The Impact of Memory Subsystem Resource Sharing on Datacenter Applications

Lingjia Tang
University of Virginia
lt8f@cs.virginia.edu

Jason Mars
University of Virginia
jom5x@cs.virginia.edu

Neil Vachharajani
Pure Storage
neil@purestorage.com

Robert Hundt
Google
rhundt@google.com

Mary Lou Soffa
University of Virginia
soffa@cs.virginia.edu

ABSTRACT

In this paper we study the impact of sharing memory resources on five Google datacenter applications: *a web search engine*, *bigtable*, *content analyzer*, *image stitching*, and *protocol buffer*. While prior work has found neither positive nor negative effects from cache sharing across the PARSEC benchmark suite, we find that across these datacenter applications, there is both a sizable benefit and a potential degradation from improperly sharing resources. In this paper, we first present a study of the importance of thread-to-core mappings for applications in the datacenter as threads can be mapped to share or to not share caches and bus bandwidth. Second, we investigate the impact of co-locating threads from multiple applications with diverse memory behavior and discover that the best mapping for a given application changes depending on its co-runner. Third, we investigate the application characteristics that impact performance in the various thread-to-core mapping scenarios. Finally, we present both a heuristics-based and an adaptive approach to arrive at good thread-to-core decisions in the datacenter. We observe performance swings of up to 25% for web search and 40% for other key applications, simply based on how application threads are mapped to cores. By employing our adaptive thread-to-core mapper, the performance of the datacenter applications presented in this work improved by up to 22% over status quo thread-to-core mapping and performs within 3% of optimal.

Categories and Subject Descriptors

B.3.3 [**Hardware**]: Memory Structures—*Performance Analysis and Design Aids*; C.4 [**Computer Systems Organization**]: Performance of Systems—*Design studies*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Performance, Experimentation, Measurement

ISCA'11, June 4–8, 2011, San Jose, California, USA.

Keywords

contention, multicore, thread-to-core mapping, thread scheduling, datacenter, workload characterization

1. INTRODUCTION

Webservice datacenters and cloud computing economies of scale have gained significant momentum in today's computing environments. Modern datacenters are constructed using commodity components as they are cheap and easily replaceable. Machines have multiple sockets hosting processors with multiple cores. The processing cores in a single machine share a number of caches and buses. Figures 1 and 2 show typical memory subsystem topologies found in state-of-the-art server processors. Figure 1 shows a dual-socket Intel Xeon (Clovertown) system comprising 8 total cores. Each socket is connected to the Memory Controller Hub (MCH) through a Front Side Bus (FSB). The MCH has four channels to four RAM banks. Each socket on this system has two separate L2 caches shared by a pair of cores and all four cores on a socket share a FSB. The Dunnington shown in Figure 2 shows a similar organization with 24 cores divided across four sockets each of which has both second level caches shared between pairs of cores, and third level caches shared between six cores. These types of machine organizations are commonplace in state-of-the art server processors, and future platforms will continue to hierarchically share resources between the cores.

When multiple cores share a resource, the threads running on those cores can *constructively* use this resource in a number of ways. For example, when threads share a cache, data sharing requires only one copy of the data in the shared cache, rather than multiple copies spread out across private caches. Further, memory bus and coherence traffic are reduced since data is fetched from memory only once and does not ping-pong back and forth between separate private caches. Even threads not sharing a cache can interfere constructively. For example, coherence traffic localized to caches within a single socket can avoid costly messages on a hotly contended system bus. Coherence traffic between two caches that share a bus is less costly than coherence traffic between sockets.

On the other hand, if the processing threads on neighboring cores do not share data, they can *destructively* interfere. Multiple threads can contend for shared resources. A thread can bring its own data into a shared cache, evicting the data of a neighboring thread and resulting in reduced

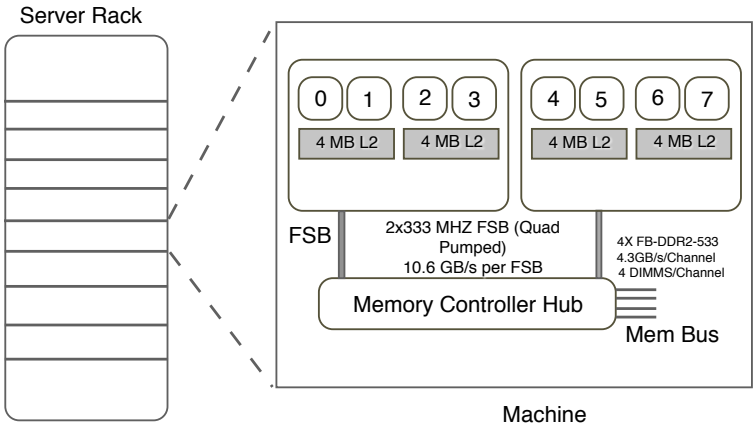

Figure 1: Topology of a Dual Socket Intel Clovertown

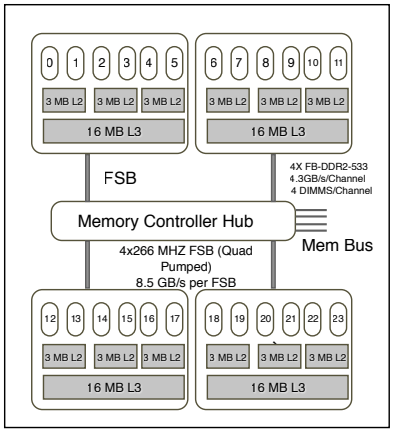

Figure 2: Topology of a Quad Socket Intel Dunnington

performance of both threads. Threads can also contend for bus bandwidth, detrimentally affecting application performance.

[**Problem**] As the datacenter that provides large scale web service emerges as a very important computing domain, understanding the interaction between datacenter applications and the underlying machine architecture is important for both hardware designers and system software designers [26, 37]. Characterizing how *constructive* or *destructive* interferences from memory resource sharing manifest in these industry-strength applications is an important step to designing architectures for these emerging workloads. For example, the characterization can help provide insights for architects to select appropriate sizes and topologies of shared and private caches when designing multicores for these workloads. Alternatively, system software may be able to map threads across the cores of a machine with a policy to co-locate threads on a shared resource, or distribute the work across these resources to minimize sharing. Currently there is little understanding about the interaction between datacenter applications and the underlying memory resources sharing topology. Recent work [45] concludes that cache sharing does not have a significant impact on contemporary multithreaded workloads using PARSEC benchmark suite. However, the commonly used benchmark suites (SPEC, PARSEC, etc) may not represent emerging datacenter application workloads. As a result of this lack of understanding, modern datacenters assign threads to cores in an ad-hoc fashion.

