CPI^2 : CPU performance isolation for shared compute clusters

1. Xiao Zhang
2. Eric Tune
3. Robert Hagmann
4. Rohit Jnagal
5. Vrigo Gokhale
6. John Wilkes

Class Presentation : Siddhartha Biswas

Abstract:

1. Performance isolation is a key challenge in cloud computing.
2. Linux has few defenses against performance interference in shared resources such as processor caches and memory buses.

Result : Applications experience unpredictable performance for other programs.

Solution : **CPI^2** - **CPU performance isolation** - Using **cycles-per-instruction** (CPI) data from hardware performance counters to

A. Identify Problems.
B. Select the likely perpetrators.
C. Throttle the perpetrators (Optionally).
D. Helping the victim to return to their expected behavior.

Introduction:

1. Google's compute clusters share machine between applications to increase resource utilization .
2. Most Google machine run multiple tasks.
3. Interference can occur in any processor that is shared between threads of different jobs.
4. This interference can negatively affect the performance of latency sensitive applications.
5. Performance isolation in Linux is limited.

No of tasks in standard Google machine

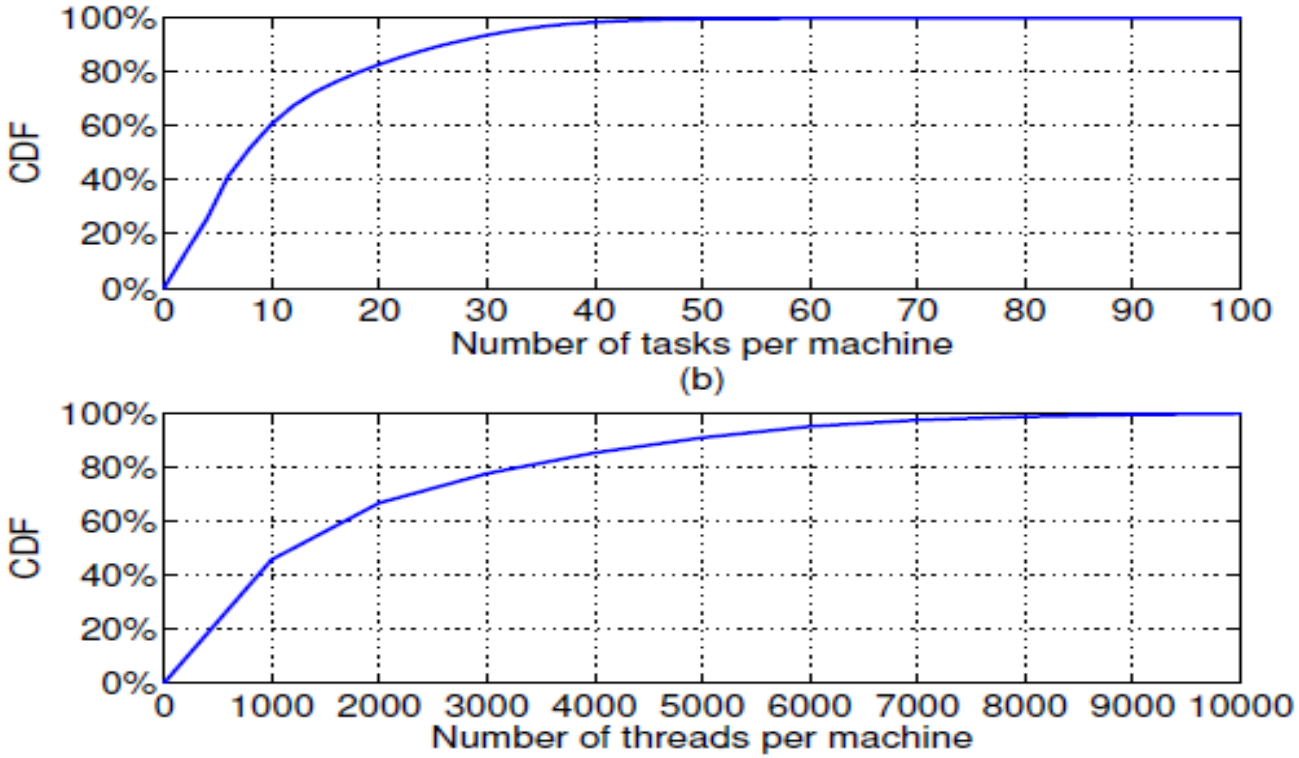

Figure 1: *The number of tasks and threads running on a machine, as cumulative distribution functions (CDFs).*

High probability of getting interference because of shared hardware.

Solving interference problem by statistical approach:

1. Google's compute clusters run thousands of similar tasks .
2. Find statistical performance of each task (**CPI^2**).
3. Need to find performance outliers among them (**Victim**) .
4. Need to reduce the interference on them from other tasks (**Antagonist**) .
6. Determining which antagonist is the likely cause and throttle it.
5. Checking new performance and continue the same procedure over time .

