

Mitigating Memory Resource Contention in Warehouse Scale Computers

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

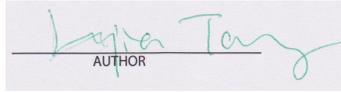
by

Lingjia Tang

May 2012

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science



AUTHOR

Lingjia Tang

Approved:

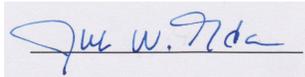


Advisor

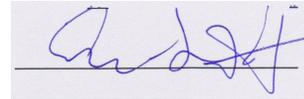
Mary Lou Soffa (Advisor)



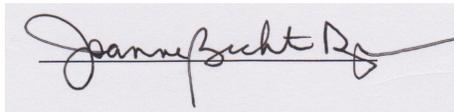
Westley Weimer (Chair)



Jack W. Davidson

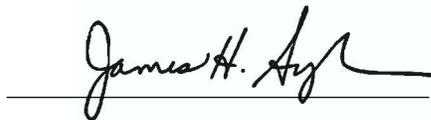


Kamin Whitehouse



Joanne Bechta Dugan

Accepted by the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

May 2012

Abstract

The class of modern datacenters hosting large-scale Internet services such as web-search, mail, and social networking has gained significant momentum in today's computing environments. However, these datacenters, recently coined as *warehouse scale computers* (WSCs), are extremely expensive to construct and operate. Improving *software performance* and *server utilization* is key to improving the efficiency and reducing the enormous cost in WSCs.

Modern WSCs are constructed using commodity multicore processors, on which part of the memory subsystem is shared. When multiple applications are co-located on a multicore machine, contention for the shared memory resources, such as caches and memory bandwidth, may occur. This contention can cause severe cross-core performance interference and significantly degrade application performance. Mitigating resource contention is critical for improving application performance. However, despite the wealth of research effort on contention management, little is known about how emerging large-scale web-service applications interact with the shared memory resources on commodity processors and how this contention can be mitigated to improve the performance of these applications.

In addition to performance, mitigating contention is also critical for improving the server utilization in WSCs. As multicore processors with expanding core counts continue to dominate the server market, the overall utilization of WSCs depends heavily on the consolidation of workloads to take advantage of the total computing potential provided by modern processors. However, many of the applications running in WSCs are user-facing, latency-sensitive applications with quality of service (QoS) requirements. These QoS requirements can be violated by the performance interference that can occur when multiple applications are consolidated on a single machine. As a result, the current common practice in WSCs is

to disallow the co-location of latency-sensitive applications with other applications. This approach is undesirable as it results in low machine utilization in WSCs and millions of dollars wasted.

This dissertation presents novel compilation and runtime approaches to significantly mitigate contention and thus improve performance, QoS and machine utilization in datacenters. Specifically, the main contributions of this dissertation include: 1) comprehensive investigation and characterization of the impact of memory resource sharing on industry-strength large-scale datacenter workloads, which expose new characteristics and insights contrary to recent literature; 2) the design of a heuristic based system and a runtime system to intelligently map application threads to cores to promote positive resource sharing and mitigate resource contention to improve application performance; and 3) the design of novel compilation techniques and runtime systems that statically and dynamically manipulate applications' contentious nature to enable the co-location of applications with varying QoS requirements and as a result, greatly improve server utilization in WSCs.

Acknowledgements

Many many people have helped me and contributed to my growth along the way. I first want to thank my advisor Mary Lou Soffa for always having faith in me even when no one else did. Thanks for your invaluable insights, guidance, support and encouragement.

Thanks to my dear husband, my awesome colleague and my best friend, Jason Mars, who encourages me to carry on when times are difficult. Without you and Dr. Soffa, I would have left the graduate school. So thanks for the all the enthusiastic pep talks, brainstormings, working with me together for every paper deadline, and your healthy criticisms.

Also, I thank Jack Davidson, Westley Weimer, Kamin Whitehouse and Joanne Bechta Dugan for serving on my dissertation committee. Their unique perspectives and comments helped enrich my dissertation work. I also thank Jack and Kevin Skadron for their help during my job search.

I thank Robert Hundt and Neil Vachharajani who were fantastic hosts for me at Google. I learned a great deal from them during those two summers. Thanks Robert for being super supportive all along. Thanks Dick Sites and all the Google folks for the insightful discussions. It was great fun to work with you all.

Thanks to my fellow graduate students, especially Tanima Dey and Wei Wang, who are also my coauthors and worked really hard with me on paper submissions.

Thanks to my friends in Charlottesville, Rui, Songbin, Jie, Xiaohua, Chong, Guofeng and many more for their support and for making Charlottesville a fun place.

Last but not least, many thanks to my parents, Anan Liu and Fusheng Tang for trying their best to give me the best education and always being there for me. Love you both!

Contents

1	Introduction	1
1.1	Memory Resource Sharing and Contention	2
1.2	Implications of Memory Resource Contention	3
1.2.1	The Impact of Contention on Performance	3
1.2.2	The Impact of Contention on Server Utilization	4
1.2.3	Trade-offs Between Performance and Utilization	4
1.3	Mitigating Contention	6
1.4	Two Strategies for Mitigating Contention	6
1.4.1	Mitigating Contention to Improve Performance	7
1.4.2	Mitigating Contention to Improve Utilization	9
1.5	Summary of Contributions	10
2	Background and Related Work	13
2.1	Warehouse Scale Computers	13
2.1.1	Cost	14
2.1.2	Application QoS	14
2.1.3	Job Scheduling, Application Colocation and Utilization	15
2.1.4	Machine Level	18
2.2	Related Work	18
2.2.1	Impact of Memory Resource Sharing	18
2.2.2	Novel Hardware Solutions to Mitigate Contention	19
2.2.3	Software Runtime and OS Approaches to Mitigating Contention	19
2.2.4	Cache Contention Aware Compilation	21

3	The Impact of Memory Resource Sharing	23
3.1	Memory Resource Sharing	24
3.2	Intra-Application Sharing	26
3.2.1	Experiment Methodology	26
3.2.2	Measurement and Findings	28
3.2.3	Investigating Performance Variability	29
3.2.4	Summary	32
3.3	Inter-Application Sharing	33
3.3.1	Experiment Design	33
3.3.2	Measurement and Findings	34
3.3.3	Varying Thread Count and Architecture	38
3.3.4	Summary	40
4	Thread-to-core Mapping	43
4.1	A Heuristic Approach to TTC Mapping	44
4.1.1	Evaluating the Heuristics	48
4.2	An Adaptive Approach to TTC Mapping	49
4.2.1	Evaluating AToM	50
5	Compiling for Niceness	52
5.1	QoS-Compile Overview	53
5.2	Identify Contentious Code Regions	55
5.2.1	Contentiousness and Sensitivity	56
5.2.2	Identify Contentious Regions	61
5.3	Compiler Transformations for Rate Reduction	64
5.3.1	Padding	64
5.3.2	Nap Insertion	66
5.3.3	Understanding Cooldown and Warmup	68
5.4	Evaluation	68
5.4.1	Setup and Methodology	69
5.4.2	Model for Code Region Identification	69

5.4.3	Compiler Transformations	73
5.4.4	QoS-Compile: Put it All Together	77
5.4.5	Google Applications	82
5.5	Summary	83
6	Reactive Niceness	84
6.1	Reactive-Niceness Overview	85
6.2	RN-Compile: Compiling for Reactive Niceness	88
6.3	RN-Runtime: Dynamic Detection and Reaction to QoS Degradation	89
6.3.1	Runtime	90
6.3.2	Detection and Reaction	92
6.4	Evaluation	95
6.4.1	Setup and Methodology	96
6.4.2	Effectiveness of Reactive-Niceness: Simple Heuristic	96
6.4.3	Effectiveness of Reactive-Niceness: Targeted Heuristic	99
6.4.4	Effectiveness of Reactive-Niceness: Phase Level Behavior	101
6.4.5	Overhead	105
6.4.6	Energy Efficiency of using Reactive-Niceness	106
6.4.7	Varying Architecture	107
6.5	Summary	109
7	Conclusions and Future Directions	110
7.1	Summary of Themes and Results	111
7.2	Future Direction	113
7.2.1	Managed Runtime for QoS and utilization in WSCs	113
7.2.2	Runtime systems and research infrastructure for WSCs	113

List of Figures

1.1	Topology of a Dual Socket Intel Clovertown	2
1.2	Current Two Options in WSCs. Option A, disallowing collocation of applications, achieves peak application performance at the sacrifice of machine utilization. Option B improves machine utilization and reduces the number of server machines needed. However, applications may suffer significant performance degradation, which can impair latency-sensitive applications' capability to deliver acceptable QoS.	5
1.3	Dissertation Overview - Understanding the impact of contention and 2 strategies to mitigate contention to improve performance and utilization	7
1.4	Enabled capabilities by our software systems. (A) illustrates the strategy 1, the TTC mapper's effect - improved performance comparing to baseline collocation situation, shown in Figure 1.2. (B) illustrates the strategy 2, Static/Dynamic Compilation for Niceness's effect. Strategy 2 improves the QoS of the high priority application to meet its QoS requirement. By doing so, strategy 2 turns previous forbidden collocations into "safe" collocations and thus improves the server utilization, comparing to the baseline of disallowing collocation shown in Figure 1.2.	8
2.1	TCO (Total Cost of Ownership) cost breakdown for a datacenter using commodity servers	14
2.2	Task placement in a cluster. The cluster manager does not co-locate latency-sensitive applications with others to protect their QoS from performance interference, causing low machine utilization.	16

2.3	Server Utilization Histogram from HP datacenters.	17
2.4	Activity profile of a sample of 5,000 Google servers over a period of 6 months.	17
3.1	Separate Caches, Separate FSBs (X.X.X.X.)	25
3.2	Sharing Cache, Separate FSBs (XX..XX..)	25
3.3	Sharing Cache, Sharing FSB (XXXX....)	25
3.4	Performance of different thread-to-core mappings when each application is running alone. The higher the bars, the better the 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 <i>bigtable</i> is benefiting from sharing last level cache; while <i>contentAnalyzer</i> and <i>webSearch</i> suffer from the contention for memory resource among sibling threads.	28
3.5	LLC misses per million instrs	29
3.6	Normalized average LLC misses per million instructions	30
3.7	Bus ratio	30
3.8	Normalized bus ratio	31
3.9	L2 Requests in MESI States and in Prefetch State	31
3.10	ContentAnalyzer. Normalized to solo performance	35
3.11	Websearch. Normalized to solo performance	35
3.12	Bigtable. Normalized to solo performance	35
3.13	ContentAnalyzer. Normalized to solo performance with {X.X.X.X.}	36
3.14	Websearch. Normalized to solo performance with {X.X.X.X.}	36
3.15	Bigtable. Normalized to solo performance with {X.X.X.X.}	36
3.16	Topology of Dual Socket Intel Westmere	38
3.17	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	39

3.18	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	39
3.19	2 threads of latency sensitive applications running alone on Westmere	41
3.20	6 threads of latency sensitive applications running alone on Westmere	41
3.21	6 threads of latency sensitive applications co-running with 6 threads of batch applications on Westmere;	41
4.1	Bus Burst Transactions (full cache line) per millisecond per one thread	45
4.2	LLC misses/ms, LLC_requests_Share/ms and LLC reference/ms	46
4.3	Decision Tree	48
4.4	Adaptive Thread-To-Core Mapping on Clovertown	51
4.5	Adaptive Thread-To-Core Mapping on Westmere	51
5.1	QoS-Compile Overview	54
5.2	Contentiousness. Each bar shows the performance degradation of a corunner caused by the application across x-axis.	58
5.3	Sensitivity. Each bar shows the performance degradation of the application across x-axis caused by each of the 8 different corunners.	58
5.4	Average Contentiousness vs. Sensitivity	60
5.5	PMUs used for predicting contentiousness	63
5.6	L3 Miss Rate is not strongly correlated with the real measured contentiousness	70
5.7	L3 Reference rate is not strongly correlated with the real measured contentiousness	70
5.8	Predicted contention score using our model is highly correlated with the real measured contentiousness for SPEC benchmarks	70
5.9	Sphinx's PMU contention score calculated using our prediction model	72
5.10	Bst8mb's degradation when running with sphinx. The higher, the more degradation. Figure 7 trends similarly with this figure, indicating the profiler is identifying the correct contentious code regions.	72

5.11	This graph shows the accuracy of the contention score given by our prediction model in predicting the contentiousness of milc.	73
5.12	Padding <code>sledge_1</code> 's effect on its co-runner <code>blockie</code> and <code>bst</code> . As padding thickness increases, <code>sledge_1</code> 's execution rate decreases, <code>blockie</code> and <code>bst</code> 's QoS improves. The padding granularity is every 5 instructions	74
5.13	Napping <code>sledge_1</code> 's effect on co-runners, <code>blockie</code> and <code>bst</code> . Nap granularity is 1ms. As nap duration increases, <code>sledge_1</code> 's execution rate decreases, <code>blockie</code> and <code>bst</code> 's QoS improves.	74
5.14	Napping <code>sledge_1</code> 's effect on co-runners. Nap granularity is 10ms.	74
5.15	<code>sledge_1</code> padding vs. nap for <code>bst4mb</code>	75
5.16	<code>sledge_1</code> padding vs. nap for <code>bst8mb</code>	75
5.17	<code>sledge_1</code> padding vs. nap for <code>bst50mb</code>	75
5.18	SPEC benchmark's performance when it is co-located with the original <code>lbm</code> , <code>lbm</code> with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone	77
5.19	SPEC benchmark's performance when it is co-located with the original <code>milc</code> , <code>milc</code> with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone	78
5.20	Gained Utilization when allow co-location.	79
5.21	<code>bst8mb</code> running with <code>sphinx</code>	80
5.22	Google benchmark's performance when it is co-located with the original <code>sledge3</code> , <code>sledge3</code> with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone	81
5.23	Google benchmark's performance when it is co-located with the original <code>er-naive4mb</code> , <code>er-naive4mb</code> with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark's performance when it is running alone	82
6.1	Reactive-Niceness Overview	86
6.2	Reactive-Niceness Compilation	88

6.3	Reactive-Niceness Runtime Architecture	90
6.4	DFA for targeted Heuristic	94
6.5	QoS of each benchmark co-running with sledge , normalized to solo QoS. (simple)	97
6.6	Utilization of sledge with each configuration. (simple)	97
6.7	QoS of each benchmark co-running with lbm . (simple)	97
6.8	Utilization of lbm with each configuration. (simple)	97
6.9	QoS of each benchmark co-running with milc . (simple)	97
6.10	Utilization of milc with each configuration. (simple)	97
6.11	QoS of each benchmark co-running with sledge , normalized to solo QoS. (targeted)	99
6.12	Utilization of sledge with each configuration. (targeted)	99
6.13	QoS of each benchmark co-running with lbm . (targeted)	100
6.14	Utilization of lbm with each configuration. (targeted)	100
6.15	QoS of each benchmark co-running with milc . (targeted)	100
6.16	Utilization of milc with each configuration. (targeted)	100
6.17	Sphinx normalized IPC with original sledge and with sledge with RN_H1	102
6.18	Sphinx normalized IPC with original sledge and with sledge with RN_H2	103
6.19	Sphinx normalized IPC with original milc and with simple milc	104
6.20	Sphinx normalized IPC with original milc and with targeted milc	104
6.21	Average nap duration for milc with simple vs. milc with targeted	104
6.22	Overhead of monitoring for high-priority application.	105
6.23	Overhead of nap engine for low-priority application.	105
6.24	Efficiency of allowing co-location with Reactive-Niceness vs over-provisioning. (targeted)	106
6.25	QoS of each benchmark co-running with sledge , lbm , and milc . (targeted)	108
6.26	Utilization of sledge , lbm and milc with each configuration. (targeted) .	108

List of Tables

3.1	Sharing configurations for sets of 2 cores and sets of 4 cores	25
3.2	Experiment Platform	26
3.3	Production Datacenter Applications	27
3.4	Optimal Thread-To-Core Mapping in Solo and Co-location Situations	38
4.1	Predicted Thread-To-Core Mapping Using the Heuristic Approach	49
5.1	Contention Benchmarks Suite: SmashBench	64
5.2	Comparing our contentiousness predictor to predictors used in prior works. Our predictor was trained with the SmashBench suite of contentious kernels and tested against all SPEC 2006 benchmarks.	71
5.3	Production Warehouse Scale Computer Applications	80
6.1	Three configurations for simple heuristic	96
6.2	Three configurations of targeted heuristic	101

Chapter 1

Introduction

Contents

1.1	Memory Resource Sharing and Contention	2
1.2	Implications of Memory Resource Contention	3
1.2.1	The Impact of Contention on Performance	3
1.2.2	The Impact of Contention on Server Utilization	4
1.2.3	Trade-offs Between Performance and Utilization	4
1.3	Mitigating Contention	6
1.4	Two Strategies for Mitigating Contention	6
1.4.1	Mitigating Contention to Improve Performance	7
1.4.2	Mitigating Contention to Improve Utilization	9
1.5	Summary of Contributions	10

Web-service datacenters and cloud computing economies of scale have gained significant momentum in today’s computing environments. Companies such as Google, Microsoft, Yahoo, Facebook, Apple and Amazon host large-scale data intensive applications including search, email, maps, docs, video, social networking and other cloud services that require execution on a Warehouse Scale Computer (WSC) [6]. A warehouse scale computer often houses tens of thousands of machines to provide the computing resources needed to serve millions of users and typically costs hundreds of millions of dollars to construct and operate. This large cost stems from purchasing servers, power, cooling and other infrastructural and operational cost [13, 17].

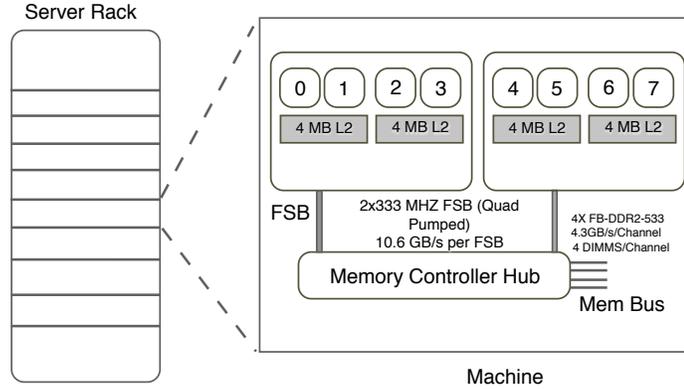


Figure 1.1: Topology of a Dual Socket Intel Clovertown

To reduce the cost and to improve the efficiency of WSCs, it is important to improve both **software performance** and **server utilization** [6]. For example, improving performance and server utilization can reduce the cost for infrastructure construction, server purchase and power consumption. At the massive scale of modern WSCs of web-services companies such as Google, 1% improvement in either performance or utilization translates to millions of dollars saved. However, memory resource contention inhibits the efficiency of WSCs.

1.1 Memory Resource Sharing and Contention

One of the major challenges that limit the efficiency in WSCs is the contention for memory subsystem resources on the servers that populate WSCs. Modern WSCs are constructed using commodity multicore machines as they are inexpensive and easily replaceable. Typically, these server machines have multiple sockets hosting processors with multiple cores. The processing cores may share a number of caches, buses and controllers. As an example, Figure 1.1 shows a typical dual-socket machine configuration found in production WSCs. 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 Front Side Bus (FSB). This type of machine organization is commonplace in state-of-the-art server processors. The sharing of these memory resources across multiple cores often has a significant impact on application performance. This impact may be *constructive* or *destructive*.

- When multiple cores share a resource, the threads running on those cores can *con-*

structively 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. Furthermore, memory bus and coherence traffic is reduced since data is fetched from memory only once and does not ping-pong back and forth between separate private caches.

- However, multiple threads, either from an individual application or multiple applications, can also contend for shared resources. **Memory resource contention** has a *destructive* impact on performance. For example, a thread can bring its own data into a shared cache, evicting the data of a neighboring thread and resulting in *performance interference* and degradation. Threads can also contend for prefetchers, memory controllers and bus bandwidth, detrimentally affecting performance.

1.2 Implications of Memory Resource Contention

Memory resource contention has a negative impact on both application performance and server utilization, significantly limiting the efficiency of modern warehouse scale computers.

1.2.1 The Impact of Contention on Performance

The destructive performance impact caused by contention can often be significant. Prior work reports up to 60% performance degradation due to contention using SPEC CPU2006 benchmarks and state-of-the-art server machines [63, 15, 40]. Therefore, it is greatly beneficial to mitigate contention and to exploit the potential positive resource sharing to improve application performance.

However, currently, there is little understanding about the interaction between the shared memory subsystem and the emerging large-scale datacenter workloads. Prior work largely relies on popular small-scale benchmark suites such as SPEC and PARSEC, and has reached conflicting conclusions about whether cache sharing has a significant performance impact, especially for contemporary multi-threaded applications [61]. To the best of our knowledge, no prior work has investigated the memory resource sharing for industry-strength emerging datacenter workloads. Although modern WSCs have generally adopted

the policy of disallowing collocation of certain applications to avoid potential performance interference, the severity of the interference due to contention is unclear. Due to the lack of understanding, current software systems in WSCs do not acknowledge or manage the resources sharing among application threads, resulting in potential performance inefficiencies.

1.2.2 The Impact of Contention on Server Utilization

In addition to performance, the interference caused by memory resource contention also proves particularly problematic to large-scale web-service applications as it may prevent these applications from providing satisfactory quality of service (QoS). On one hand, in order to reduce the machine and operational cost, it is essential for datacenters to consolidate various workloads on multicore servers to improve machine utilization [46]. On the other hand, warehouse scale computer workloads are composed of diverse applications with varying QoS requirements and priorities. Key applications, usually those that are user-facing and provide interactive service such as search, mail and maps, are latency sensitive and have fairly strict QoS requirements. Other applications such as backup service and file compression are batch applications that are not latency sensitive or have a lower QoS priority. When co-locating applications on a multicore platform, the performance and QoS of high priority applications may suffer unacceptable amounts of degradation due to resource contention [58, 37]. Moreover, high priority applications may even suffer more QoS degradation than low priority applications, resulting in unacceptable priority inversion. As a result, modern warehouse scale computers often resort to disallowing co-location of latency-sensitive applications with any other applications, which leads to costly low machine utilization [6]. This over-provisioning of compute resources is one of the major reasons the utilization in modern WSCs remains low, recently reported to be below 30% on average [42].

1.2.3 Trade-offs Between Performance and Utilization

Figure 1.2 further illustrates the performance interference caused by contention and its implications for the tradeoffs among performance, QoS and utilization in WSCs. Disallowing collocation of applications, shown as Option A, achieves peak application performance at the

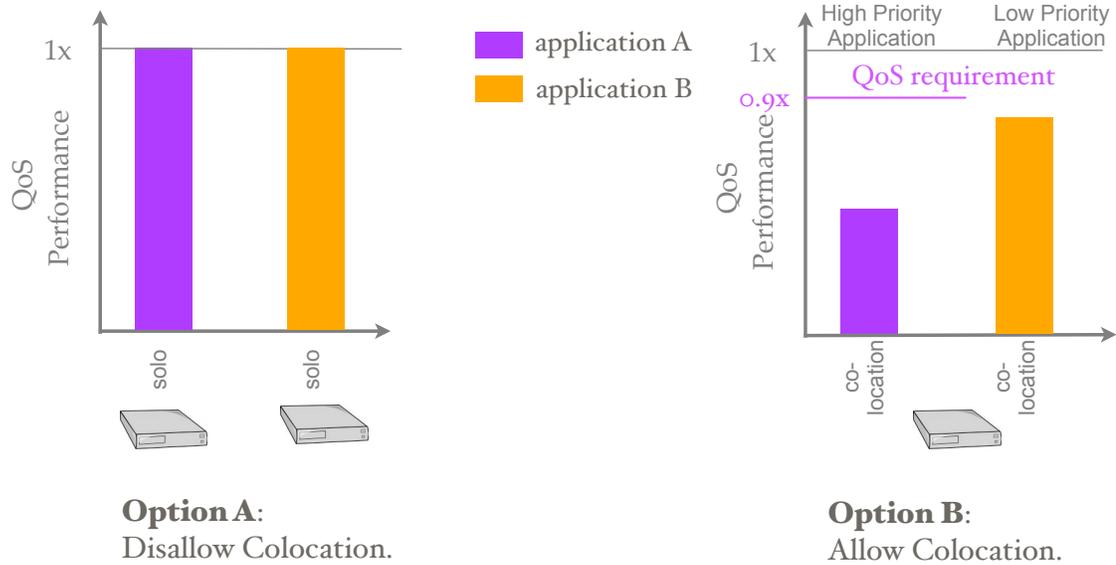


Figure 1.2: Current Two Options in WSCs. Option A, disallowing collocation of applications, achieves peak application performance at the sacrifice of machine utilization. Option B improves machine utilization and reduces the number of server machines needed. However, applications may suffer significant performance degradation, which can impair latency-sensitive applications’ capability to deliver acceptable QoS.

sacrifice of machine utilization. The alternative, allowing collocation, shown as Option B, improves machine utilization at the risk of impairing latency sensitive applications’ capability to deliver acceptable QoS. The current approach in modern warehouse scale computers is fairly ad-hoc with a mix of these two options. WSCs may allow collocation of applications that do not necessarily have strict QoS requirements, and thus simply submit to suffering performance degradation for the sake of better machine utilization. However, for latency-sensitive applications, WSCs often resort to disallowing co-location of these types of applications with any other applications, which translates to low machine utilization at the cost of millions of dollars.

In conclusion, memory resource contention has significantly limited both the application performance and server utilization in WSCs; and consequently, mitigating memory resource contention is critical for improving efficiency in WSCs.

1.3 Mitigating Contention

This dissertation argues for the design of novel software systems that are aware of the impact of resource sharing on applications, intelligently mitigate potential memory contention and promote positive resource sharing to improve software performance. In addition, this dissertation argues for novel software systems to mitigate contention to provide QoS management on multicore machines for applications with various QoS requirements to improve server utilization.

