Measuring Interference Between Live Datacenter Applications

Melanie Kambadur
Columbia University
melanie@cs.columbia.edu

Tipp Moseley
Google, Inc.
tipp@google.com

Rick Hank
Google, Inc.
rhank@google.com

Martha A. Kim
Columbia University
martha@cs.columbia.edu

***Abstract*—Application interference is prevalent in datacenters due to contention over shared hardware resources. Unfortunately, understanding interference in live datacenters is more difficult than in controlled environments or on simpler architectures. Most approaches to mitigating interference rely on data that cannot be collected efficiently in a production environment. This work exposes eight specific complexities of live datacenters that constrain measurement of interference. It then introduces new, generic measurement techniques for analyzing interference in the face of these challenges and restrictions. We use the measurement techniques to conduct the first large-scale study of application interference in live production datacenter workloads. Data is measured across 1000 12-core Google servers observed to be running 1102 unique applications. Finally, our work identifies several opportunities to improve performance that use only the available data; these opportunities are applicable to any datacenter.**

I. Introduction

Application interference occurs when multiple applications contend for shared resources such as processor time, cache space, or I/O pins. In datacenters, interference is particularly undesirable as it hurts performance and increases operating costs. Chip multi-processors (CMPs), which are widely used in datacenters, are a key contributor to interference. CMPs offer increased throughput and reduced power consumption over traditional single processor chips [41]. However, they also exacerbate interference because more applications typically run on a single physical machine. To leverage the performance benefits of CMPs, system designers must understand and prevent application interference to the greatest possible extent.

Unfortunately, the complex characteristics of datacenter workloads and architectures make application interference difficult to reason about. High heterogeneity of applications and high core utilization targets mean that datacenters' CMPs are filled with a wide variety of multi-threaded applications. Because these applications are diverse in their performance objectives, resource requirements, and inputs, and because datacenters put severe limitations on performance monitoring, it is a challenge to even measure application interference, let alone to manage it. Yet, as more applications migrate to datacenters, it has become critically important to keep negative application interference under control.

Many current approaches to monitor and combat interference work well on solitary machines, but fall short in a datacenter environment. Some techniques involve predicting application performance at a high level of detail, which is feasible in controlled settings with simple benchmarks and architectures, but becomes much more complex in datacenters. While it is possible to guess application performance at a high level and reduce interference to some degree, it is impossible to accurately predict performance to the level of precision required to eliminate it entirely. Other approaches use gladiator–style match-ups between applications to measure interference and find optimal scheduling solutions. This is not practical in a datacenter, mainly because of financial restrictions on how data can be measured. A third approach observes benchmark application performance (sometimes via simulation), then attempts to apply the observations to live applications. Some of these techniques rely on statistics that are not measurable in datacenters, while others are generous in their assumptions that noiseless and controlled offline measurements are later applicable in live, chaotic settings.

To measure live datacenter application interference, a new methodology is needed. Such a methodology should ideally be able to capture the interference effects of thousands of applications, running with real user inputs on production servers with diverse architectural platforms. Furthermore, the methodology should be financially reasonable, not requiring hundreds or thousands of machines for simulations and not disturbing the performance of production services.

In this paper, we use our experience with and exclusive access to live datacenter applications to expose the realities of measuring and analyzing interference in a datacenter. Then, we develop a methodology to measure live datacenter interference, and test the methodology on production servers at Google. Specifically:

(1) We identify eight sources of complexity in interference measurement and analysis that are either unique to datacenters or frequently not handled by previous works (Section II).
(2) We introduce a generally applicable methodology for measuring application interference in the restrictive environment of a datacenter (Section III).
(3) As a proof-of-concept, the methodology is implemented and used in the first large–scale study of measured application interference in a live datacenter. We collect data from 1102 unique applications across 1000 Google servers, each running on 12 core, 24 hyper–thread Intel Westmeres. These measurements capture the performance of production workloads, live schedules, and real user

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

interaction (Section IV).

(4) Given the information that can be measured in live datacenters, we outline opportunities to control negative application interference in datacenters (Section V).

II. Complexities of Interference in a Datacenter

Application interference in a datacenter is much more challenging to reason about, measure, or predict than in a controlled environment or on a solitary machine. It is important for scheduling experts and datacenter systems specialists to understand what performance analysts are up against. This section describes eight specific complexities that are unique to datacenters or largely unaccounted for in past work, in some cases preventing the use of established methodologies for combating application interference. For example, many past works run an application on an isolated machine to determine its baseline performance, and then run the application with a single application co-runner to measure interference effects([8], [12], [17], [23], [27], [33], [34], [39], [46], [47], [50]–[52]). The pairwise impacts are then incorporated into scheduling policies or used to fairly allocate resources between applications. Such techniques rely on well-defined, discrete applications and isolated measurements, neither of which is available in a datacenter. There are thousands of applications to test, user inputs vary in non-obvious ways (such that they cannot be simulated off-line), and applications are frequently re-written and updated.

Other approaches estimate the resource usage of applications and attempt to schedule applications with complementary needs together ([3], [5], [7], [10], [11], [15], [22], [24], [28], [36], [38], [42], [49]). While some general predictions can be made about application performance, it is challenging to make such predictions precise in the complex environment of a datacenter.

The eight complexities below are common to most datacenters; to show that they are realistic, we use experiences and data from our measurement study of production servers at Google described in Section IV.

A. Large Chips with High Core Utilizations

When slow page loads translate into lost revenue, the pressure to deliver web content quickly is high. Datacenters are driving the demand for increasingly high-core-count chips. CMPs with as many as 100 cores already exist [48], with datacenters today using CMPs with tens of cores. The 1000 Google machines profiled in Section IV are 12-core machines supporting up to 24 hyperthreads. These core-crowded chips mean more applications are sharing resources, such as cache, that they otherwise would not share. Despite this, a survey of recent work in application interference shows that many researchers validate their solutions on chips with only two or four cores ([3], [4], [10], [12], [13], [17], [21], [22], [27], [33], [46], [47], [49], [51], [52]).

In the early days of CMPs, resource contention was not the issue it is today: core counts per chip were low, and datacenters have historically struggled to use all cores on a chip (see the "bin-packing" problem discussed in [18]). Because it leads to power savings and better parallel performance, high core utilization is desirable, and it has been increasing along with per-chip core counts [26]. Today, core utilization is already high: in profiling the 24-hyperthread machines, we found that an average of about 14 hyperthreads were occupied. Figure 1 shows the full distribution of observed hyperthread occupancies.