[**Contributions**] In this paper we study the impact of shared resources on five industry-strength datacenter applications: web search, bigtable, content analyzer, image stitcher, and protocol buffer. All benchmarks are measured in Google's production datacenter environment with real query inputs. We are specifically focusing on web search, bigtable and content analyzer as they represent the emerging large-scale latency-sensitive online service workloads that are running in many of the world's largest datacenters. A detailed description of these applications is presented in Table 1. While prior work has found the performance effects from constructive and destructive resource sharing is not significant across the PARSEC benchmark suite, we find that across these datacenter applications, there is both a sizable benefit and potential degradation from these resource sharing effects. This observation leads to our three main research contributions:

- We demonstrate the impact of memory resource sharing for key datacenter applications and show the importance of identifying good thread-to-core (TTC) mappings for applications in the datacenter. Threads can be mapped to share or to not share caches and bus bandwidth, and good mapping decisions depend heavily on the applications memory characteristics. (Section 3)

- We evaluate and analyze the impact of co-locating threads from multiple applications with diverse memory behavior. In this work, we discovered that the best mapping for a given application changes when co-located with another application. This is largely due to the tradeoff between intra-application and inter-application contention. One key insight in this work is that the best mapping does not only depend on the applications sharing and memory characteristics, it is also impacted dynamically by the characteristics of co-runners. (Section 4)

- We identify the application characteristics that impact performance in the various thread-to-core mapping scenarios. These characteristics include the amount of sharing between threads, the amount of memory bandwidth the application requires, and the cache footprint of the application. These three characteristics can be used to identify good thread to core mappings. We present an algorithm for a heuristics-based thread to core mapping technique that takes advantage of these applications characteristics. We also present an adaptive approach that uses a competition heuristic to learn the best performing mapping online. (Section 5)

[**Results Summary**] At the datacenter scale, a performance improvement of 1% for key applications, such as websearch, can result in millions of dollars saved. We observe a performance swing of up to 25% for websearch, and 40% for other key applications, simply from remapping application

Table 1: Production Datacenter Applications

applications	description	metric	type
content analyzer	content and semantic analysis, used to take key words or text documents and cluster them by their semantic meanings [1]	throughput	latency-sensitive
bigtable	storage software for massive amount of data [9]	average latency	latency-sensitive
websearch	industry-strength internet search engine [5]	queries per second	latency-sensitive
stitcher	image processing and stitching, used for generating street views [43]	N/A	batch
protobuf	protocol buffer [2]	N/A	batch

threads to cores. When co-locating threads from multiple applications, the optimal thread to core mappings changes. We also find that by leveraging knowledge of an application's sharing characteristics, we can predict both how an application's threads should be mapped when running alone as well as with another application. However, we conclude that using our online adaptive learning approach is a preferable approach for arriving at good thread to core mappings in the datacenter as it arrives at near optimal decisions and is agnostic to applications' sharing characteristics. We observe a performance improvement of up to 22% over status quo thread-to-core mapping and performance within 3% of optimal mapping on average.

2. BACKGROUND AND MOTIVATION

2.1 Memory Resource Sharing

On multi-socketed multicore platforms such as the dual socket Intel Clovertown shown in Figure 1, processing cores may or may not share certain memory resources including the last level cache (LLC) and memory bandwidth as discussed in the previous section. Thus for a given subset of processing cores, there is a particular *sharing configuration* among the cores of that subset. For example, for two processing cores on the Clovertown machine shown in Figure 1, there are three possible sharing configurations among two cores, shown in Table 2. For a set of four processing cores on the same Clovertown machine, there are three different sharing configurations among the four cores. Each sharing configuration is also illustrated in Figures 3, 4, and 5. The cache hierarchy and memory topology of the specific machine determine the possible sharing configurations among multiple cores. For example, on a multi-socket Dunnington, the sharing configurations span combinations of sharing and not sharing scenarios of the three memory components: the L2 cache, L3 cache, and the front side bus (FSB).

Whether an application's performance is constructively or destructively impacted by the sharing configuration of the cores on which it is running depends on whether the application thread's data sharing characteristics mimic the sharing configuration of the cores. Figures 3, 4 and 5 show three mappings corresponding to three sharing configurations on our experimental platform, the Intel Clovertown. Here we introduce a notation for the set of cores the threads are mapped to on this Clovertown topology. We use `X` to highlight the cores the threads are mapped to. For example `{XXXX....}` indicates four threads mapped to cores {0, 1, 2, 3} on the same socket, as shown in Figure 4. To study the performance impact of resource sharing in a controlled and isolated fashion, we compare the performance differences of an application in different thread-to-core mappings. This sheds light on how sharing of each type of resource impacts performance of various applications with different data sharing patterns. For example, the performance difference between mapping `{XX..XX..}` and `{X.X.X.X.}` reflects the impact of sharing last level cache (LLC); and the performance difference between mapping `{XX..XX..}` and `{XXXX....}` reflects the impact of sharing FSB. When there is a significant performance variability, a resource-aware thread-to-core mapping is needed.

2.2 Datacenter

[Job Scheduling] In the modern datacenter, job scheduling is done in a hierarchical fashion. A global job scheduler manages a number of machines and selects a particular machine for each job based on the amount of memory or the number of CPU the job requires. Once a machine is selected, the job, and its individual threads, are then managed by the OS scheduler. The OS scheduler decides how the application threads are mapped to the individual processing cores of this machine. At this level, general purpose system software such as the Linux kernel is adapted for, and used, in the datacenter for finer grain scheduling.

Current job scheduling does not take memory resource sharing into account. The scheduler's thread-to-core mapping is determined without regard to, or knowledge of, the application characteristics or the underlying resource sharing topology. The state-of-the-art kernel scheduler focuses on load balancing and prioritizes cache affinity to reduce cache warm-up overhead. Although developers can specify which cores to use manually, this must be done on an application by application, architecture by architecture basis. As a result, this option is seldom used as it places a significant burden on the developer.

[Job Priority and Co-location] Applications in a datacenter have different priorities. Key applications such as web search and bigtable are latency sensitive and have high priority. As the number of cores per machine increases, lower-priority batch applications such as encoding or image processing are co-located with key applications. Co-locating multiple heterogeneous workloads means higher machine utilization, thus fewer machines are needed and the operation cost is reduced. However, as shown in prior work [29], the performance interference caused by lower priority applications on high priority applications needs to be minimized. There is a wealth of research on hardware and OS designs that provides QoS priority management [11, 33, 32, 15, 19]. However, managing QoS priorities on multicores remains a challenge. Table 1 shows which of the datacenter applications used in this work are latency sensitive, and which are batch. This work focuses on the performance of the key latency sensitive applications.

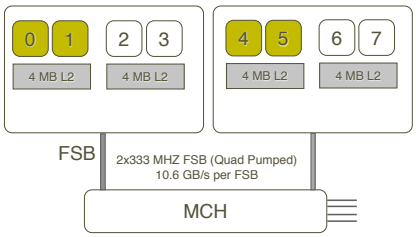

Figure 3: Sharing Cache, Separate FSBs (`XX..XX..`)

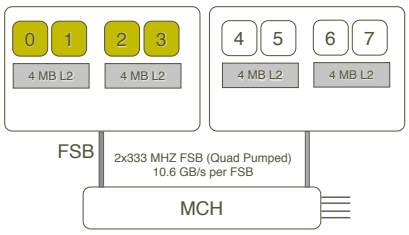

Figure 4: Sharing Cache, Sharing FSB (`XXXX....`)

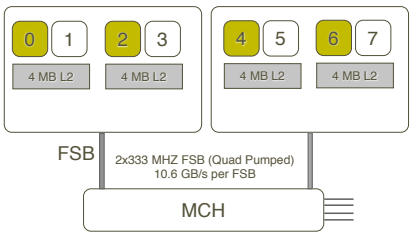

Figure 5: Separate Caches, Separate FSBs (`X.X.X.X.`)

Table 2: Sharing configurations for sets of 2 cores and sets of 4 cores

# Cores	LLC(L2)	FSB	set of cores
2 Cores	**Share**: 2 cores - 1 L2	**Share**: 2 cores - 1 FSB	{0,1}, {2,3}, {4,5} {6,7}
	Distribute: 2 x (1 core - 1 L2)	**Share**: 2 cores - 1 FSB	{0,2}, {1,3}, {4,6}, {5,7}
	Distribute: 2 x (1 core - 1 L2)	**Distribute**: 2 x (1 core - 1 FSB)	{0,4},{0,5},{0,6},{0,7},{1,4},{1,5}...
4 Cores	**Share**: 2 x (2 cores - 1 L2)	**Share**: 4 cores - 1 FSB	{0,1,2,3}, {4,5,6,7}
	Share: 2 x (2 cores - 1 L2)	**Distribute**: 2 x (2 cores - 1 FSB)	{0,1,4,5}, {2,3,6,7}
	Distribute: 4 x (1 core - 1 L2)	**Distribute**: 2 x (2 cores - 1 FSB)	{0,2,4,6}, {1,3,5,7}