In addition to the current lack of understanding, there are multiple challenges for designing systems that can mitigate contention to improve performance or QoS on multicore platforms. Applications may contend for a plethora of memory components including a hierarchy of caches, prefetchers, memory controllers and buses. The interaction between the applications and these various components can be fairly complicated. In addition, due to the current limited transparency and monitoring capabilities for hardware behaviors, it is challenging for system software to dynamically detect and diagnose the occurrences of memory resource contention. System software also does not have control over hardware resources such as caches and memory bandwidth, which renders responding to contention and managing applications' QoS quite challenging. As a result, despite the amount of research attention given to contention problems on multicore platforms [28, 45, 10, 19, 44, 51, 52, 27, 32, 15, 29, 24, 63, 23, 24, 7, 4, 60], mitigating the impact of contention on an application's performance and QoS, enforcing the relative QoS priorities of co-running applications, while maximizing machine utilization, remains key challenges in modern warehouse scale computers.

1.4 Two Strategies for Mitigating Contention

This dissertation first comprehensively investigates the impact of memory resource sharing on industry-strength large-scale datacenter workloads and provides new information and insights. The result of our investigations demonstrate that, contrary to conclusions from recent work [61], across several key datacenter applications including web-search, there is both a sizable benefit and a potential degradation from improperly sharing microarchitec-

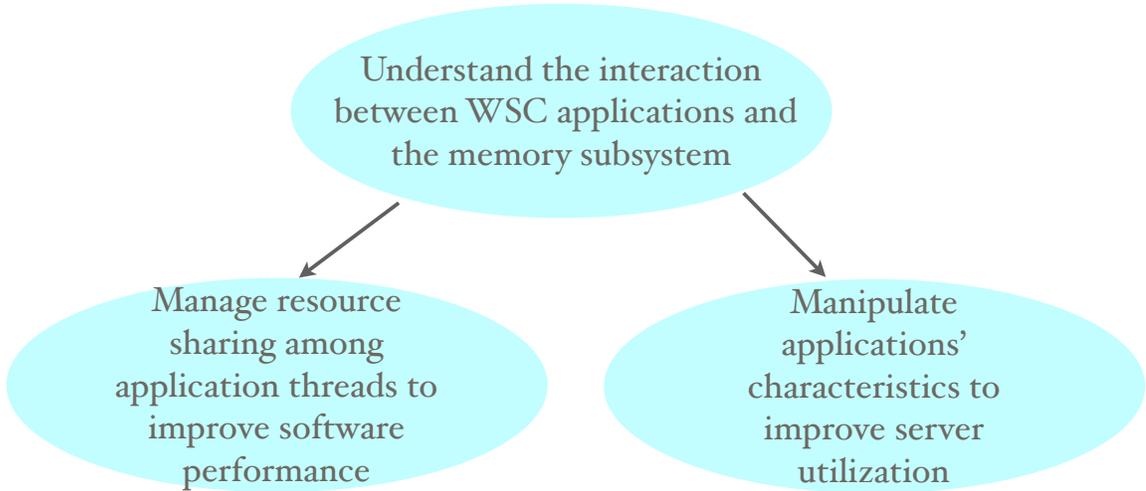


Figure 1.3: Dissertation Overview - Understanding the impact of contention and 2 strategies to mitigate contention to improve performance and utilization

tural resources on a single machine, such as on-chip caches and bandwidth to memory. This dissertation then presents two complementary software strategies, shown in Figure 1.3, to mitigate memory resource contention for improving performance and server utilization of WSCs.

1.4.1 Mitigating Contention to Improve Performance

*Strategy 1: Manage resource sharing among threads to improve performance using **Intelligent Thread-to-Core Mapping**.*

The basic idea of an intelligent thread-to-core mapper is to take advantage of the memory resource topologies (an example is shown in Figure 1.1) to promote more positive sharing and reduce negative sharing among threads. The processing cores on this machine do not share the same resources. For example, as shown in the figure, core 0 and 1 share a L2 cache, front side bus (FSB) and a memory controller. However, core 0 and 2 do not share L2 cache, and core 0 and 4 do not share FSB. Therefore, when mapping threads to cores, the mapper essentially manages what resources are shared among threads. For example, some threads share data and may benefit from cache sharing. Thus these threads should be mapped to cores that share a cache. Others threads are contentious with each other and thus should be mapped to cores that do not share a cache or even FSB to mitigate

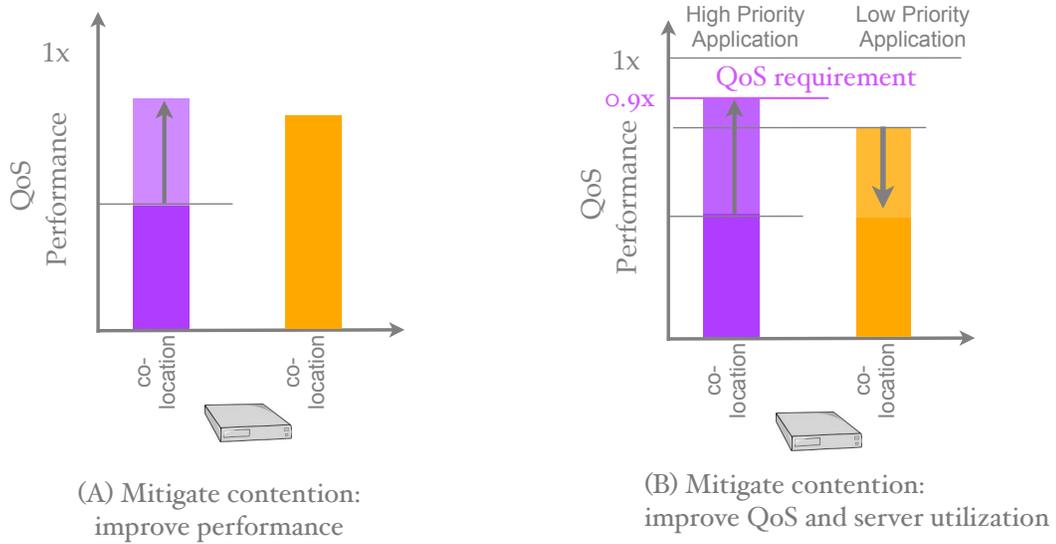


Figure 1.4: Enabled capabilities by our software systems. (A) illustrates the strategy 1, the TTC mapper’s effect - improved performance comparing to baseline colocation situation, shown in Figure 1.2. (B) illustrates the strategy 2, Static/Dynamic Compilation for Niceness’s effect. Strategy 2 improves the QoS of the high priority application to meet its QoS requirement. By doing so, strategy 2 turns previous forbidden colocations into “safe” colocations and thus improves the server utilization, comparing to the baseline of disallowing colocation shown in Figure 1.2.

interference. An intelligent thread-to-core mapper can take advantage of these application characteristics and the memory topologies to mitigate contention and improve performance.

This research finds that the performance variability between the worst and the optimal thread-to-core mappings can be significant for datacenter workloads. More interestingly, the best thread-to-core mapping for a given application does not only depend on the application’s sharing and memory characteristics; it is also impacted dynamically by the characteristics of other applications that are co-running on the same machine simultaneously. Based on this insight, we design two approaches for intelligent thread-to-core (TTC) mapping. The desired outcome of our strategy for mitigating contention to improve *performance* is illustrated in Figure 1.4 (A).

- **Heuristic-based approach**

The application characteristics that impact performance in various thread-to-core mapping scenarios are identified. These characteristics include the amount of data sharing among threads, the amount of memory bandwidth an application requires, and the cache footprint of the application. We present an algorithm that takes advan-

tage of these applications characteristics to identify efficient thread-to-core mappings.

- **Dynamic approach**

We also present an adaptive approach, AtoM, which uses a competition heuristic to search for the best performing mapping online. The approach includes two phases: a learning phase, when AToM empirically has various TTC mappings compete to learn which mapping performs best, and an execution phase, when the winning TTC mapping is run for a fixed or adaptive period of time.

1.4.2 Mitigating Contention to Improve Utilization

*Strategy 2: Manipulate applications’ characteristics to improve server utilization using **Static/Dynamic Compilation for Niceness***

The second strategy comprises novel compilation and runtime systems to directly manipulate an application’s contentious nature and reduce the interference it can cause to its corunning applications. By doing so, we facilitate more “safe” colocations where contention is mitigated so that latency-sensitive applications can provide acceptable QoS. The desired outcome of our strategy for mitigating contention to improve *utilization* is illustrated in Figure 1.4 (B).

- **Static Approach: Compiling for Niceness**

Two key insights underlie this compiling approach. Firstly, a compilation-based approach is both well-suited and desirable for WSCs. Large-scale web-service applications such as web-search, maps, email and video are developed and hosted by the companies that operate the WSCs, and their source code is available and recompiled regularly. Secondly, in the era of multicore and the emerging computing domain of WSCs, the objectives of compiler optimization ought to be multifaceted. In addition to optimizing each application for its own individual performance, we argue for the additional objective of optimizing for an application’s “*niceness*”; that is, to reduce its potential interference to its co-running applications.

Our approach, **QoS-Compile**, uses novel compilation techniques to directly manipulate the contentiousness of low priority applications to ensure the QoS of a higher

priority co-runner. With this, high machine utilization can be achieved through allowing colocation while providing satisfactory QoS. QoS-Compile uses a prediction model to pinpoint code regions that aggressively demand memory resources. It then targets these regions, transforms their code layouts to reduce their contentiousness by throttling down their memory request rate. Thus its interference to the QoS of its co-runners is reduced. To the best of our knowledge, QoS-Compile is the first compilation approach to address the QoS challenges caused by contention for multi-programmed workloads.

- **Dynamic: Reactive Niceness**

Enabled by the above static approach, we design a lightweight dynamic approach, **Reactive-Niceness**, to further improve QoS and utilization. Reactive-Niceness instruments the contentious regions to enable the flexible manipulation of their contentiousness at runtime. Dynamically, Reactive-Niceness detects contention-caused QoS degradation and adaptively throttles down the memory request rate of those contentious regions in the low-priority application. The degree of execution rate reduction on low-priority applications is based on the severity of observed QoS degradation of the high-priority application and a feed-back control, enabling the flexibility needed to further improve machine utilization and achieve more precise QoS management.

In summary, this dissertation advances the state-of-the-art for understanding and managing the impact of memory resource contention on large-scale emerging WSC workloads and provides effective software systems to mitigate contention to significantly improve both **application performance** and **server utilization** in WSCs.

1.5 Summary of Contributions

We first conduct a thorough investigation of the impact of sharing memory resources (e.g., shared caches and memory bandwidth) on key commercial datacenter applications including Google’s web-search engine and bigtable (a peta-scale data storage software). This work is the first to characterize the impact of memory resource sharing on real-world large-

scale datacenter applications, exposing new insights about these emerging workloads and demonstrating the significant impact of memory contention. Chapter 3 examines:

- **Intra-application sharing:** We investigate the impact of memory resource sharing on threads that belong to a single multithreaded application [58] (Section 3.2). While prior work has found neither positive nor negative effects from cache sharing across benchmark suites, we find that across these datacenter applications, there is both a sizable benefit and a potential degradation from improperly sharing resources.
- **Inter-application sharing:** We investigate the impact of memory resource sharing on threads that belong to multiple multithreaded applications [58] (Section 3.3). Our investigation demonstrates that memory resource contention among multiple applications often cause significant performance degradation. Our investigation also shows that, contrary to common intuition, the optimal thread-to-core mapping for a given application changes depending on its execution environment, including the underlying machine, whether it is running alone, and if not, which application is its corunner.

We then apply the discoveries and insights from the investigation and design intelligent thread-to-core mappers to mitigate contention and improve software performance in WSCs. Chapter 4 presents:

- **Heuristics based thread-to-core mapper:** We identify the application characteristics that impact performance in the various thread-to-core mapping scenarios and provide a technique for deriving algorithms from these characteristics for heuristics based thread-to-core mapping to improve performance efficiency in WSCs [58] (Section 4.1).
- **Automatic thread-to-core mapper (AToM):** We also present the design of an adaptive approach that uses a competition heuristic to learn the best performing mapping online to improve performance. This approach is agnostic to the underlying microarchitecture [58] (Section 4.2).

To address the server utilization and QoS challenges, we design a static compilation approach, **QoS-Compile**, for effective QoS management on multicores to facilitate workload

consolidation and improve server utilization in WSCs. QoS-Compile is the first compilation approach to addressing the QoS challenges caused by contention for multiple co-running applications. Chapter 5 presents:

- **Profiling techniques to identify contentious code regions:** We design a prediction model that is based on the performance counters to pinpoint code regions that aggressively demand memory resources [57, 56] (Section 5.2).
- **Novel code transformations to dampen code regions' contentious nature:** We design two compilation techniques that reduce a code region's contentiousness and the potential performance interference it can cause to co-runners [57] (Section 5.3).

In addition to static compilation techniques, we design a statically-enabled runtime system, **Reactive-Niceness** (RN), to further improve server utilization and achieve more accurate QoS management of latency-sensitive applications. Chapter 6 presents:

- **RN-Compiler:** We present a profiling guided compilation approach that enables the adaptive manipulation of contentiousness of the low-priority application. The RN-Compiler identifies the contentious code regions of an application and inserts hooks in these regions that are used to invoke runtime manipulation [55] (Section 6.2).
- **RN-Runtime:** We present a runtime system that continuously monitors the QoS of high-priority applications, detects when contention is occurring dynamically, and directs the manipulation of the contentiousness of low-priority applications based on an adaptation policy. We also present two adaptation policies for flexible adjustments of the tradeoffs between QoS and utilization [55] (Section 6.3).

Collectively, this dissertation takes a major leap forward in understanding and mitigating memory resource contention to improve efficiency in the emerging domains of large-scale web-services and modern warehouse scale computers. We demonstrate the need for new types of software systems in modern WSCs and design effective mechanisms to fundamentally address the detrimental impact of memory resource contention on efficiency of WSCs to greatly improve both performance and server utilization.

Chapter 2

Background and Related Work

Contents

2.1 Warehouse Scale Computers	13
2.1.1 Cost	14
2.1.2 Application QoS	14
2.1.3 Job Scheduling, Application Colocation and Utilization	15
2.1.4 Machine Level	18
2.2 Related Work	18
2.2.1 Impact of Memory Resource Sharing	18
2.2.2 Novel Hardware Solutions to Mitigate Contention	19
2.2.3 Software Runtime and OS Approaches to Mitigating Contention	19
2.2.4 Cache Contention Aware Compilation	21

2.1 Warehouse Scale Computers

This chapter reviews the background and related work of this dissertation. We first present the background of modern warehouse scale computers including its cost breakdown (Section 2.1.1), the QoS metrics of applications that are running in these datacenters (Section 2.1.2), and the job scheduling on a cluster level (Section 2.1.3) as well as on a machine level (Section 2.1.4). We then present related work, especially on the topic of memory resource contention from both software and hardware communities (Section 2.2).

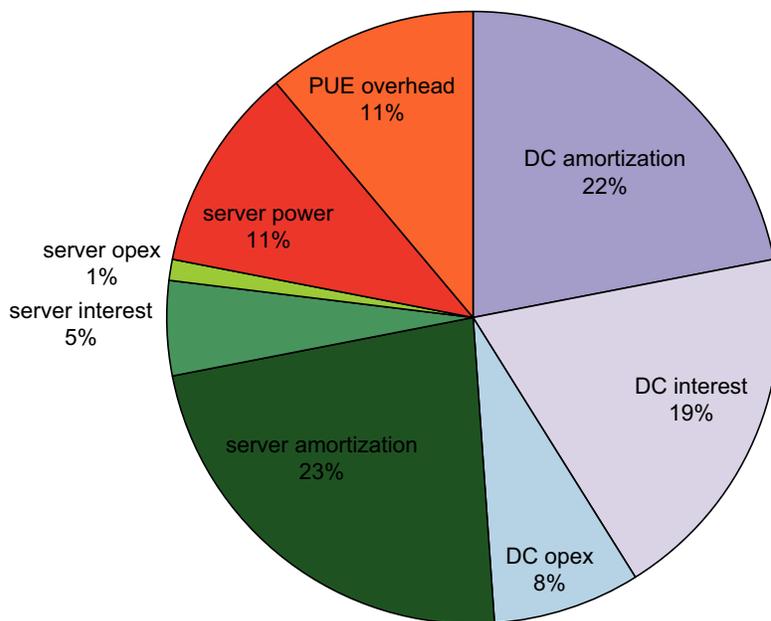


Figure 2.1: TCO (Total Cost of Ownership) cost breakdown for a datacenter using commodity servers

2.1.1 Cost

To better understand the importance of improving performance and server utilization, let us first examine the cost of a modern warehouse computer. Figure 2.1 is from “The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines” by Barroso et al. [6]. It presents a 3-year total cost of ownership (TCO) breakdown of a datacenter housing commodity servers, including both capital cost and operational cost. As shown in the figure, servers, power and datacenter construction are several major components of the TCO. Improving performance and utilization can reduce the cost for all these components. Better performance and higher server utilization indicate less servers needed for a given amount of work, less power consumed by these servers and a smaller datacenter to host these servers.

2.1.2 Application QoS

Applications that are running in a warehouse scale computer often have various quality-of-service (QoS) priorities. User-facing applications such as web-search, maps, email and other Internet services are *latency-sensitive*, and have high QoS priorities. Applications

such as file backup, offline image processing, and video compression are *batch* applications that often have no QoS constraints. For these, latency is not as important.

In this dissertation, we define the QoS of an application in terms of the relevant performance metric specified in its internal service level requirements (SLAs). For example, the QoS of Google’s web-search is measured using query latency and queries-per-second. This is in contrast to Bing’s [22, 30], which uses the quality of search results provided. And a job’s QoS level of 95% corresponds to the normalized 95% of its performance when an entire machine is dedicated to that job. More details and examples of applications and their QoS or performance metrics are presented in Section 3.1 Table 3.3.

2.1.3 Job Scheduling, Application Colocation and Utilization

In this section, we examine the current job scheduling in a modern WSC, the application colocation policy and the current server utilization. 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. Once a job is mapped to a machine, the machine-level scheduler then decides the mapping and scheduling of the job and its threads. In this section, we focus on the cluster level scheduler. We discuss the machine level scheduler in the next section.

In modern warehouse scale computers, each *web-service* is composed of one to hundreds of application tasks, and each task runs on a single machine. An application task is composed of the application binary, associated data, and a configuration file that specifies the machine level resources required. These resources include the number of cores, amount of memory, and disk space that are to be allocated to the task. The configuration file for a task may also include special rules for the cluster manager such as whether to disallow colocations with other tasks. Based on the resource requirement, the cluster manager, which is responsible for a cluster of servers, uses an algorithm similar to bin-packing to place each task on a single machine [43].

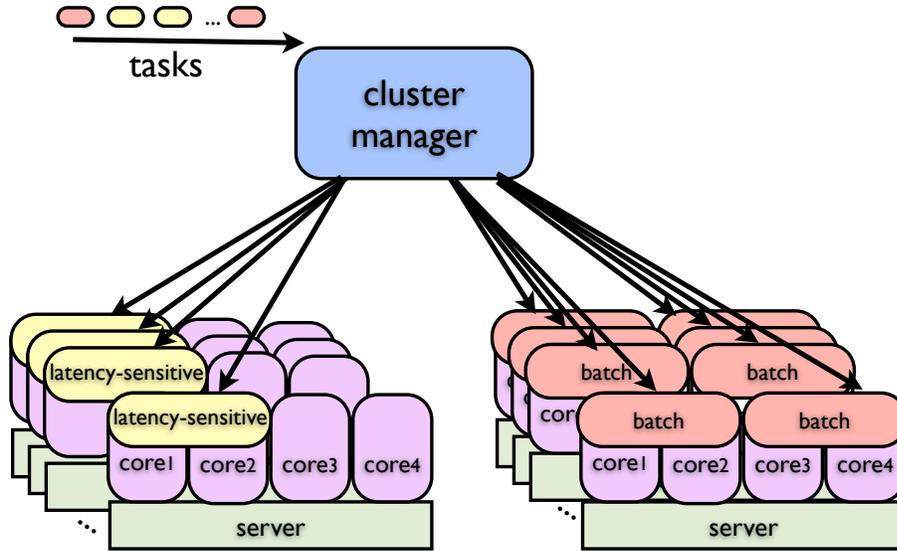


Figure 2.2: Task placement in a cluster. The cluster manager does not co-locate latency-sensitive applications with others to protect their QoS from performance interference, causing low machine utilization.

Application Colocation

As multicores become widely adopted in datacenters, the cluster manager often consolidates multiple disparate tasks on a single server to improve the machine utilization. However, to avoid the performance interference, latency-sensitive applications that have strict QoS are not co-located with any other applications. A simplified illustration of the application task placement process is shown in Figure 2.2 [37, 38, 36]. Latency-sensitive tasks that disallow co-location inadvertently occupy more resources on a server leading to unnecessary server overprovisioning and low machine utilization.

Server Utilization

The current server utilization in warehouse-scale computers is typically quite low, often below 30%. Figure 2.3 presents a histogram of utilization for two production workloads, “web 2.0” applications and enterprise IT applications, from enterprise-scale commercial deployments [42]. The data presented is from utilization traces collected over many days, aggregated over more than 120 servers (production utilization traces were provided courtesy of HP Labs). As shown here, the servers spend the vast majority of time under 10% utilization.

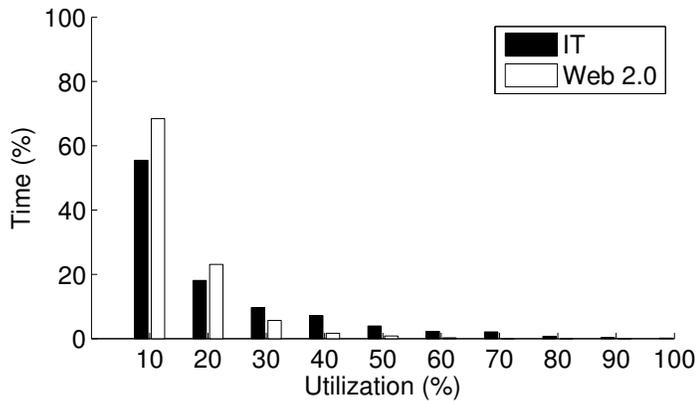


Figure 2.3: Server Utilization Histogram from HP datacenters.

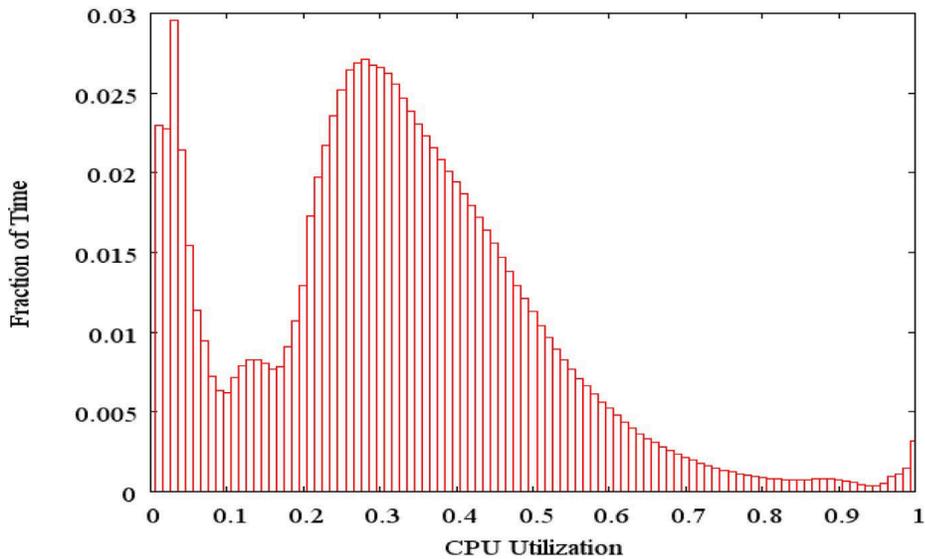


Figure 2.4: Activity profile of a sample of 5,000 Google servers over a period of 6 months.

Figure 2.4 presents a histogram of average CPU utilization of more than 5,000 servers during a six-month period in Google production datacenters [6]. As shown in the figure, servers rarely operate near their maximum utilization. The average utilization shown here is around 30%. The policy of disallowing collocation of latency-sensitive applications is one of the main reasons for the low utilization.

2.1.4 Machine Level

Once application tasks are mapped on a machine, on the individual machine level, general purpose system software such as the Linux kernel is adapted for and used in the datacenter for finer grain scheduling. The state-of-the-art kernel scheduler focuses on load balancing and prioritizes cache affinity to reduce cache warm-up overhead. It 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. 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.

2.2 Related Work

2.2.1 Impact of Memory Resource Sharing

Currently, little is known about the impact of memory resource sharing on large-scale web-service datacenter applications. A great amount of prior work has concluded that contention has a significant impact on the performance of traditional workloads using common benchmark suites such as SPEC [63, 15, 40]. However, recent work by Zhang et al. concludes that contemporary multithreaded applications are not affected by cache sharing using a multithreaded benchmark suite PARSEC [61]. To the best of our knowledge, no prior work has investigated the impact of memory resource sharing for industry-strength emerging datacenter workloads. The commonly used benchmark suites (such as SPEC, PARSEC) do not necessarily represent these workloads, and thus may be misleading.

2.2.2 Novel Hardware Solutions to Mitigate Contention

Hardware techniques such as cache partitioning and bandwidth partitioning to reduce resource contention and improve performance and fairness on multicores have received much research attention [28, 45, 10, 19, 44, 51, 52, 27, 32]. In addition, there also have been a number of work aimed at better modeling cache contention [8] and monitoring cache contention [62]. Most of the prior work focuses on improving the system’s overall performance or performance fairness and does not address the QoS challenges.

Recently, platform support that enforces different QoS priorities has been proposed [16, 21, 12, 18]. Among these studies, a promising direction for QoS management is hardware execution throttling. Herdirch et al. [18] use what is likely to be future hardware capabilities, core specific dynamic voltage scaling and clock modulation to throttle down low priority applications to reduce their performance interference on the high priority applications. Ebrahimi et al. [12] and Iyer et al. [21] propose hardware changes to throttle memory requests to provide QoS management. Although these hardware solutions have shown promising results using simulations, they require significant changes to current commodity micro-architectures and cannot be applied to multicore platforms that are already in production or to be deployed in the near future.