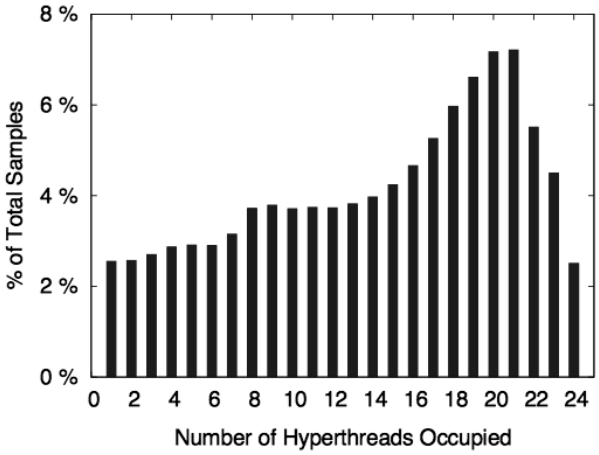


Fig. 1. **Datacenter machines are filled with applications.** Profiling 1000 12-core, 24 hyperthread Google servers running production workloads, we found the average machine had more than 14 of the 24 hyperthreads in use. These results reveal the extent of multi-way interference, which is largely un-handled by existing interference management techniques.

B. Heterogeneous Application Mixes

Datacenter servers not only support many application threads at once, but frequently also execute a diverse mix of applications on each machine. This is not surprising considering the massive number of different applications that run in datacenters today. For example, our profiling of the Google servers revealed 1102 unique applications. While a couple of these were system support applications and thus constantly or periodically running on all machines, the vast majority could be flexibly scheduled among servers in the fleet. Our measurements also showed that a machine runs at least five applications half of the time, and sometimes runs as many as 20 (see Figure 2). Characterizing interference is much simpler if only a couple of unique applications are scheduled together, so a lot of prior work assumes only two applications running on a machine at a time. According to Figure 2, such methodologies would apply only about 20% of the time.

C. Fuzzy Application Delineations

Sometimes, even trivial issues become complex in datacenter settings. To measure application interference, there must be some definition of an application. Applications might be defined as narrowly as on a per process basis, or they can be delineated by user, input, or code segment. The division of applications is tricky though; define them too narrowly, and there will be insufficient data to get useful interference

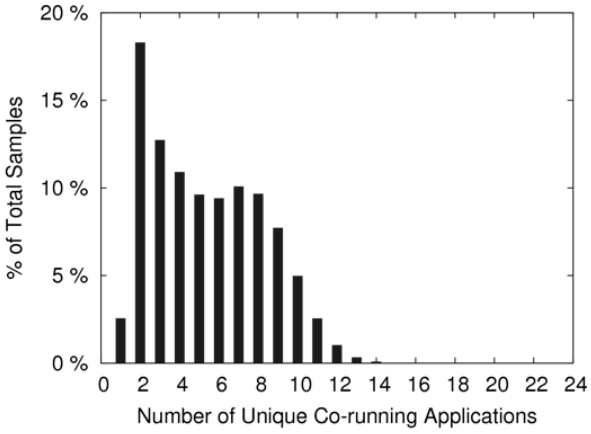


Fig. 2. **Datacenter servers have diverse application mixes.** Google server profiling reveals that most machines run five or more unique applications at once, and sometimes as many as 20. Many past works consider only two applications running together at a time, a scenario present only 20% of the time in to this data.

information. Define them too coarsely, and performance variations unrelated to application interference may inadvertently be captured. There is no clear right choice for how applications should be delineated. In the Section IV study and in Figure 2, each unique binary is considered to be an application, which is a fairly coarse-grained classification.

D. Varying and Sometimes Unpredictable Inputs

Unlike in controlled environments, applications in a datacenter are added or refactored frequently. Many applications accept user inputs and can experience significant performance swings based on usage, sometimes with predictable periodicity, and sometimes without. It is intuitive that input could affect how an application interferes or is interfered with (Jiang and Shen [22] show this formally), but most prior studies use just single–input benchmarks.

E. Varying Micro-architectural Platforms

Performance changes depend on the micro-architectural platform as well as inputs. In a large datacenter, it is uncommon for all servers to use the same micro-architecture. As new chips become available, datacenters incrementally update their servers, resulting in an evolving, heterogeneous mix of platforms. Most past work does not consider this, but interference measurement and mitigation techniques should ideally be micro-architecture independent.

F. Unknown Optimal Performance

Many existing interference solutions rely on knowing an application's optimal performance without interference. For static input benchmarks, this is as simple as running the application on a dedicated machine. At a datacenter, isolating a production application on a dedicated machine is a prohibitively expensive way to find baseline performance, especially given the number of applications to evaluate and the need for frequent re-evaluation as inputs, architectures, or even the applications themselves change. When we conducted our measurement study, Google would not allow us to measure the baseline performance of applications on isolated machines due to the cost.

G. Limited Measurement Capabilities

Performance analysts at datacenters are restricted in other ways as well. For example, an extremely limiting restriction that we had to work around in developing our methodology for the Google study was that we had to keep our profiling overhead as low as possible, and preferably well under one percent. Google's rationale, which is likely to be echoed by other datacenter companies, is that excessive overhead in measuring is not always a worthwhile investment. The financial losses caused by too much measurement perturbation in the present may outweigh future performance gains that are discoverable with the additional measurements.

H. Corporate Policy and Structure

Other difficulties relate to corporate policy and the often large size of datacenter companies. For example, performance analysts and scheduling policy makers might work in completely separate teams. That means performance analysis results must be sufficiently flexible to be fed into completely independent scheduling tools. A large company might also delay the deployment of new performance monitoring tools for strategic or accounting reasons. As a result, new solutions might not be testable or applicable for months. Performance objectives of an individual application may also compete with system-wide goals. Even if it were easy to identify and quantify every instance of negative interference, it is not always clear how each instance should be resolved. For example, in most cases a latency-sensitive application's performance is prioritized over less important applications, but performance must also be balanced with cost-efficiency. Thus, even latency-sensitive applications are likely to be co-scheduled with other applications to keep utilization up.