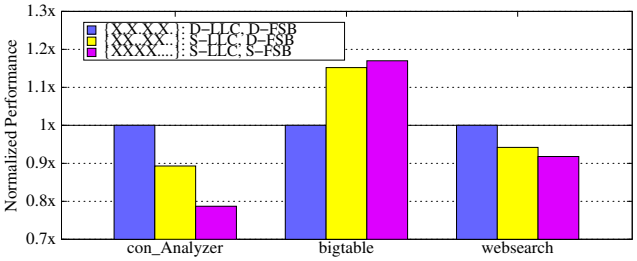

Figure 6: Performance of different thread-to-core mappings when each application is running alone. The higher the bars, the better performance. The performance variability is up to 20% for each application, indicating that the memory resource sharing has a significant performance impact on these applications. Also, notice that *bigtable* is benefiting from sharing last level cache; while *contentAnalyzer* and *webSearch* suffer from the contention for memory resource among sibling threads.

3. INTRA-APPLICATION SHARING

In this section, we investigate the performance impact of memory resource sharing for three key datacenter applications. Experiments and measurement are conducted using different thread-to-core (TTC) mappings to study the impact of intra-application sharing, defined as resource sharing among the threads of an individual multi-threaded application.

3.1 Experiment Methodology

The primary platform used in this paper is a dual socket Intel Clovertown (Xeon E5345), shown in Figure 1. Each socket has 4 cores. Each 2 cores on the same socket are sharing a 4MB 16 way last level cache (L2). The platform is running Linux kernel version 2.6.26 and a customized GCC 4.4.3. We also conducted experiments on the Intel Westmere, which is presented in Section 4.3.

Table 1 presents a detailed description of the five datacenter applications we used in our study. In the datacenter latency-sensitive applications are either run alone on a machine or co-located with a batch application to improve machine utilization. Our study mirrors this execution policy as we focus on the latency-sensitive applications shown in Table 1. Also, instead of measuring instruction per cycle (IPC) or execution time, we use each application's specified performance metric in this study. Application-specific metrics more accurately describe performance than application agnostic metrics such as IPC [3]. The performance metrics are also shown in Table 1. The load for each application is real world query traces in production datacenters. A load generator is set up to test the peak capacity behavior of these applications. The performance shown is applications' stable behavior after the initialization phase. Because our measurements use a large amount of queries from production, these applications' behaviors and characteristics are representative of real-world execution.

In this section we describe experiments when the application is running alone to study the interaction within a multi-threaded application with the underlying resource sharing and the resulting performance variability. Three measurements are conducted with three thread-to-core mappings: `{XXXX....}`, `{XX..XX..}`, and `{X.X.X.X.}`. The performance difference between mapping configurations demonstrates how sharing LLC, sharing FSB, or sharing both can constructively or destructively impact the performance of applications of interest. In each mapping, we use `taskset` to map threads to cores. This allows us to study the resource sharing outside of the default OS scheduler's algorithm. This methodology is shown to be valid for measuring the impact of cache sharing by prior work [45]. Applications are parameterized to have a fixed load execute across 4 cores. All experiments were run three times and the average measurement is presented.

3.2 Measurement and Findings

Figure 6 demonstrates the performance variability due to different TTC mappings for the latency sensitive applica-

tions presented in Table 1. For each applicaton, the x axis shows the subset of cores to which the application is mapped. The y axis shows each application's performance in each TTC mapping scenario, normalized by its performance using the mapping {X.X.X.X.}.

The results show that the performance impact of memory resource sharing for these applications is significant, up to 22% for *contentAnalyzer*, 18% for *bigtable* and 8% for *webSearch*. Secondly, each application prefers different sharing configurations. Both *contentAnalyzer* and *webSearch* prefer to run on separate LLCs and separate FSBs. The mapping {X.X.X.X.} has 10% performance improvement for *webSearch* and 20% for *contentAnalyzer* compared to mapping {XXXX....}, when all threads are on the same socket sharing 2 LLCs and a single FSB. On the other hand, *bigtable* achieves the best performance when running on the same socket sharing 2 LLCs and a FSB, and the {X.X.X.X.} mapping has a 18% degradation. When taking a deeper look, for *contentAnalyzer* and *webSearch*, the difference between the 1st bar and the 2nd bar indicates the impact of cache sharing when available FSB bandwidth remains the same; the difference between the 2nd and the 3rd bar indicates the impact of sharing FSB versus having separate FSBs.

For *bigtable*, sharing LLC has a constructive impact on performance. The 3rd bar is slightly higher than the 2nd bar, indicating that FSB bandwidth may not be a main bottleneck from *bigtable*. On the other hand, the reduced coherence latency on the same socket may give mapping {XXXX....} a slight advantage over {XX..XX..}.

3.3 Investigating Performance Variability

To confirm that different memory sharing configurations provided by the different thread-to-core mapping is the main cause of the performance variability, we also conducted experiments to collect performance counters information. Performance counters including last level cache misses, bus transactions, MESI states of LLC requests are collected using `pfmon` [12].

Last Level Cache Misses: Figure 7 shows the average number of last level cache misses per million instructions for each application's execution in each TTC mapping scenario normalized to the scenario {X.X.X.X.}. Misses per million instructions is used because in this experiments we are comparing the misses caused by a fixed section of code. Figure 7 shows that the LLC misses trend is fairly consistent with the performance trend in the different mapping scenarios. *Content Analyzer* and *webSearch* both have an increase in last level cache misses when transitioning from not sharing LLC to sharing LLC, indicating contention for LLC occurs among threads. This explains the performance degradation from these two applications' 1st bar to 2nd and 3rd bars in Figure 6. *Bigtable* on the other hand, has a decrease in LLC misses when transitioning from not sharing LLC to sharing LLC, indicating the cache sharing is constructive and threads are sharing data that fits in the LLC. This explains the performance improvement from *Bigtable* 1st bar to 2nd and 3rd bar in Figure 6.

FSB Bandwidth Consumption: Figure 8 shows the average number of bus transaction requests per million instructions in different mapping scenarios, normalized by the rate in scenario {X.X.X.X.}. The number of bus transactions is measured using the `BUS_TRANS_BURST` event, which counts the number of full cache line requests (64 bytes). The bus bandwidth consumption is consistent with the last level cache misses and performance trends. The increase in last level cache misses causes the increase in bus requests which degrades performance. For *contentAnalyzer* and *webSearch*, bus requests per million instructions in mapping scenarios {XX..XX..} and {XXXX....} are similar. However, their performance is worse in the mapping scenario {XXXX....}. This is due to the contention for the FSB. For the same amount of bus requests, having 2 FSBs provides a performance advantage. This is also supported by the observation that *contentAnalyzer* has higher bus requests than *webSearch*, and *contentAnalyzer* suffers a bigger degradation transitioning from using 2 FSBs to sharing a single FSB on one socket.

Data Sharing: We further investigated the level of data sharing within each application to explain why some applications are benefiting and others are suffering from cache sharing. Figure 9 shows the number of L2 Requests per millisecond in five states: Modified, Exclusive, Shared, Invalid and Prefetch. This figures shows that *bigtable* has the most amount of sharing between data in the LLC, which is also consistent with the observation that *bigtable* benefits from cache sharing.

Summary: In this section we show that the impact of sharing the last level cache can either be positive or negative and can be significant (up to 10%). Bus contention also has a fairly significant impact on performance and contributes another 10% performance variability. For applications that have higher levels of sharing, a positive side effect of placing all threads close to each other and sharing a bus is observed. These results demonstrate the importance of a good thread-to-core mapping that mimics the application's inherent data sharing pattern.