2.2.3 Software Runtime and OS Approaches to Mitigating Contention

[Contention Aware Scheduling] A research direction that recently attracted a wealth of attention is contention aware scheduling [15, 29, 24, 63, 23, 24, 7, 4, 60]. Contention aware scheduling techniques use predictors or models to decide what applications should be co-running together to minimize the performance degradation or to improve performance isolation. Although contention-aware scheduling may improve the overall performance or fairness of a workload composed of a mixture of highly contentious and not contentious applications, its effectiveness is highly dependent on the composition of the workload. In addition, the current work does not address thread-to-core (TTC) mapping for multiple multi-threaded applications when the resource sharing can have either positive or negative impact. Moreover, scheduling approaches do not provide direct manipulation of the

contentious nature of an application.

- **[TTC Mapping]** So far there is little work on software approaches to intelligently mapping threads to cores to promote positive resource sharing, reduce contention and improve performance. Most of the above work uses single threaded applications and focus on only on the resource contention aspect between multiple applications, ignoring the contention and potential benefit among threads that belong to a single application. For example, Banikazemi et al. [4] present a scheduler to adaptively schedule threads of single-threaded applications to cores to take advantage of the cache topology to alleviate resource contention. Tam et al. [54] present a technique to cluster communicating threads onto the same chip to reduce the communicating latency. Their approach targets multithreaded-applications. However, their technique only focuses on the constructive effect of sharing resource without considering the potential resource contention.
- **[QoS management for improving utilization]** There are currently no general software solutions for achieving the QoS management and enforcement of QoS priorities as described in Figure 1.4 (B). Our static/dynamic compilation approaches differ from the scheduling efforts in that, firstly, our approaches do not decide which application should be running with which. Instead we focus on the complimentary question: when multiple applications are already scheduled to be running simultaneously on a multicore platform, how to reduce the contention and guarantee their QoS? Secondly, most of the above studies on contention aware scheduling focus on improving overall performance or performance fairness instead of guaranteeing different QoS priorities for different applications. In this dissertation, our static/dynamic compilation system provides a mechanism to trade a small amount of performance of low priority applications to enable more “safe” co-locations and thus improve machine utilization in the WSC.
- **[Indicators for Application Contention Characteristics]** One important component for contention-aware scheduling and adaptive runtimes is the indicators for application contention characteristics. This is related to the dissertation because our

profiling technique relies on indicators to identify contentious code regions. Knauerhase et al. [29] use last level cache miss rate as an indicator of contentiousness of an application. Zhuravlev et al. [63] demonstrate that cache contention is not the dominant cause for performance degradation of co-running applications on CMPs; contentious behaviors that happen in many components of the memory sub-system all contribute to the performance degradation. They also conclude that last level cache miss ratio is one of the best predictor for co-running applications' performance degradation. Mars et al. [40] use changes in LLC miss rate as an indicator to detect contention. Approaches that use prediction models are also proposed. Jiang et al. [24] estimate an application's contentiousness by estimating the extra cache misses caused by co-location based on the application's reuse distance profile. This dissertation presents a more accurate prediction model than the state-of-the-art approaches.

In addition to contention-aware scheduling mentioned above, software solutions to provide QoS guarantee using page coloring/remapping have been proposed [31, 11, 49]. Most page coloring methods require significant modifications to the kernel and knowledge of details of cache designs. However, cache designs on modern multicores are highly guarded industry secrets. Mars et al. [40] design CAER, an adaptive runtime that detects and responds to contention online. This is the first proposed software approach to detect contention, which periodically pauses the execution of an application, monitors the difference in last level cache miss rate to infer contention.

2.2.4 Cache Contention Aware Compilation

Researchers recently have started to explore using code transformations and restructuring to improve cache sharing and reduce contention on multicores [26, 25, 61, 50]. Most such research focuses on compilation techniques to improve data sharing for multi-threaded programs. Kandemir et al. [25] propose a code restructuring scheme for improving cache locality by optimizing loop iterations distribution and scheduling on multicores. Our static/dynamic compilation approaches for niceness do not address the interaction of cache sharing or contention within an application. Instead, our techniques regulate the memory

pressure an application puts on the shared resources, changing how applications interact with each other in terms of contending for the shared memory resources. Methods to reduce cache pollution by using special instructions to manage cache are also proposed. Sandberg et al. [48] use non-temporal prefetch instructions to improve performance of cache sharing among a mix of workloads on commodity multicores.

Chapter 3

The Impact of Memory Resource Sharing

Contents

3.1	Memory Resource Sharing	24
3.2	Intra-Application Sharing	26
3.2.1	Experiment Methodology	26
3.2.2	Measurement and Findings	28
3.2.3	Investigating Performance Variability	29
3.2.4	Summary	32
3.3	Inter-Application Sharing	33
3.3.1	Experiment Design	33
3.3.2	Measurement and Findings	34
3.3.3	Varying Thread Count and Architecture	38
3.3.4	Summary	40

This chapter characterizes the performance impact of memory resource sharing on key workloads in modern warehouse scale computers. As mentioned in Chapter 2, currently, little is known about the impact of memory resource sharing on these large-scale industry-strength applications. In this chapter, we first investigate *intra-application sharing* (Section 3.2), characterizing the impact of resource sharing on threads that belong to a single multithreaded application. In this case, the threads may share data so it may either benefit or degrade from the resource sharing. We then investigate *inter-application sharing* (Section 3.3), characterizing the impact on threads that belong to multiple multithreaded

applications.

As shown in this chapter, our investigations demonstrate that, contrary to conclusions from recent work [61], across several key datacenter applications including websearch, there is both a sizable benefit and a potential degradation from improperly sharing microarchitectural resources on a single machine, such as on-chip caches and bandwidth to memory. More interestingly, the best thread-to-core mapping for a given application does not only depend on the application’s sharing and memory characteristics; it is also impacted dynamically by the characteristics of other applications that are co-running on the same machine simultaneously.

3.1 Memory Resource Sharing

On multi-socketed multicore platforms such as the dual socket Intel Clovertown shown in Figure 1.1, 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 1.1, there are three possible sharing configurations between two cores, shown in Table 3.1. 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 3.1, 3.2 and 3.3. 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 3.1, 3.2 and 3.3 show three mappings corresponding to three sharing configurations on our experimental platform, the Intel Clovertown. Here we introduce a notation for the

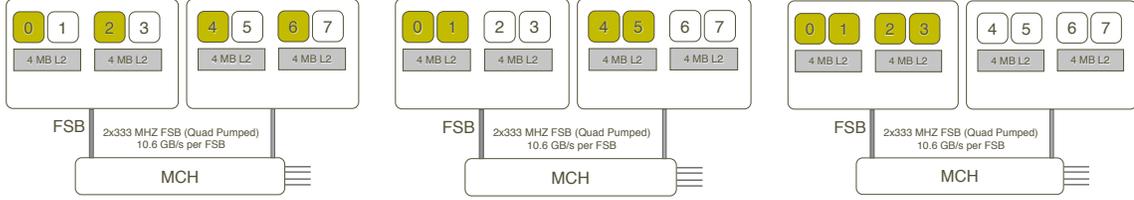


Figure 3.1: Separate Caches, **Figure 3.2:** Sharing Cache, Separate FSBs (X.X.X.X.) **Figure 3.3:** Sharing Cache, Sharing FSB (XXXX....)

Table 3.1: Sharing configurations for sets of 2 cores and sets of 4 cores

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

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, $\{XXXX\dots\}$ indicates four threads mapped to cores $\{0, 1, 2, 3\}$ on the same socket, as shown in Figure 3.3. 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 $\{XX\dots XX\dots\}$ and $\{X.X.X.X.\}$ reflects the impact of sharing last level cache (LLC); and the performance difference between mapping $\{XX\dots XX\dots\}$ and $\{XXXX\dots\}$ reflects the impact of sharing FSB. When there is significant performance variability, a resource-aware thread-to-core mapping is needed.

Table 3.2: Experiment Platform

CPU	Xeon E5345
GHz	2.33ghz
Cores	2x4
L2	4x4MB 16 way
FSB	2 x 10.6GB/s
Memory Bus	4 x 4.3GB/s
Memory	32GB

3.2 Intra-Application Sharing

We first investigate the performance impact of memory resource sharing for several key data-center 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.

3.2.1 Experiment Methodology

We first describe our experiment setup including the platform, benchmark applications, performance metrics and experiment design.

The primary platform used for this investigation is a dual socket Intel Clovertown (Xeon E5345), shown in Figure 1.1 and Table 3.2. 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. This platform is commonly deployed in Google production datacenters. We also conducted experiments on the Intel Westmere, which is presented in Section 3.3.3.

Table 3.3 presents a detailed description of the five datacenter applications we used in this study. It also 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. We use each application’s specified performance metric in this investigation. The performance metrics are also shown in Table 3.3. 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

Table 3.3: 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 [2]	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 [53]	N/A	batch
protobuf	protocol buffer [3]	N/A	batch

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.

[Experiment Design] In this section we conduct 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 [61]. Applications are parameterized to have a fixed load execute across 4 cores. All experiments were run three times and the average measurement is presented. The performance variability between runs for each configuration is around 1%.

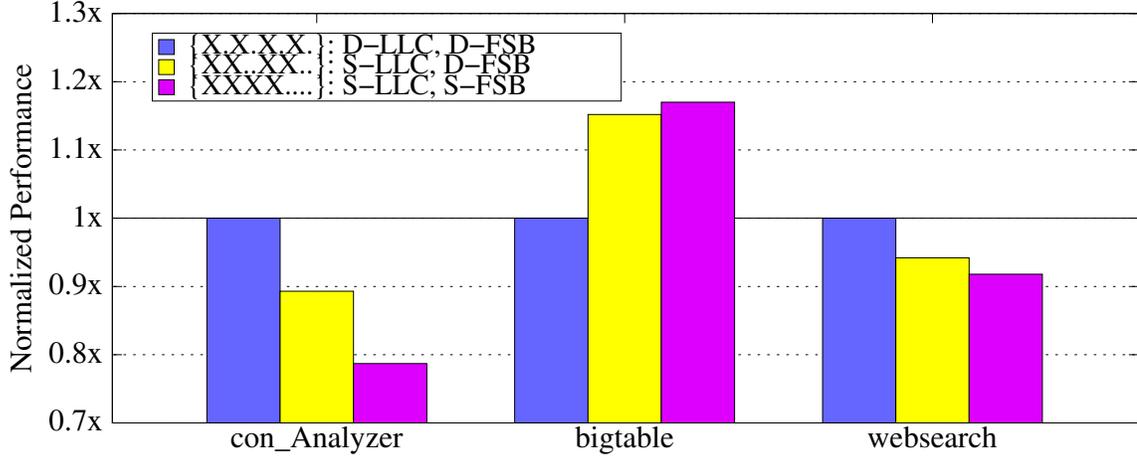


Figure 3.4: Performance of different thread-to-core mappings when each application is running alone. The higher the bars, the better the performance. The performance variability is up to 20% for each application, indicating that the memory resource sharing has a significant performance impact on these 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.

3.2.2 Measurement and Findings

Figure 3.4 demonstrates the performance variability due to different TTC mappings for the latency sensitive applications presented in Table 3.3. For each application, 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

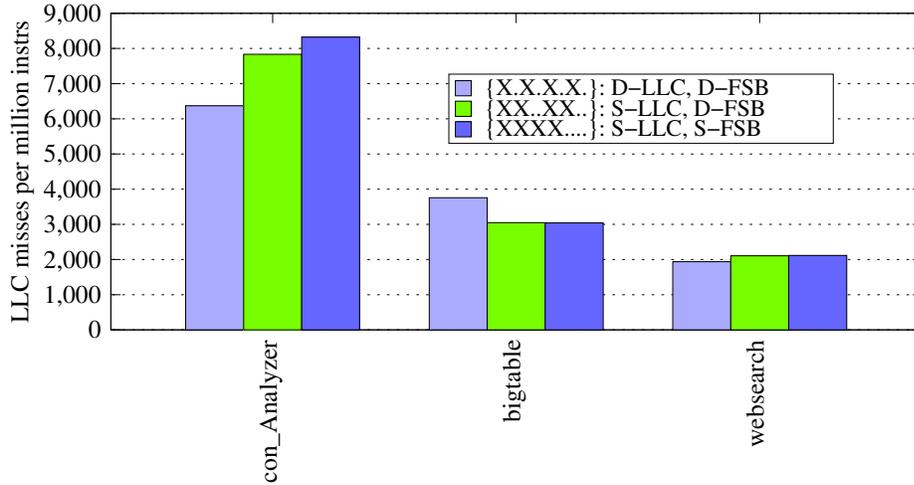


Figure 3.5: LLC misses per million instrs

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..}.

3.2.3 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` [14].

[Last Level Cache Misses] Figure 3.6 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 3.6 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

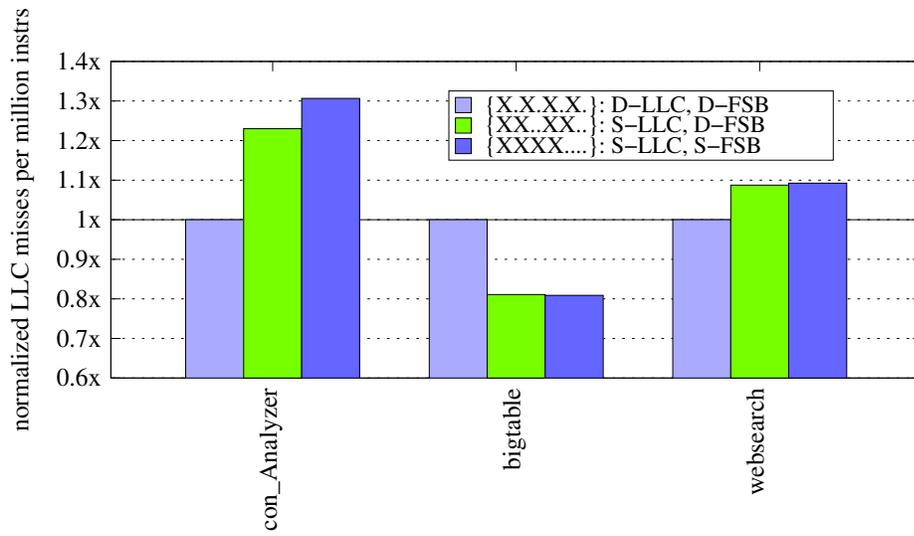


Figure 3.6: Normalized average LLC misses per million instructions

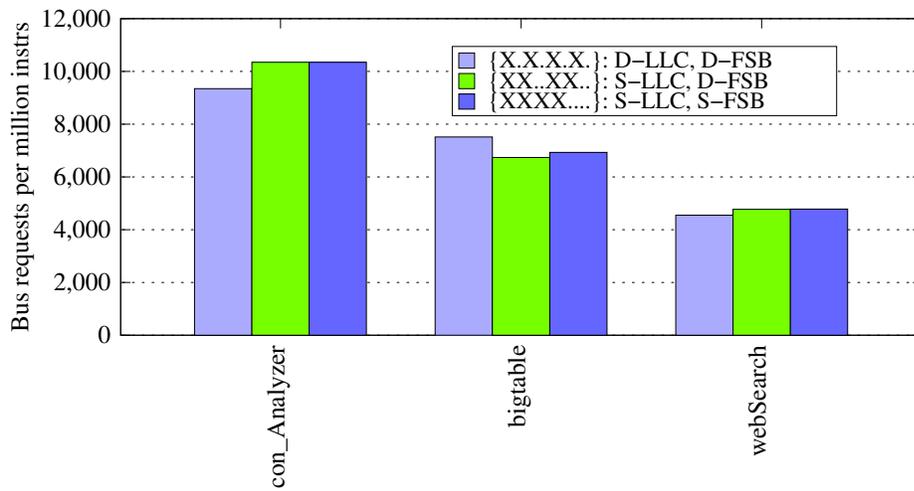


Figure 3.7: Bus ratio

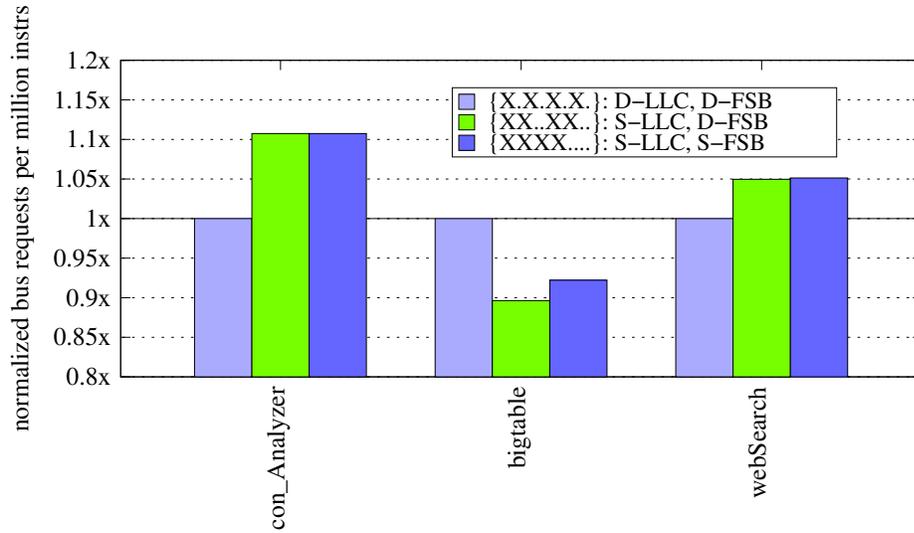


Figure 3.8: Normalized bus ratio

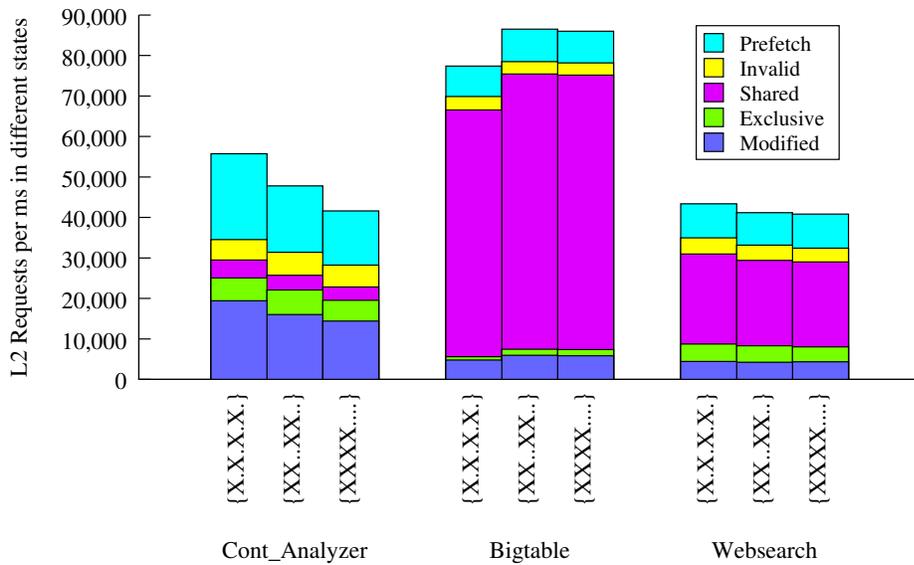


Figure 3.9: L2 Requests in MESI States and in Prefetch State

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 3.4. *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 3.4.

[FSB Bandwidth Consumption] Figure 3.8 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 3.9 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.

3.2.4 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 an effective thread-to-core mapping that mimics the application’s inherent data sharing pattern.

3.3 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 Chapter 1, 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 is 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.

3.3.1 Experiment Design

Similar to Section 3.2, 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 3.3) 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.

3.3.2 Measurement and Findings

Figure 3.10 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 3.13 shows *contentAnalyzer*'s performance when it is running alone and it is

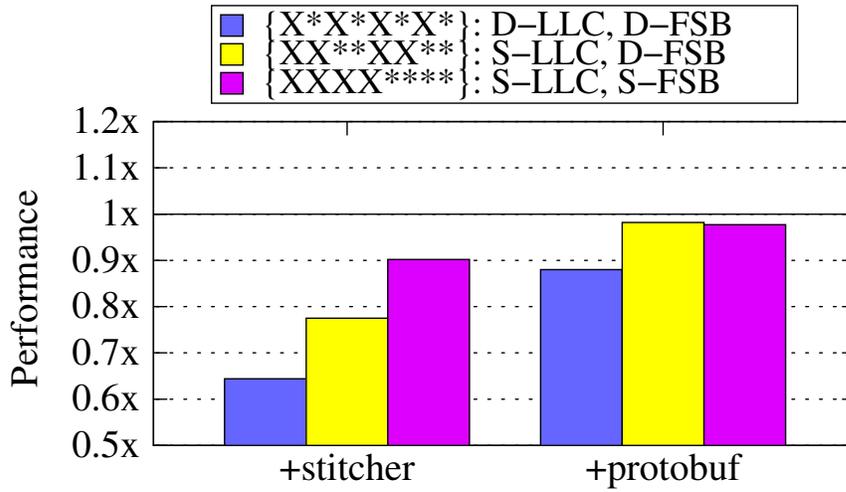


Figure 3.10: ContentAnalyzer. Normalized to solo performance

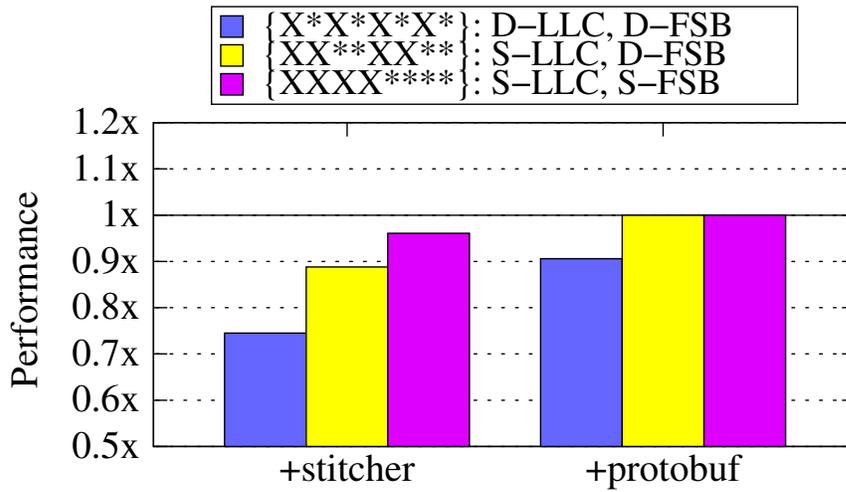


Figure 3.11: Websearch. Normalized to solo performance

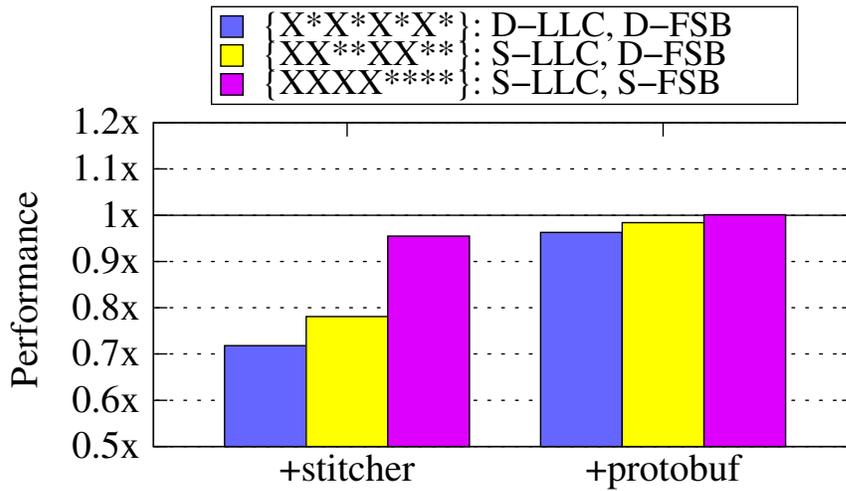


Figure 3.12: Bigtable. Normalized to solo performance

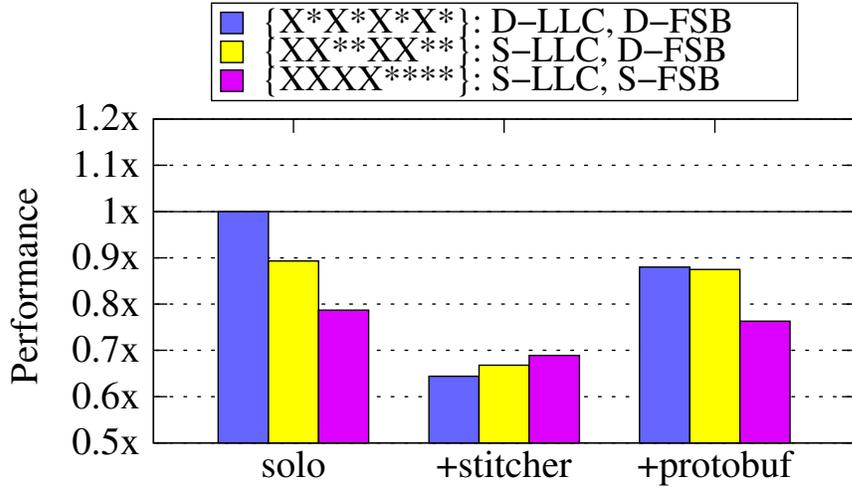


Figure 3.13: ContentAnalyzer. Normalized to solo performance with {X.X.X.X.X}

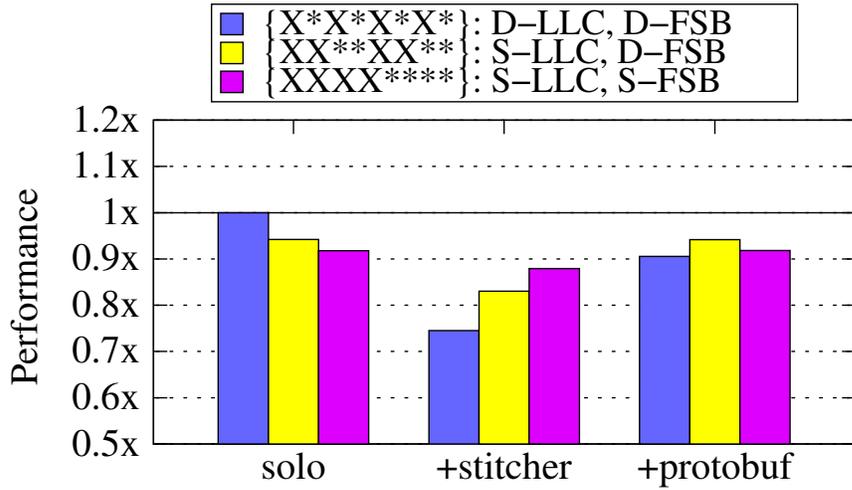


Figure 3.14: Websearch. Normalized to solo performance with {X.X.X.X.X}

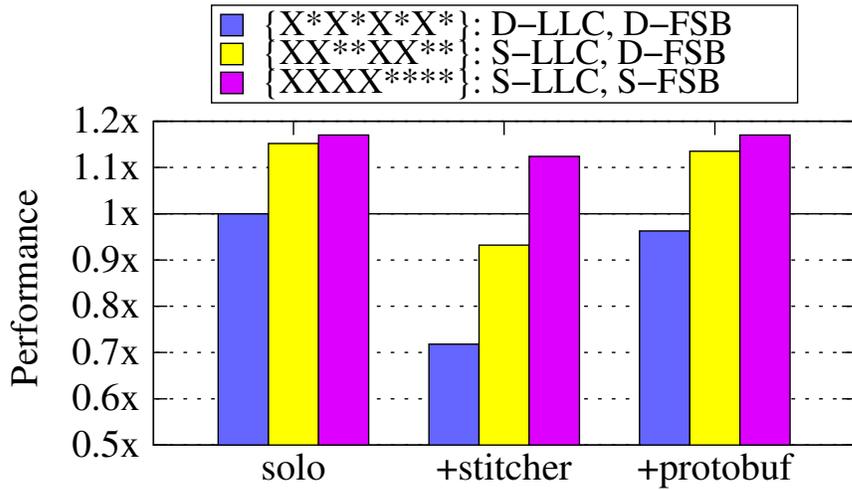


Figure 3.15: Bigtable. Normalized to solo performance with {X.X.X.X.X}

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 3.11 and 3.14 show *webSearch*'s performance when it is co-running with *stitcher* and *protobuf*. Similar to Figure 3.10, in Figure 3.11, 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 3.14 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 3.11 and 3.14 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 3.12 and 3.15. 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 3.4. This table presents the optimal mapping for each application and 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 [61] (presented in Table 3.4's last row), our experiments demonstrate that industry-strength datacenter applications have diverse preferences in resource sharing and TTC mappings.

