Figure 13 shows an example. The throttled antagonist is a task from a MapReduce job that survived the first

Job	*Type*	*Correlation*
a production service	latency-sensitive	0.66
compilation	latency-sensitive	0.63
security service	latency-sensitive	0.58
statistics	latency-sensitive	0.53
data query/analysis	latency-sensitive	0.53
maps service	latency-sensitive	0.43
image render	latency-sensitive	0.37
ads serving	latency-sensitive	0.37
scientific simulation	batch	0.36

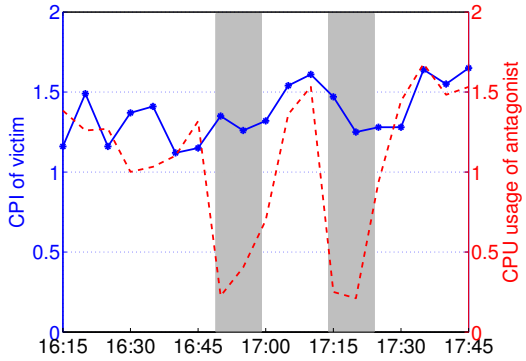

Figure 11: *Case 4: (a) The top 9 antagonist suspects. (b) CPI of the victim and CPU usage of the throttled suspect antagonist (scientific simulation). CPU hard-capping was applied twice: from 16:49 to 16:59 and from 17:14 to 17:24, indicated by shaded area.*

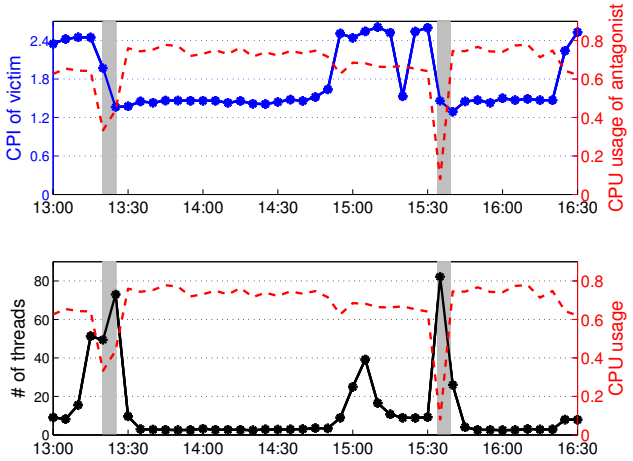

Figure 12: *Case 5: (a) CPU usage of an antagonist (replayer-batch) and CPI of a victim (a query serving service). (b) CPU usage and thread count for the antagonist. Shading shows when the antagonist CPU is hard-capped.*

hard-capping (perhaps because it was inactive at the time) but during the second one it either quit or was terminated by the MapReduce master.

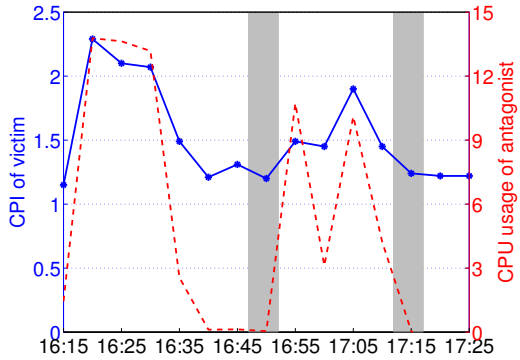

Figure 13: *Case 6: CPU usage of a throttled suspect antagonist (a MapReduce worker) and the CPI of a victim (a latency-sensitive service). The MapReduce batch survived the first throttling (from 16:48 to 16:53, indicated by shaded area) but exited abruptly during the second throttling (from 17:12 to 17:17, indicated by shaded area).*

In our experiments we hard-capped the antagonists to only 0.01 CPU-sec/sec. That may be too harsh; a feedback-driven throttling that dynamically set the hard-capping target would be more appropriate; this is future work. At the other extreme, we have also discussed extending CPI^2 so that if throttling didn't work, it would ask the cluster scheduler to kill and restart an antagonist task on another machine.

7. Large-scale evaluation

The measurement part of CPI^2 has now been rolled out to all of Google's production machines. It is identifying antagonists at an average rate of 0.37 times per machine-day: that's a few thousand times per day in a cluster like the one analyzed here [30].