4. INTER-APPLICATION SHARING

In this section, we present the performance impact of memory resource sharing when co-locating multiple applications on a machine. We define inter-application resource sharing as resource sharing between applications. As we discussed in Section 2, co-location is important for improving machine utilization, especially for a multi-socketed multicore machine. However, co-location may introduce detrimental performance degradation due to contention for shared resources. For some platforms, certain memory components may be always shared among all threads from all applications. For example, for the dual-socket Clovertown used in our experiments, the memory controller hub and memory bus are always shared among all execution threads. However, for resources such as the LLC and FSB, an application can, share only LLC(s) or only FSB(s) or both, among its own threads, or with another application. As we show in the previous section, resource sharing within an application may have either constructive or destructive impact. However, when there is no explicit inter-process communication between applications, resource sharing between applications are either neutral or destructive. Depending on the application and its co-runners, the amount of impact from sharing different resources between applications may vary. In this section, we study the impact of LLC and FSB sharing and how the impact affects thread-to-core mapping decisions in the presence of co-location.

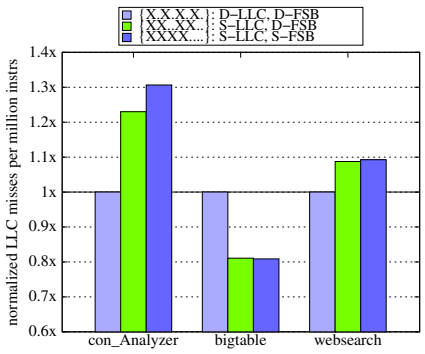

Figure 7: Average LLC misses per million instructions

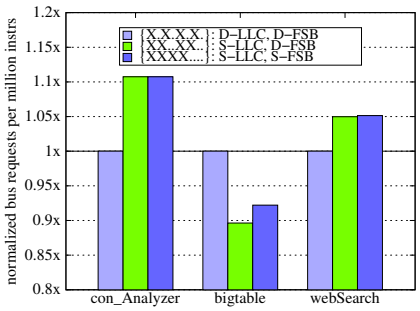

Figure 8: Average bus transactions per million instructions

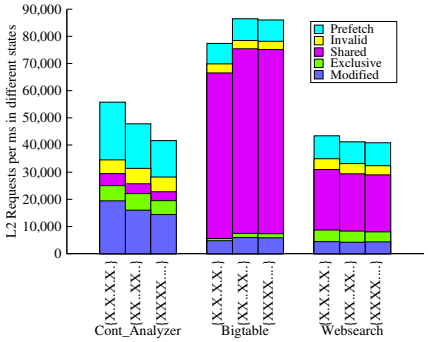

Figure 9: L2 Requests in MESI States and in Prefetch State

4.1 Experiment Design

Similar to Section 3, we study the impact of resource sharing by comparing the performance variability for key applications in three TTC mappings scenarios. The three TTC mappings are: `{XXXX****}`, `{XX**XX**}`, and `{X*X*X*X*}`. For this study, we use `*` to denote a thread of the co-running application. We use the batch applications *stitcher* and *protobuf* (described in Table 1) as co-running applications. Since batch applications are often co-located with key latency sensitive applications in production, and we are focusing on the three important latency sensitive applications, we measure the performance for each of the three latency sensitive applications, *contentAnalyzer*, *bigtable* and *webSearch*, when sharing resources with each co-runner in each of the three sharing configurations.

4.2 Measurement and Findings

Figure 10 shows *contentAnalyzer*'s performance when it is co-running with other applications. The first cluster of bars show *contentAnalyzer*'s performance when it is co-located with *stitcher*, and the second cluster, when it is co-located with *protobuf*. In this figure, the *contentAnalyzer*'s performance is normalized by three different baselines. Specifically, its performance when co-located in each thread-to-core mapping scenario is normalized by its performance when running alone in the corresponding mapping scenario. For example, the first bar in the first cluster shows *contentAnalyzer*'s performance when it is running with *stitcher*. The thread-to-core mapping is denoted as `{X*X*X*X*}`, `X` denoting *contentAnalyzer*'s threads and `*` denotes *stitcher*'s threads. This performance is normalized by *contentAnalyzer*'s performance when it is running alone using mapping `{X.X.X.X.}`. This figure demonstrates the performance interference caused by adding *stitcher* and by adding *protobuf* when *contentAnalyzer* is bound to a certain subset of cores. Interestingly, in different TTC mapping scenarios, the same co-runner causes different amounts of degradation to *contentAnalyzer*. This is because in different mapping scenarios, co-locating a co-runner to the available idle cores leads to sharing of different resources between co-running applications. The first bar in the first cluster shows a degradation of 35% caused by sharing both LLC and FSB between *contentAnalyzer* and *stitcher*. The second bar shows a degradation of 22% caused by only sharing the FSB bandwidth between the two applications. Note that the performance degradation shown by the third bar is due to interference caused by *stitcher* for sharing the memory controller hub and the rest of the memory system with *contentAnalyzer*, which is unavoidable in the topology of the platform in our experiment.

Figure 13 shows *contentAnalyzer*'s performance when it is running alone and it is co-running, normalized by a single baseline: its performance when running alone in the mapping `{X.X.X.X.}`. Our key observation here is that the best thread-to-core mapping for *contentAnalyzer* changes. When it is running alone its best mapping is `{X.X.X.X.}`. When running with *protobuf*, it is still `{X*X*X*X*}`. When running with *stitcher* the best mapping changes to `{XXXX****}`. With the same co-runner, the performance variability of *contentAnalyzer* between the worst and the best mapping can be fairly significant. When running with *protobuf*, the performance swing between different mappings is around 11%.

Figures 11 and 14 show *webSearch*'s performance when it is co-running with *stitcher* and *protobuf*. Similar to Figure 10, in Figure 11, each bar represents the performance of *webSearch* when co-located in a certain mapping scenario, normalized by its performance when running alone in the same mapping scenario. Figure 14 shows its performance when co-located, normalized by a single baseline, namely, when it is running alone and mapped to `{X.X.X.X.}`. Figures 11 and 14 show that *WebSearch*'s performance variability has a similar trend as *contentAnalyzer*'s. Also similarly, the optimal mapping for *webSearch* changes depending on if it is running alone or which application it is running with. One difference worth noticing between *contentAnalyzer* and *webSearch* is that when *webSearch* is co-located with *protobuf*, its best mapping is `{XX**XX**}`.

In contrast to both *contentAnalyzer* and *webSearch*, *bigtable* prefers to share the LLC and FSB among its own threads *both* when it is running alone and when running with other applications as shown in Figure 12 and 15. However, there is a significant performance swing between thread-to-core mappings. When it is running with *stitcher*, there is a 40% performance difference between the three mappings.