CPI as a metric:

1. Cycles per instruction (CPI) is used as a performance indicator for detecting interference.

2. CPI can be measured directly from existing hardware and does not require application level input.

Concerns about CPI (as a metric):

1. CPI might not be well correlated with application-level behavior.

2. Instructions required to accomplish a fixed amount of work may vary between tasks of the same job, or over time in one task . Will CPI be proper performance indicator for these tasks?
   - It was found not an issue , in practice.

3. CPI only shows a symptom, not the root cause.
   - True. But treating symptoms can restore good performance.

4. CPI doesn't measure network or disk interference effects.
   - True. Other techniques required to detect I/O interference

CPI might not be well correlated with application-level behavior.

Observation – It wil show correct beha-viour (Batch job).

Correlation between TPS & IPS is about 97%.

IPS =
CPU Cycle Speed / CPI

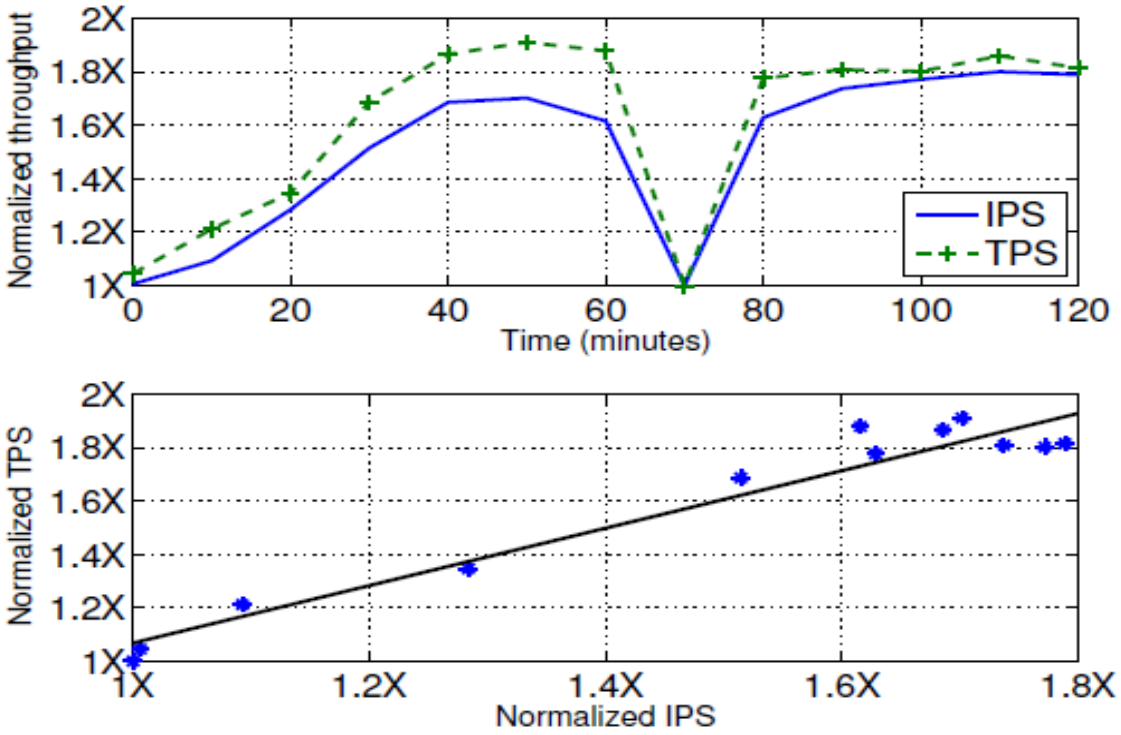

Figure 2: *Normalized application transactions per second (TPS) and instructions per second (IPS) for a representative batch job: (a) normalized rates against running time; (b) scatter plot of the two rates, which have a correlation coefficient of 0.97. Each data point is the mean across a few thousand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

CPI might not be well correlated with application-level behavior.

Observation – It will show correct beha-viour (Latency Sensitive Application).

Correlation between CPI & Request Latency is about 97% .