III. A Methodology for Measuring Interference in Live Datacenters

Put together, all of the complications outlined in the previous section make for intricate interference scenarios with restricted means to collect data about interference. Here we outline a series of techniques that form the first complete methodology for measuring application interference in the restrictive environment of a live production datacenter. Figure 3 shows an overview of this methodology. First, performance data is measured in small samples on live production servers using a small number of remote collection machines. Next, the data is examined to find per-application baseline performance comparators and to identify interference relationships between applications. These relationships are then made to be architecture independent so that performance data can be aggregated across all of the machines monitored. Afterwards, the aggregated performance data and the baseline performance indicators can be used together to analyze system-wide application interference.

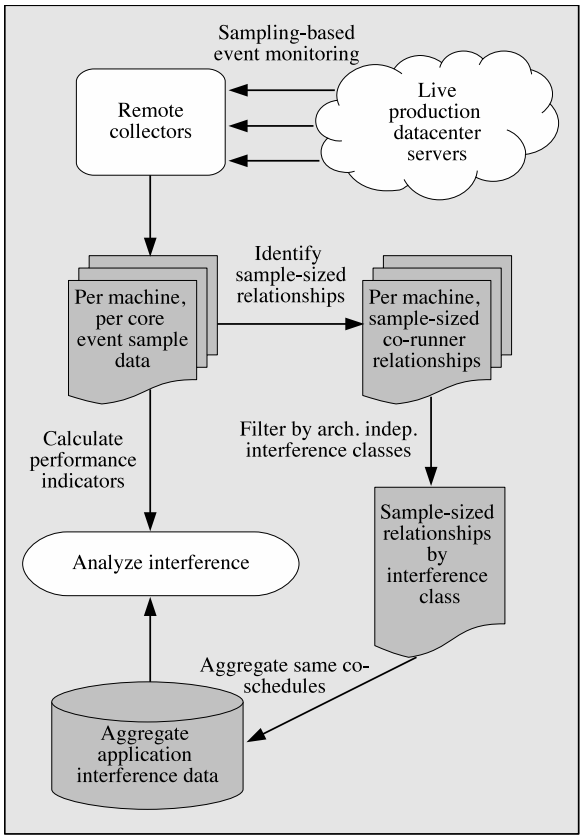


Fig. 3. **A methodology for measuring application interference on live production servers is described in Section III.**

A. Collecting Low-Overhead Performance Metrics

The most accurate way of capturing interference relationships in a datacenter is to measure them live. Since it is critical not to degrade performance, all measurements taken must have as little overhead as possible. Past work shows that sampling-based performance monitoring minimally perturbs applications. For example, the Google-Wide Profiling (GWP) tool [44], from which we borrow some measurement ideas, profiles live applications with less than 0.01% overhead using sampling-based monitoring. GWP samples performance data using `perf` [1], a Linux performance monitoring tool. Perf not only has low overhead, but it also provides abstractions over hardware capabilities, meaning the same monitoring commands can be issued on many different hardware platforms in a datacenter. The tool samples a number of measurable *events* including software events that interface with the kernel (such as page faults) and hardware events reported from the processor (such as CPU-cycles and various types of cache misses).

To further limit overheads, performance information can be reported to a small number of remote, non-production machines for later analysis. Also, sampling periods and frequencies — the number of occurrences of an event per sample, and the average rate of samples per second, respectively — and collection duration per machine can be tuned so that they are high enough to record useful information, but not so high that performance monitoring is overly intrusive.

B. Statistical Performance Indicators

One challenge of assessing interference relationships in datacenters is that the optimal performance of applications is usually unknown. Because the cost of isolating an application on a machine is high, it is rarely possible to find out how an application would perform with no application interference, so performance measurements of an application in the wild are usually clouded by several co-running applications. Instead of using optimal performance as a baseline, we can use a statistical performance indicator.

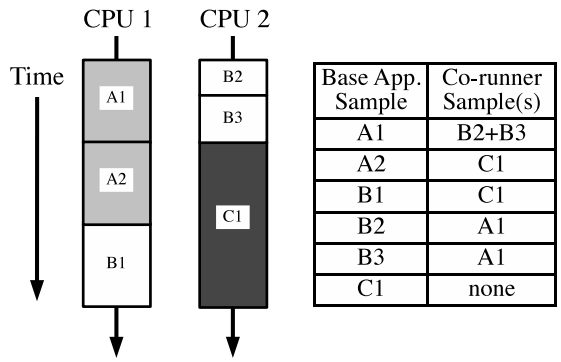
After collecting sampled performance metrics, a statistical estimator that aggregates these fine grained measurements can be used as a comparator for future observed samples. An example statistical indicator is the mean cycles per instruction (CPI) of a large number of samples. Although some dimensionality is lost in aggregation, a statistical performance indicator works well for a couple of reasons. First, only one hardware counter needs to be monitored, so the necessary information can be safely collected without perturbing live applications. Second, the indicator can be compactly stored and updated for large numbers of samples and applications. The biggest risk of using performance indicators is that phase changes of an application may be mistaken for application interference. We outline a workaround in the discussion in Section VI.

C. Identifying Sample-Sized Interference Relationships

In a controlled experiment, two applications can be run simultaneously on a machine, with applications' performance interactions monitored for the duration of their execution. As Section II explained, such co-scheduling cannot be forced in a datacenter. Another complicating factor in determining interference relationships is that applications run for extremely varying amounts of time. One application may run for a week, for example, during which time many different sets of other applications may alternately share the same machine. Thus, it is difficult to attribute the original application's performance to any one (or even any one set of) co-running applications. To learn specific interference relationships, live data must be carefully filtered.

Each performance sample includes a time-stamp, which can be used to identify which samples overlap in runtime, and eventually reveal interference relationships. Specifically, for a given *base sample*, we compile a list of the given sample's *co-runners*. A co-runner is a sample that ran for the entire duration of the base sample. We use an algorithm similar to liveness analysis in compilers to identify co-runners. The input is the starting time of each base sample, from which we work backwards to find other samples that were "live" for the duration of the base sample.