At the time of writing, the enforcement part of CPI^2 is not widely deployed (due to a deliberately conservative rollout policy), so to evaluate what enforcement would do if it were more widely deployed, we periodically look for recently-reported antagonists and manually cap their CPU rate for 5 minutes, and examine the victim's CPI to see if it improves. We collected data for about 400 such trials and present our analysis below.

7.1 Is antagonism correlated with machine load?

It might be thought that antagonists occur more frequently, and have larger effects, on overloaded machines. However, our data does not support this.

Figure 14 shows machine CPU utilization and victim CPI relative to the job's mean at the moment when an antagonist was reported. Antagonism is not correlated with machine load: it happens fairly uniformly at all utilization levels and

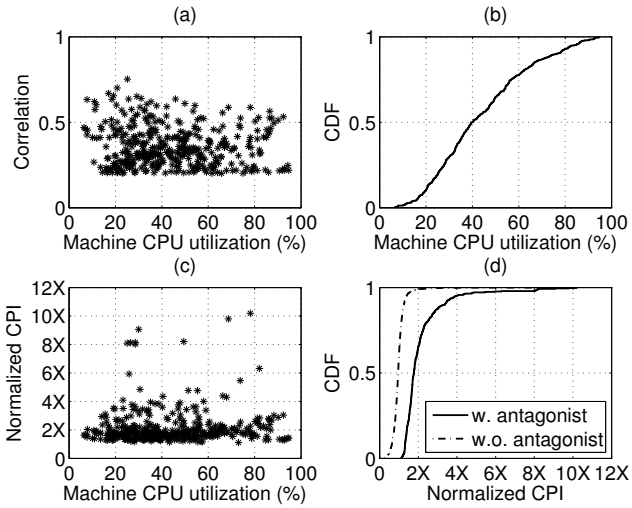

Figure 14: *CPU utilization, antagonist detection, and increase in CPI. (a) Calculated antagonist correlation versus observed CPU utilization. (b) CDF of observed CPU utilization on the machine. (c) Observed victim CPI divided by the job's mean CPI versus the observed CPU utilization. (d) CDFs of observed CPI divided by the job's mean CPI, in cases where an antagonist was identified and when no antagonist could be identified. All but graph (d) show data points sampled at the time when an antagonist was detected.*

the extent of damage to victims is also not related to the utilization.

Figure 14(d) shows CDFs of CPIs when an antagonist was reported versus when no antagonist was reported. It shows that CPI^2 is indeed capturing cases where CPI has increased and that the increase has quite a long tail.

7.2 Accuracy of antagonist identification

To evaluate the quality of our algorithm to detect antagonists, we look at the true and false positive rates for the detector. To do this, we compare the victim's CPI when an antagonist was first reported and the victim CPI that resulted when the antagonist was throttled. If the latter is smaller (by some margin) than the former, we say it is a true positive; if the latter is larger by the same margin, we say it is a false positive; any other case is considered noise. A natural choice of margin is the standard deviation (*cpi_stddev*) in the CPI spec.

Our workload is divided into two priority bands (production and non-production), and we break down our results the same way in Figure 15. The production jobs show a much better true positive rate than non-production ones. We think this is because non-production jobs' behaviors are less uniform (e.g., engineers testing experimental features) so it is harder to identify performance fluctuations due to interference.

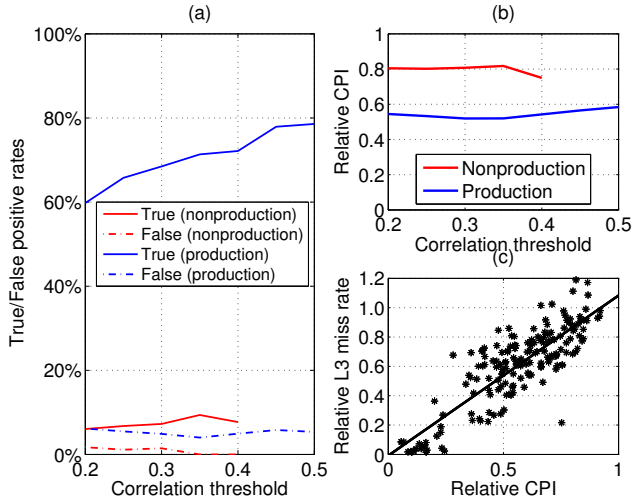

Figure 15: *Antagonist-detection accuracy for all jobs. (a) Detection rates versus the antagonist correlation threshold value. (b) Observed* relative *CPI for true-positive cases versus antagonist correlation. (c) The* relative *L3 cache misses/instruction versus relative CPI for the true-positive cases. A relative rate is calculated by dividing the rate during throttling by the rate seen before throttling begins. We did not collect antagonist correlation values larger than 0.4 for non-production jobs in (a) and (b).*