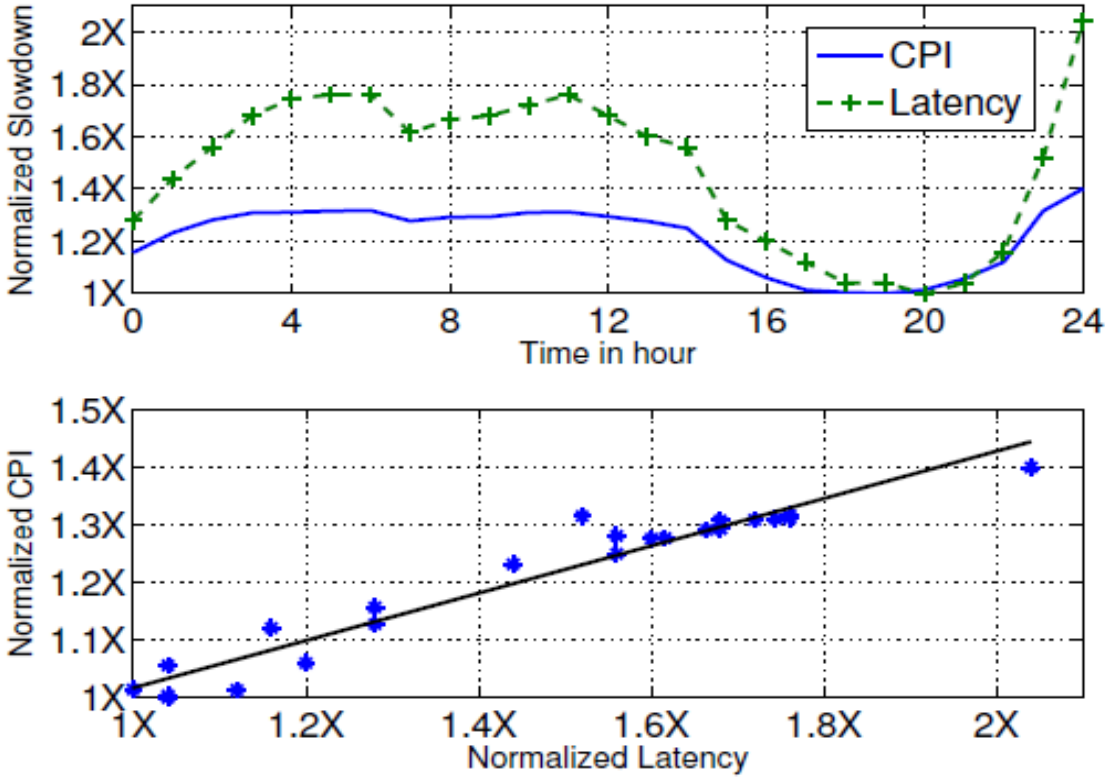

Figure 3: *Normalized application request latency and CPI for a leaf node in a user-facing web-search job: (a) request latency and CPI versus time; (b) request latency versus CPI; the correlation coefficient is 0.97. The latency is reported by the search job; the CPI is measured by CPI^2. The results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

CPI is a function of Hardware Platform:

A . Computation intensive application.

B. Computation intensive application.

C. I/O dependent application

Observation:
Job C shows poor correlation because CPI does not capture I/O behavior.

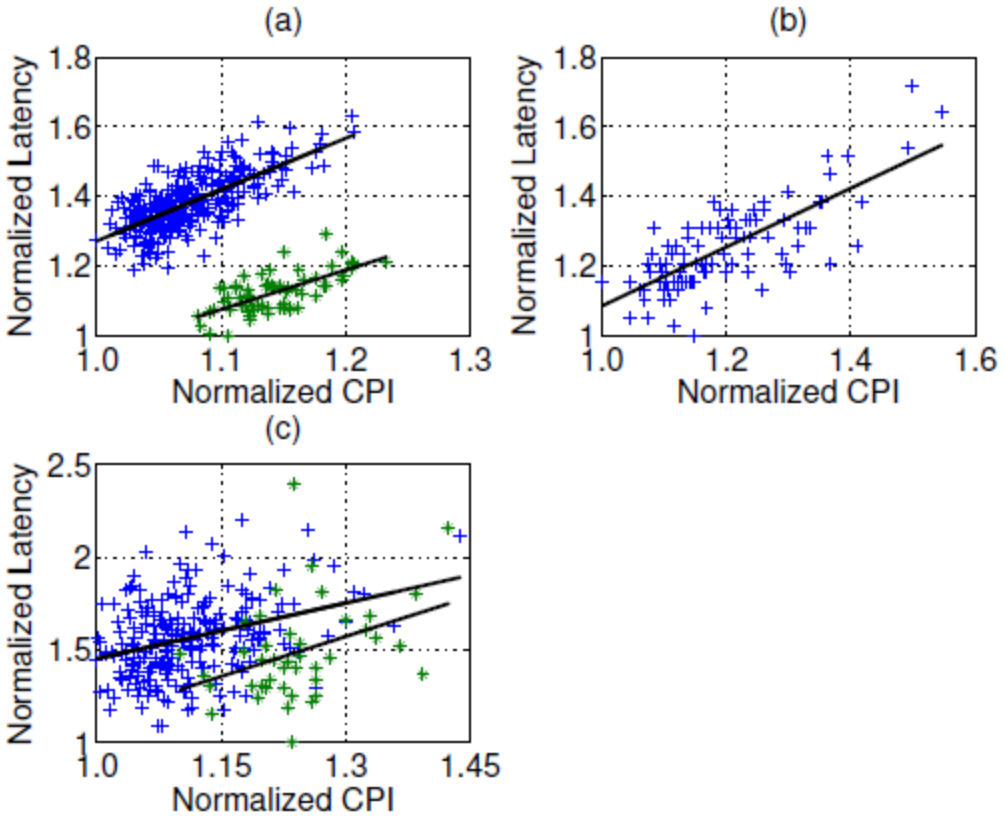

Figure 4: *Normalized request latency and CPI of tasks in three web-search jobs: (a) a leaf node; (b) an intermediate node; (c) a root node. Each point represents a 5-minute sample of a task's execution. Different colors indicate different hardware platforms.*

CPI changes slowly over time as the instruction mix that gets executed changes.