Figure 4 shows an example of samples from two CPUs and the corresponding co-runner relationships between those



Base App. Sample	Co-runner Sample(s)
A1	B2+B3
A2	C1
B1	C1
B2	A1
B3	A1
C1	none

Fig. 4. **Sample sized co-runners.** Timelines of two CPUs on the same machine are shown to the left. Each segment represents a performance sample (e.g., 2 million instructions) from an application. For example, *A1* is the first sample of application *A*. The table to the right shows the *co-runner* samples for each *base application* sample. Application *A1* has two co-runners because two consecutive samples of application *B* run for its duration. In this contrived example, sample *C1* is especially long to illustrate the uncommon case of a sample having no co-runners.

samples. Each segment in the figure is a different sample, and letter labels are application names so that *A1* is the first sample of an application *A*. Since by definition, co-running samples must run for the same amount of time or longer than the base sample, it may not be possible to identify co-runners for long samples. This can be mitigated by combining successive samples when we are looking for co-runners of a base sample. In Figure 4, sample *A1* has two co-runners because two successive samples of application *B* run for its duration. Some samples still may not have co-runners (as illustrated by the long sample *C1*). When applying this methodology (Section IV), we found that this is the case for just 0.6% of the samples. This number can be kept low if the number of samples per scheduling context-switch is relatively high; if many samples in a row are of the same application, it is more likely that co-runner relationships can be identified.

Extrapolating application-level interference relationships from a collection of sample-sized relationships is straightforward. First, all of the base samples for the base application are identified. Those samples are then sorted by their identified co-runners. Any base samples with the same sets of co-runners can be aggregated to determine the interference relationship between the base application and a set of co-running applications. With enough samples, this technique becomes schedule-independent. Depending on the schedule, more samples may be collected that represent a certain interference relationship, but with prolonged sampling, all interference relationships that occur can eventually be identified. Thus, *interference relationships can be determined without any prior knowledge of the scheduling policy*. This is extremely useful in a datacenter, because scheduling policies may be very complex, and may even be unknown to those trying to understand interference.

D. Interference Classes

Interference depends on the resources that two applications are contending for. Depending on the topology of the architectural platform, all applications sharing a chip may not have equal influence on one another. Consider, for example, two applications which share all of their cache versus two applications that share only interfaces to peripheral devices (like an I/O hub). Our analysis distinguishes between such types of interference using architecture independent *interference classes*. An interference class defines the closest relationship (in terms of resource sharing) that two applications running on the same chip might have. The closest interference relationship is between two applications running on different hyperthreads of a single core. Such applications contend for everything from execution slots to cache to memory control and I/O resources. A more distant relationship would be between applications which share the same last level cache and resources beyond. The loosest interference class is between two applications which are on the same chip, but which do not share any resources except their interface to peripheral devices. Others have used interference classes to estimate the potential amount of interference in various assignments of applications to a machine (see contention groups in [19] for example). We see a few additional reasons that defining interference classes can be beneficial. First, it allows for data to be aggregated simply across samples on many-core machines — all shared core co-runners, for example, can be considered equivalent. Next, it allows for the aggregation of data across machines with different (but similarly symmetric) architectural platforms. Finally, interference classes help reduce the complexity when considering the range of possible co-schedules of multiple applications at a time.

IV. Applying the Measurement Methodology

We now apply the general application interference measurement techniques established in the previous section to conduct the first large-scale study of interference on production Google servers running workloads with live user interaction. Unlike past work, this study does not rely on benchmarks or simulation. The study illustrates the noisiness of production interference that any datacenter interference analyst must negotiate. It also reveals that some interference patterns are visible above the noise, leading to exploitable performance opportunities, which are discussed in Section V.

A. Collecting Performance Metrics

We used the perf tool and remote collection methodology described in Section III to collect samples across 1000, 12-core production servers at Google. As described, the basic methodology allows for a choice between a number of different performance events to monitor. Unfortunately, there is no single perfect hardware counter that accurately indicates performance across a variety of applications. There is substantial debate about what, if any, hardware counter event can accurately indicate performance across a variety of applications. With such a large number of applications to compare, it

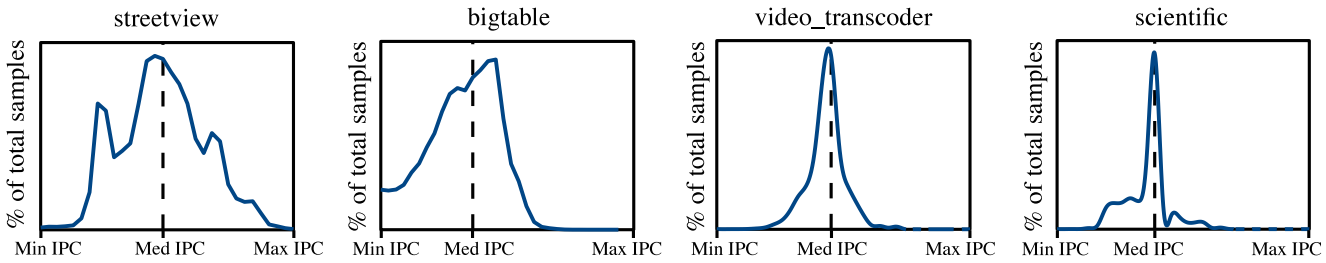


Fig. 5. **Median IPC is a good performance indicator for the Google data collected.** Each graph shows the performance variations of the specified application when scheduled with eight of their most common co-runners. The overall median IPCs for each base application correspond well to their performance curves.

is nearly impossible to use application-specific metrics (such as time per transaction) for this study. Application run time cannot be used either because it is not necessarily related to performance in datacenters (for example, an ads server might run continually until stopped for an update). Some have suggested that last level cache (LLC) miss rates are the best indicators for interference studies [8], while others note that LLC will not accurately monitor all workloads, especially those that are memory bound [46]. Other work suggests that contention for memory bandwidth and buses might be a good indicator [28], [38], [40]. To capture the effects of cache and memory contention, we use instructions per cycle (IPC) to indicate performance in this study. Although it has been widely used in past interference studies (e.g., [7], [16], [33], [36], [37], [42]), there is debate about IPC too. In particular, Alameldeen and Wood found that architectural enhancements can cause IPC to improve even as application performance worsens, or vice versa — especially for multi-threaded applications [2]. To avoid such unexpected discrepancies, we ensured that the profiled servers were identical in all respects, including chip type, clock speed, RAM, and operating system. If future studies are conducted across multiple architectural platforms, it may be necessary to consider metrics other than IPC.

Application IPC was sampled every 2.5 million instructions. After 2.5 million instructions executed on a production server's core, a remote profiling machine recorded the time-stamp, the location of the core on its machine, and the application executing. In post-processing, the elapsed time per sample was connected with the machines' clock speed to get the IPC of each sample. Over the course of the study, the remote profiler encountered 1102 unique binaries and collected nearly 350 million samples. See Table I for a summary of the collection statistics.