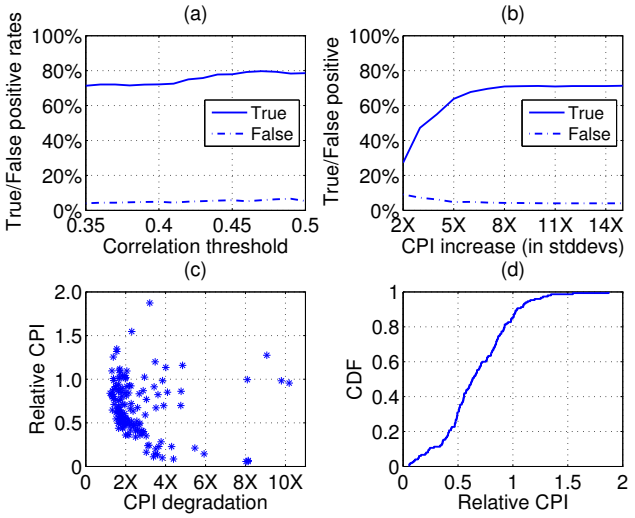

Figure 16: *Antagonist-detection accuracy and CPI improvement for production jobs. (a) Detection rates versus the antagonist correlation threshold value. (b) Detection rates versus how much the CPI increases, expressed in standard deviations. (c) Observed relative victim CPI (see Figure 15) versus the victim's CPI degradation (CPI before throttling divided by the job's mean CPI). (d) CDF of victim's relative CPI. The antagonist correlation threshold is 0.35 in (b), (c) and (d).*

Based on these results, declaring an antagonist only when the detector correlation is 0.35 or above seems a good threshold. Using it, throttling just the single most-suspected antagonist reduces the victim CPI to 0.82× (non-production jobs) and 0.52× (production jobs) its pre-throttling value in the case of a true positive (Figure 15(b)).

We looked at correlations between CPI improvement and several memory metrics such as L2 cache misses/instruction, L3 misses/instruction, and memory-requests/cycle, and found that L3 misses/instruction shows strongest correlation (with 0.87 linear correlation coefficient in Figure 15(c)).

Figure 16(a) shows a true positive rate in identifying the right antagonist of ∼70% for production jobs. This is independent of the antagonist correlation value when it is above a threshold of 0.35. Figure 16(b) suggests that an anomalous event should not be declared until the victim has a CPI that is at least 3 standard deviations above the mean.

7.3 Benefits to victim jobs

A victim's *relative CPI* (the CPI during throttling divided by the CPI before it) provides a useful measure of how much the victim's performance is improved by throttling. Figure 16(c) shows that it is significantly lower than 1 across a wide range of CPI degradation values (CPI before throttling divided by mean CPI).

Figure 16(d) shows that the median victim production job's CPI is reduced to 0.63× it's pre-throttling value when throttling the top antagonist suspect, including both true and false positive cases.

8. Related Work

The pure-software approach taken by CPI^2 complements work in the architecture community on cache usage monitoring and partitioning (e.g., [35, 36, 9, 42, 4, 32]). It has one major advantage: we can deploy it now, on existing commodity hardware.

CPI^2 is part of a larger body of work on making the performance of applications in shared compute clusters more predictable and scalable. For example, Q-cloud [28] aims to provide a QoS-aware cloud for applications by profiling applications' performance in a standalone mode and using that to provide a baseline target when consolidating them onto a shared host.