Table 3.4: Optimal Thread-To-Core Mapping in Solo and Co-location Situations

Benchmark	Solo	w/ <i>Stitcher</i>	w/ <i>Protobuf</i>
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

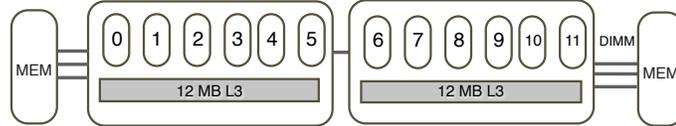


Figure 3.16: Topology of Dual Socket Intel Westmere

3.3.3 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.

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 3.2.1. Figure 3.17 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 3.18 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*

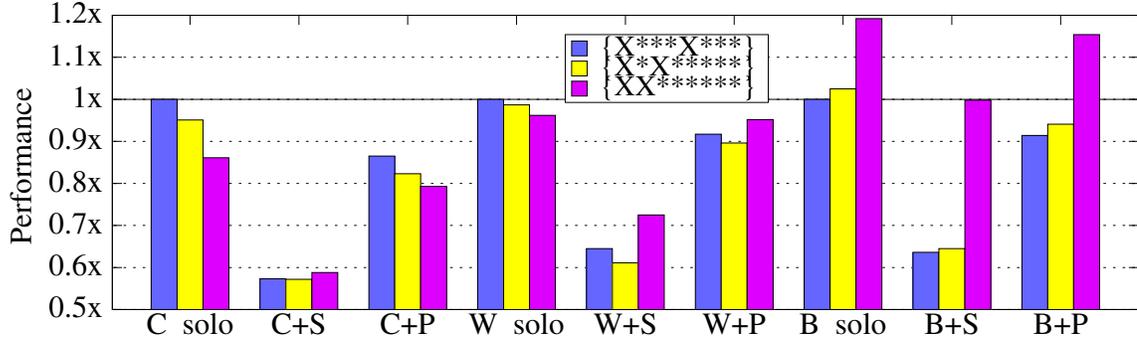


Figure 3.17: 2 threads of a latency sensitive application collocated with 6 threads of a batch application, normalized to the latency sensitive application’s solo performance in $\{X...X...\}$ mapping

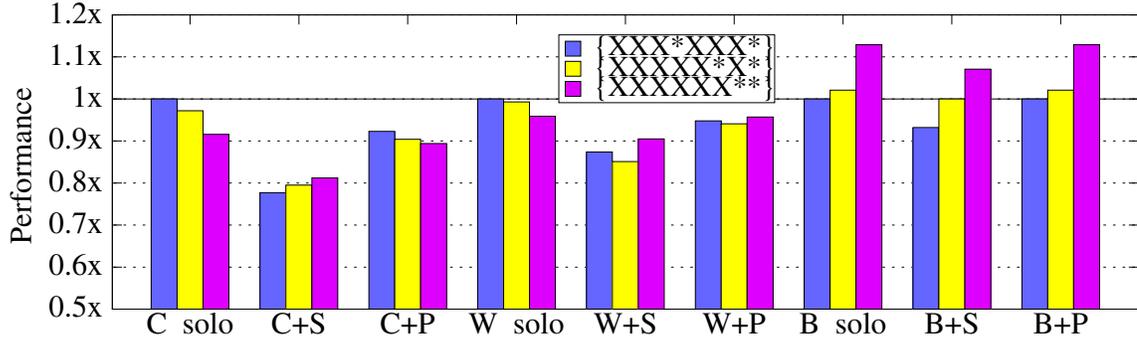


Figure 3.18: 6 threads of a latency sensitive application collocated with 2 threads of a batch application, normalized to the latency sensitive application’s solo performance in $\{XXX.,XXX.\}$ mapping

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.

Varying Architecture

We also conducted experiments on a Intel’s Westmere platform, shown in Figure 3.16. 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 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 3.19 and 3.20 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.

For the co-location study, we present the results when 6 threads of latency sensitive application co-running with 6 threads of corunner (Figure 3.21). 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.

3.3.4 Summary

In this section, our investigations 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

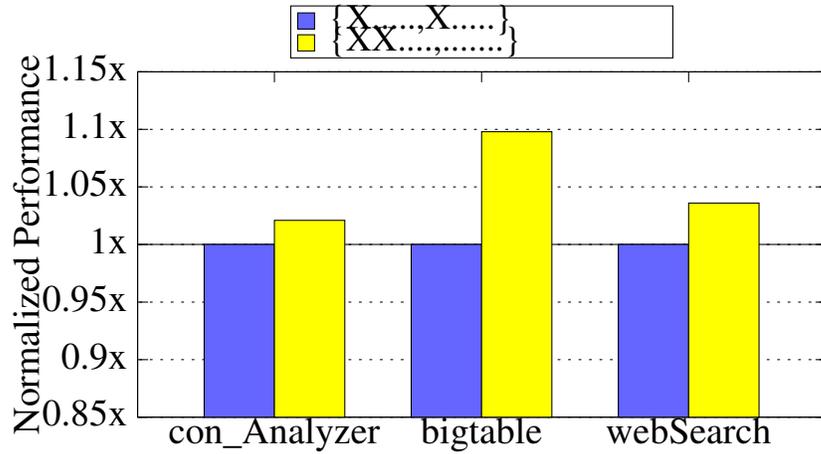


Figure 3.19: 2 threads of latency sensitive applications running alone on Westmere

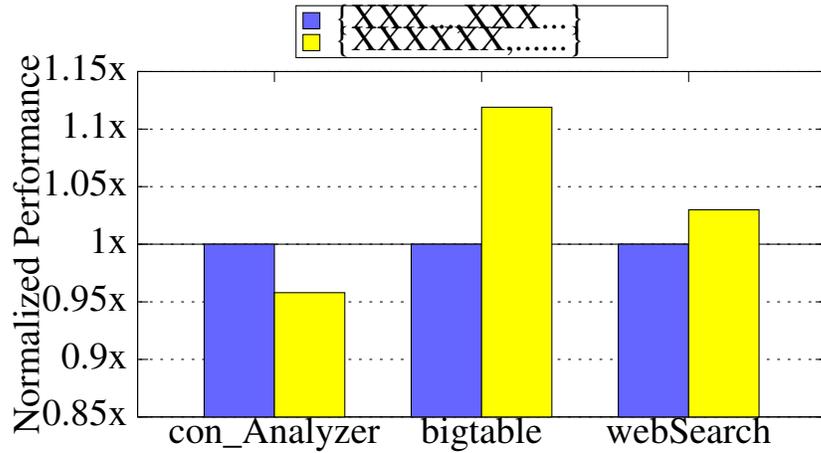


Figure 3.20: 6 threads of latency sensitive applications running alone on Westmere

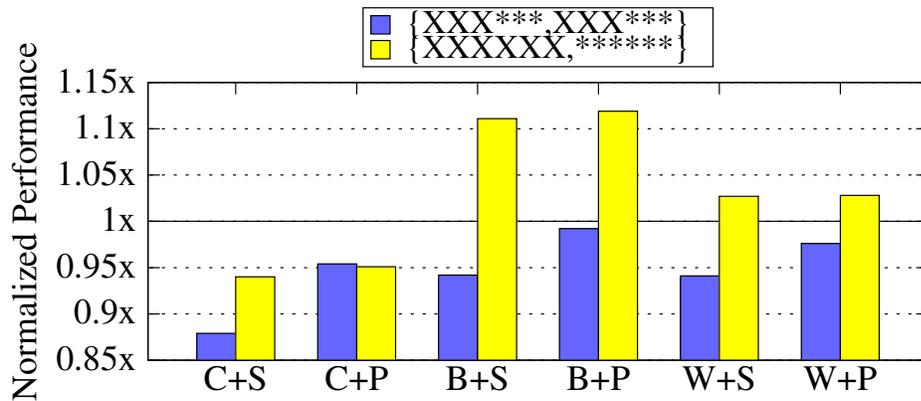


Figure 3.21: 6 threads of latency sensitive applications co-running with 6 threads of batch applications on Westmere;

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.

Chapter 4

Thread-to-core Mapping

Contents

4.1 A Heuristic Approach to TTC Mapping	44
4.1.1 Evaluating the Heuristics	48
4.2 An Adaptive Approach to TTC Mapping	49
4.2.1 Evaluating AToM	50

At the datacenter scale, a performance improvement of 1% for key applications, such as websearch, can result in millions of dollars saved. In Chapter 3, we demonstrate a performance swing of up to 25% for websearch, and 40% for other key applications, simply from remapping application threads to cores. The optimal thread-to-core mapping decision is also dynamic. It changes when the application changes, when the number of threads changes, when the architecture changes and when co-located application changes. These observations necessitate an intelligent thread-to-core mapping system to provide the performance that is currently left on the table.

In this chapter, we present two approaches to mitigating resource contention, exploiting positive resource sharing and ultimately improving performance using intelligent thread-to-core mapping:

- Heuristics approach: 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 (Section 4.1).

- Adaptive approach: an online system for arriving at the intelligent thread-to-core mapping (Section 4.2).

Finally, we conclude that our adaptive learning approach is a preferable approach for identifying good thread to core mappings in the datacenter. It arrives at near optimal decisions and is agnostic to applications’ sharing characteristics. When using the adaptive approach, 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

4.1 A Heuristic Approach to TTC 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. This dissertation has 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 Chapter 3 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 4.1 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 $15,000 \times 64\text{Byte} = 0.96\text{GB}/s$. The total bus transactions/ms for all four threads running on four cores can be as high as $0.96\text{GB}/s \times 4 = 3.8\text{GB}/s$. The theoretical FSB peak bandwidth on this platform is 10.6 GB/s. When using a micro-benchmark that measures peak bandwidth, `STREAM` [41], the observed maximum sustained bandwidth is 5.6GB/s. When four threads of *contentAnalyzer*

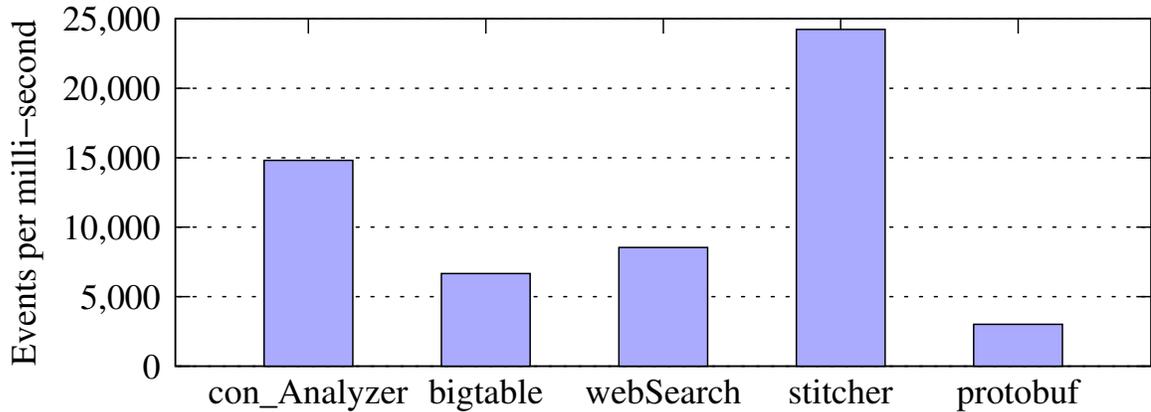


Figure 4.1: Bus Burst Transactions (full cache line) per millisecond per one thread

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 3.4. When *webSearch* and *contentAnalyzer* are running alone, because of the medium-high bus demand, 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 4.2 presents the

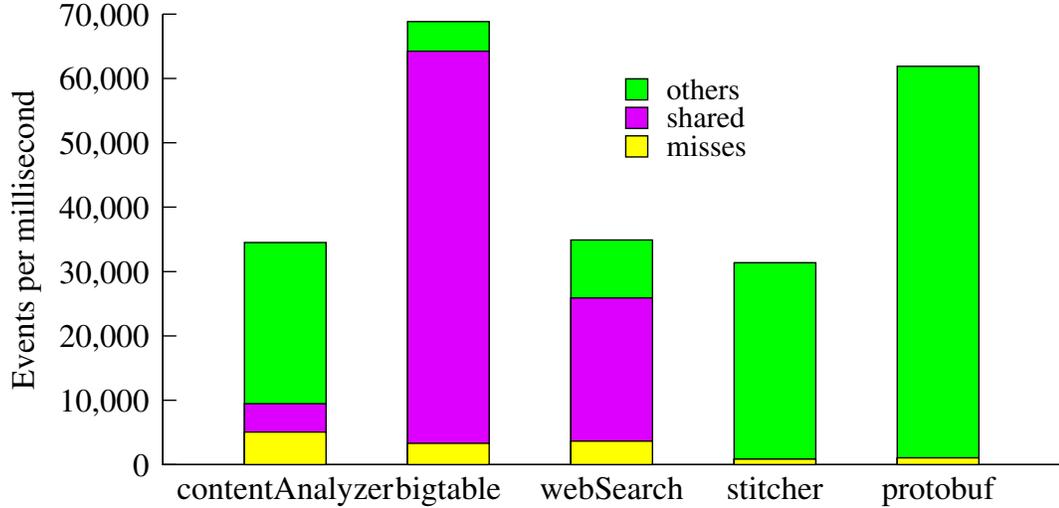


Figure 4.2: LLC misses/ms, LLC_requests_Share/ms and LLC reference/ms

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 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 [63, 29, 40] show that last level cache miss rate can be used as an indicator to estimate the footprint size of an application and to predict the potential performance degradation an application may cause to its co-runners. Figure 4.2 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 3.4. An application's cache characteristics

are important when deciding a good TTC mapping. And both data sharing and cache footprint need to be considered.