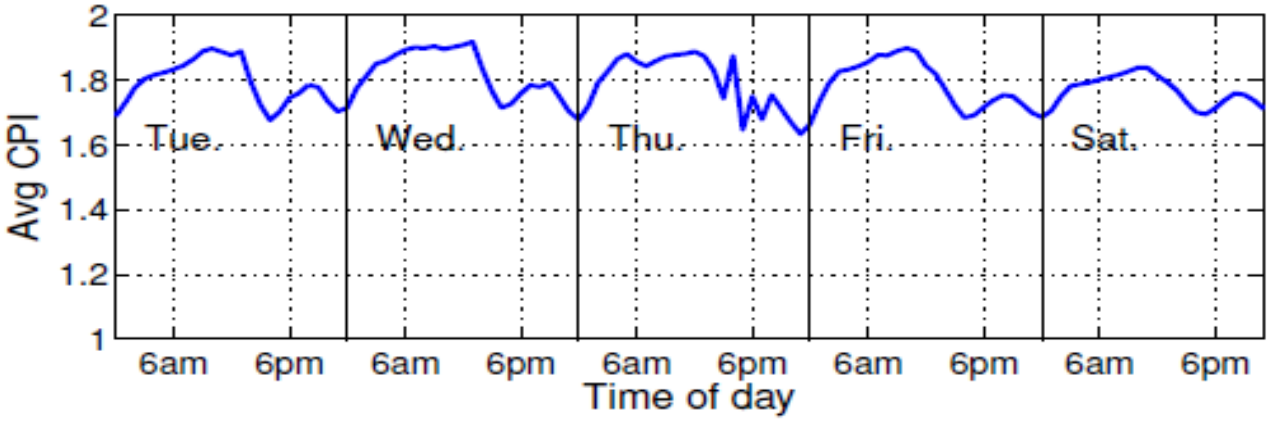

Figure 5: *Average CPI across thousands of web-search leaf tasks over time. The first day is 2011.11.01.*

Observation:

1. CPI of a web search job over five days.
2. Almost same pattern everyday .
3. Only 4% coefficient of variation (standard deviation divided by mean).

Conclusion (CPI as a metric) :

1. Positive correlation between changes in CPI and changes in compute intensive application.

2. CPI is reasonably stable measure over time.

Collecting CPI Data:

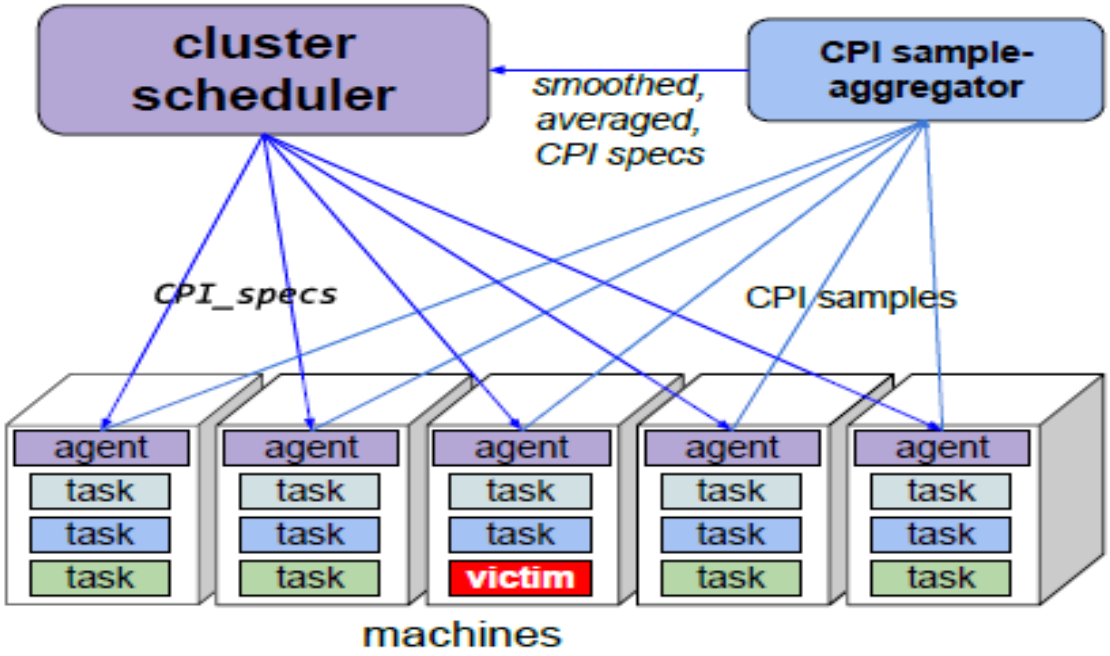

Figure 6: *The CPI^2 data pipeline.*

1.CPI is gathered for every task on a machine.
2.Collected data is sent to a service where data for related task is aggregated .
3. Per job, per-platform aggregated CPI is sent back to each machine.
4. Anomalies are detected locally which enables rapid response.

CPI Sampling:

1.CPI data is derived from hardware counters.

2.CPI = (CPU CLK UNHALTED.REF counter / INSTRUCTIONS RETIRED counter) .

3. Data is collected per Cgroup basis.

4.CPI data is sampled periodically – usually 10 second period a minute.

CPI data aggregation:

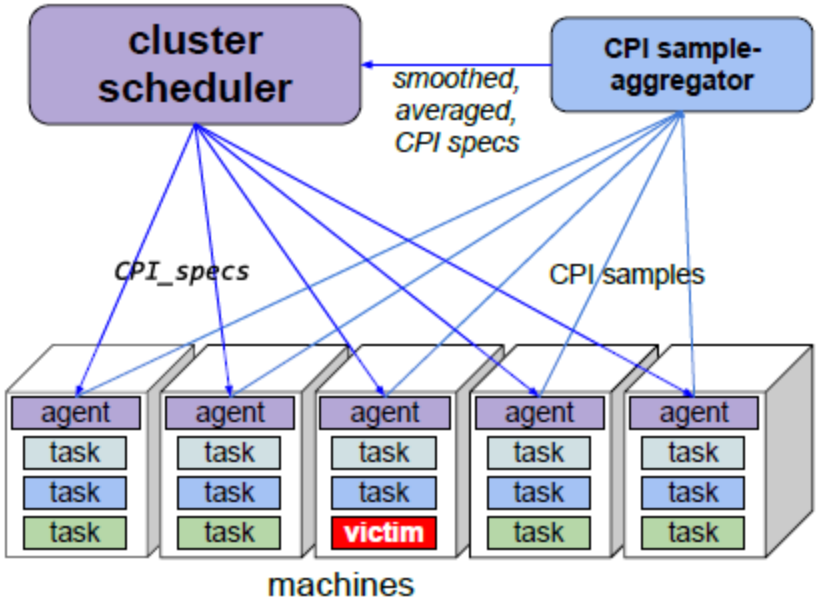

Figure 6: *The CPI^2 data pipeline.*

1.The data aggregation component of CPI^2 calculates the **mean** and **standard deviation** of each job's CPI – called CPI spec.

2.Information is updated every 24 hours .

3.Since CPI changes with time very slowly , CPI spec acts like predicted CPI.

Identifying antagonists:

1.CPI values are measured and analyzed locally by a management agent that runs in every machine.

2.A predicted CPI distribution is provided to this management agent .

3.A CPI measurement is flagged as an outliner if it is larger than the 2 times of standard deviation point of predicted CPI distribution.

4.Tasks which take less than 0.25 CPU-sec/ sec are also ignored because default CPI value for these tasks are very high.

5.A list of suspects is made from the other high CPU usage tasks.

6.Correlation is checked between the **victim's CPI value** and **Antagonist's CPU usage.**

7.A good correlation means the suspect is highly likely to be a real antagonist – higher the correlation value (near to 1), the greater the accuracy in identifying an antagonist . This value is **> 0.35** in practice.

Dealing with antagonists:

1.Find the first job from the list of jobs which has the biggest correlation with victim.

2.Forcibly reduce antagonist's CPU usage by applying CPU hard-capping.

3.Check the victim's performance whether it is improved or not?

4.If yes- then kill the current antagonist .

5.If performance of victim is not improved , do second round of same checking.