Alameldeen et al. [1] argue that IPC (the inverse of CPI) is not a good performance metric because changes in instruction count might have little effect on the amount of useful work a user program actually accomplishes. This was not the case for our production jobs. Where CPI^2 uses CPI increases to indicate conflicts, other work has used application-level metrics, which may be more precise but are less general and may require application modifications. For example, Mantri [3] looks for MapReduce stragglers, which it identifies using progress reported by the MapReduce job, so that it can du-

plicate or restart them. PRESS and CloudScale [16, 34] use a combination of application-level signals and CPU usage data to identify and control usage-driven performance interference. We do not have a universal definition of a standard application transaction, but even we did, it would be a large effort to change all the user programs to report against it.

Google-Wide Profiling (GWP) [31] gathers performance-counter sampled profiles of both software and hardware performance events on Google's machines. It is active on a tiny fraction of machines at any time, due to concerns about the overhead of profiling. In contrast, CPI^2 uses hardware performance counters in counting mode, rather than sampling, which lowers the cost of profiling enough that it can be enabled on every shared production machine at Google at all times.

HiTune [10] uses similar instrumentation techniques to GWP, but focuses on building a high-level dataflow-based model of application behavior. It helps application developers identify problems in their deployments, but it doesn't automatically identify antagonists and deal with them.

Mars et al. [23] focused on detecting when cache contention happens, rather than who causes contention. We go further by selecting an antagonist out of tens of candidates, and our solution applies whatever the type of CPU resource contention.

Kambadur et al. [21] collected per-CPU samples at millisecond granularity and analyzed interference offline. Our per-task CPI samples are aggregated over a few seconds and antagonist identification is conducted online.

CPI^2 uses hard-capping to control antagonists, motivated by observations that adjusting CPU scheduling can achieve fair cache sharing among competing threads [14]. An alternative would be to use hardware mechanisms like duty-cycle modulation [41]. This offers fine-grain control of throttling (in microseconds by hardware gating rather than milliseconds in the OS kernel scheduler), but it is Intel-specific and operates on a per-core basis, forcing hyper-threaded cores to the same duty-cycle level, so we chose not to use it.

Similarly, CPI^2 consciously uses a simple metric (CPI) for its input, rather than probing for the root cause. This could certainly be done: for example, Zhang et al. [40] used memory reference counts to approximate memory bandwidth consumption on SMP machines; West et al. [38] used cache miss and reference counts to estimate cache occupancy of competing threads on multicore machines; VM^3 [20] profiled applications' cache-misses per instruction to estimate effective cache sizes in a consolidated virtual machine environment; Cuanta [17] introduced a cache loader micro-benchmark to profile application performance under varying cache-usage pressure; and Blagodurov [7] and Zhuravlev [43] applied heuristics based on cache miss rates to guide contention-aware scheduling. Koh et al. [22] and Matthews et al. [24] studied performance interference of co-hosting multiple VMs on a single physical machine. Instead, CPI^2 focuses on managing the effects of interference and leaves detailed diagnostics to other tools.

There are many potential causes of contention. For example, Barker et al. [5] studied interference due to background disk and network loads in an Amazon EC2 cloud [2] and found performance degradation can be up to 75% for latency-sensitive applications. CPI^2 focuses just on CPU; it could usefully be complemented by approaches that handle other shared resources.

TRACON [8] uses statistical machine learning techniques to predict interference of data-intensive applications, which it uses to guide placement decisions. Its models are trained by driving applications with a workload generator; CPI^2 uses data from application tasks in production to build its models.

9. Future Work

Disk and network I/O conflicts are managed at Google using mechanisms outside the scope of this paper, but the idea of correlation-based antagonist identification could be applied to this I/O realm as well.

Our cluster scheduler will not place a task on the same machine as a user-specified antagonist job, but few users manually provide this information. In the future, we hope to provide this information to the scheduler automatically.

Our fixed hard-capping limits are rather crude. We hope to introduce a feedback-driven policy that dynamically adjusts the amount of throttling to keep the victim CPI degradation just below an acceptable threshold. Other amelioration techniques like page coloring, hardware-based cache partitioning, and affinity-based placement may also be valuable directions to explore.

10. Conclusion

We have presented the design, implementation, and evaluation of CPI^2, a CPI-based system for large clusters to detect and handle CPU performance isolation faults. We showed that CPI is a reasonable performance indicator and described the data-gathering pipeline and local analyses that CPI^2 performs to detect and ameliorate CPU-related performance anomalies, automatically, using CPU hard-capping of antagonist tasks.