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  $P.BusUsage < C.BusUsage$  then
4     | | map(P, [share.LLC, sharing.FSB]) ;
5     | else
6     | | map(P, [share.LLC, distributed.FSB]) ;
7     | end
8   | end
9   | else
10  | | if  $P.BusUsage < C.BusUsage$  then
11  | | | map(P, sharing.FSB) ;
12  | | | if  $P.LLC_Footprint = high$  then
13  | | | | map(P, [distributed.LLC, sharing.FSB]);
14  | | | else
15  | | | | map(P, [share.LLC, sharing.FSB]);
16  | | | end
17  | | | else
18  | | | | if  $P.LLC_Footprint < C.LLC_Footprint$  then
19  | | | | | map(P, [share.LLC, distributed.FSB]);
20  | | | | else
21  | | | | | map(P, [distributed.LLC, distributed.FSB]);
22  | | | | end
23  | | | end
24  | end

```

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 decision tree this algorithm is based on is shown in Figure 4.3.

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 selects the thread(s) that have the least usage of the same resource to co-locate.

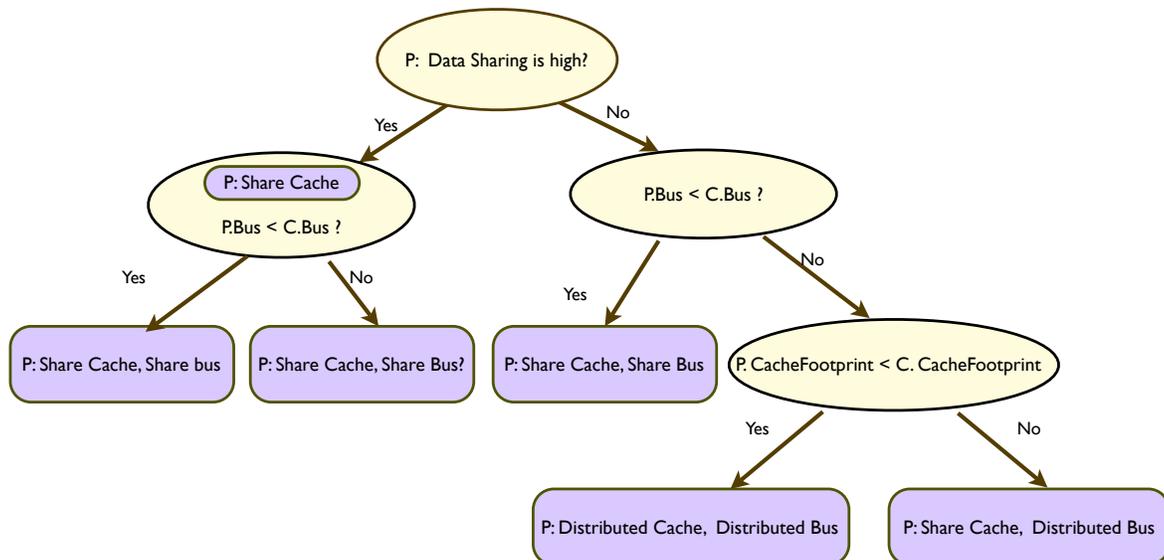


Figure 4.3: Decision Tree

4.1.1 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 multithreaded application, we use

$$LLC_MissRate \times \left(1 - \frac{LLC_shared_state_requests}{LLC_all_requests}\right) \quad (4.1)$$

The prediction result using heuristic algorithm is presented in Table 4.1.

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 3.14 and 3.15). The advantages of our heuristic approach are 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

Table 4.1: Predicted Thread-To-Core Mapping Using the Heuristic Approach

Benchmark	w/ <i>Stitcher</i>	w/ <i>Protobuf</i>
bigtable	Optimal: {XXXX***};	Optimal: {XXXX***};
	Predicted: {XXXX***}	Predicted: {XX**XX**} (sub-optimal: 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*} (sub-optimal: 1% worse)

to be generated on an architecture by architecture basis. Also, characteristics of an application may not be perfectly captured. For example, using LLC miss rate to approximate the cache footprint is not perfect [39], especially when there is data sharing between threads. These limitations motivate an adaptive approach that is more flexible and portable.

4.2 An Adaptive Approach to TTC Mapping

In this section we present AToM, an **A**daptive **T**hread-to-core **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 3.1) 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.

4.2.1 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 4.4 and 4.5 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, normalized by its performance when running alone in the $\{X \dots X \dots\}$ 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.

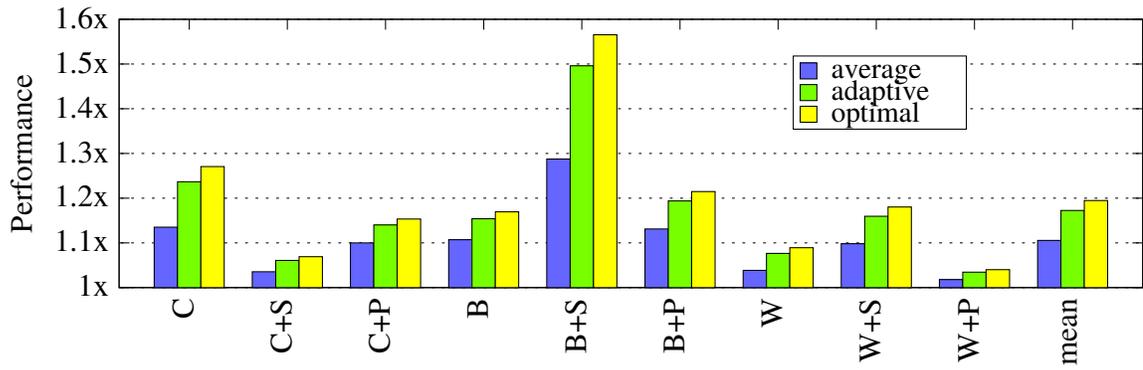


Figure 4.4: Adaptive Thread-To-Core Mapping on Clovertown

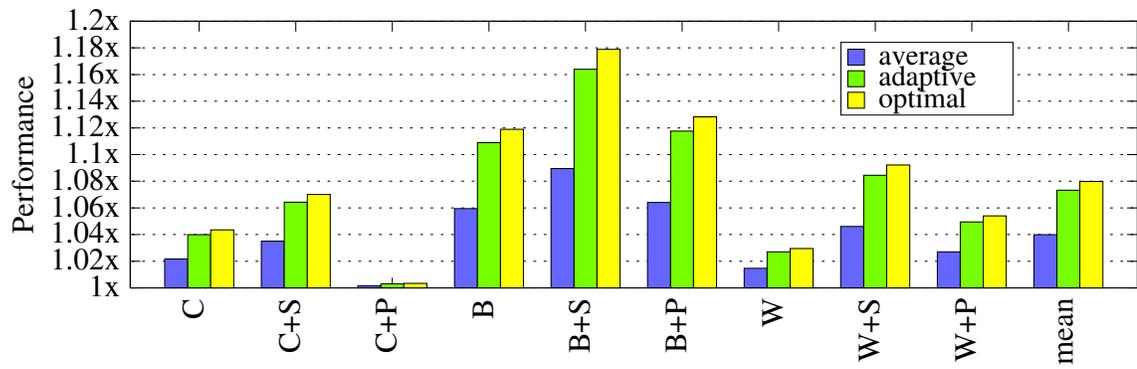


Figure 4.5: Adaptive Thread-To-Core Mapping on Westmere

Chapter 5

Compiling for Niceness

Contents

5.1 QoS-Compile Overview	53
5.2 Identify Contentious Code Regions	55
5.2.1 Contentiousness and Sensitivity	56
5.2.2 Identify Contentious Regions	61
5.3 Compiler Transformations for Rate Reduction	64
5.3.1 Padding	64
5.3.2 Nap Insertion	66
5.3.3 Understanding Cooldown and Warmup	68
5.4 Evaluation	68
5.4.1 Setup and Methodology	69
5.4.2 Model for Code Region Identification	69
5.4.3 Compiler Transformations	73
5.4.4 QoS-Compile: Put it All Together	77
5.4.5 Google Applications	82
5.5 Summary	83

Chapter 4 presents intelligent thread-to-core mapping for mitigating memory resource contention and improving **application performance**. In this chapter, we present a novel approach to mitigating contention and improving **server utilization** in modern warehouse scale computers.

As modern warehouse scale computers continue to leverage commodity multicore processors with increasing core counts, there is a growing need to consolidate various workloads on these machines to fully utilize their computation power. However, in Chapter 3, we demonstrate that when multiple applications are co-located on a multicore machine, contention for shared memory resources can cause severe cross-core performance interference. To ensure that the *quality of service* (QoS) of user-facing applications does not suffer from performance interference, WSC operators resort to disallowing co-location of latency-sensitive applications with other applications. This policy translates to low machine utilization and millions of dollars wasted in WSCs.

In this chapter, we present **QoS-Compile**, the first compilation approach that statically manipulates application contentiousness to enable the co-location of applications with varying QoS requirements, and as a result, can greatly improve machine utilization. In essence, to co-locate applications of different QoS priorities, our compilation technique uses *pessimizing* transformations to throttle down the memory access rate of the contentious regions in low priority applications to reduce their interference to high priority applications. Our evaluation using synthetic benchmarks, SPEC benchmarks and large-scale Google applications show that QoS-Compile can greatly reduce contention, improve QoS of applications, and improve machine utilization. Our experiments show that our technique improves applications’ QoS performance by 21% and machine utilization by 36% on average.

5.1 QoS-Compile Overview

There are two key insights of QoS-Compile. Firstly, WSCs typically house a known set of long running applications, such as web search and maps, running for weeks and months at a time. A cluster-level scheduler maps multiple applications to each individual machine, and thus the co-location persists for this period until a job finishes running. The various QoS priorities of these applications are known throughout the lifetime of the WSC. In addition, binaries of these applications are available and the profiling can be performed continuously both in production and in test settings. Within this environment, a compilation approach is particularly useful for tailoring the binaries of these applications to “play nice” together.

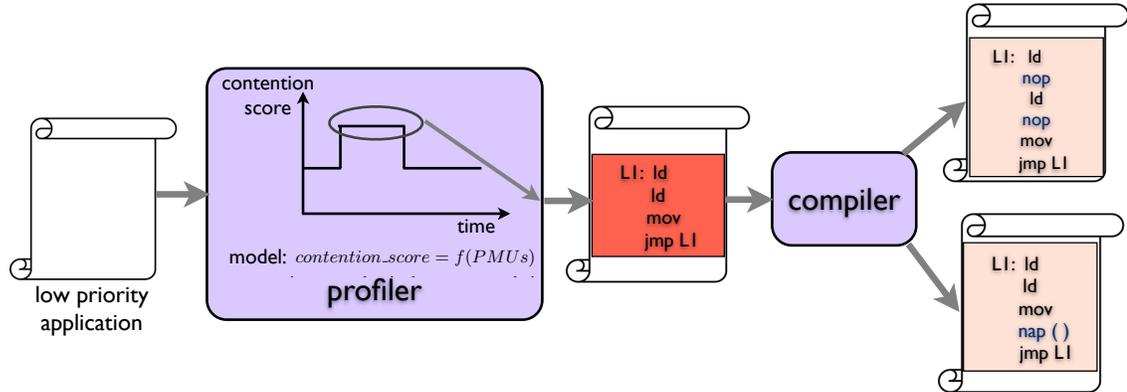


Figure 5.1: QoS-Compile Overview

Secondly, in the era of multicores and the emerging computing domain of WSCs, the objectives of compiler optimization ought to be multifaceted. Simply optimizing each application for its own individual performance irrespective of the surrounding execution environment may not be ideal. In this work, we argue for the additional objective of optimizing for an application’s “niceness,” to reduce its potential interference to its co-running applications.

QoS-Compile consists of two steps. First, the application is profiled to identify its contentious code regions. Second, transformations are applied to these regions to reduce their contentiousness.

[Identifying Contentious Regions] Resource contention is only manifested during runtime, and as a result, a static code analysis to identify such code regions may not be feasible. Our technique uses a profiling analysis to characterize the memory resources usage of an application when it is running alone. The intuition is that if a code region aggressively uses shared memory resources (shared caches and memory bandwidth, etc) when executing, this region may interfere with a co-runner that is sensitive to contention. To predict a code region’s contentiousness, we established a performance monitoring unit (PMU) based prediction model via regression. As Figure 5.1 shows, our profiler dynamically samples PMUs when an application is executing, estimates the contentiousness of code regions using the prediction model, and selects code regions that are above a certain contention threshold. Being able to pinpoint just the regions responsible for contention is a key benefit of QoS-Compile as we only throttle down the memory access rate of these regions. Applications may have short bursts of contentiousness or be contentious only during certain phases. Our

compiler transformations are applied only to the code that is responsible for these bursts or phases. The details of the profiling technique are presented in Section 5.2.2.

[**Compiling for “Niceness”**] After identifying the contentious code regions, QoS-Compile then specializes the code layout of these regions to reduce their contentious nature, as shown in Figure 5.1. QoS-Compile is essentially a software *rate-based* technique as it throttles down the memory request rate of a low priority application, reducing the resulting pressure on the memory subsystem and allowing the neighboring high priority applications to consume more of these resources. In this work, we develop two transformations for memory request rate reduction: **padding** and **nap insertion**. Using these two transformations, QoS-Compile provides a wide range of *throttling granularities*. These granularities range from intermittent bursts of just a few instructions before a brief pause, to thousands of instructions before each longer nap. Our padding transformation provides fine granularity throttling while nap insertion provides coarser granularities. Both of these transformations include parameters for adjusting the amount of rate reduction, which in turn controls the amount of interference and QoS degradation suffered by co-runners. This tunability is important for achieving the desirable balance between QoS and machine utilization. The details of the compilation transformations are presented in Section 5.3.

[**Using QoS-Compile in a Modern WSC**] In modern WSCs, high priority latency-sensitive jobs, such as web-search and maps, are run on machines for weeks and months at a time. These jobs often use a fraction of the cores on a single machine. However, to protect their QoS, the co-location of other jobs on these machines is often disallowed. QoS-Compile can be used, on demand, to compile low priority batch jobs, such as video encoding/decoding and compression, to enable their co-location on these underutilized machine resources. QoS-Compile can also be composed with a number of multi-versioning schemes [34] to enable its rate reduction transformations only when co-running with a high priority application.

5.2 Identify Contentious Code Regions

In this section we present our novel profiling technique and its core component, a performance counter based prediction model, to identify contentious code regions. Section 5.2.1

investigates and answers two key questions about a code regions’ contentious nature that prior work has no answer or conflicting answers for. Section 5.2.2 then presents our technique to identify contentious code regions based on the insights gained from our investigation.

5.2.1 Contentiousness and Sensitivity

To identify contentious code regions, it is important to first have an in-depth understanding of application contention characteristics, including an application’s *contentiousness*, which is the potential performance degradation it can cause to its co-runners, and an application’s *sensitivity* to contention, which is the potential degradation it can suffer from its co-runners. In this section we present formal definitions of both *contentiousness* and *contention sensitivity*, and investigate key questions about the nature of each and how they relate.

We first investigated whether contention characteristics (both contentiousness and sensitivity to contention) are *consistent* characteristics of an application. One hypothesis for identifying contentious code regions is that contentiousness is a consistent characteristic of a code region. We define consistent as, for a given machine, the relative ordering between all applications’ contentiousness and sensitivity in general does not change across different co-runners.

Secondly, we investigated the correlation between an application’s contentiousness and its sensitivity to contention. An important observation is that both an application’s contentiousness, and its sensitivity to contention, involve the usage of shared resources. One intuition is that contentious applications may also be sensitive to contention and vice versa. Prior work has had conflicting conclusions about the relations between an application’s contentiousness and contention sensitivity. There are four possible outcomes. An application can be 1) contentious and sensitive; 2) not contentious and insensitive; 3) contentious but not sensitive; and 4) not contentious but sensitive. Among these four outcomes, Jiang et al. [24, 35] conclude that typical applications’ contentiousness and sensitivity are strongly correlated and should be classified as either contentious and sensitive, or not contentious and insensitive. Xie et al. [59] on the other hand, argue the existence of applications that are not contentious but sensitive. Meanwhile, other recent works [63, 29] argue that a

contentious application that has high cache misses is likely to be very sensitive as well.

[Definition] Before answering these questions, we first present formal definitions of both contentiousness and contention sensitivity. On multicore processors, an application’s *contentiousness* is defined as the potential performance degradation it can cause to co-running application(s) due to its heavy demand on shared resources. On the other hand, an application’s *sensitivity* to contention is defined by its potential to suffer performance degradation from the interference caused by its *contentious* co-runners.

As demonstrated in previous work [24], an application A’s sensitivity is formally defined using the following formula,

$$Sensitivity_A = \frac{IPC_{A(solo)} - \overline{IPC_{A(co-run)}}}{IPC_{A(solo)}} \quad (5.1)$$

where $IPC_{A(solo)}$ is A’s IPC when it is running alone and $\overline{IPC_{A(co-run)}}$ is the statistical expectation of the A’s IPC when it co-runs with random co-runners. We extend this definition to include A’s contentiousness as,

$$Contentiousness_A = \frac{\overline{IPC_{B_i(solo)}} - \overline{IPC_{B_i(co-run_A)}}}{\overline{IPC_{B_i(solo)}}} \quad (5.2)$$

where A’s contentiousness is quantified as the statistical expectation of the IPC degradation A causes to its random co-runner.

We can estimate $Sensitivity_A$ and $Contentiousness_A$ by co-locating A with various co-runners B_i , and take the average of A’s measured contentiousness and contention sensitivity. A’s sensitivity to corunner B_i can be defined as,

$$Sensitivity_{A(co-run_{B_i})} = \frac{IPC_{A(solo)} - IPC_{A(co-run_{B_i})}}{IPC_{A(solo)}} \quad (5.3)$$

and the A’s average measured sensitivity is,

$$Sensitivity_{A(avg)} = \frac{\sum_i^n Sensitivity_{A(co-run_{B_i})}}{n} \quad (5.4)$$

Similarly, we can define A’s contentiousness when it is co-running with B_i and its average

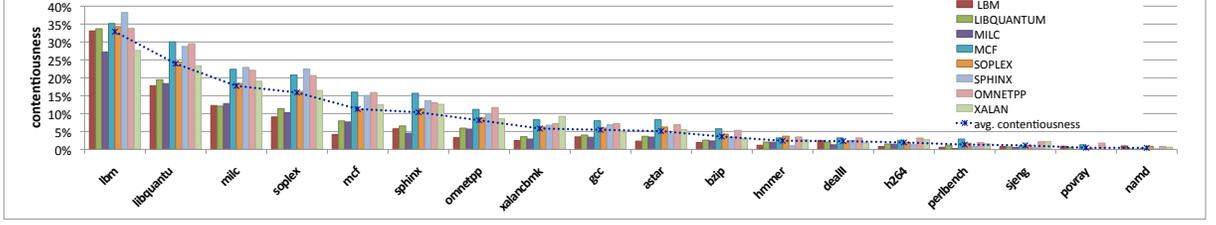


Figure 5.2: Contentiousness. Each bar shows the performance degradation of a corunner caused by the application across x-axis.

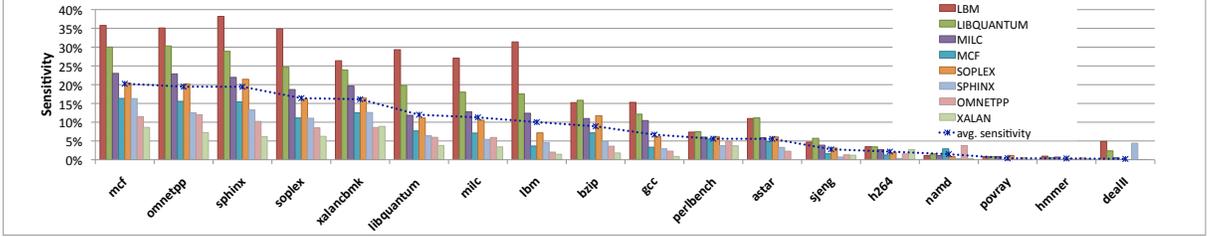


Figure 5.3: Sensitivity. Each bar shows the performance degradation of the application across x-axis caused by each of the 8 different corunners.

contentiousness as,

$$Contentiousness_{A(co-run_{B_i})} = \frac{IPC_{B_i(solo)} - IPC_{i(co-run_A)}}{IPC_{B_i(solo)}} \quad (5.5)$$

$$Contentiousness_{A(avg)} = \frac{\sum_i^n Contentiousness_{A(co-run_{B_i})}}{n} \quad (5.6)$$

In this work we use Equation 5.4 to estimate $sensitivity_A$, and Equation 5.6 to estimate $contentiousness_A$.

[Experiment Design] To evaluate these key questions as regards to the nature of contention characteristics of an application, we have performed a series of experiments using 18 benchmarks of SPEC CPU2006 benchmarks suite. These benchmarks represent a diverse range of application workloads and memory behaviors, including different working set sizes, cache misses, and offcore traffic. All experiments were conducted on Intel Core i7 920 (Nehalem) Quad Core with 2.67GHZ processors, 8MB last level cache shared by four cores and 4GB memory. For each experiment, we selected two of the 18 benchmarks, co-located them on neighboring two cores, and measured each benchmark’s contentiousness and sensitivity in each experiment using Equation 5.3 and Equation 5.5. We then calculated each benchmark’s average contentiousness and sensitivity using Equation 5.4 and Equation 5.6.

We conducted exhaustive co-running of all possible co-running pairs, which is a total of 162 ($\frac{18 \times 18}{2}$) co-running experiments executed to completion on `ref` inputs. Each experiment was conducted three times to calculate the average. Note that SPEC runs are fairly stable and there is little variance between runs.

[Is contentiousness a consistent characteristic of an application?] Figure 5.2 presents our benchmarks’ *contentiousness*. This contentiousness is calculated using Equation 5.5, which indicates the performance degradation each of the 18 benchmarks causes to its co-runner. The 18 benchmarks are shown on the x-axis. For each of the 18 benchmarks, we show its measured contentiousness when it is co-running with each of the eight most contentious co-runners respectively. Each bar represents a co-runner. Only 8 co-runners are shown in the figure because of the space limit. The dotted line shows the average contentiousness of each benchmark, computed by averaging each benchmark’s 18 contentiousness values across 18 co-runners using Equation 5.6. The 18 benchmarks on the x-axis are then sorted by their average contentiousness. The line graph for average contentiousness shows a general descending trend.

Figure 5.2 demonstrates that contentiousness is a consistent characteristic of an application. The relative order of benchmarks’ contentiousness stays fairly consistent regardless of which co-runner is present. For example, when comparing each benchmark’s contentiousness when it is co-running with `lbm`, shown by the first bar for each 18 benchmark, we notice that the contentiousness of 18 benchmarks are almost all in descending order along the y-axis mirroring the dotted line. This also applies to all other co-runners as well. The graph also shows that `lbm` is the most contentious benchmark among the 18 benchmarks.

[Is sensitivity a consistent characteristics of an application?] Similar to Figure 5.2, Figure 5.3 shows the sensitivity to contention of each of the 18 benchmarks when co-located with the most contentious applications. This sensitivity is calculated using Equation 5.3, indicating how much degradation the eight co-runners cause to each of the 18 benchmarks. These 18 benchmarks are sorted according to their average sensitivity, calculated using Equation 5.4. Similar to Figure 5.2, this figure shows that sensitivity is also consistent for each application. Although the descending trend is not as consistent as Figure 5.2, the general trend is strong.

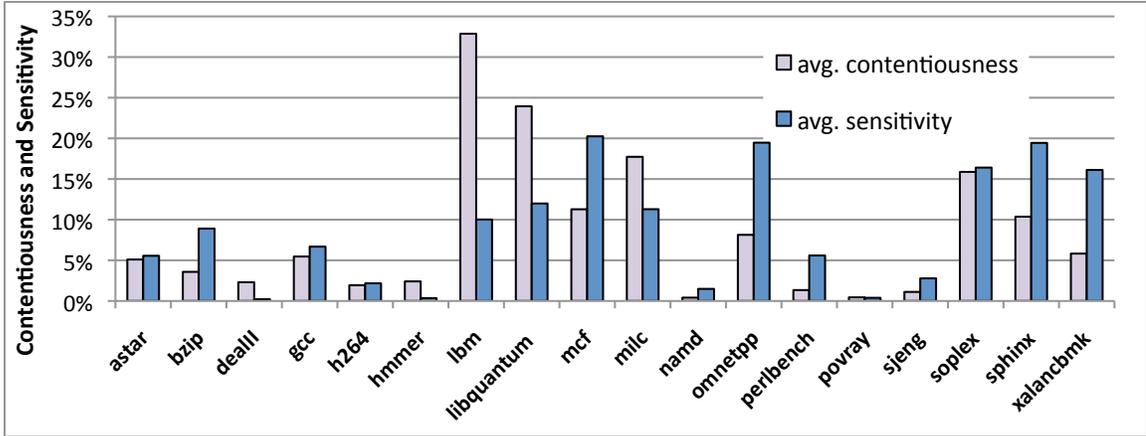


Figure 5.4: Average Contentiousness vs. Sensitivity

[Contentiousness vs. Sensitivity: are they strongly correlated?] In Figure 5.4, we juxtapose contentiousness and sensitivity. In this graph, for each application across the x-axis, the first bar shows the average contentiousness of this application with the eighteen co-runners presented in Figures 5.2 and 5.3. The second bar shows each benchmark’s average sensitivity to the same set of co-runners. Figure 5.4 clearly demonstrates a large disparity between application contentiousness and sensitivity. As shown in the figure, applications such as `lbm` and `libquantum` are highly contentious and only mildly sensitive, while other applications such as `omnetpp` and `xalan` are highly sensitive, and slightly contentious. Also notice that, in Figures 5.2 and 5.3, the sorted orderings of the 18 benchmarks (x-axis) are almost completely different. In fact, the correlation coefficient between contentiousness and sensitivity using linear regression is 0.48, which further shows they are not strongly correlated.

Summary To summarize, through our experimentation we find,

1. Contentiousness and sensitivity are an application’s consistent characteristics. Figure 5.2 shows that applications with higher contentiousness tend to be consistently more contentious regardless of co-runners. This general trend also applies to sensitivity, as shown in Figure 5.3.
2. Contentiousness and sensitivity of general purpose applications are not strongly correlated as shown in Figure 5.4. While we do not observe applications that are only sensitive or only contentious, four outcomes occur in practice; applications can be 1)

contentious and sensitive; 2) not contentious and insensitive; 3) contentious but not highly sensitive; 4) not highly contentious but sensitive.

5.2.2 Identify Contentious Regions

In the previous section, we show that contentiousness is an inherent consistent characteristic of an application or a code region. In this section, we present the profiling analysis used to identify contentious code regions of an application. The core component of our analysis is a model based on hardware performance counters for the dynamic scoring of sequences of executed code. The intuition behind using the information provided by hardware performance counters is that if a code region aggressively consumes certain memory resources, it is likely to be contentious for the resource when it is co-running with other applications. In this section, we first discuss how we constructed the model. We then describe how this model is used during a profiling run to identify the static code regions that are most contentious.

[General Model for Contentiousness] We use a linear model to combine the impact of contention in multiple shared resources, including last level cache (LLC), memory bandwidth and prefetchers. The contentiousness of a dynamically executed code region is determined by the amount of pressure the region puts on the shared memory subsystem. Thus, it can be predicted based on usage of shared resources, shown as the following equation,

$$C = a_1 \times LLC_usage + b_1 \times BW_usage + c_1 \times Pref_usage, \quad (5.7)$$

where C is contention score, BW is bandwidth and $Pref$ is prefetchers.

Each code region may have a different combination of cache, bandwidth and prefetch usage. How contentious each code region is relative to other regions depends on the relative importance between cache, bandwidth and prefetcher contention. The relative importance is reflected as coefficients a_1 , b_1 and c_1 .

[Leveraging PMUs] Modern architectures provide numerous performance counters for various aspects of the microarchitecture. Our second step is to identify the appropriate performance monitoring units (PMUs) to estimate the terms in Equation 5.7.

BW_usage: It is fairly easy to quantify and measure bandwidth usage using PMUs. For

example, we can use the number of cache lines the last level cache brings in from memory per second.

LLC_usage: It is challenging to measure cache usage using PMUs. PMUs can provide information on the cache access frequency and the cache miss rate, but currently they do not provide information on the cache footprint or occupancy. To approximate LLC usage, we measure how much data is fetched from the shared cache and not the memory for a given interval.

Prefetcher_usage: Not all architectures provide performance counters for all prefetchers. However, the main impact of prefetchers is reflected as increased bandwidth and cache usage. Thus, prefetcher usage can be estimated using cache and memory bandwidth usage.

Guided by the above insights, we identify the appropriate PMUs on the Intel Core i7 (Nehalem). On this platform, we identify the number of cache lines the last level cache brings in per millisecond (*LLCLinesIn/ms*), as shown in Figure 5.5, to capture the aggregate pressure an application puts on the bandwidth. We identify $(L2LinesIn - L3LinesIn)/ms$ to estimate the shared L3 cache usage. It reports the rate of data being fetched into private caches from the shared cache. Because both *L3LinesIn* and *L2LinesIn* include the prefetchers’ traffic, we do not need an extra PMU to measure the prefetcher usage. Using the above PMUs, Equation 5.7 becomes:

$$C = a_1 \times (L2LinesIn_rate - L3LinesIn_rate) + b_1 \times L3LinesIn_rate \quad (5.8)$$

where *C* is contention score.

[Regression to Establish the Prediction Model] After identifying the appropriate PMUs, we use multiple regression to determine the coefficients in Equation 5.8. We use the **SmashBench** suite [37, 38] (Table 5.1), developed in Google, to train our model. SmashBench is composed of contentious kernels that span a spectrum of contentious memory access patterns and working set sizes. We measure each kernel’s contentiousness using the average performance degradation it causes to other kernels within the suite when co-running. Using the measured contentiousness and the measured PMUs profile, including the average *L2LinesIn/ms* and *L3LinesIn/ms*, we then conduct regression analysis to determine the

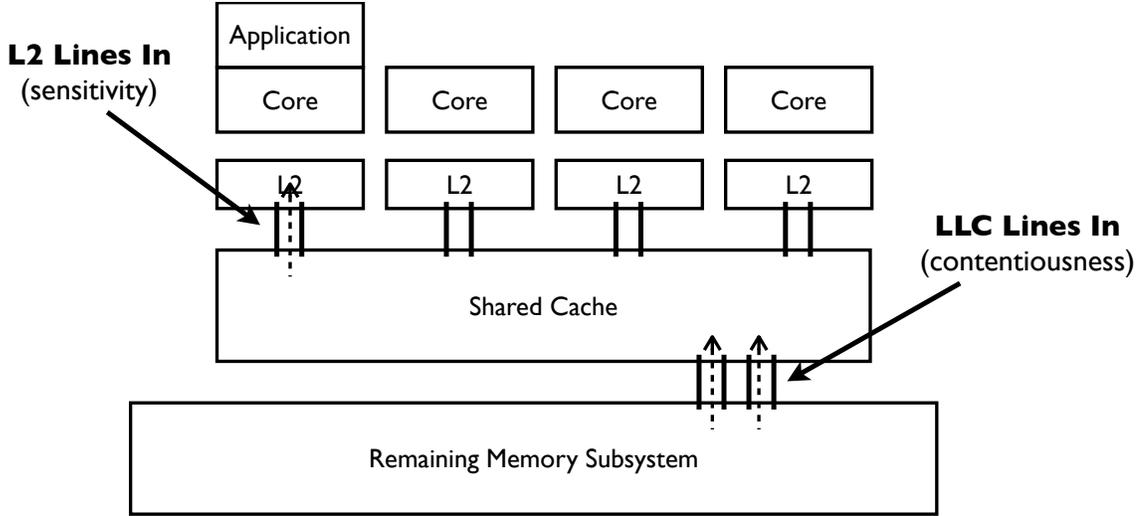


Figure 5.5: PMUs used for predicting contentiousness

model coefficients (Equation 5.7). The regression result is:

$$\begin{aligned}
 C = & 1.663 \times (L2LinesIn/ns - L3LinesIn/ns) \\
 & + 8.890 \times L3LinesIn/ns + 0.044
 \end{aligned}
 \tag{5.9}$$

The p value for $(L2LinesIn/ns - L3LinesIn/ns)$ is 0.018, $5.11e-07$ for $L3LinesIn/ns$, and $2.015e-06$ for the entire regression. All are smaller than 0.5, indicating statistically significant effects. The R-squared is 0.8876, indicating a strong fit. The coefficients show the relative importance between the bandwidth usage and the LLC usage, indicating that memory bandwidth contention has a more dominating effect.

The regression results show that our model combines the contention of multiple resources and is highly indicative of the performance interference a code region may cause. The prediction accuracy of the model is evaluated in Section 5.4.

[Identifying Code Regions] Identifying code regions based on the PMU model is fairly straightforward, and involves correlating PMU information with its corresponding source code. There are a number of approaches for conducting the correlation. In this work, we use a simple approach. We first record the application’s PMU statistics (L2 and L3 lines in rate) every 1ms. Meanwhile, we record the number of instructions executed in every sample interval. These serve as markers in the dynamic instruction trace for the

Benchmark	Footprint	Description
bst	4mb, 8mb, 50mb	random accessing a binary search tree
naive	4mb, 8mb, 50mb	random accessing an array
er-naive	4mb, 8mb, 50mb	fast random accessing an array
blockie	small, medium, large	a number of large 3D arrays. A portion of one array is continuously copied to another.
sledge	small, medium, large	two large arrays, copies data back and forth between arrays with this sledgehammer pattern.

Table 5.1: Contention Benchmarks Suite: SmashBench

sequence of instructions that are responsible for the PMU data. We use the collected PMU profile and Equation 5.9 to calculate a contention score for every 1ms instruction interval. We then use a PIN [33] tool to replay the execution. Using the recorded interval markers we analyze the set of source level basic blocks that comprise the 1ms interval. We select the hottest set of basic blocks of that region, typically comprising more than 90% coverage of the interval, and assign these blocks the corresponding contentiousness score that was produced by our model.

5.3 Compiler Transformations for Rate Reduction

QoS-Compile provides two compilation techniques, *padding* and *nap insertion*, for both fine-grain and coarse-grain memory request rate reduction. In this section, we describe both of these techniques and discuss the tradeoffs between them.

5.3.1 Padding

Our *padding* transformation inserts non-memory instructions between memory instructions in a contentious code region. These instructions consume CPU cycles but do not issue memory requests. Therefore, in essence, they limit the amount of memory requests issued in a

given time interval. When the amount of padding increases, the code region’s pressure on the memory subsystem decreases. We implement padding by inserting no operation instructions (`nop`) in contentious code regions at the basic block level using MAO [20]. Padding provides a fine grain mechanism for reducing a code region’s execution rate, memory request rate, and its interference to co-runners. Inserting these `nops` artificially inflicts a slowdown that can be as small as the number of cycles consumed by a single `nop`.

Application specific and microarchitecture specific factors need to be considered when deciding a sensible padding policy for a given interference reduction goal. The application specific factors include:

1. *The code region’s memory characteristics.* The contentious level of a code region affects the amount of padding needed. The more contentious, the more padding needed. In addition, many memory characteristics such as the footprint affect the latency of memory instructions, which in turn affects the amount of padding needed. We discuss more about this effect shortly.
2. *Binary instruction characteristics.* The instruction mix, for example, the ratio of memory instructions (loads, etc) versus other instructions (CPU instructions) also needs to be considered. For a given amount of instructions, the more dense memory instructions are, the more padding may be required to reduce the pressure they cause to the memory system.

In addition, microarchitecture specific factors include:

1. How `nops` are executed on the architecture;
2. The memory hierarchy design and the access latencies for different levels in the memory hierarchy.

Many of the above factors essentially affect the memory latency of instructions, which is important when deciding a padding policy for a given interference reduction goal. This is mostly because that an application can be stalled on the memory instructions when the data is being fetched. During this period, `nops` may not have an effect on slowing down the application execution rate or memory request rate because the program is already stalled.

For example, a `load` may take hundreds of cycles to complete. When stalled on a use, a large amount of `nops` after this `load` may be useless for rate reduction. Therefore, each `nop`, depending on where it is inserted and the latency of memory instructions before it, may have a different impact on the memory request rate. This makes it difficult to accurately predict the rate reduction effect for a padding policy.

There are two main parameters for padding: *granularity* and *thickness*. Padding granularity is how often to pad (for example, every 3 instructions) and the thickness is how much `nops` to insert at every insertion point. In this paper, given a list of contentious basic blocks identified by the QoS-Compile’s profiler, we instrument padding at the beginning of each basic block. If a basic block contains more instructions than the specified padding granularity, we instrument within the basic block as well. The amount of padding inserted is determined by the thickness parameter. Generally, as discussed, the more dense memory instructions are, the longer latency they incur, the thicker padding is needed.

5.3.2 Nap Insertion

Our *nap insertion* technique inserts intermittent sleep to contentious code regions. Putting a contentious code region to epochal short “nap” mode reduces the pressure it puts on the memory subsystem and the interference it can cause to its co-runners. Similar to padding, two important parameters for nap insertions are *granularity* (how often the contentious code should nap) and *nap duration* (how long a nap interval should be, which is similar to padding thickness). However, comparing to padding, nap insertion is a much coarser-grain rate control as naps can occur for milliseconds at a time.

Another difference between nap insertion and padding is that, while padding indirectly controls the execution rate by inserting instructions to prolong the execution time, nap insertion on the other hand, directly controls the time allotted between naps and the duration of the nap, thus having a more accurate and predictable rate reduction control than padding. To estimate the effect of nap insertion on memory request or execution rate reduction, we use the following equation:

$$R_{execution} = \frac{nap_granularity}{nap_granularity + nap_duration} \quad (5.10)$$

where $nap_granularity$ is the duration of the execution interval between inserted naps and $nap_duration$ is the length of a nap. Given the execution rate $R_{execution}$ of a low priority application, L , we can estimate the improved QoS of its high priority co-runner, H . We denote H 's improved QoS using QoS_{imprd_co-run} :

$$QoS_{imprd_co-run} = 1 - (1 - QoS_{orig_co-run}) \times R_{execution} \quad (5.11)$$

where QoS_{imprd_co-run} and QoS_{orig_co-run} are both normalized by H 's QoS when running alone, and QoS_{orig_co-run} is H 's QoS when co-running with the original L ; QoS_{imprd_co-run} is H 's QoS when co-running with the napping L . Padding can also use Equation 5.11 to predict the improved QoS when padded code region is reducing to a certain execution rate $R_{execution}$. However, as we discuss later, because of the coarse grain control, nap insertion is less skewed by the cooldown/ warmup effect.

Algorithm 2: NapInsertion

Input : *Binary*, *nap_granularity*, *nap_duration*
Output: *Binary* with inserted nap