Based on these experiment results, we can categorize these applications based on the underlying sharing configurations they prefer when running alone and running with other applications. The categorization is shown in Table 3. This table presents the optimal mapping for each application and

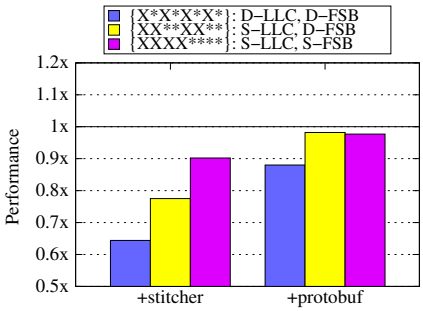

Figure 10: ContentAnalyzer. Normalized to solo performance

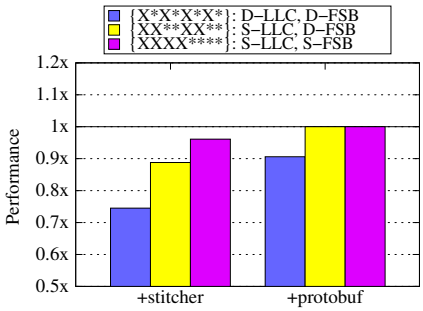

Figure 11: Websearch. Normalized to solo performance

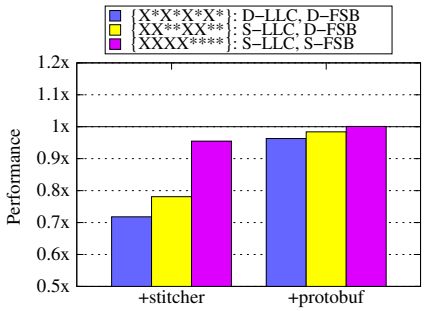

Figure 12: Bigtable. Normalized to solo performance

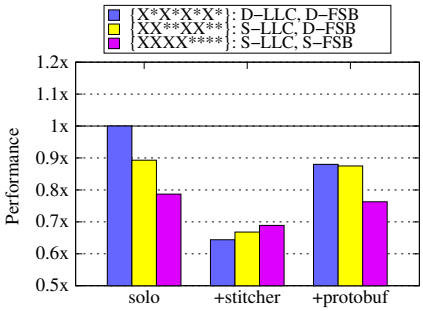

Figure 13: ContentAnalyzer. Normalized to solo performance with `{X.X.X.X.}`

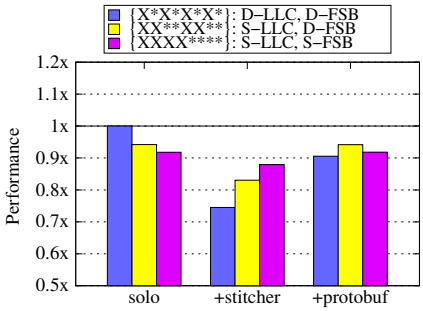

Figure 14: Websearch. Normalized to solo performance with `{X.X.X.X.}`

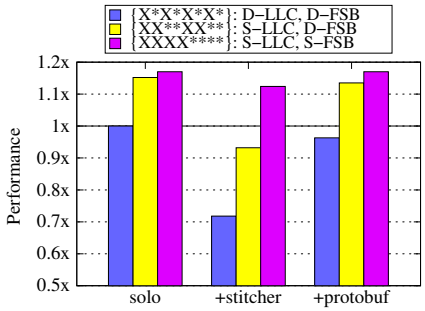

Figure 15: Bigtable. Normalized to solo performance with `{X.X.X.X.}`

highlights the changes in mapping preferences in different situations. In the table, `S` stands for "shared" and `D` stands for "distributed". In contrast to the conclusions about PARSEC suite in prior work [45] (presented in Table 3's last row), our experiments demonstrate that industry-strength datacenter applications have diverse preferences in resource sharing and TTC mappings.

Summary: In this section, our experiments show that, depending on the co-runner, sharing LLC and FSB with the corunner can have a significant impact. An application's performance swing between its best and worst thread-to-core mapping can be significant. Also, the optimal thread-to-core mapping is different for each application and may change when the application's corunner changes. This result indicates the importance of an intelligent system for thread-to-core mapping that is aware of the underlying resource topology and possible sharing configurations.

4.3 Varying Thread Count and Architecture

In this section, we describe experiments to evaluate whether the above observations are also applicable when the number of threads, the architecture or the memory topology changes.

4.3.1 Varying Number of Threads

We studied the impact of memory resource sharing when the latency sensitive applications have 2 and 6 threads. All experiments are conducted on Clovertown described in Section 3.1. Figure 16 presents the scenario when each latency sensitive application is configured to have 2 threads. This figure presents the latency sensitive application's performance when it is running alone, co-located with 6 threads of *stitcher*, and co-located with 6 threads of *protobuf*. In the figure, we use C for *contentAnalyzer*, W for *webSearch*, B for *bigtable*, S for *stitcher* and P for *protobuf*. The y axis shows each of the three latency sensitive applications' performances normalized by the performance when running alone in the `{X...X...}` mapping. Figure 17 presents the scenario when each latency sensitive application is configured to have 6 threads. In this figure, the performance of each latency sensitive application is measured when it is running alone, co-located with 2 threads of *stitcher*, and co-located with 2 threads of *protobuf*.

In general, our results show that in both 2-thread and 6-thread cases, each application's sharing preferences are similar to its preferences in the 4-thread case. For example, *bigtable* prefers sharing cache among its threads when it has 2 threads, 4 threads and 6 threads. *ContentAnalyzer* prefers sharing cache and FSB with its own thread when running with *stitcher* and prefers distributing its threads when running alone or with *protobuf*. For *webSearch*, when running with *stitcher*, the optimal mapping is always sharing with its own threads. Moreover, similar to the 4-thread case, for each application, its optimal thread-to-core mapping changes when its co-runners change.

4.3.2 Varying Architecture

We also conducted experiments on a Intel's Westmere platform. Our experiment platform is a dual-socket Intel Xeon X5660. Each socket has 6 cores. The memory topology of this architecture is quite different from Clovertown used in previous sections. All six cores on the same socket share a 12 MB last level cache. Each chip has its own integrated

Table 3: Optimal Thread-To-Core Mapping in Solo and Co-location Situations

Benchmark	Solo	w/ *Stitcher*	w/ *Protobuf*
bigtable	{XXXX....}: S-LLC, S-FSB	{XXXX****}: S-LLC, S-FSB	{XXXX****}: S-LLC, S-FSB
contentAnalyzer	{X.X.X.X.}: D-LLC, D-FSB	{XXXX****}: S-LLC, S-FSB	{X*X*X*X*}: D-LLC, D-FSB
webSearch	{X.X.X.X.}: D-LLC, D-FSB	{XXXX****}: S-LLC, S-FSB	{XX**XX**}: S-LLC, D-FSB
PARSEC	does not matter	N/A	N/A

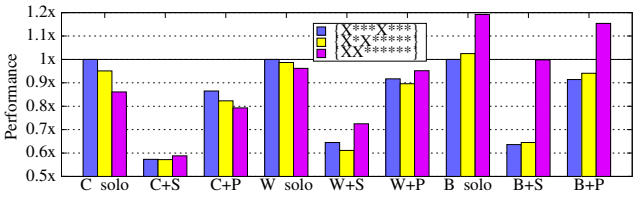

Figure 16: 2 threads of a latency sensitive application colocated with 6 threads of a batch application, normalized to the latency sensitive application's solo performance in {X...X...} mapping

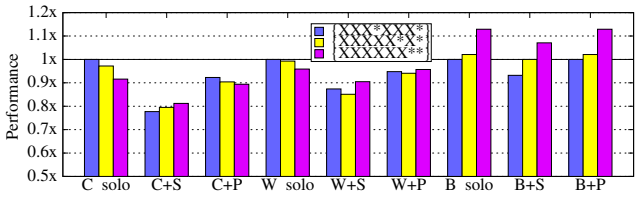

Figure 17: 6 threads of a latency sensitive application colocated with 2 threads of a batch application, normalized to the latency sensitive application's solo performance in {XXX.,XXX.} mapping

memory controller, and has 3 channels of 8.5GB/s/channel bus connecting to DIMM. Processors are connected through QuickPath interconnect (QPI). We conduct experiments to evaluate the performance impact of sharing the LLC and memory bandwidth on the same socket versus distributing threads to two sockets for our three key latency sensitive datacenter applications. Figures 18 and 19 present the results when each application is running alone with 2 threads and 6 threads. We use a similar notation to present the thread-to-core mapping. For example, {X.....X.....} indicates two threads are mapped to two different sockets on this architecture. In both figures, each application's performance is normalized to its performance when its threads are evenly distributed across 2 sockets. These results show that, due to the different memory resource sharing patterns, different thread-to-core mappings can cause significant performance variability. This is similar to results on Clovertown. On Westmere, the performance swing is as high as 10%. *Bigtable* behaves similarly on both architectures as it always benefits from cache sharing. However, interestingly, while *contentAnalyzer* on Westmere benefits from cache sharing in the 2-thread case, in the 6-thread case, it suffers from cache sharing. In the 8-thread case, which we do not show here, its performance degradation due to cache sharing is over 20%. On the other hand, on Clovertown, it always suffers from cache sharing. This discrepancy between its sharing preference on two architectures may be due to the fact that Westmere has a 12MB LLC instead of 4MB LLCs on Clovertown. Whether an application can benefit from last level cache sharing also depends on the size of the cache and the number of threads that are executing.