TABLE I
PROFILING AND COLLECTION STATISTICS

Performance Sample Size	2.5×10^6 instructions
Monitored Indicator	Instructions per cycle (IPC)
Number of Machines*	1000
**Machines identical in all respects (e.g., clock speed, RAM, O/S)*	
Threads / Core	2
Cores / Socket	6
Sockets / Machine	2
Threads / Machine	24
Unique Binaries Encountered	1102
Samples Collected (all 1102 applications)	3.45×10^8

B. Statistical Performance Indicators

From the raw samples we calculated a statistical performance indicator to estimate a baseline performance for each application. Because the collected IPCs did not form a normal distribution, we use medians rather than mean as an indicator. For each application and for each sample, we calculated and recorded the median IPC. Note that this aggregated metric is scheduling *dependent*, and we did not examine the schedule in our calculations. There are two reasons for this. First, provided our samples are representative of the system as a whole, a scheduling dependent performance indicator tells us what the normal performance of an application is in the datacenter overall. We believe the samples were representative, as our collections spanned 1000 international machines and a period of twelve hours. Second, it did not make sense for us to try to account for the scheduling system, because the policies in place at Google are not only highly complex, but also highly secretive. If scheduling policies change in the future, the methodology does not need to be revised, but new statistical performance indicators should be calculated. To evaluate the choice of medians, we can look at where medians fall on the performance curves of the data collected. Figure 5 shows the distributions of performance samples for four common Google applications (`streetview`, `bigtable`, `video_transcoder`, and `scientific`). The y-axes on the graph show the percentage of samples that range from the minimum to maximum IPC of each application on the x-axes. The graphs reveal that medians are a representative aggregate indicator. (Note that all absolute and relative IPC values have been anonymized at Google's request.)

C. Identifying Sample-Sized Interference Relationships

Returning to the raw, unaggregrated performance samples, the next step was to find co-runners among application samples. As explained in Section III-C, by definition co-running samples must be longer running than or equal length to the base application sample. Because of this, we were concerned that the samples dropped due to lack of co-runner might be biased towards the slower samples. However, the effects were

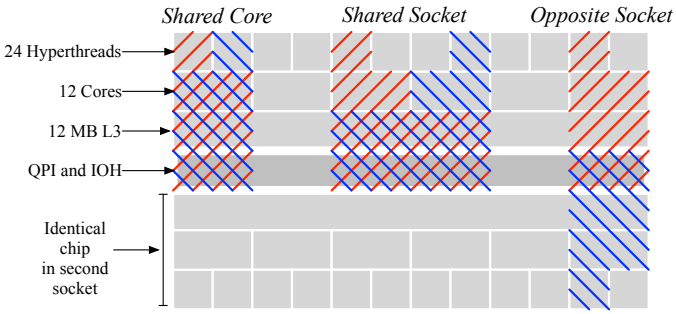


Fig. 6. **Westmere Interference Classes.** The profiled Intel Westmeres are dual-socket machines, supporting 12 hyperthreads per socket. Interference relationships are partitioned into three classes as depicted here: *shared core*, *shared socket*, and *opposite socket*.

not significant in the data collected. Across the most frequently occurring eight applications only 0.6% of the samples were dropped, with the peak being 3.47% for `search`. The impact on median IPC was negligible; dropping samples reduced it by just 0.23% on average.

D. Defining Interference Classes

The machines used for collection in this study all have the same chip, so only one set of interference classes was needed. The chips are Intel Westmeres, which have two hyperthreads per core and six cores sharing an L3 cache for a total of 12 hyperthreads per socket as pictured in Figure 6. With two sockets connected by an Intel Quick Path Interconnect (QPI) and to an I/O hub (IOH), each Westmere supports a total of 24 hyperthreads. Given this topology, there are three discernible interference classes, also depicted in Figure 6. The closest is between two applications on hyperthreads which share a core (*shared core*); then between two application threads on different cores but sharing a socket and thus an L3 cache (*shared socket*); and finally between two threads on the same machine but on different sockets (*opposite sockets*) which share only the QPI and IOH.

For each of the sample co-runners previously identified, we looked at the relative core locations of the applications. Using these core locations, we assigned each pair of co-runners the appropriate interference class label. Between eight of the most commonly running applications we encountered, the average number of shared core samples ranged from 2000 to 45 million, with about 1 million samples on average. Between the same applications, the number of shared socket samples ranged from 12,000 to 400 million per application and 9.5 million on average. The opposite socket relationships ranged from to 14,000 to 500 million samples with 11 million on average.

E. Analyzing Interference

A primary question in past work is *how does a base application's performance change with a particular co-runner?* This is a very challenging question to answer in a datacenter. One approach is to examine the performance effects of a given application on another by aggregating all of the performance metrics from the sample-sized relationships of a particular base application and a particular co-running application. However, up to 22 other hyperthreads may be occupied with various unrelated applications during each of the samples, so this must be taken into account. It was rare to find only two applications running together on a machine, which is not surprising considering our earlier observation that Google maintains a high thread occupation rate (Figure 1) and runs diverse applications together on a single machine (Figure 2). The shared core interference relationship is especially important to understand as it is likely the strongest. Finding two applications running in isolation on the same core with the remaining threads empty was an extremely rare occurrence; probably due to intentional scheduling decisions to distribute resources.

Regardless of the reasons, it is clear that noiseless data is hard to come by in a datacenter. Thus, pairwise comparisons can never fully capture all the causes of interference. Still, we wanted to attempt to see if shared core influences were strong enough to be apparent over the noise of applications scheduled on the rest of the machine. Though necessarily incomplete, if pairwise comparisons can yield any information, they are attractive for two reasons. First, reducing the comparison space makes the resulting information easier to collect, understand, and analyze. Also, some schedulers — including Google's — are already prepared to accept pairwise scheduling information but not information about more complex relationships.

To find shared-core influences, we aggregated the previously identified pairwise relationships of eight commonly running applications, filtering the samples to use only those that were labelled as shared core. To reduce random performance variations, we required that a minimum of 1000 samples be present for each aggregated metric to be significant; all 64 cross-pairings satisfied this minimum.