```

1 instrument a global variable counter;
2 foreach BasicBlock in Binary do
3   if (BasicBlock.contention_score > contention_threshold) and (BasicBlock.coverage >
   coverage_threshold) then
4     | InstrumentNap(BasicBlock, nap_granularity, nap_duration);
5   | end
6 end
```

Algorithm 3: InstrumentNap

Input : *BasicBlock*, *nap_granularity*, *nap_duration*
Output: *BasicBlock* with inserted nap

```

1 At the beginning of the BasicBlock, instrument the following code: counter ++;
2 if (counter > counter_threshold) then
3   | cur_time ← read_time_stamp_register ;
4   | if (cur_time - pre_time >= nap_granularity) then
5   | | sleep(nap_duration);
6   | | prev_time ← read_time_stamp_register ;
7   | | counter ← 0;
8   | end
9 end
```

The main algorithm to conduct nap insertion is presented in Algorithm 2 and the instrumentation function is presented in Algorithm 3. The nap is only inserted to top basic blocks that are above a contention score threshold and are above a certain execution time coverage. The contention score of each basic block is generated by our profiling approach in

Section 5.2.2. To reduce the overhead of checking the time stamp, we also use a counter to keep track of how many times the selected contentious basic blocks are executed and only to check the elapsed execution time when the counter is above a threshold.

5.3.3 Understanding Cooldown and Warmup

When applying a given amount of rate reduction to a code region, it may seem intuitive that it should provide the same amount of the interference reduction to a given co-runner. However, the granularity at which the intermittent rate reduction is conducted indeed matters. This is because of the *memory pressure cool-down and cache warm-up* effect. Again, we use L to denote a low priority application to which we conduct padding or nap insertion, and H to denote a high QoS priority application whose QoS we are aiming to improve. When padding or a nap just starts to throttle down memory requests, it would take a while for L 's pressure on the memory subsystem to cool down, especially if the data are residing below the cache. The memory system will still be serving L 's requests issued before the padding or nap for a short period of time. Meanwhile, it takes a while for H to warm up the cache to achieve its optimal performance when it is running alone. We call this period the *cooldown/warmup window*. During this window, the yielding of shared resources is not instant and may negatively impact the effectiveness of the rate reduction mechanism. This effect may not be negligible, especially for padding, because padding happens at a fine granularity (a number of cycles or ns). However, the severity of this window may be greatly reduced for nap insertion because nap insertion can be at a coarser granularity. In Evaluation (Section 5.4), we will further investigate the interaction of nap granularity and this cooldown/warmup effect.

5.4 Evaluation

In this section, we first evaluate the effectiveness of our prediction model and profiling technique in identifying contentious code regions. We then evaluate the application of our *padding* and *nap insertion* compiler transformations to reduce the contentiousness of an application and improve its co-runner's QoS. We then investigate the impact of leveraging

QoS-Compile to improve utilization using both SPEC benchmarks and Google applications.

5.4.1 Setup and Methodology

Our evaluation is conducted on two platforms:

- *Intel Nehalem.* Intel Core i7 920 Quad Core with 2.67GHZ processors, 8MB last level cache shared by four cores and 4GB memory. This platform runs Linux 2.6.29.6 and GCC 4.4.6.
- *Intel Clovertown.* A dual socket Intel Clovertown (Xeon E5345). Each socket has 4 cores. Each 2 cores on the same socket are sharing a 4MB 16 way last level cache (L2). This platform runs Linux kernel version 2.6.26 and a customized GCC 4.4.3.

The workloads used in our evaluation include the *SmashBench* contentious kernel suite (summarized in Table 5.1), SPEC CPU2006, and large-scale Google applications such as websearch. SmashBench and SPEC experiments are conducted on the Intel Nehalem configuration and the Google experiments are conducted on production servers hosting the Intel Clovertown configuration. Each benchmark is compiled using GCC at the O2 level. All SPEC applications are run using `ref` inputs. Each experiment was conducted three times to calculate the average performance. SmashBench, SPEC and Google benchmark runs are fairly stable with a variance of 1% or less between runs.

5.4.2 Model for Code Region Identification

The key component of the profiling system is the PMU model used to correlate the memory subsystem activity of a code region to its contentious nature and potential for causing interference.

[Model Accuracy] To evaluate the accuracy of our PMU model (Equation 5.9), we compare our PMU model’s predicted contentiousness of SPEC benchmarks with their real measured contentiousness. We profile each benchmark’s PMUs (*L2LinesIn_rate* and *L3LinesIn_rate*), and calculate the predicted contentiousness using Equation 5.9 with the acquired PMU profiles. The prediction is then compared against each benchmark’s observed contentiousness, measured as the average performance degradation it causes to a set

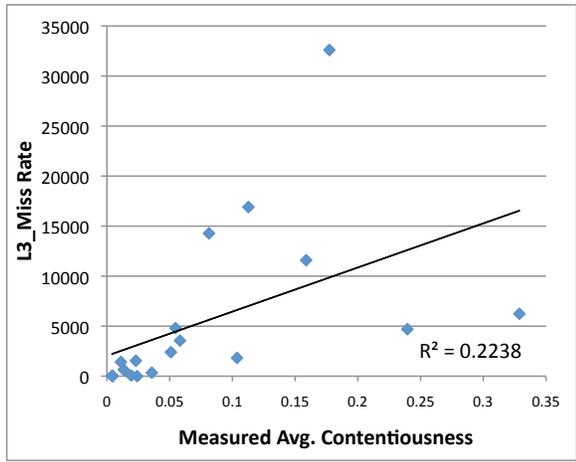


Figure 5.6: L3 Miss Rate is not strongly correlated with the real measured contentiousness

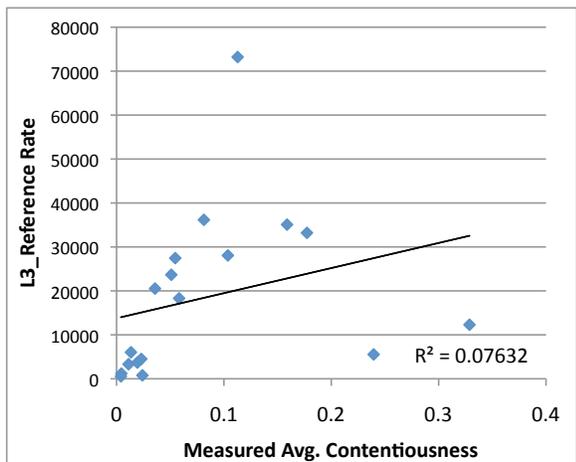


Figure 5.7: L3 Reference rate is not strongly correlated with the real measured contentiousness

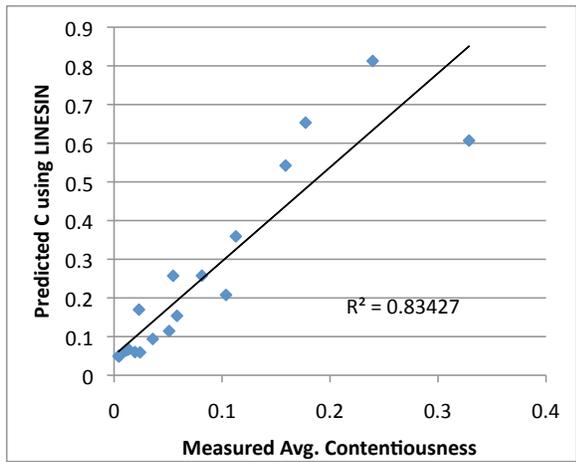


Figure 5.8: Predicted contention score using our model is highly correlated with the real measured contentiousness for SPEC benchmarks

Predictor	R^2
LLC Miss Rate	0.2238
LLC Reference Rate	0.07632
Our Prediction Model	0.83427

Table 5.2: Comparing our contentiousness predictor to predictors used in prior works. Our predictor was trained with the SmashBench suite of contentious kernels and tested against all SPEC 2006 benchmarks.

of co-runners.

As a baseline, we compare our predictive model to state of the art estimators proposed by prior work [63]. Figure 5.6 and 5.7 show the results when using LLC miss rate and LLC reference rate to predict applications’ contentiousness. The correlation coefficients (R) are 0.47 and 0.28, respectively, showing that neither LLC miss rate nor LLC reference rate alone can accurately indicate application contentiousness. Figure 5.8 presents our prediction results compared to the real measured contentiousness for SPEC CPU2006 benchmarks. Recall that our model is trained using a different set of benchmarks (e.g., SmashBench) and here we evaluate it on SPEC. For SPEC, the prediction’s linear correlation coefficient R is 0.91, indicating that our prediction model can accurately score contentiousness. Table 5.2 summarizes the correlation results and the correlation coefficient R of each model. Table 5.2 shows that our model is significantly better than prediction using LLC miss rate or LLC reference rate, as proposed in prior work. Keep in mind that our prediction model is trained with a separate set of applications, the SmashBench suite, and evaluated here on SPEC2006.

[Pinpointing Code Regions] To evaluate the effectiveness of pinpointing the contentious code regions using our PMU model, we compare benchmarks’ PMU model results with the degradation they cause to their co-runners. Figure 5.9 presents `sphinx`’s contention score calculated using its performance counter profile when it is running alone, based on Equation 5.9. The x-axis is time. Here `sphinx` is using `ref` input. The y-axis is the contention score using PMU model of `sphinx`’s execution phases. Figure 5.9 shows that `sphinx` is not evenly contentious through the entire execution, but, instead, there are several phases (humps in the figure) that are more contentious than the rest. Figure 5.10 presents `bst8mb`’s degradation when running with `sphinx`. This figure also presents the entire execution of `sphinx` using `ref` input. Comparison between Figure 5.9 and 5.10 shows that PMU contention score correctly identifies execution phases that are contentious (e.g cause

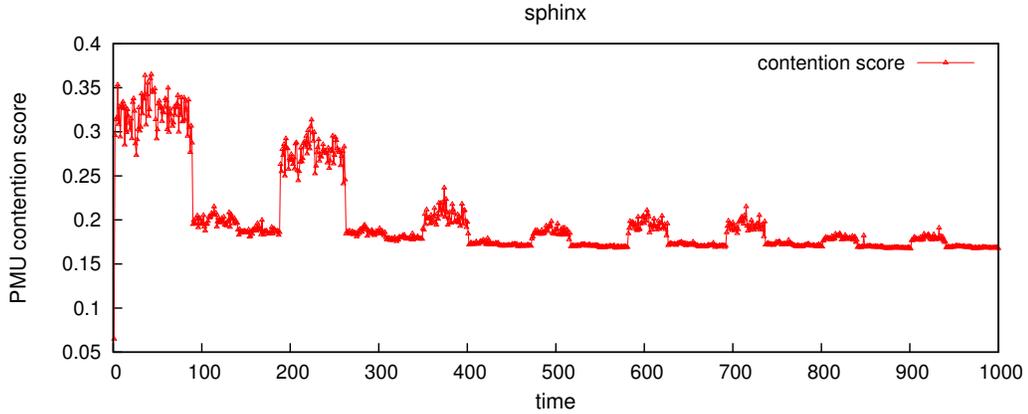


Figure 5.9: Sphinx’s PMU contention score calculated using our prediction model

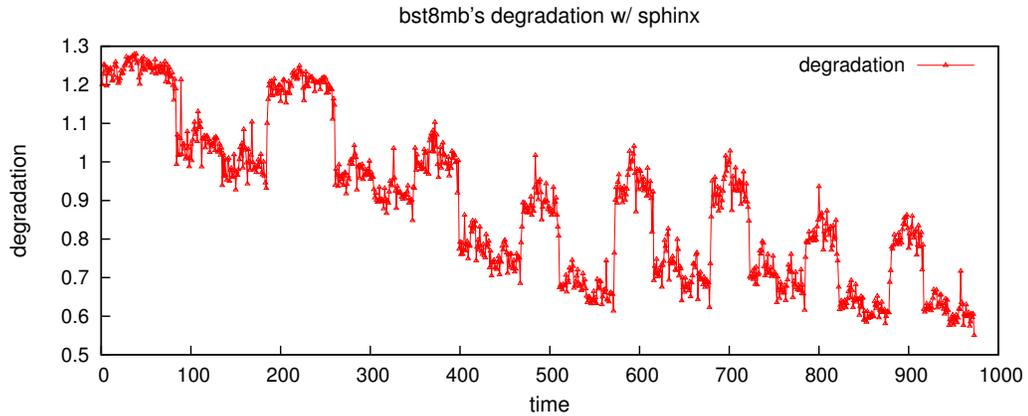


Figure 5.10: Bst8mb’s degradation when running with sphinx. The higher, the more degradation. Figure 7 trends similarly with this figure, indicating the profiler is identifying the correct contentious code regions.

more degradation to a co-runner). The execution phases with higher PMU contention score (humps in Figure 5.9 are consistent with the higher degradation (humps in Figure 5.10).

Similarly, Figure 5.11 presents the results for benchmark milc. The y-axis shows the actual measured slowdown of sledge caused by milc through the entire execution of milc using ref input. Sledge is selected because its performance is stable, which facilitates clear demonstration of the contention phases of milc. The y-axis also presents the phase-level contention score of milc. We overlay these two lines in the figure for better comparison. Note that contention score does not aim to predict the real degradation. Instead, it is designed to indicate the level of contentiousness of various code regions. Figure 5.11 shows that the shapes of two lines match consistently, indicating that the predicted contention score accurately captures the contentious phases of milc.

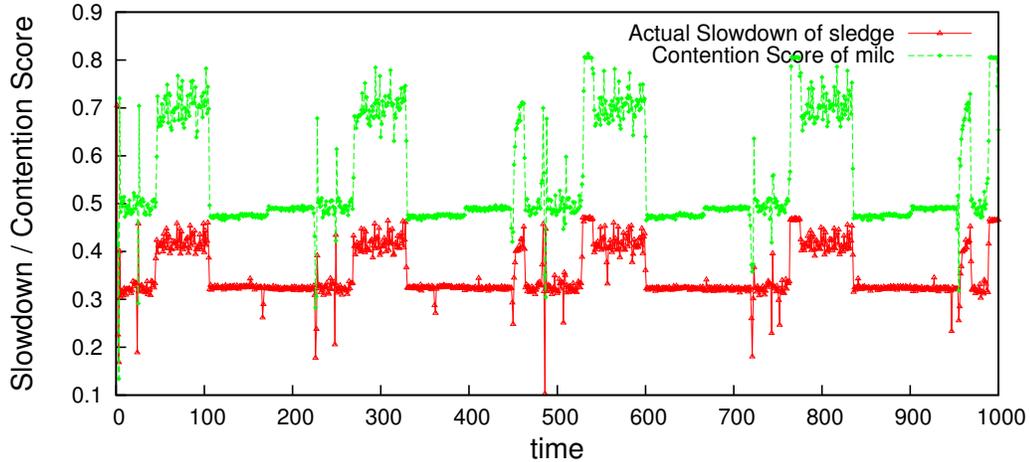


Figure 5.11: This graph shows the accuracy of the contention score given by our prediction model in predicting the contentiousness of milc.

5.4.3 Compiler Transformations

In this section, we evaluate the two transformations used in QoS-Compile, *padding* and *nap insertion*, using the SmashBench suite. This evaluation focuses on the effectiveness of our transformations for improving a co-running application’s QoS. We applied our transformations to the whole program of the contentious kernels without the use of the model to identify specific regions. All experiments in this section were conducted on the Intel Nehalem described in Section 5.4.1.

In Figures 5.12, 5.13, and 5.14 we show the QoS (in terms of execution rate) impact of allowing pairwise co-location of `sledge_1` (`sledge large`) with 6 co-runners when leveraging QoS compile. The dashed line shows the QoS of `sledge_1` and the solid lines shows the QoS of each of the 6 corunners when colocated with `sledge_1`. In these experiments, `sledge_1` is assumed to be our low priority applications while each of its 6 co-runners are assumed to be high priority. The x-axis shows various settings for padding and nap insertion. Figure 5.12 presents the results of applying padding to `sledge_1`, and Figures 5.13 and 5.14 show the results when applying nap insertion to `sledge_1`.

Figure 5.12 shows that, as the padding thickness increases, `sledge`’s execution rate decreases, and the QoS of `blockie` and `bst` improves. For example, when running with the original `sledge_1`, `blockie_1`’s normalized QoS is 0.6x of its solo optimal QoS. After we apply padding to `sledge_1`, `blockie_1`’s QoS is improved to almost 0.9x , which is a

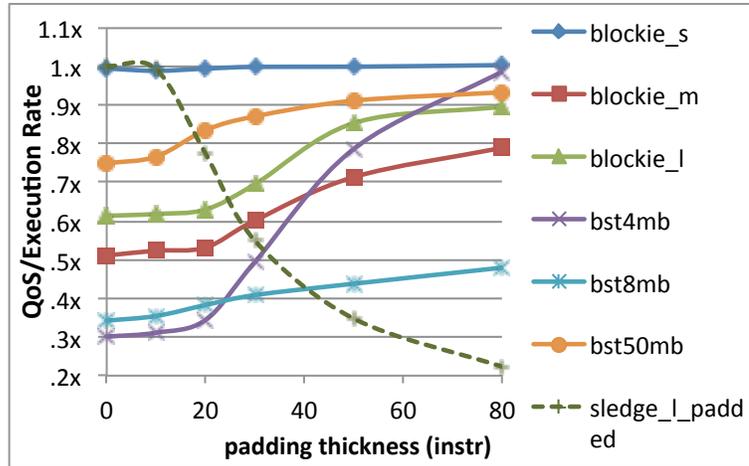


Figure 5.12: Padding `sledge_l`'s effect on its co-runner `blockie` and `bst`. As padding thickness increases, `sledge_l`'s execution rate decreases, `blockie` and `bst`'s QoS improves. The padding granularity is every 5 instructions

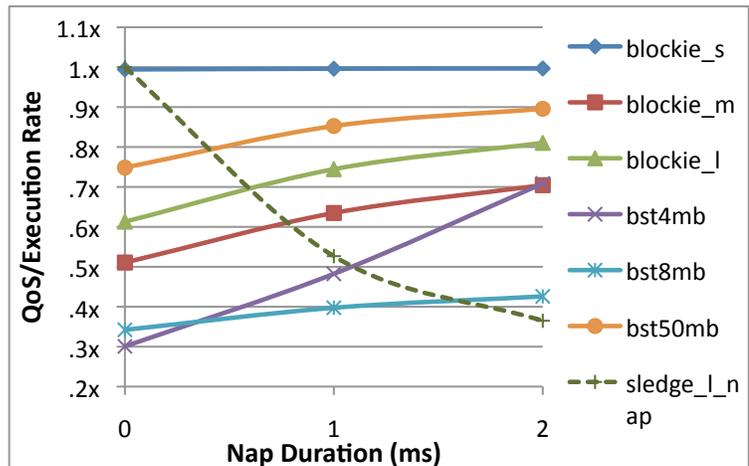


Figure 5.13: Napping `sledge_l`'s effect on co-runners, `blockie` and `bst`. Nap granularity is 1ms. As nap duration increases, `sledge_l`'s execution rate decreases, `blockie` and `bst`'s QoS improves.

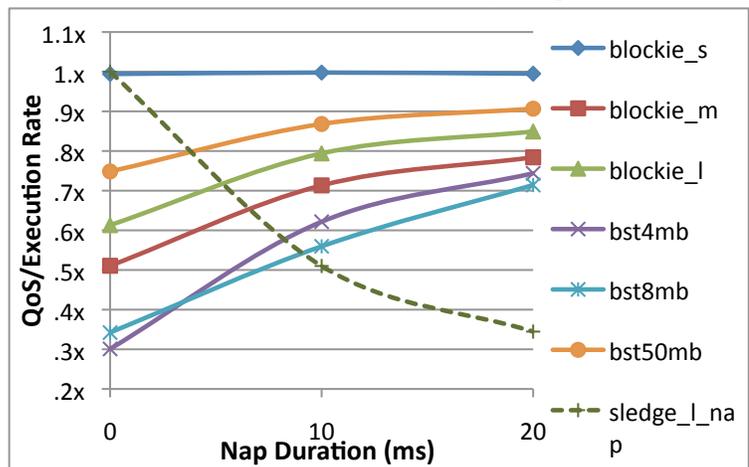


Figure 5.14: Napping `sledge_l`'s effect on co-runners. Nap granularity is 10ms.

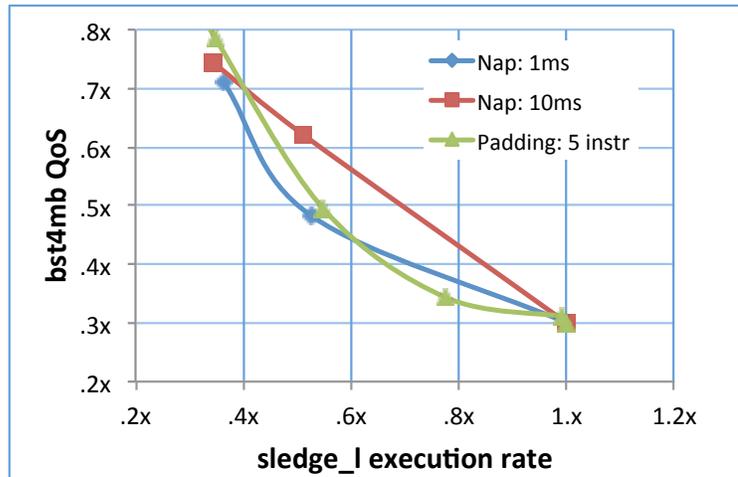


Figure 5.15: sledge_l padding vs. nap for bst4mb

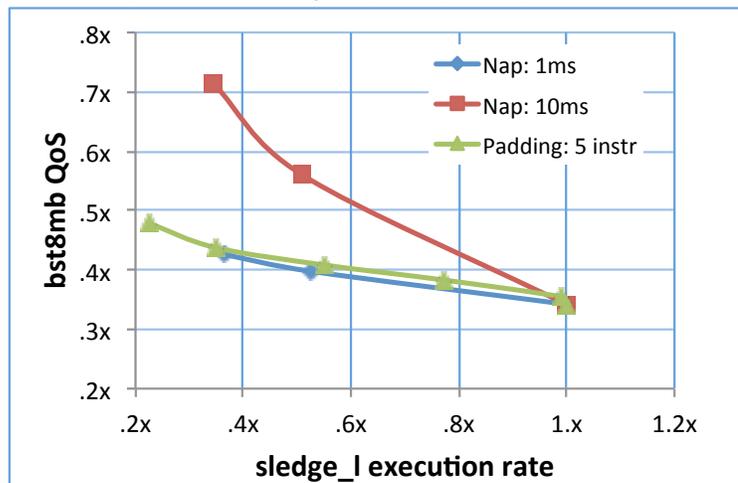


Figure 5.16: sledge_l padding vs. nap for bst8mb

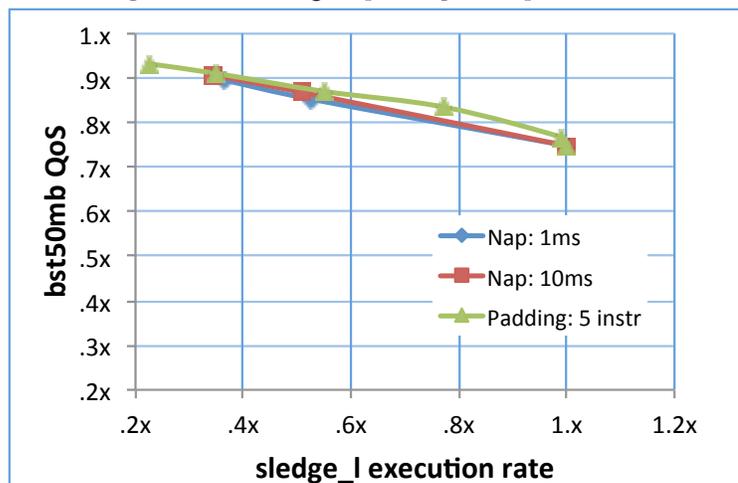


Figure 5.17: sledge_l padding vs. nap for bst50mb.

50% improvement. An interesting observation is that the amount of improvement is not the same for various co-runners. For example, `bst8mb`'s normalized QoS when running with the original `sledge_1` is 0.35x, almost 3 times slower than when it is running alone. However after applying padding, its QoS is only improved to 0.5x. Another interesting observation is that the amount of interference reduction and QoS improvement slows down as padding thickness increases. The improvement is more significant around padding thickness 30 to 50, but for some benchmarks the improvement plateaus after 50. This indicates a potentially diminishing return of increasing padding thickness beyond a certain point.

Figures 5.13 and 5.14 show the results when applying nap insertion to `sledge_1`. The difference between these two figures is the napping granularity. Figure 5.13's granularity is 1ms, meaning that nap is inserted every 1ms of the execution. The x-axis shows the nap duration, ranging from no nap at all to 2 ms nap every 1ms of execution. Figure 5.14 shows the results when the nap granularity is 10ms. These figures demonstrate the effectiveness of nap insertion: as nap duration increases, co-runner's QoS improves. Comparing Figure 5.13 and Figure 5.14 also demonstrates the impact of the nap granularity. Interestingly, napping every 10ms performs significantly better than napping every 1ms for several co-runners. For example, for `bst8mb`, when running with `sledge_1_nap_10ms_20ms` (nap 10 ms every 20ms), its normalized QoS is above 0.7x of its solo optimal QoS, compared to only 0.5x when it is running with `sledge_1_nap_1ms_2ms`. This improvement is consistent with the *cooldown and warmup effect*.

Figures 5.15, 5.16 and 5.17 further illustrate the different impact of padding and nap with various configurations. In each figure, the x-axis shows the `sledge_1`'s normalized execution rate. The y-axis shows its co-runners' normalized QoS. In each figure, we plot three lines showing the effect of three compilation techniques, padding, `nap_1ms` and `nap_10ms`. From these figures we can compare, with the same reduced execution rate for `sledge_1`, which technique achieves the best QoS improvement. Figures 5.15 and 5.17 show that nap and padding perform similarly for `bst4mb` and `bst50mb` as the three lines are very close to each other. However, Figure 5.16 shows that `nap_10ms` performs significantly better than the other two. For example, when `sledge_1` is running at 0.4x (40% of its original execution speed), `nap_10ms` improves `bst8mb`'s QoS to 0.65x compared to only 0.4x for both

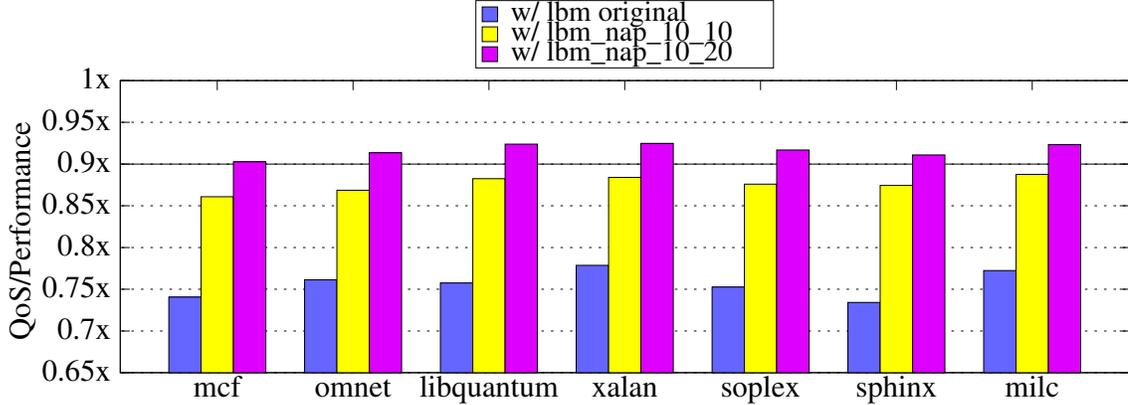


Figure 5.18: SPEC benchmark’s performance when it is co-located with the original lbm, lbm with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark’s performance when it is running alone

padding and nap_1ms. This result is consistent with the *cooldown and warmup* discussion in Section 5.3.2. Longer padding or napping granularity allows co-runners to warm up the cache and achieve better QoS performance. Since the experimental platform has a 8MB last level cache, among `bst4mb`, `bst8mb` and `bst50mb`, `bst8mb` is the most cache contentious benchmark, and therefore benefit the most from longer nap granularity. We also observe similar results when applying padding and nap insertion to other synthetic benchmarks, which are not shown here.

5.4.4 QoS-Compile: Put it All Together

In this section, we evaluate QoS-Compile, the combination of profiling to identify contentious code regions and compilation techniques to dampen contentiousness and improve the QoS of co-runners. The goal of this evaluation is to study the effectiveness of QoS-Compile in 1) improving the QoS of high priority applications when running with low priority applications; and 2) improving machine utilization, meaning that the low priority applications can still reasonably utilize the machine under the constraints of maintaining the QoS of high priority applications at a satisfactory level. We conduct this series of experiments using 8 memory-intensive benchmarks from SPEC CPU 2006 on the Intel Nehalem described in Section 5.4.1. Our evaluation in Section 5.4.3 shows that nap insertion performs better than padding. As such, we focus on nap insertion in this section.

[**Application level**] For each benchmark, we first profiled to sample its PMUs and

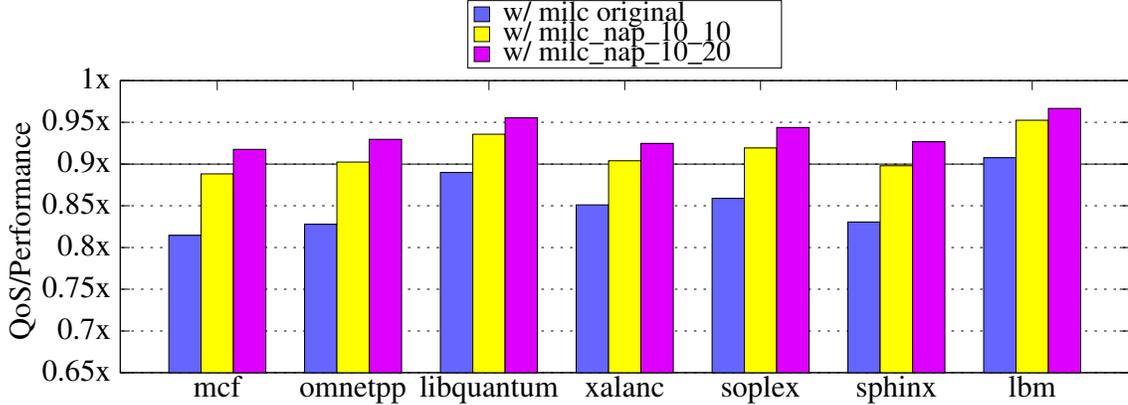


Figure 5.19: SPEC benchmark’s performance when it is co-located with the original milc, milc with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark’s performance when it is running alone

calculated its contention score using our PMU model (Equation 5.9). We then identified its code regions (basic blocks) with contention scores that are above a specified threshold. In our experimentation, we used 0.3 as the threshold. We conducted nap insertion to those basic blocks using the algorithm presented in Section 5.3.2. To evaluate QoS-Compile’s effectiveness, we conducted pair-wise co-run experiments to co-locate a benchmark, presumed to be our low priority application, with 7 other benchmarks, presumed to be the high priority application, and measured the QoS degradation due to its interference.

Figures 5.18 and 5.19 present results for lbm and milc. Figure 5.18 shows the normalized performance of each SPEC benchmark when it is running with lbm. The x-axis shows each benchmark presumed to be the high priority co-runner. The y-axis shows its normalized performance. The higher the bars, the better. For each co-runner benchmark, a cluster of three bars show its performance when it is running with lbm, with lbm_10_10 (lbm is napping 10ms every 10 ms) and with lbm_10_20, normalized by its performance when it is running alone. These 7 co-runner benchmarks are the memory-intensive SPEC benchmarks. We did not present results for other CPU bound SPEC benchmarks because in general they do not suffer degradation from memory resource contention. These figures demonstrate the effectiveness of QoS-Compile. QoS-Compile greatly improves lbm’s “niceness”: reducing lbm’s interference to its co-runner and improving co-runner’s QoS performance. For example, mcf’s QoS is improved 22%, from only 0.74x of its solo optimal QoS when it is running with the original lbm to above 0.9x of the optimal when it is running with the napping lbm.

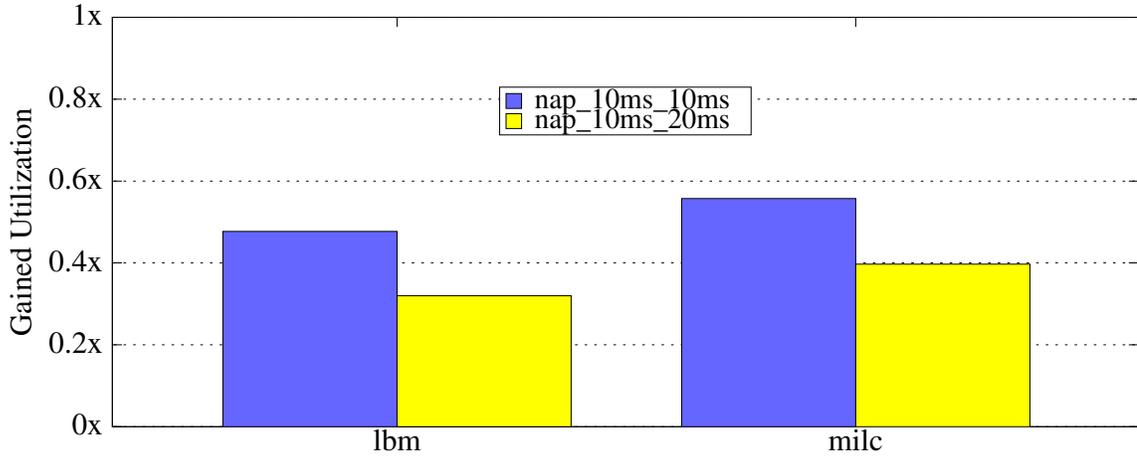


Figure 5.20: Gained Utilization when allow co-location.

In general, every benchmark’s QoS when running with `lbm_10_20` is above 90% of the solo optimal QoS. Figure 5.19 presents similar results for `milc`.

Because QoS-Compile can greatly improve QoS, it provides opportunities for warehouse scale computers to allow co-location knowing that using QoS-Compile, the QoS degradation of the co-located high priority application would be within an acceptable threshold (10%, for example). Figure 5.20 shows the gained machine utilization when allowing co-location facilitated by QoS-Compile. Utilization is measured using `lbm_nap`’s normalized performance (execution rate normalized by the original `lbm` performance when it is running alone). For example, 48% gained utilization for napping `lbm_10_10` indicates that `lbm` is running at 48% of its original execution rate. That is, as opposed to disallowing co-location to ensure the QoS of the high priority application, using QoS-Compile, we allow 48% additional computation while protecting the QoS of its co-runner.

As we mentioned previously in Chapter 1, without QoS-Compile, WSC operators currently have only two options, either allow co-location and suffer a significant QoS penalty or disallow co-location and suffer a utilization penalty. As these figures together demonstrate, QoS-Compile allows users to trade a small amount of QoS to improve machine utilization. In this experiment, we allow 10% QoS degradation, and in return, gain 40% of utilization of the extra otherwise idle core. Changing the `nap` granularity and `nap` interval provides a knob that can be used to tune the tradeoff between QoS degradation and the amount of utilization gained. The more QoS degradation headroom, the more utilization.

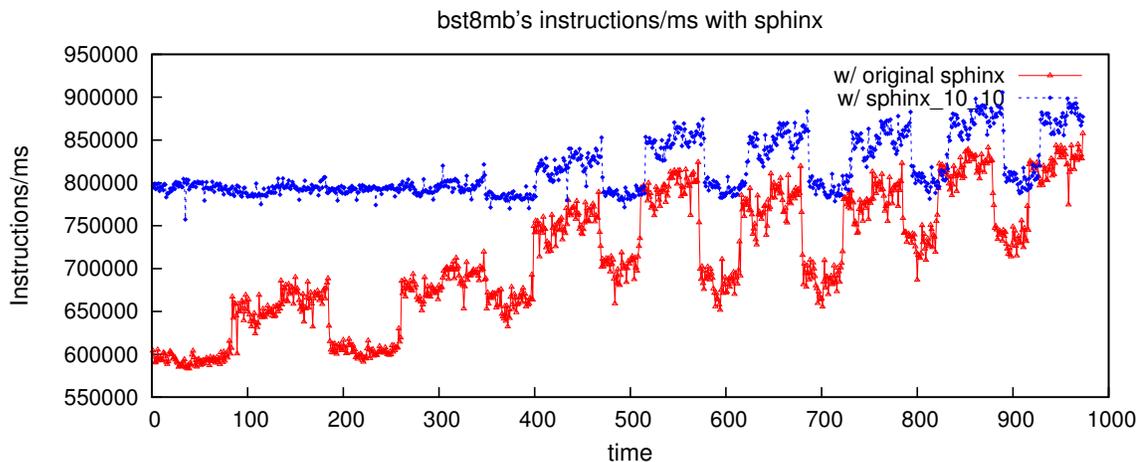


Figure 5.21: bst8mb running with sphinx

workload	description	metric
websearch	Websearch scoring and retrieval	(QPS) queries per sec
cluster-docs	Unsupervised Bayesian clustering tool to take keywords or text documents and "explain" them with meaningful clusters.	throughput
cluster-keywords	Unsupervised Bayesian clustering tool to take keywords or text documents and "explain" them with meaningful clusters.	throughput
goog-retrieval	Web indexing	query latency (ms)
maps-detect-face	Face detection for streetview automatic face blurring	user time (secs)
maps-detect-lp	OCR and text extraction from streetview	user time (secs)
maps-stitch	Image stitching for streetview	user time (secs)

Table 5.3: Production Warehouse Scale Computer Applications

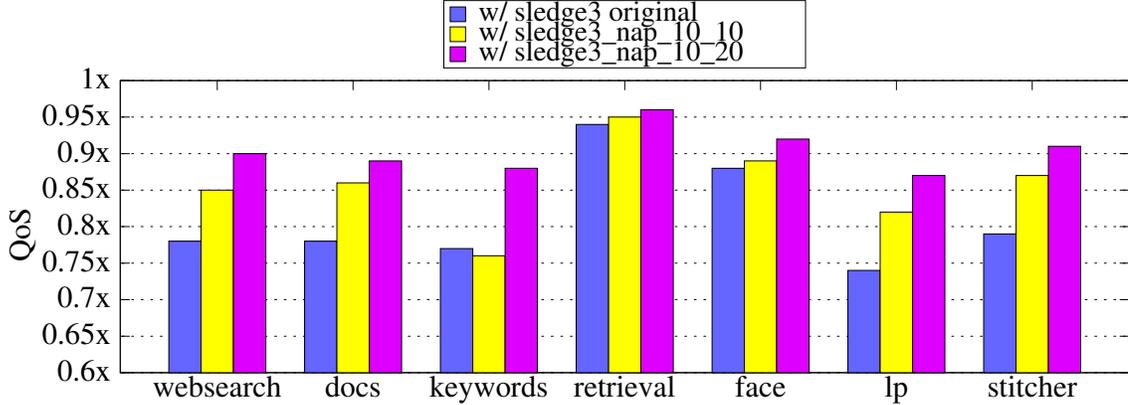


Figure 5.22: Google benchmark’s performance when it is co-located with the original sledge3, sledge3 with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark’s performance when it is running alone