In light of the space constraint, for the co-location study, we only present the results when 6 threads of latency sensitive application co-running with 6 threads of corunner (Figure 20). The y axis shows each latency sensitive application's performance, normalized to its performance when running alone in mapping scenario {XXX...XXX...}. This result shows that on Westmere, depending on the co-runner, the optimal thread-to-core mapping may also change. This is also consistent with the observation on Clovertown.

5. THREAD-TO-CORE MAPPING

To achieve a good thread-to-core mapping to best utilize shared resources, it is important to characterize applications' interaction with these shared resources, and pinpoint the potential bottlenecks among the shared resources. In this work, we have identified three important memory characteristics of an application that can be exploited to understand the preferences in memory resource sharing configurations, including: its memory bandwidth consumption, the amount of data sharing within the application, and its footprint in the shared cache.

[**Memory Bandwidth Usage**] We first investigate our applications' memory bandwidth usage. On Clovertown, we focus on the FSB bandwidth because FSB is a main sharing point for memory bandwidth on this architecture. Our previous experiments in Sections 3 and 4 show that when threads are sharing the FSB, their performance may degrade. The amount of degradation may differ for each application, depending on which application is co-located with it. We hypothesize that the amount of bus bandwidth usage for each application is a good indicator for determining its proper FSB sharing configuration.

Figure 21 presents the bus bandwidth consumption per thread pinned to one core for all five applications. The bus request rate is measured using the BUS_TRANS_BURST event. 15,000 bus transactions/ms for a thread of *contentAnalyzer* translates to $15,000 \times 64Byte = 0.96GB/s$. The total bus transactions/ms for all fours threads running on four cores can be as high as $0.96GB/s \times 4 = 3.8GB/s$. The theoretical FSB peak bandwidth on this platform is 10.6 GB/s. When using a micro-benchmark that measures peak bandwidth, STREAM [30], the observed maximum sustained bandwdith is 5.6GB/s. When four threads of *contentAnalyzer* are sharing a single FSB, the bus utilization is close to 70%. Using a similar calculation, *stitcher*'s bandwidth demand is 1.6GB/s per core. This figure shows that *stitcher* has the highest bus bandwidth usage. *WebSearch* and *bigtable* have medium bus demands and *protobuf* has the lowest bus bandwidth demand. This is consistent with the mapping preferences shown in Table 3. When *webSearch* and *contentAnalyzer* are running alone, because of the medium-high bus demand,

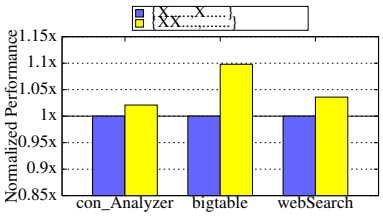

Figure 18: 2 threads of latency sensitive applications running alone on Westmere

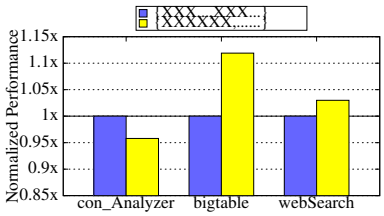

Figure 19: 6 threads of latency sensitive applications running alone on Westmere

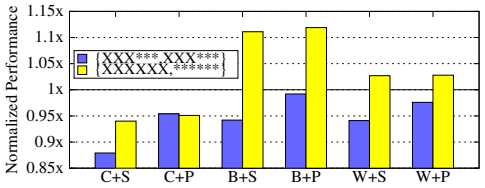

Figure 20: 6 threads of latency sensitive applications co-running with 6 threads of batch applications on Westmere;

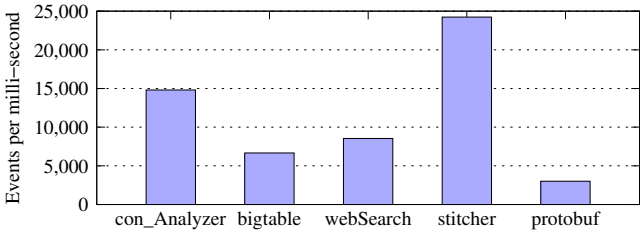

Figure 21: Bus Burst Transactions (full cache line) per millisecond per one thread

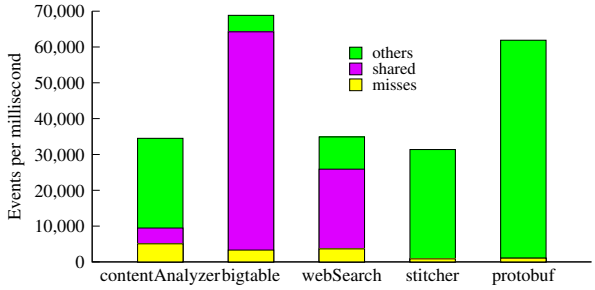

Figure 22: LLC misses/ms, LLC_requests_Share/ms and LLC reference/ms

it is preferable to spread threads on two sockets and use 2 FSBs. However, when they run with *stitcher*, both prefer not to share a FSB with *stitcher* because *stitcher* has a much higher bus demand and can cause more performance degradation. On the other hand, when running with *protobuf*, both *webSearch* and *contentAnalyzer* both benefit from sharing FSB with *protobuf* instead of their own threads. *Bigtable* benefits from sharing last level cache and FSB when it is running alone, thus it is preferable for *bigtable* to share these two resources with its own threads when running with other applications. This experiment demonstrates that bus bandwidth consumption is an important characteristic when determining good thread-to-core mappings.

Our experiments in Sections 3 and 4 also demonstrate that sharing a cache can cause significant performance impact. There are two key characteristics to consider when studying the interaction between an application and a shared cache: the amount of data sharing among an application's threads and the application's footprint in the shared cache.

[Data sharing] In Section 3 we show that the percentage of cache lines that are in the "share" states can indicate an application's level of data sharing. Figure 22 presents the average LLC reference rate for a thread of each application. In this figure, we bin LLC references into three categories: LLC misses, LLC references that are in "share" state, and others (including prefetch state and cache hit that are not in "share" state). *Bigtable* has the highest percentage of cache requests that are in the share state and *contentAnalyzer* has the lowest. This is consistent with our findings that *bigtable* prefers to share LLC when it is running alone as well as when it is running with other applications while *contentAnalyzer* does not. On the other hand,*webSearch* has a relatively high level of data sharing. However, sharing the last level cache among its threads would cause a performance degradation. This is because when deciding if sharing a cache would improve or degrade an application's performance and which thread the application should share the cache with, we need to consider not only data sharing but also the potential of cache contention.