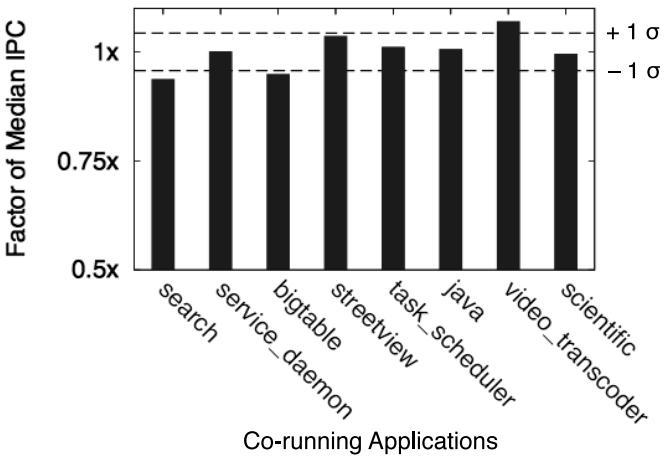


Fig. 7. **Streetview's performance variations across co–runners.** Bars represent `streetview`'s normalized median performance when co–located with eight common co-runner applications. Dashed horizontal lines show overall variance of all measured `streetview` samples.

Figure 7 shows `streetview` as it shares a core with eight other applications (including different co-running instances of

its own binary). Other applications exhibit similar performance effects in their shared core co-runner graphs. In Figure 7, bars along the x-axis show the shared core co-runner of `streetview`, and the y-axis gives the normalized median IPC across each of the aggregated `streetview` and shared core co-runner samples. The dotted horizontal lines show the average variance across all of the measured (co-runner independent) `streetview` samples. We note that while *it is difficult to tell an exact ordering of `streetview`'s best to worst co-runners* given the large variance of the samples, it is clear that a few shared core co-runners interfere beyond the noise.

We collected data on shared socket and opposite socket pairwise interference using similar techniques. The additional data is not included here because it does not add much insight. In part, this is because the pairwise influence of sharing a socket or machine can be weaker than when sharing a core. Consider, for example, a co-runner sharing a socket with a base application. The base application has one shared core co-runner and ten shared socket co-runners on a Westmere (recall Figure 6). So, if we try to examine the effects of a single shared socket co-runner on the base application, we are also capturing the effects of at least ten other co-runners sharing as many or more resources with the base application. Fully understanding shared socket and shared machine influences would require examining interference patterns between larger groups of co-runners than pairs.

V. Performance Opportunities

Given a total ordering of interference relationships, some past works are able to find optimal schedules and sometimes nearly eliminate negative interference. An important goal of this work was to show that such solutions cannot be immediately successful when applied to datacenters, primarily because the precision required to determine a total ordering of relationships is not available. The measurement techniques in Section III outline a path towards better understanding application interference in datacenters, where the measurable information is necessarily more limited. Although it is disappointing that many insightful techniques cannot be immediately applied in datacenters, the good news is that in a datacenter even small reductions in application interference are valuable. In this section, we outline two techniques that are immediately applicable in a datacenter once the data outlined earlier in this paper has been collected.

A. Restricting Beyond Noisy Interferers

With many applications running on live machines, it is difficult to observe isolated (noise-free) interactions. Moreover, measurement restrictions make the discovery of a full ordering of co-runner preferences difficult. Despite the noise, the data still allow us to recognize that some applications interfere. We define *beyond noisy interferers* (BNIs) as applications that can be clearly seen to hamper another application's performance despite the noisy data. To identify BNIs, we find the average variance from the mean performance of a base application that incorporates all possible co-schedules. This metric indicates the average expected performance fluctuation of an application across diverse scheduling scenarios. Next, the measured samples of a particular co-scheduling relationship can be compared to the overall variance. If a co-schedule affects an application beyond its normal variance, it is classified as a BNI.

We applied this procedure to the Google data to see if any shared-core co-runners could be classified as BNIs. Figure 8 shows the performance of eight common Google applications when they were observed to be sharing a core with one of the other eight applications. Boxes in the matrix show the difference from the average variance (across all 1102 applications encountered in the study) of each base application (on the y-axis) for each co-runner (on the x-axis). A white box indicates that the shared-core co-runner positively interferes with the base application beyond the average variance, while a black box indicates negative interference beyond the average variance. Several negative BNIs (6 of 64 possible, or nearly 10%) emerge despite the fact that most of the observed data includes noise from other applications interfering outside of the shared core.

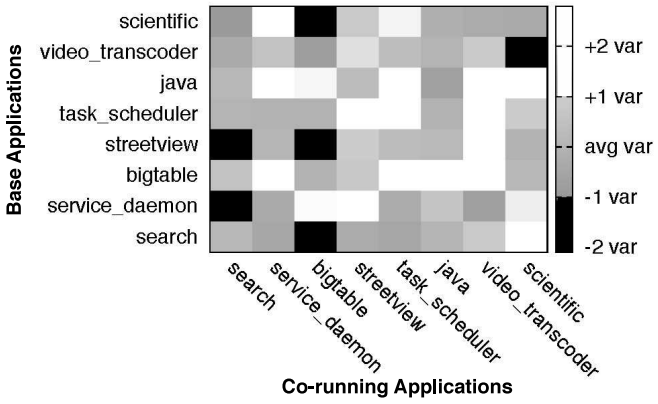


Fig. 8. **Beyond noisy interferers in the Google data.** Shared core co-runner applications along the x-axis affect the performance of base applications along the y-axis. White boxes show co-runners that positively interfere beyond the average variance with base applications, while black boxes show co-runners that negatively interfere beyond the average variance.

Such observed BNIs do not yield a complete ordering of application co-schedule preferences, and thus do not allow allow the compilation of an optimal schedule. Negative BNIs can, however, indicate specific applications that should not run together. A simple scheduling policy change to restrict negative BNIs from running alongside the base application could result in significant performance gains. Similarly, positive BNIs might be purposely scheduled with a base application to improve its average performance.

In some cases, even eliminating one or two negative co-runners could result in significant performance improvements for an application. The potential for improvement can be estimated if we assume that in the absence of samples with the negative co-runner, the base application would perform

at its median performance with all other co-runners. Then, the improved median performance can be reverse engineered from the performance data already available as follows: first, calculate the fraction of samples where the base application runs with the negative co-runner and call it the "negative fraction". Call the remaining samples the "neutral fraction". Next, multiply the negative fraction by the median performance of the base application when running with the negative co-runner and subtract this value from the overall median where the base application runs with any application including the negative co-runner. Finally, divide this value by the neutral fraction to get the new expected median performance. In the data pictured in Figure 8 for example, the `bigtable` application is a negative BNI for `streetview`. If we eliminate all instances of `bigtable` running with `streetview` and assume that `streetview` will then perform at its median, then `streetview`'s overall performance will have improved by about 1.3%. If we also exclude `search` from running with `streetview` and make the same assumption, `streetview`'s performance could jump as much as 2.2% system-wide. Though these effects may seem small, when multiplied across weeks or months of application execution on thousands of servers, such improvements could result in sizable monetary savings.