[Phase level] QoS-Compile not only reduces the overall average QoS degradation, it also pinpoints the contentious regions and mitigates the QoS degradation those regions can cause when executing. This makes QoS-Compile also suitable for applications that only have phases of contention. To further evaluate QoS-Compile’s effectiveness in pinpointing and managing the contentious phases, we sample the performance of co-runners throughout the entire execution to observe their performance variability due to interference. Figure 5.21 presents `bst8mb`’s performance (instructions/ms) when it is running with the original `sphinx`, compared to its performance when running with napping `sphinx` (`sphinx_10_10`, napping 10ms every 10ms). The x-axis shows time. We sample the entire execution of `sphinx` with `ref` input. The y-axis is `bst8mb`’s performance. `Bst8mb` is a contentious kernel and when it is running alone it has quite stable performance. Therefore the performance variability shown in the figure is purely due to interference from `sphinx`. As the figure shows, during the early half of the execution, original `sphinx` causes significant performance degradation to `bst8mb`, demonstrated by the low IPS during the first 400 samples. QoS-Compile correctly identifies the contentious phase and improves the `bst8mb`’s IPS greatly. For the later half of the execution, the QoS-Compile also identifies `bst8mb`’s performance valleys and improves it greatly.

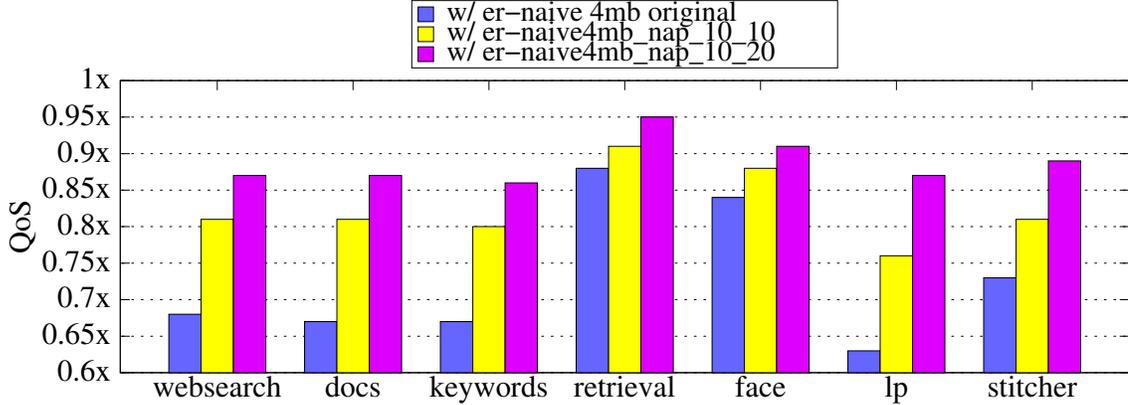


Figure 5.23: Google benchmark’s performance when it is co-located with the original er-naive4mb, er-naive4mb with nap insertion (10ms, 10ms) and nap insertion (10ms, 20ms), normalized by each benchmark’s performance when it is running alone

5.4.5 Google Applications

To evaluate our compilation technique’s effectiveness in improving co-runner’s QoS, we also conducted experiments using several large-scale warehouse scale computer applications. The experimental platform is an Intel Clovertown machine used in production (as described in Section 5.4.1). The production applications are presented in Table 5.3. The QoS metric for each application is the application-specific performance metric in its internal SLA, also presented in Table 5.3. The load for each application is a trace of large amount of real world queries in production WSCs. A load generator was set up to feed the queries to these applications. The performance shown is applications’ stable behavior after the initialization phase, and the performance is stable between runs. Figure 5.22 and Figure 5.23 present results. In these experiments, each Google application is co-located with 2 threads of Smash-Bench benchmarks. Figure 5.22 presents Google applications’ QoS when co-located with `sledge.l`. The x-axis shows each Google application. And the y-axis is each application’s normalized performance. Each application’s QoS are measured in 3 running scenarios presented as a cluster of three bars: when it is co-located with 2 threads of original `sledge.l`, with 2 threads of napping `sledge` that naps 10ms every 10ms, and with `sledge` that naps 20ms every 10ms. Each application’s QoS performance is normalized to its performance when it is running alone. Figure 5.23 presents Google application’s QoS performance when co-located with a cache contentious benchmark, `er-naive4mb`. Figure 5.22 and Figure 5.23

demonstrate that nap insertion is effective in improving an application’s “niceness” and improving its co-running Google applications’ QoS. For example, nap insertion improves `websearch`’s QoS from 0.77x to 0.9x when running with `sledge_1`, and from 0.68x to 0.87x when running with `er-naive4mb`. QoS-Compile can improve QoS significantly and provides warehouse scale computer operators with flexibility of allowing co-location with a slight hit on QoS. For example, if warehouse scale computer scheduler specifies that 0.9x of the optimal peak QoS is an acceptable threshold for `websearch`, with QoS-Compile, we can allow co-location of `websearch` with other co-runner such as `sledge_1` to improve the machine utilization. Without QoS-Compile, 0.65x of its solo QoS when running with the original `sledge_1` may be too significant to allow co-location, and thus leaving the machine under-utilized.

5.5 Summary

In this chapter, we have presented QoS-Compile, the first compilation approach that statically manipulates application contentiousness to enable the co-location of applications with varying QoS requirements, and as a result, can greatly improve machine utilization. Using a novel prediction model, QoS-Compile first pinpoints an application’s contentious code regions that tend to cause performance interference. QoS-Compile then transforms those regions to reduce their contentious level. In this work we have shown that binary code transformations to throttle down the execution rate and the memory access rate of the contentious regions in low priority applications is an effective approach to reduce their interference to high priority applications. Through our experimentation, we find that QoS-Compile improves applications’ QoS performance by 21% and machine utilization 36% on average. In the era of multicores and the emerging computing domain of WSCs, the objectives of compiler optimization ought to be multifaceted. In this work, we argue for the additional objective of optimizing for an application’s “niceness”, to reduce its potential interference to its co-running applications.

Chapter 6

Reactive Niceness

Contents

6.1	Reactive-Niceness Overview	85
6.2	RN-Compile: Compiling for Reactive Niceness	88
6.3	RN-Runtime: Dynamic Detection and Reaction to QoS Degradation	89
6.3.1	Runtime	90
6.3.2	Detection and Reaction	92
6.4	Evaluation	95
6.4.1	Setup and Methodology	96
6.4.2	Effectiveness of Reactive-Niceness: Simple Heuristic	96
6.4.3	Effectiveness of Reactive-Niceness: Targeted Heuristic	99
6.4.4	Effectiveness of Reactive-Niceness: Phase Level Behavior	101
6.4.5	Overhead	105
6.4.6	Energy Efficiency of using Reactive-Niceness	106
6.4.7	Varying Architecture	107
6.5	Summary	109

Chapter 5 presents a compilation approach, QoS-Compile, for statically manipulating an application's contention characteristics to reduce the performance interference it may cause to corunning applications and ultimately facilitate workload consolidation and improve server utilization. Essentially, QoS-Compile is a conservative approach. It throttles down the execution of an application's contentious regions, regardless of whether the QoS of the

corunning high priority application actually suffers from performance interference or not. In this chapter we present a statically enabled dynamic approach, **Reactive-Niceness**, to enable the adaptive manipulation of the contentiousness of low-priority applications to ensure the QoS of high-priority co-runners. Reactive-Niceness monitors the QoS degradation of the high-priority applications online and diagnoses whether resource contention among applications is the root cause of the degradation. If so, it prescribes the necessary amount of throttling down dynamically and by doing so, reducing the QoS degradation of high-priority application. The biggest advantage of this online approach is its dynamic detection and reaction, which helps achieve better server utilization and more accurate QoS control than the static approach. Using Reactive-Niceness on SPEC2006 and SmashBench workloads, we are able to improve utilization by more than 70% in many cases, and more than 50% on average, while enforcing a 90% QoS threshold. We also improve the energy efficiency of modern multicore machines by 47% on average over a policy of disallowing co-locations.

6.1 Reactive-Niceness Overview

Reactive-Niceness provides a software mechanism that automatically and adaptively regulates the pressure that a low-priority *batch* application applies to shared memory subsystem resources to ensure the QoS of high-priority *latency-sensitive* application. One key insight of Reactive-Niceness is that a dynamic approach is needed to effectively detect contention at runtime and reactively adjust the “niceness” of a low priority application only when contention with high-priority co-runners is occurring. This reactive “niceness” enables the flexibility needed to further improve machine utilization and more accurately manage QoS.

Reactive-Niceness combines both static compilation and dynamic adaptation. Reactive-Niceness first uses a profile guided compilation approach to identify the code regions in low-priority applications that aggressively demand memory resources and may cause resource contention, and instruments those regions to enable the flexible manipulation of their contentiousness. The profiling and static compilation enable the dynamic engine to manipulate the execution of the low-priority application. They also assist in the diagnosis of contention and trigger the runtime system only during phases when problematic code

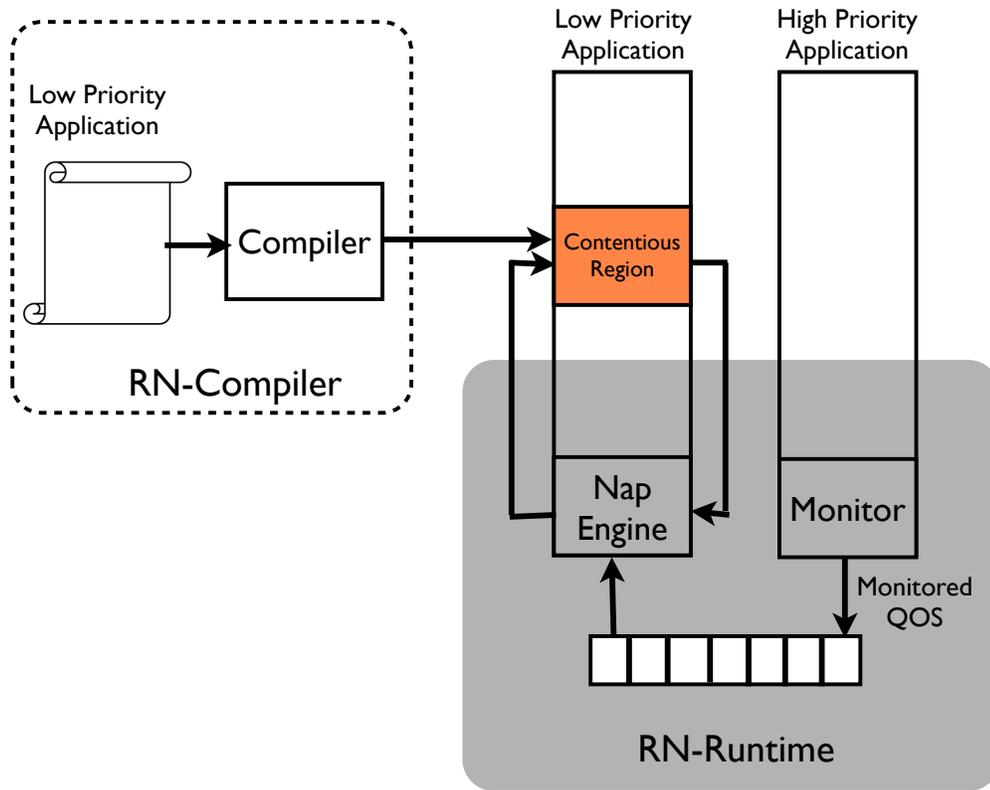


Figure 6.1: Reactive-Niceness Overview

regions are executing.

At runtime, Reactive-Niceness dynamically detects contention-caused QoS degradation and adaptively throttles down the execution rate and memory request rate of those contentious regions in the low-priority application. The particular high priority application a low priority application will be co-located with is not known when it is compiled. Therefore, a runtime approach that can dynamically adjust its execution rate based on its interference to the corunner is especially desirable. The degree of execution rate reduction on low-priority applications is based on the severity of observed QoS degradation of the high-priority application, allowing for more drastic responses to higher levels of contention. As contention lessens dynamically, the execution rate of the low-priority application is then increased to maximize machine utilization. The dynamic execution rate manipulation facilitates “safe” colocation; cores that would otherwise be idle to avoid the unpredictable and potentially significant QoS degradation are now utilized.

Reactive-Niceness consists of two components, RN-Compiler and RN-Runtime, as shown

in Figure 6.1.

[RN-Compiler] The RN-Compiler is a static profile-driven compiler approach that uses a performance counter based profiling analysis to identify contentious code regions and insert markers on those regions to steer the runtime adaptation. The profiling analysis used to identify contentious code regions is similar to the analysis in QoS-Compile, presented in Section 5.2.2. As shown in Figure 6.1, these inserted markers trigger the RN-Runtime, via the *Nap Engine*, when contentious code regions are executed. These triggers call upon the runtime to directly manipulate the rate of memory accesses generated by the low-priority application through the Nap Engine interface. The binaries produced by the RN-Compiler can also be run without the RN-Runtime. In this case, the inserted markers are benign, and the application runs as normal. The overhead of having these markers present in the binary are minimal, and a full evaluation of these overheads is presented in Section 6.4.

The advantage of a profiling guided approach is to pinpoint the potentially problematic code regions for better dynamic contention detection. Dynamically detecting resource contention is quite challenging for system software. Purely relying on the dynamic observation of the QoS degradation may lead to false positives for contention detection, as contention may not be the only reason for QoS degradation. The profiling guided approach facilitates an more effective and low-overhead contention detection. The RN-Compiler is described in more detail in Section 6.2.

[RN-Runtime] The RN-Runtime is responsible for monitoring the QoS of high-priority applications, detecting when a low-priority application is interfering with the performance of the high-priority application, and dynamically deciding the degree of memory access rate reduction to apply to alleviate the performance interference. As shown in Figure 6.1, a lightweight dynamic runtime that monitors application QoS is attached to the high priority application. This runtime periodically reports an application’s QoS through a shared memory buffer. The Nap Engine that is attached to the application binary of the low-priority application reads the most recent QoS reports from this buffer to steer the online contention response. The RN-Runtime and the adaptive policies are described in detail in Section 6.3.

[Using Reactive-Niceness in a Modern WSC] Figure 6.1 also illustrates how Reactive-Niceness is used in the context of a WSC. All low-priority applications in the

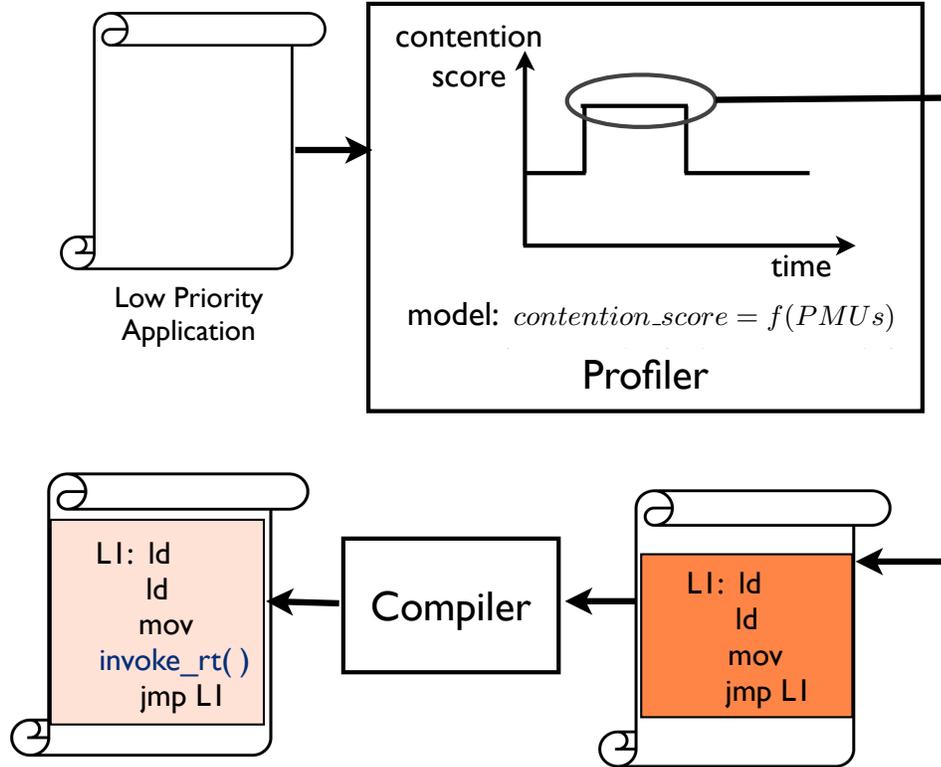


Figure 6.2: Reactive-Niceness Compilation

WSC are compiled with a flag denoting that it is a low-priority application. The applications are then compatible for execution with Reactive-Niceness enabled. When these applications are scheduled to co-run with a high-priority application, QoS monitoring is turned on, and the Nap Engine enacts the adaptation policy.

6.2 RN-Compile: Compiling for Reactive Niceness

In this section we present RN-Compile, our static compilation to enable dynamic contention mitigation and QoS improvement at runtime. The RN-Compile process is illustrated in Figure 6.2. To compile a low-priority application, we first identify its contentious code regions using a profiler that scores code regions as they execute. We then insert markers in those regions that periodically invoke the RN-Runtime. Because markers target the problematic regions, the runtime engine is only triggered when the contentious regions are executing.

Our approach to identifying the contentious code regions is fairly straightforward and is based on the prediction model using performance counters presented in Section 5.2.2. During profiling, performance counters (L2 and L3 cache lines in rate) are sampled every 1 ms and the contention score is calculated using Equation 5.9. To correlate the contention score to the corresponding static code regions, the number of instructions retired in each 1 ms execution interval is also sampled and recorded. After the profiling run, a PIN [33] tool is used to replay the execution. Based on the recorded instruction profile, our PIN tool identifies the hottest basic blocks that are executed during each 1 ms execution interval and assigns the corresponding contention score to these basic blocks. The PIN tool then selects the basic blocks with high contention score.

After these highly contentious basic blocks are identified, instead of applying compilation transformations to these regions as in QoS-Compile (Section 5.3), we instrument markers, *invoke_rt()*, to the contentious code, shown in Figure 6.2. At runtime, these markers invoke the RN-Runtime to dynamically decide the throttling policy. To minimize the potential overhead of frequent calls to the runtime, we have implemented a number of optimizations. Most notably, we use a self-tuning global checker that allows the call to the runtime to be executed only after a sufficient execution iterations of the same basic block. This is especially helpful when a large number of markers are inserted in the critical path of execution. Instead of executing the function call every time, an increment and compare is executed in the average case.

6.3 RN-Runtime: Dynamic Detection and Reaction to QoS Degradation

In this section, we present RN-Runtime, our runtime engine that dynamically detects the QoS degradation of high priority applications due to resource contention, and adaptively manipulates the contentiousness of low-priority applications to mitigate QoS degradation.

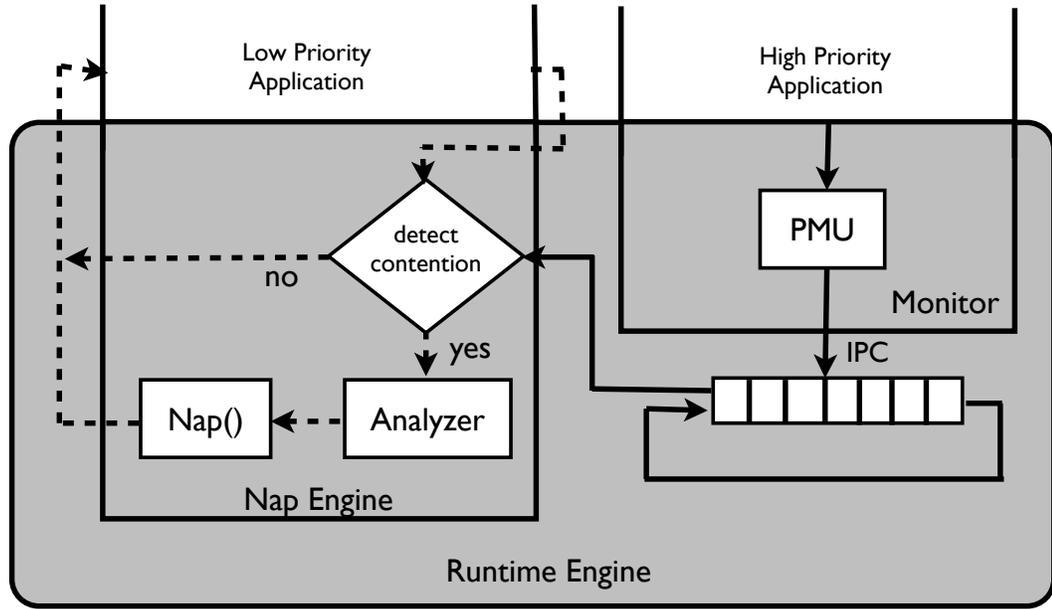


Figure 6.3: Reactive-Niceness Runtime Architecture

6.3.1 Runtime

Figure 6.3 illustrates the design for RN-Runtime. The runtime engine is composed of two main components: *Monitor* and *Nap Engine*. In our implementation, the Nap Engine is linked into the low-priority application and the Monitor is either linked into, or attached to the PID of, the high-priority application. The Nap Engine and the Monitor communicate through a shared memory buffer.

[Monitor] The Monitor is responsible for monitoring the QoS of high priority applications. In this design we use instruction-per-cycle (IPC) as a proxy for QoS. The IPC is often used in production datacenters as a QoS proxy because it is readily available using hardware performance counters and can be sampled with little overhead. For example, Google Wide Profiling (GWP) is currently deployed in Google’s fleet to collect IPC and other counters for performance monitoring and debugging [47]. The Monitor uses a *periodic probing* technique, leveraging a timer interrupt to sample the hardware performance counters every 1 ms, and storing the recent sequence of IPC samples in a circular buffer in the shared memory. As we show in Section 6.4, this period probing technique incurs a minimal overhead (often less than 1%).

[Nap engine] Based on the monitored QoS, the Nap Engine detects resource contention and QoS degradation, and accordingly reacts by deciding the appropriate execution rate reduction for the low-priority application. The Nap Engine is only invoked by the instrumented markers when the low-priority application is executing the contentious regions. Instead of invoking the Nap Engine every time an instrumented contentious basic block is executing, a timer based on the time stamp register, read using the RDTSC instruction [1], is used in the instrumentation to only yield control from the low-priority application to the Nap Engine periodically (2 ms in our experiments). To further reduce the overhead of timer checking, we also use this timer to adapt the global checker mentioned in Section 6.2 by approximating the amount of runtime invocations to skip before reading the timestamp counter again. This approximation requires a simple calculation based on the time past since the prior invocations and is adaptively adjusted upon every timestamp read. Due in part to these optimizations, the overhead of invoking the Nap Engine is low, never exceeding 5%, and is evaluated in Section 6.4.

When invoked, the Nap Engine’s main tasks are to firstly detect contention and QoS degradation based on the information provided by the Monitor, and secondly if contention is detected, analyzes and decides how to appropriately throttle down the low-priority application to mitigate the degradation. The Nap Engine controls the execution rate of a low-priority application by putting the execution of a contentious code region to epochal intermittent short “nap” mode. Naps reduce the memory request rate and execution rate of the low-priority application and the pressure it puts on the shared memory subsystem. This in turn prioritizes the memory requests of the co-running high-priority applications, and the QoS degradation it suffers due to the resource contention with the low-priority application is greatly reduced or eliminated for the duration of the nap. Two main parameters that affect the behavior and the effectiveness of napping include the *frequency* and the *duration* of naps. The Nap Engine controls these parameters and decides whether and when a nap should occur (essentially how long the low-priority application should execute at a normal rate) and how long of a nap it should take to effectively improve the QoS of the corunning high-priority application. Flexible policies and heuristics for contention detection and reaction can be implemented in RN-Runtime, which are further discussed in the next

section.

6.3.2 Detection and Reaction

In this section, we present two adaptation policies used in RN-Runtime to detect resource contention and QoS degradation, and to reactively control the execution rate of the low-priority application to mitigate contention if necessary. It is challenging to design a software approach to detecting contention as it occurs. This is mostly due to the fact that contention in various hardware components such as shared caches and memory controllers is not exposed to the software. For example, during runtime, information such as the amount of data belonging to the high-priority application that is evicted by a corunning low-priority application from the shared cache is not visible to the software. We design probabilistic empirical approaches to tackling the challenge of dynamic contention detection based on the online monitoring and feedback control. Once contention and QoS degradation are detected, the Nap Engine is also tasked to decide the appropriate rate reduction to apply to the low-priority application to reduce the QoS degradation.

In this work, we design two heuristics for the Nap Engine: `simple` and `targeted`. The `simple` heuristic directly relies on QoS monitoring information of the high-priority application and is designed to provide users with a flexible, tunable “knob” to manage the tradeoffs between QoS and utilization. For example, the heuristic can be configured to prioritize QoS and conservatively reduce utilization or prioritize utilization and risk the QoS. However, the `simple` heuristic does not strive for a strict QoS goal, such as improving the QoS of high priority application to above 90% of its normal QoS when running alone. The `targeted` heuristic on the other hand, is designed to accommodate a pre-specified QoS target. `Targeted` makes the detection based on closely monitoring the impact of throttling of the low-priority application on the QoS and uses an analytical model to adjust the appropriate nap duration adaptively.

[Heuristic 1: Simple] The basic idea of `simple` is to detect and react purely based on the monitored QoS of the high priority application. In our runtime implementation, we use instruction-per-cycle (IPC) as a proxy for QoS and `simple` adjusts the nap duration based on the dynamically monitored IPC. The details of our algorithm are described in

Algorithm 4: Nap_Engine (Heuristic 1: Simple)

Input : $threshold_low, threshold_high, nap_ratio_low, nap_ratio_mid, nap_ratio_high$