We demonstrated CPI^2's usefulness in solving real production issues. It has been deployed in Google's fleet. The beneficiaries include end users, who experience fewer performance outliers; system operators, who have a greatly reduced load tracking down transient performance problems; and application developers, who experience a more predictable deployment environment.

In future work, we will be exploring adaptive throttling and making job placement antagonist-aware automatically. Even before these enhancements are applied, we believe that CPI^2 is a powerful, useful tool.

Acknowledgements

This work would not have been possible without the help and support of many colleagues at Google. In particular, the data pipeline was largely built by Adam Czepil, Paweł Stradomski, and Weiran Liu. They, along with Kenji Kaneda, Jarek Kusmierek, and Przemek Broniek, were involved in many of the design discussions. We are grateful to Stephane Eranian for implementing per-cgroup performance counts in Linux, and to him and David Levinthal for their help on capturing performance counter data. We also thank Paul Turner for pointing us to Linux CPU bandwidth control.

References

[1] ALAMELDEEN, A. R., AND WOOD, D. A. IPC considered harmful for multiprocessor workloads. *IEEE Micro 26*, 4 (July 2006), 8–17.

[2] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`, 2008.

[3] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Nov. 2010).

[4] AWASTHI, M., SUDAN, K., BALASUBRAMONIAN, R., AND CARTER, J. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proc. Int'l Symp. on High Performance Computer Architecture (HPCA)* (Raleigh, NC, Feb. 2009).

[5] BARKER, S. K., AND SHENOY, P. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proc. 1st ACM Multimedia Systems (MMSys)* (Phoenix, AZ, Feb. 2010).

[6] BARROSO, L. A., DEAN, J., AND HOLZLE, U. Web search for a planet: the Google cluster architecture. In *IEEE Micro* (2003), pp. 22–28.

[7] BLAGODUROV, S., ZHURAVLEV, S., DASHTI, M., AND FEDOROVA, A. A case for NUMA-aware contention management on multicore systems. In *Proc. USENIX Annual Technical Conf. (USENIX ATC)* (Portland, OR, June 2011).

[8] CHIANG, R. C., AND HUANG, H. H. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)* (Seattle, WA, Nov. 2011).

[9] CHO, S., AND JIN, L. Managing distributed, shared L2 caches through OS-level page allocation. In *Proc. Int'l Symp. on Microarchitecture (Micro)* (Orlando, FL, Dec. 2006), pp. 455–468.

[10] DAI, J., HUANG, J., HUANG, S., HUANG, B., AND LIU, Y. HiTune: Dataflow-based performance analysis for big data cloud. In *Proc. USENIX Annual Technical Conf. (USENIX ATC)* (Portland, OR, June 2011).

[11] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (Feb. 2012), 74–80.

[12] DEAN, J., AND GHEMAWAT, S. MapReduce: simplified data processing on large clusters. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004), pp. 137–150.

[13] ERANIAN, S. perfmon2: the hardware-based performance monitoring interface for Linux. `http://perfmon2.sourceforge.net/`, 2008.

[14] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (Brasov, Romania, Sept. 2007), pp. 25–36.

[15] Wikipedia: Generalized extreme value distribution. `http://en.wikipedia.org/wiki/Generalized_extreme_value_distribution`, 2011.

[16] GONG, Z., GU, X., AND WILKES, J. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proc. 6th IEEE/IFIP Int'l Conf. on Network and Service Management (CNSM 2010)* (Niagara Falls, Canada, Oct. 2010).

[17] GOVINDAN, S., LIU, J., KANSAL, A., AND SIVASUBRAMANIAM, A. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proc. ACM Symp. on Cloud Computing (SoCC)* (Cascais, Portugal, Oct. 2011).

[18] Apache Hadoop Project. `http://hadoop.apache.org/`, 2009.

[19] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. European Conf. on Computer Systems (EuroSys)* (Lisbon, Portugal, Apr. 2007).

[20] IYER, R., ILLIKKAL, R., TICKOO, O., ZHAO, L., APPARAO, P., AND NEWELL, D. VM3: measuring, modeling and managing VM shared resources. In *Computer Networks* (Dec. 2009), vol. 53, pp. 2873–2887.