B. Isolating Sensitive Applications and Exiling Antagonists

It is interesting to know how *sensitive* an application is to performance changes. Several previous studies have looked at application sensitivities in the context of resource contention ([24], [28], [31], [32], [46]), some of them using datacenter workload benchmarks. In these studies, sensitivity is defined in terms of an application's optimal performance. As explained in Section II, it is difficult to ascertain a datacenter application's optimal performance, but we can extend the earlier work to comply with the available data. Specifically, the variance data used to determine BNI application relationships in Figure 8 can also be used to determine an application's overall sensitivity. Base applications with large performance variations across co-runners can be identified as sensitive to performance changes. For example, in Figure 8 the `scientific` and `streetview` applications have shared core co-runners that cause their performance to swing both above and below one average variance. If the performance of these two applications (or any sensitive application) is important to the datacenter, systems managers can decide to isolate the applications on their own core, or even their own machine.

Antagonistic applications can be identified in a similar manner. A co-running application is antagonistic if it frequently causes base applications to exhibit negative performance swings beyond their average variances. In the figure, `bigtable` is a negative BNI for three applications, so it can be classified as antagonistic. Again, depending on the performance goals of the datacenter, it might make sense to exile such antagonistic applications to their own core or machine so that they do not negatively interfere with other applications' performance.

VI. Future Opportunities

Using the data collected in the Google study, it is possible to identify BNIs and to find sensitive and antagonistic applications that can be isolated or exiled, respectively. With extensions to the methodology outlined here, there are further opportunities to minimize interference and improve performance.

A. Multi-dimensional Scheduling Constraints

This initial study focuses on pairwise interference effects, for simplicity and because Google's scheduler was already ready to accept pairwise scheduling inputs. There may also be significant trios or even larger sets of application co-schedules with relevant interference patterns. For example, some application A might not perform poorly with either B *or* C as a co-runner, but may perform poorly when B *and* C are both co-runners. One could identify triplet (or larger) BNIs using the same techniques as for pairwise BNIs. Once identified, larger groups of BNIs could be employed in all the same ways as pairwise BNIs. As discussed in Section IV-E this would be particularly useful when examining the effects of interference beyond shared core.

B. More Fine-grained Application Definitions

It is well known that some applications exhibit distinct phases with different performance characteristics. Such phases might obfuscate the process of identifying performance effects. In our Google study, we were able to observe fairly stable performance (Figure 5) by limiting our measurement study to twelve hours because most of the applications had diurnal phases based on the peak and off-peak usage of users. For important applications, it may be worth the additional complexity to identify distinct phases more precisely. Then, each phase of the applications could be considered as separate "applications" when analyzing co-runner relationships. Similarly, if a given application's performance is known to vary widely based on input, the application could be broken apart according to its usage pattern.

C. Correlating Multiple Performance Events

While data collection is limited to one performance event at a time, multiple events could be collected on separate trials and compared to give a fuller picture of application performance and interference. Correlating IPC with metrics such as LLC misses and I/O contention, could lead to more insight than examining any one metric on its own. The challenge of correlating multiple performance events is that application co-schedules have to be matched across trials. When we analyzed the Google data, we were able to greatly reduce the aggregation complexity by combining sample data across same shared-core co-runners without filtering based on the rest of the applications co-scheduled on the machine. This method is a starting point for correlating multiple events, but it would be more precise to match the full machine co-schedules instead of just matching shared-core co-runners.

VII. Related Works

Several papers and textbook chapters highlight challenges associated with CMPs in datacenters. Ranganathan and Jouppi discuss challenges related to general trends in changing infrastructures at large datacenters [43]. Kas writes about problems that must be solved as datacenters adopt CMPs, but does not specifically address the difficulties involved in measuring application interference [26]. One relevant description of the challenges of resource interference between applications can be found in Illikkal et al.'s work which discusses potential performance problems due to shared resource interference but does not detail the challenges of measuring interference [20].

While this work is the first to conduct a datacenter scale application interference study on live production workloads, other researchers have conducted application interference studies geared towards datacenters. Rather than measuring live applications with user interaction, the following studies use benchmarks, simulations, and offline analysis of server workloads. While a benchmark runs, Mars et al. use performance counters to detect cache miss changes and identify contention so that schedules can be adaptively updated [34]. Another paper by Mars et al. measures changes in instruction rate to detect cross-core interference and adapt schedules accordingly [33]. Tang et al. try different thread-to-core mappings of benchmarks to methodically find the best co-schedules [47]. Another large scale study models resource interference of server consolidation workloads, finding core and cache contention [3]. This methodology requires estimates of cache usage and considers only two jobs co-scheduled at a time. Bilgir et al. simulate Facebook workloads to look for energy and performance benefits in assigning the correct number of cores and mapping applications effectively across CMPs [6]. The works by Carter et al. [9] and Levesque et al. [30] evaluate whether increasing core counts on Cray machines will improve scientific applications' performance by estimating their memory bandwidth contention. Finally, Hood et al. [19] and Jin et al. [25] break down expected contention by class for different architectural platforms using microbenchmarks. They then estimate how real applications will perform on different architectural configurations.

A number of other works have measured the use of shared resources on single machines. Moseley measured resource sharing between threads in simultaneous multithreading (SMT) processors using hardware performance monitoring [37]. Snavely and Tullsen conduct an impressively thorough study of application co-scheduling on SMT architectures [45]. Like us, they use sample-based performance monitoring, but their work uses simulation and benchmarks rather than live workloads and relies on testing a significant number of permutations of all jobs co-scheduled together. Azimi et al. also use hardware sampling of benchmarks to study how threads share resources so that they can optimize cache locality and determine how caches should be partitioned on SMT machines [4]. Zhang et al. perform an extensive examination of cache contention between applications on varying CMP platforms [50], while Zhao et al. took a more detailed approach, monitoring not just cache sharing but occupancy and interference as well [52].