[Cache Footprint] When the total size of two or more threads' footprints is larger than the shared cache, *contention* occurs. Previous work has studied how to identify an application's cache contention characteristics. Zhuravlev et. al [46], Knauerhase et. al [25] and Mars et. al [29] show that last level cache miss rate is a good indicator to estimate the footprint size and predict the potential performance degradation an application may cause to its co-runners. Figure 22 presents the LLC miss rate for all five applications. This figure shows that *contentAnalyzer* has a higher LLC miss rate than *webSearch* and less percentage of share state cache lines. This is consistent with the fact that *contentAnalyzer* suffers more from cache contention than *webSearch*, shown in Figure 6. An application's cache characteristics are important when deciding a good TTC mapping. And both data sharing and cache footprint need to be considered.

5.1 A Heuristic Approach to TTC Mapping

Based on an application's characteristics in terms of their resource usage when running alone, we can predict a good thread-to-core mapping that takes advantage of the memory sharing topology when applications are co-located. Algorithm 1 shows a heuristic algorithm to make such a decision.

The basic idea behind the heuristic is that since we can characterize applications based on their potential bottlenecks (bus usage, shared cache usage and the level of data sharing), when co-locating, we should maximize the potential benefit from sharing and avoid mapping threads that have the same resource bottleneck. For example, if the application has a high level of data sharing, the mapping should allow its threads to share resources such as LLC.

We also prioritize the latency-sensitive application's performance (denoted as P in the algorithm) over its corunner (C) 's. For example, the heuristic algorithm compares the resource usage of P's threads with that of the co-running applications' threads and select the thread(s) that have the least usage of the same resource to co-locate.

Algorithm 1: Resource-Characteristics-Based Mapping Heuristics

```
   Input: P: Latency-sensitive app; C: Corunning app
   Output: a thread-to-core mapping
 1 if P.DataSharing = high then
 2 |   map(P, share_LLC);
 3 |   if P.Bus_Usage < C.Bus_Usage then
 4 |   |   map(P, [share_LLC, sharing_FSB]) ;
 5 |   else
 6 |   |   map(P, [share_LLC, distributed_FSB]) ;
 7 |   end
 8 else
 9 |   if P.Bus_Usage < C.Bus_Usage then
10 |   |   map(P, sharing_FSB) ;
11 |   |   if P.LLC_Footprint = high then
12 |   |   |   map(P, [distributed_LLC, sharing_FSB]);
13 |   |   else
14 |   |   |   map(P, [share_LLC, sharing_FSB]);
15 |   |   end
16 |   else
17 |   |   if P.LLC_Footprint < C. LLC_Footprint then
18 |   |   |   map(P, [share_LLC, distributed_FSB]);
19 |   |   else
20 |   |   |   map(P, [distributed_LLC, distributed_FSB]);
21 |   |   end
22 |   end
23 end
```

5.1.1 Evaluating the Heuristics

To evaluate the heuristic algorithm, we apply it to six co-running application pairs and compare the predicted best mapping with the ground-truth best mapping. We use FSB bandwidth consumption to compare the `Bus_Usage` in the algorithm; and use LLC miss rate as an approximate proxy to compare applications' `LLC_Footprint`. To take data sharing into account when comparing cache footprints for an multi-threaded application, we use

$$LLC\_MissRate \times (1 - \frac{LLC\_shared\_state\_requests}{LLC\_all\_requests}) \qquad (1)$$

The prediction result using heuristic algorithm is presented in Table 4.

Our heuristic approach correctly predicts the best mapping in 4 out of 6 co-running pairs. In two cases, the heuristic algorithm also makes fairly good decisions: the performance difference between the predicted mapping and the optimal mapping is less than 2% for both (Figures 14 and 15). The advantages of our heuristic approach is that it is effective and requires only simple runtime support. However, there are two main limitations of this approach. First, these characteristics must be collected for each application. Second, because each architecture has different topologies and sharing points, a new algorithm needs to be generated on an architecture by architecture basis. Also, an application characteristics may not be perfectly captured. For example, using LLC miss rate to approximate the cache footprint is not perfect [28], especially when there is data sharing between threads. These limitations motivate an adaptive approach that is more flexible and portable.

Table 4: Predicted Thread-To-Core Mapping Using the Heuristic Approach

Benchmark	w/ *Stitcher*	w/ *Protobuf*
bigtable	**Optimal:** `{XXXX****}`;	**Optimal:** `{XXXX****}`;
	Predicted: `{XXXX****}`	**Predicted:** `{XX**XX**}` (suboptimal: 1% worse)
contentAnalyzer	**Optimal:** `{XXXX****}`;	**Optimal:** `{X*X*X*X*}`;
	Predicted: `{XXXX****}`	**Predicted:** `{X*X*X*X*}`
webSearch	**Optimal:** `{XXXX****}`;	**Optimal:** `{XX**XX**}`;
	Predicted: `{XXXX****}`	**Predicted:** `{X*X*X*X*}` (suboptimal: 1% worse)

5.2 An Adaptive Approach to TTC Mapping

In this section we present `AToM`, an **A**daptive **T**hread-to-c**o**re **M**apper. Our experiments in the previous sections show that the optimal thread-to-core mapping may change when the number of threads, co-running application, or architecture changes. These variations indicate that an adaptive learning approach is promising for the intelligent thread-to-core mapping. `AToM` uses a *competition heuristic* to adaptively search for the optimal thread-to-core assignment for a given set of threads. This approach includes two phases: a learning phase and an execution phase.

[**Learning Phase**] During the learning phase, AToM empirically puts various thread-to-core mappings against each other to learn which mapping performs best. Each thread-to-core mapping is given an equal amount of time, and the best performing mapping is selected as the winner of the competition. Although randomly mapping threads to cores may generate a large amount of varying mappings, because most of memory topologies are symmetric, the search space for equivalent mappings is greatly reduced. For example, for 2 core mapping cases, there are only three classes of mappings (Table 2) that represent three different sharing configurations.

[**Execution Phase**] During this phase the winning thread-to-core mapping is run for a fixed or adaptive period of time before another competition is held. In this work, we allow our execution phase to run indefinitely. The datacenter applications presented in this work have steady phases, and each competition produces the same winner. Therefore, reentering the learning phase only produces an unnecessary overhead.

5.2.1 Evaluating AToM

In this work, we have constructed a prototype of AToM tuned for the datacenter. During the learning phase, AToM cycles through three taskset configurations for a period of 10 minutes each. For an application in the datacenter we use a long period to minimize noise in our competition. The datacenter applications presented in this work are long running programs, running for days and weeks at a time; however for our experimental runs we allow only 2 hours of execution. Figures 23 and 24 present the results of our experimentation on both Clovertown and Westmere. In the figures, we use C for *contentAnalyzer*, W for *webSearch*, B for *bigtable*, S for *stitcher* and P for *protobuf*. The y axis shows each of the three latency sensitive applications' performance,

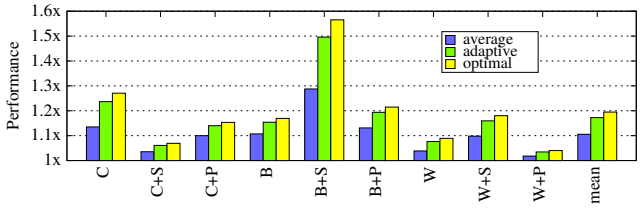

Figure 23: Adaptive Thread-To-Core Mapping on Clovertown

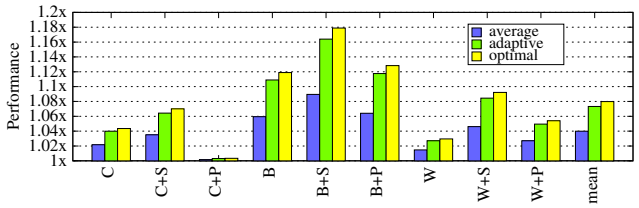

Figure 24: Adaptive Thread-To-Core Mapping on Westmere

normalized by its performance when running alone in the `{X...X...}` mapping. In both figures, the x axis shows 9 machine loads, including each of our latency sensitive applications running alone and co-located with our batch applications. Each application is configured to run on 4 cores. The y axis shows the performance of our latency sensitive application normalized to the worst assignment. As this figure shows, AToM is quite effective, achieving near optimal performance. In each case, AToM outperforms the average case (average random assignment) by up to 22%, and is significantly better performing than the worse case assignments.