Case Study : Effectiveness of the antagonist identification algorithm:

Case 1:

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

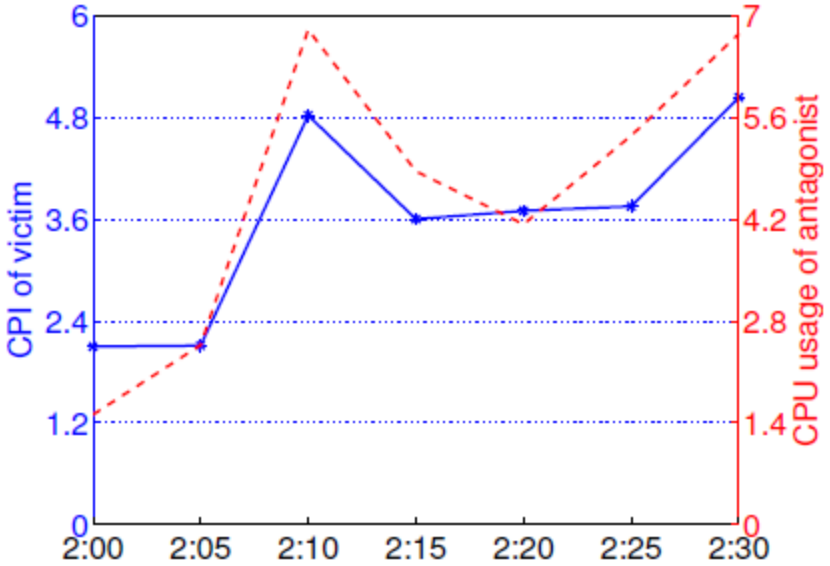

Figure 8: *Case 1: (a) The top 5 antagonist suspects. (b) The CPI of the victim and the CPU usage of the top antagonist.*

Case Study : Effectiveness of the antagonist identification algorithm:

Case 2:

Observation:
15 minute CPU hard capping was done here to check the victim's performance.

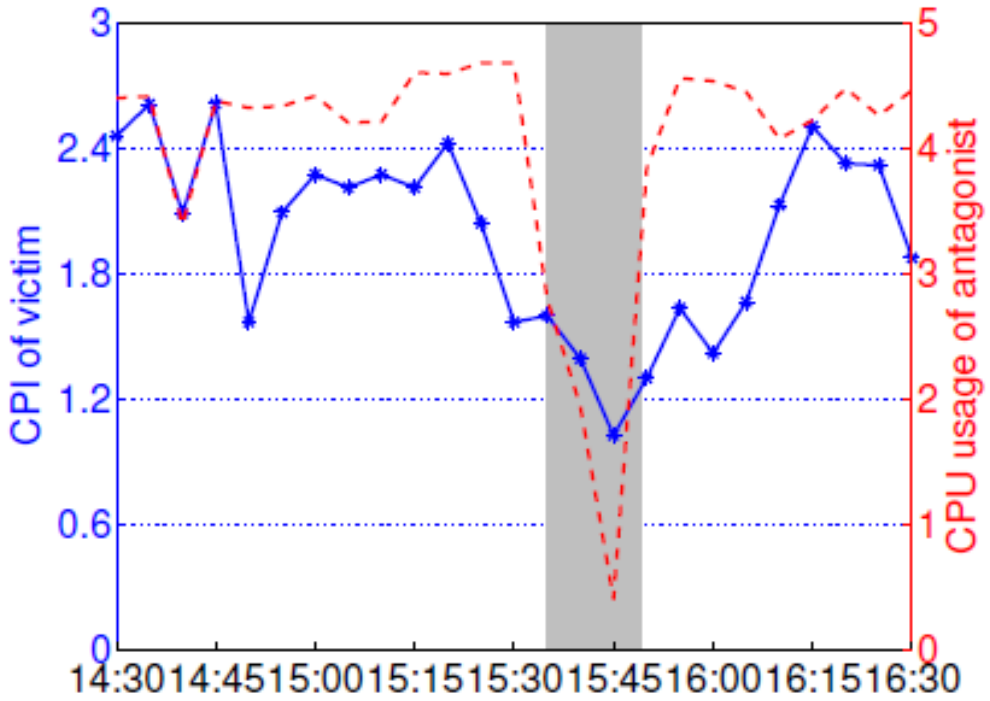

Figure 9: *Case 2: CPI of the victim and CPU usage of the prime suspect antagonist (a best-effort batch job). CPU hard-capping was applied from 15:35 to 15:49 (indicated by the shaded area).*

Case Study : Effectiveness of the antagonist identification algorithm:

Case 3:

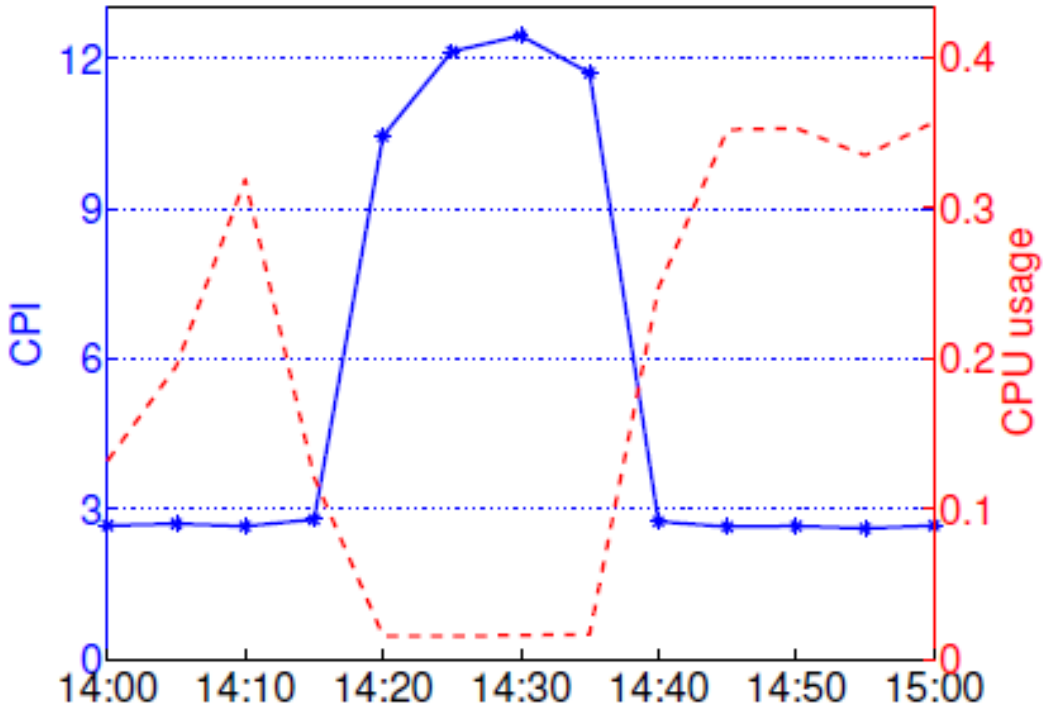

Figure 10: *Case 3: the CPI and CPU usage of the "victim": the CPI changes are self-inflicted.*

Large Scale Evaluation:

Is antagonism correlated with machine load? No.

Observation:

1. Correlation > 0.35 for various loads (distributed evenly).

2.High CPI for machines with low CPU utilization.

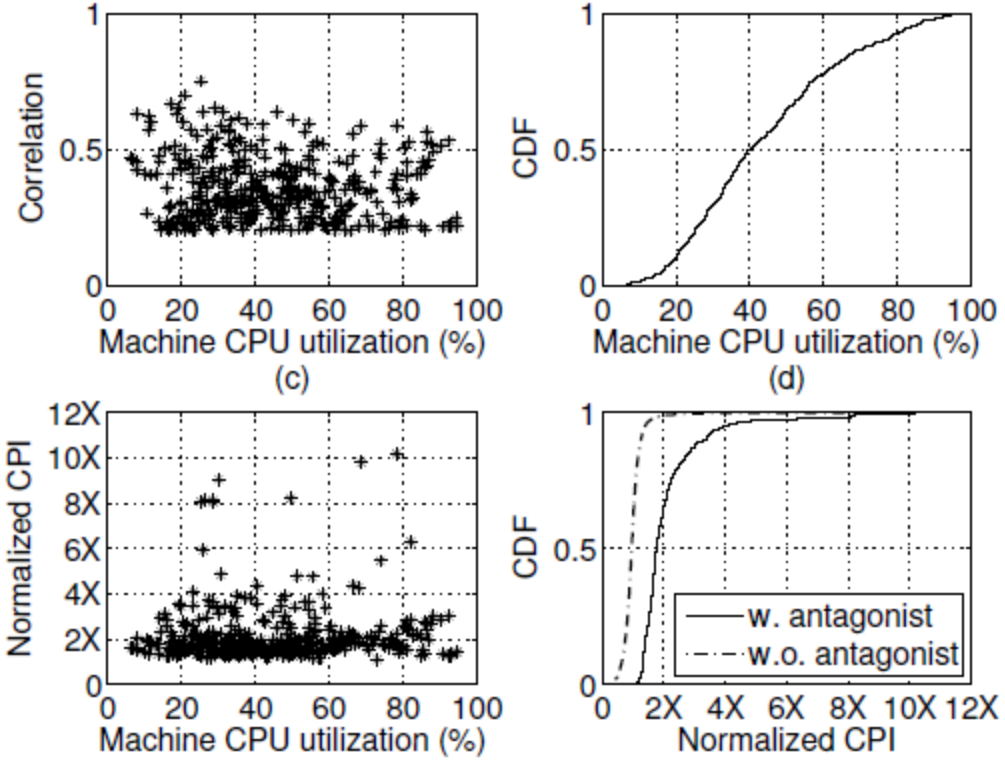

Figure 14: *CPU utilization, antagonist detection, and increase in CPI. (a) Calculated antagonist correlation versus observed CPU utilization. (b) CDF of observed CPU utilization on the machine. (c) Observed victim CPI divided by the job's mean CPI versus the observed CPU utilization. (d) CDFs of observed 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.*

Large Scale Evaluation:

Benefits to victim jobs ? Yes.