[21] KAMBADUR, M., MOSELEY, T., HANK, R., AND KIM, M. A. Measuring interference between live datacenter applications. In *Proc. Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)* (Salt Lake City, UT, Nov. 2012).

[22] KOH, Y., KNAUERHASE, R., BRETT, P., BOWMAN, M., WEN, Z., AND PU, C. An analysis of performance interference effects in virtual environments. In *Proc. IEEE Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)* (San Jose, CA, Apr. 2007).

[23] MARS, J., VACHHARAJANI, N., HUNDT, R., AND SOFFA, M. L. Contention aware execution: online contention detection and response. In *Int'l Symposium on Code Generation and Optimization (CGO)* (Toronto, Canada, Apr. 2010).

[24] MATTHEWS, J. N., HU, W., HAPUARACHCHI, M., DESHANE, T., DIMATOS, D., HAMILTON, G., MCCABE, M., AND OWENS, J. Quantifying the performance isolation properties of virtualization systems. In *Proc. Workshop on Experimental Computer Science* (San Diego, California, June 2007).

[25] MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. Power management of online data-intensive services. In *Proc. Int'l Symposium on Computer Architecture (ISCA)* (San Jose, CA, June 2011).

[26] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)* (Singapore, Sept. 2010), pp. 330–339.

[27] MENAGE, P. Linux control groups. `http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, 2007.

[28] NATHUJI, R., KANSAL, A., AND GHAFFARKHAH, A. Q-Clouds: managing performance interference effects for QoS-aware clouds. In *Proc. European Conf. on Computer Systems (EuroSys)* (Paris, France, Apr. 2010).

[29] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign language for data processing. In *Proc. ACM SIGMOD Conference* (Vancouver, Canada, June 2008).

[30] REISS, C., TUMANOV, A., GANGER, G., KATZ, R., AND KOZUCH, M. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proc. ACM Symp. on Cloud Computing (SoCC)* (San Jose, CA, Oct. 2012).

[31] REN, G., TUNE, E., MOSELEY, T., SHI, Y., RUS, S., AND HUNDT, R. Google-Wide Profiling: a continuous profiling infrastructure for data centers. *IEEE Micro*, 4 (July 2010), 65–79.

[32] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: scalable and efficient fine-grain cache partitioning. In *Proc. Int'l Symposium on Computer Architecture (ISCA)* (San Jose, CA, 2011).

[33] SCHURMAN, E., AND BRUTLAG, J. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Proc. Velocity, Web Performance and Operations Conference* (2009).

[34] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proc. ACM Symp. on Cloud Computing (SoCC)* (Cascais, Portugal, Oct. 2011).

[35] SUH, G. E., DEVADAS, S., AND RUDOLPH, L. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. Int'l Symp. on High Performance Computer Architecture (HPCA)* (Boston, MA, Feb 2002).

[36] SUH, G. E., RUDOLPH, L., AND DEVADAS, S. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing 28* (2004), 7–26.

[37] TURNER, P., RAO, B., AND RAO, N. CPU bandwidth control for CFS. In *Proc. Linux Symposium* (July 2010), pp. 245–254.

[38] WEST, R., ZAROO, P., WALDSPURGER, C. A., AND ZHANG, X. Online cache modeling for commodity multicore processors. *Operating Systems Review 44*, 4 (Dec. 2010).

[39] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving MapReduce performance in heterogeneous environments. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008).

[40] ZHANG, X., DWARKADAS, S., FOLKMANIS, G., AND SHEN, K. Processor hardware counter statistics as a first-class system resource. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS)* (San Diego, CA, May 2007).

[41] ZHANG, X., DWARKADAS, S., AND SHEN, K. Hardware execution throttling for multi-core resource management. In *Proc. USENIX Annual Technical Conf. (USENIX ATC)* (Santa Diego, CA, June 2009).

[42] ZHAO, L., IYER, R., ILLIKKAL, R., MOSES, J., NEWELL, D., AND MAKINENI, S. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (Brasov, Romania, Sept. 2007), pp. 339–352.

[43] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Managing contention for shared resources on multicore processors. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Pittsburgh, PA, Mar. 2010), pp. 129–142.