6. RELATED WORK

Kozyrakis et al. [26] present a study of emerging large scale online services workloads and show how their characteristics effect the datacenter system design for architects. Other studies that characterize the interaction between emerging datacenter workloads and underlying architectures include studies by Reddi et al. [37] and by Soundararajan et. al. [40]. Hardavellas et al. [17, 16] investigate sharing characteristics for database workloads and webservice workloads. Our work looks at large scale Google workloads and focuses on their interactions with the underlying memory resource sharing. Zhang et al. [45] examine the influence of cache sharing on multithreaded applications using the PARSEC suite, and conclude that there is neither significant constructive or destructive influence from cache sharing. Our work expands the study to both cache and bus sharing using commercial datacenter applications and shows that for these applications, there is both positive and negative significant performance impact. We also present approaches to take advantage of the performance variability.

In the architecture community, much work has investigated and proposed approaches to addressing memory resource sharing and contention. New architectural supports for cache and memory bus partitioning, management and monitoring are proposed and evaluated [10, 34, 36, 42, 27, 24, 41, 11, 19]. Most studies evaluate the performance impact of resource sharing and different schemes of partitioning using simulations. While simulations are important and necessary when designing new architectures, examining large-scale applications on existing hardware is important for improving our understanding of emerging workloads as well as improving performance in the deployed systems. Effective scheduling approaches to addressing resource sharing on SMT processors are also proposed [39, 13, 35].

In the area of OS and runtime scheduling, most studies focus on job co-scheduling to avoid co-locating cache contentious applications together to improve performance and fairness [25, 46, 14, 6, 31, 21, 20]. Approaches to alleviating resource contention and guaranteeing QoS by controlling the execution rate through hardware features or a runtime are proposed [18, 29]. Banikazemi et al. [4] present a scheduler to adaptively schedule threads to cores to take advantage of the cache topology to alleviate resource contention. Most of the above works use single threaded applications and focus on the resource contention aspect. Tam et al. [44] present a technique to cluster communicating threads onto the same chip to reduce the communicating latency. However, their approach focuses on the constructive effect of sharing resource without considering the potential resource contention. There is also theoretical work to model sharing caches among threads [7, 8]. In addition, compilation techniques that are aware of the cache sharing and cache topology also proposed [23, 22, 45, 38].

7. CONCLUSION

In this work, we present an in depth study of the interaction of industry-strength datacenter applications and the shared resources in the underlying memory subsystem. We have found a performance swing of up to 25% for web search and 40% for other key applications, simply based on how application threads are mapped to cores. This is particularly significant considering that at the datacenter scale, a performance improvement of even 1% can results in millions of dollars saved. This finding demonstrates the importance of performing intelligent thread-to-core mapping for applications in the datacenter. In this work, we have also presented key application characteristics that impact the optimal thread-to-core mapping decisions, and show how these characteristics can be used to build heuristics to perform thread-to-core mappings. One key insight and observation from this study is the fact that when threads of multiple applications with diverse memory behaviors are co-located, the ideal mapping for a given application is different than if running alone on the system. In addition to the heuristic approach, we present an adaptive approach, and conclude that an adaptive approach is more attractive as it is simple to implement and, at least for long running datacenter applications, is quite effective. Using this approach, the performance of datacenter workloads improved by up to 22% over status quo thread-to-core mapping and performs within 3% of optimal mapping on average.

8. REFERENCES

[1] Latent semantic analysis. http://en.wikipedia.org/wiki/Latent_semantic_analysis.
[2] Protocol buffer. http://code.google.com/p/protobuf/.
[3] A. Alameldeen and D. Wood. Ipc considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8, 2006.
[4] M. Banikazemi, D. Poff, and B. Abali. Pam: a novel

performance/power aware meta-scheduler for multi-core systems. *In SC '08*, Nov 2008.

[5] L. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2), 2003.

[6] M. Bhadauria and S. McKee. An approach to resource-aware co-scheduling for cmps. *In ICS '10*, Jun 2010.

[7] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. *In SPAA '04*, Jun 2004.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. *In HPCA-11*, pages 340– 351, 2005.

[9] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[10] J. Chang and G. Sohi. Cooperative cache partitioning for chip multiprocessors. *Proceedings of the 21st annual international conference on Supercomputing*, page 252, 2007.

[11] E. Ebrahimi, C. Lee, O. Mutlu, and Y. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *In ASPLOS '10*, Mar 2010.

[12] S. Eranian. What can performance counters do for memory subsystem analysis? *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*, pages 26–30, 2008.

[13] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. *In ASPLOS '10*, Mar 2010.

[14] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. *In PACT '07*, Sep 2007.

[15] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. *In Micro '07*, pages 343–355, 2007.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive nuca: near-optimal block placement and replication in distributed caches. *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, Jun 2009.

[17] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. *Proc. CIDR*, 2007.

[18] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. *In ICS '09*, Jun 2009.

[19] 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. *In SIGMETRICS '07*, Jun 2007.

[20] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. *In PACT '08*, Oct 2008.

[21] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. *High Performance Embedded Architectures and Compilers*, pages 201–215, 2010.

[22] M. Kamruzzaman, S. Swanson, and D. Tullsen. Software data spreading: leveraging distributed caches to improve single thread performance. *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, Jun 2010.

[23] M. Kandemir, S. Muralidhara, S. Narayanan, Y. Zhang, and O. Ozturk. Optimizing shared cache behavior of chip multiprocessors. *In MICRO-42*, pages 505–516, 2009.

[24] D. Kaseridis, J. Stuecheli, J. Chen, and L. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. *In HPCA-16*, pages 1–11, 2010.

[25] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *Micro, IEEE*, 28(3):54–66, 2008.

[26] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *Micro, IEEE*, 30(4):8–19, 2010.

[27] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. *In HPCA-16*, pages 1–12, 2010.

[28] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 167–176, New York, NY, USA, 2011. ACM.

[29] J. Mars, N. Vachharajani, R. Hundt, and M. Soffa. Contention aware execution: online contention detection and response. *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, Apr 2010.

[30] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/, Feburary 2005.

[31] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Apr 2010.

[32] M. Moreto, F. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. Flexdcp: a qos framework for cmp architectures. *SIGOPS Operating Systems Review*, 43(2), Apr 2009.

[33] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28:6–16, May 2008.

[34] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *In MICRO 39*, Dec 2006.

[35] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, F. Cazorla, M. Nemirovsky, and M. Valero. Thread to strand binding of parallel network applications in massive multi-threaded systems. *In PPoPP '10*, Jan 2010.

[36] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. *In PACT '06*, Sep 2006.

[37] V. Reddi, B. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, Jun 2010.

[38] S. Sarkar and D. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, pages 353–368, 2008.

[39] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, Dec 2000.

[40] V. Soundararajan and J. Anderson. The impact of management operations on the virtualized datacenter. *Proceedings of the 37th annual international symposium on Computer architecture*, pages 326–337, 2010.

[41] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *In ASPLOS XIII*, Mar 2008.

[42] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. *In HPCA '02*, pages 117– 128, 2002.

[43] R. Szeliski. Image alignment and stitching: a tutorial. *Found. Trends. Comput. Graph. Vis.*, 2(1):1–104, 2006.

[44] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, Jun 2007.

[45] E. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? *In PPoPP '10*, pages 203–212, 2010.

[46] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *In ASPLOS '10*, Mar 2010.