There is no dearth of related previous research proposing operating systems or hardware solutions to mitigate application interference. Unfortunately, many of the proposed ideas cannot accommodate the complexities outlined in Section II. It is difficult to give credit to everyone who has contributed to such a well studied area. We have already discussed a number of works in this area that use measured performance monitoring as input; another relevant body of work estimates applications' resource usage to improve scheduling ([5], [10], [11], [15], [23], [24], [28], [29], [38], [42]). There is also a series of work that adjusts access to computing resources like CPU processing speed and cache partitioning size to make resource sharing more fair ([14], [16], [17], [20], [21], [27], [35], [36], [39], [49], [51]).

VIII. Conclusions

This paper encourages researchers to develop scalable application interference solutions, and begins to pave the way for such work. To establish the difficult nature of this task, we first detail the challenges of measuring and analyzing application interference at datacenter scale, exposing eight specific challenges that are unique to datacenters or that remain largely un-addressed in past research. These factors combine to make interference effects in a datacenter exceedingly difficult to predict, measure, and correct. To assist in the efforts of understanding interference between datacenter applications, we suggest a collection of measurement techniques to work around the complexities. The new techniques are generically applicable for any datacenter, but as a proof-of-concept, we implement them to conduct an application interference study on production Google servers. The study is the first large-scale measurement study of application interference, revealing application interference "in the wild" on 1000 12-core machines running live commercial datacenter workloads. Using just data that is feasible to collect in the restrictive environment of a datacenter, we have outlined several opportunities to improve performance by reducing negative application interference.

IX. Acknowledgments

We would like to thank Google for providing the resources that made this study possible. The work was also supported in part by the National Science Foundation (CNS-1117135). We thank Lingjia Tang, Dave Levinthal, Stephane Eranian, Amer Diwan, and other Google colleagues for their insights and suggestions as we worked on this project. Finally, we want to acknowledge William Kramer and our anonymous reviewers for their helpful feedback on the paper.

References

[1] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/, July 2011.

[2] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, July 2006.

[3] P. Apparao, R. Iyer, and D. Newell. Towards modeling & analysis of consolidated CMP servers. *ACM SIGARCH Computer Architecture News*, 36:38–45, May 2008.

[4] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review*, 43:56–65, April 2009.

[5] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 189–199, 2010.

[6] O. Bilgir, M. Martonosi, and Q. Wu. Exploring the potential of CMP core count management on data center energy savings. *Proceedings of the Workshop on Energy Efficient Design (WEED)*, June 2011.

[7] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 318–329, 2008.

[8] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *Transactions on Computer Systems (TOCS)*, 28, December 2010.

[9] J. Carter, Y. He, J. Shalf, H. Shan, E. Strohmaier, and H. Wasserman. The performance effect of multi-core on scientific applications. In *Cray User Group Meeting*, Seattle, WA, USA, 2007.

[10] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351, 2005.

[11] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 105–115, 2007.

[12] R. C. Chiang and H. H. Huang. TRACON: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 47:1–47:12, 2011.

[13] M. Devuyst, R. Kumar, and D. M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[14] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 335–346, 2010.

[15] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. *ACM SIGPLAN Notices*, 45:91–102, March 2010.

[16] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 25–38, 2007.

[17] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 479–488, 2009.

[18] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[19] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance impact of resource contention in multicore systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1 –12, April 2010.

[20] R. Illikkal, V. Chadha, A. Herdrich, R. Iyer, and D. Newell. PIRATE: QoS and performance management in CMP architectures. *SIGMETRICS Performance Evaluation Review*, 37:3–10, March 2010.

[21] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *SIGMETRICS Performance Evaluation Review*, 35:25–36, June 2007.

[22] Y. Jiang and X. Shen. Exploration of the influence of program inputs on CMP co-scheduling. In *European Conference on Parallel Processing (EUROPAR)*, volume 5168 of *Lecture Notes in Computer Science*, pages 263–273. 2008.

[23] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–229, 2008.

[24] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 201–215, 2010.

[25] H. Jin, R. Hood, J. Chang, J. Djomehri, D. Jespersen, K. Taylor, R. Biswas, and P. Mehrotra. Characterizing application performance sensitivity to resource contention in multicore architectures. Technical Report NAS-09-002, NASA Ames Research Center, 2009.

[26] M. Kas. Towards on-chip datacenters: A perspective on general trends and on-chip particulars. *The Journal of SuperComputing (SCI)*, October 2011.

[27] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, 2004.

[28] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 65–76, 2010.

[29] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, W. Zhihua, and C. Pu. An analysis of performance interference effects in virtual environments. In *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, april 2007.

[30] J. Levesque, J. Larkin, M. Foster, J. Glenski, G. Geissler, S. Whalen, B. Waldecker, J. Carter, D. Skinner, H. He, H. Wasserman, J. Shalf, H. Shan, and E. Strohmaier. Understanding and mitigating multicore performance issues on the AMD Opteron architecture. Technical Report LBNL-62500, Lawrence Berkeley National Laboratory, 2007.

[31] J. Mars, L. Tang, and R. Hundt. Heterogeneity in homogeneous warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 10(2):29–32, July 2011.

[32] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[33] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 167–176, 2011.

[34] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 257–265, 2010.

[35] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. *ACM SIGARCH Computer Architecture News*, 35:46–56, June 2007.

[36] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: a QoS framework for CMP architectures. *ACM SIGOPS Operating Systems Review*, 43:86–96, April 2009.

[37] T. Moseley. Adaptive thread scheduling for simultaneous multithreading processors. Master's thesis, University of Colorado, 2006.

[38] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 146–160, 2007.

[39] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 237–250, 2010.

[40] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 208–222, 2006.

[41] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, September 2005.

[42] K. K. Pusukuri, D. Vengerov, A. Fedorova, and V. Kalogeraki. Fact: a framework for adaptive contention-aware thread migrations. In *Proceedings of the International Conference on Computing Frontiers (CF)*, 2011.

[43] P. Ranganathan and N. Jouppi. Enterprise IT trends and implications for architecture research. In *Proceedings of the Symposium on High Performance Computer Architecture (HPCA)*, pages 253–256, 2005.

[44] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[45] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, 2000.

[46] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT)*, pages 12–21, 2011.

[47] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 283–294, 2011.

[48] Tilera Corporation. Tile-Gx Processor Family. http://www.tilera.com/products/processors/TILE-Gx_Family/, 2012.

[49] C. Xu, X. Chen, R. Dick, and Z. Mao. Cache contention and application performance prediction for multi-core systems. In *Proceedings of the International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 76 –86, march 2010.

[50] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 203–212, 2010.

[51] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 23–23, 2009.

[52] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in CMP platforms. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 339–352, 2007.