Observation:

1. Relative CPI is < 1 in most cases.
   Relative CPI = CPI after throttling / Actual CPI .

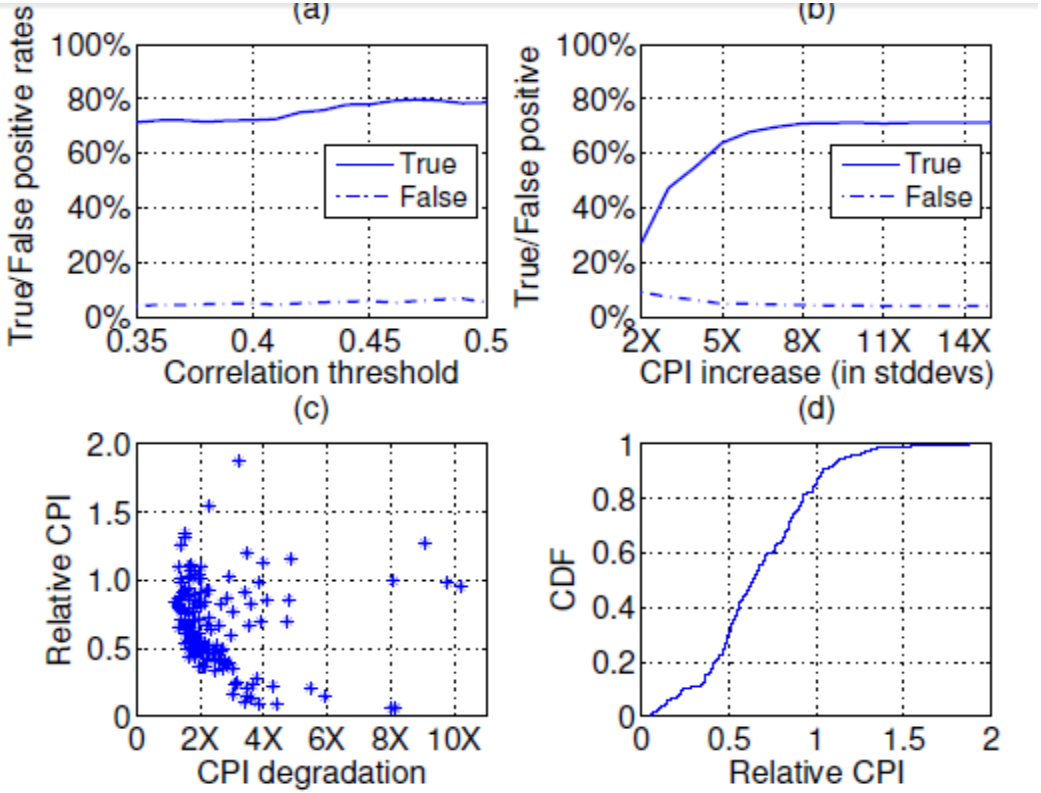

Figure 16: *Antagonist-detection accuracy and CPI improvement for production jobs. (a) Detection rates versus the antagonist correlation threshold value. (b) Detection rates versus how much the CPI increases, expressed in standard deviations. (c) Observed relative victim CPI (see Figure 15) versus the 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).*

Related Work:

1. Pure software approach taken by CPI^2 complements work in the architecture community on cache monitoring and partitioning. But CPI^2 is deployable in existing hardware.

2. CPI2 is a larger body of work on making performance of applications in shared computer clusters more predictable : Q-cloud is such a system which aims to provide QoS to cloud computing applications.

3. Where CPI2 uses CPI increases to indicate conflicts, there are other related works which use application level metrics, which is more precise than CPI, but less general and need application modification.

4. Google-Wide Profiling gathers performance counter sampled profile of both hardware and software performance events, but it is enabled only for a tiny fraction of a second in order to reduce overhead of profiling.

Future Work:

1. Disk and network I/O conflicts can be resolve by correlation-based antagonist identification.

2. Exploring adaptive throttling and making job placement antagonist-aware automatically.

3. In this algorithm , the antagonist is throttled only to 0.01 CPU-sec/sec. This is quite harsh . A feedback driven throttling which dynamically set the hard capping would be more appropriate.

4. This algorithm is very simple – it will not work well if a group of antagonists together cause significant performance issue , which individually did not have much effect on the victim . In future work , it is required to reduce the number of antagonists or thinking antagonists as a group.

Conclusion:

1. CPI^2 is a CPI-based system for large clusters to detect and handle CPU performance isolation faults.

2. The design, implementation, and evaluation of CPI^2 is presented in this paper.

3. The authors demonstrated CPI^2's usefulness in solving real production issues – it has been deployed in Google's fleet .

4. The beneficiaries include

   A. End users, who experience fewer performance outliers.

   B. System operators, who have a greatly reduced load tracking down transient performance problems.

   C. Application developers, who experience a more predictable deployment environment.

Class Discussion:

1. What is good and bad in this model?

2. If you have any question for me regarding this paper.