```
1  $ipc$  = latest IPC sample from the shared IPC buffer;  
2 if ( $ipc < threshold\_low$ ) then  
3   |  $nap\_duration \leftarrow nap\_ratio\_low \times exec\_duration$  ;  
4 else if ( $ipc < threshold\_high$ ) then  
5   |  $nap\_duration \leftarrow nap\_ratio\_mid \times exec\_duration$  ;  
6 else  
7   |  $nap\_duration \leftarrow nap\_ratio\_high \times exec\_duration$  ;  
8 end  
9  $nap(nap\_duration)$ ;
```

Algorithm 4. When the contentious code region is executing, the Nap Engine is invoked periodically (each `execution duration`). The Nap Engine then reads the latest IPC sample of the high-priority application (HP). Two thresholds (`threshold_low`, `threshold_high`) are used to bucket the monitored IPC into low, medium and high. The nap duration is decided based on which bucket the IPC is in. The lower the IPC, the longer the nap duration. In addition to IPC, application specific performance metrics such as query latency can also be monitored and bucketed. The rationale of this heuristic is that although many factors other than contention may cause QoS degradation (such as load temporal changes), we will conservatively throttle down the low priority application (LP) once the QoS degradation is observed. The parameter configurations (IPC thresholds and nap ratio) decide how QoS-biased (conservative) or utilization-biased (optimistic) the `Simple` heuristic is. The sensitivity of those parameters is discussed in Section 6.4.

[**Heuristic 2: Targeted**] The primary design goal of `targeted` is to adaptively adjust the nap duration to improve the QoS to a user-specified goal (such as a minimum QoS of normalized 90% of a specific QoS target). The basic idea of `targeted` is to detect and react based on measuring how the QoS of high-priority application is effected by naps of low-priority applications. Figure 6.4 illustrates the logic of `targeted`. There are three basic states for LP, which the Nap Engine tracks. Periodically, the nap engine is invoked to analyze the QoS samples of HP and the analysis result triggers potential state transitions.

- **Intermittent nap state.** The `intermittent nap state` indicates that naps are inserted to throttle LP’s execution rate. The QoS of the HP is sampled both when the LP is napping and when the LP later wakes up from the nap and is executing.

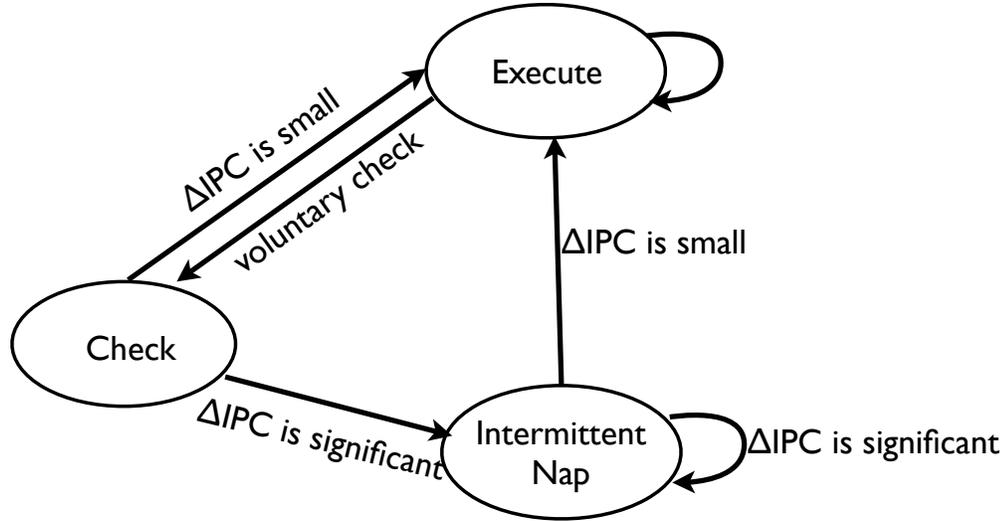


Figure 6.4: DFA for targeted Heuristic

The difference between the two samples, ΔIPC is used to adjust the next nap duration. The bigger the difference, the more significant the impact of contention is, and the longer the nap duration should be. The detailed model of adapting the nap duration is shown later. When the IPC delta is smaller than the pre-specified QoS degradation threshold, it indicates that napping does not have a significant impact. And LP transition to the execution state.

- **Execution state.** In execution state, LP executes at the full rate with no naps inserted. However, LP does not stay in execution state indefinitely. A countdown is set to trigger the transition to voluntary check state after a pre-specified execution period.
- **Check state.** The purpose of the check state is to periodically detect if contention occurs after a period of execution. The detection is similar to intermittent nap state. LP is put to nap for a short interval and then is run for a short interval. The difference of the IPC samples of HP during these two intervals is used to decide if contention is occurring. If so, LP transitions to the intermittent nap state; if not, the execution state.

The algorithm for targeted heuristics is described in Algorithm 5. A parameter conservative factor is used to guard how close the monitored QoS degradation is to the pre-specified

threshold ($\mathit{delta_IPC}$, for example) before the napping is used to throttle down the LP.

In Algorithm 5, we estimate the appropriate nap duration based on the QoS goal (the degradation threshold QoS_{thresh} , such as 90% of optimal QoS) and the observed difference between IPC of HP when LP is napping (IPC_{nap}) and when LP is executing (IPC_{exec}). To estimate the appropriate nap duration we solve the following equation:

$$\frac{QoS_{thresh}}{1 - QoS_{thresh}} = \frac{IPC_{nap} - IPC_{exec}}{IPC_{exec}} \times \frac{exec_duration}{exec_duration + nap_duration} \quad (6.1)$$

where $exec_duration$ is the duration of the execution interval between inserted naps.

Algorithm 5: Nap_Engine (Heuristic 2: Targeted)

```

Input :  $QoS\_goal, conservative\_factor, execute\_period$ 
1 if ( $LP\_state == execution\_state \ \&\& \ execute\_countdown > 0$ ) then
2   |  $execute\_countdown --$ ;
3   | return; /* no napping, running ahead */;
4 else if ( $LP\_state == execution\_state \ \&\& \ execute\_countdown = 0$ ) then
5   |  $nap(check\_interval)$ ;
6   |  $ipc\_nap \leftarrow$  read the average IPC of HP when LP is the last napping duration from the shared
   | IPC buffer;
7   |  $LP\_state \leftarrow check\_state$ ; /* voluntarily nap for a short period to test if contention
   | is back by checking  $\mathit{delta\_IPC}$  */;
8 else if ( $LP\_state == nap\_state \ || \ LP\_state == check\_state$ ) then
9   |  $ipc\_exec \leftarrow$  read the average IPC of HP when LP is the execution interval from the shared IPC
   | buffer;
10  |  $\mathit{delta\_ipc} \leftarrow (ipc\_nap - ipc\_exec)/ipc\_nap$ ;
11  | if ( $\mathit{delta\_ipc} < conservative\_factor * QoS\_goal$ ) then
12  |   |  $execute\_countdown \leftarrow execute\_period$ ;
13  |   |  $LP\_state \leftarrow execution\_state$ ;
14  |   | return; /* significant contention is not detected nap does not seem to have a
   | big enough effect on IPC */;
15  | else
16  |   |  $nap\_duration \leftarrow calculate\_duration(\mathit{delta\_ipc}, QoS\_goal)$ ;
17  |   |  $nap(nap\_duration)$ ;
18  |   |  $ipc\_nap \leftarrow$  read the average IPC of HP when LP is the last napping duration from the
   | shared IPC buffer;
19  |   |  $LP\_state \leftarrow nap\_state$ ;
20  | end
21 end

```

6.4 Evaluation

We use the same prediction model as in QoS-Compile, which is evaluated in Section 5.4. So in this section, we focus on evaluating the effectiveness of both heuristics in controlling the

Configurations	thresh_low	thresh_high	nap_ratio
util_biased	0.5	1.0	{0, 1, 2}
balanced	0.8	1.5	{0, 1, 2}
QoS_biased	0.5	1.0	{1, 2, 3}

Table 6.1: Three configurations for `simple` heuristic

QoS degradation that results from resource contention and in improving server utilization. We also take a comprehensive look into the dynamic behavior of Reactive-Niceness and its reaction to contentious phases throughout the execution. Lastly, we evaluate the overhead and power efficiency of Reactive-Niceness.

6.4.1 Setup and Methodology

Our evaluation is conducted on a 2.67 GHz Quad Core *Intel Nehalem* processor, described in Section 5.4.1 with an 8MB last level cache (L3) shared by four cores with 4GBs of main memory. This platform runs Linux 2.6.29.6 and a customized GCC 4.4.6. The workloads used in our evaluation include the `sledge` application from the **SmashBench** contentious kernel suite [37, 38] (developed at Google, summarized in Table 5.1), and applications from SPEC CPU2006. All benchmarks are compiled using GCC at the O2 level. All SPEC applications are run using `ref` inputs. In our evaluations in Section 5.4, we have shown that SPEC 2006 and Google applications have similar amounts of performance degradation due to contention. In addition, throttling down low-priority applications at millisecond granularity has similar effect on improving the QoS of a high-priority SPEC or Google application. Due to the lack of workload access during our experimentation, in this section we use SPEC as our main experimental benchmark suite. Each experiment is conducted three times to calculate the average performance. Benchmark runs are fairly stable with a performance variance of 1% or less between runs.

6.4.2 Effectiveness of Reactive-Niceness: Simple Heuristic

As mentioned in Section 6.3.2, our `simple` heuristic provides “knobs” that control whether the emphasis of Reactive-Niceness is biased towards QoS or machine utilization. Table 6.1 presents the three configurations we use in our evaluation. These include `util_bias`,

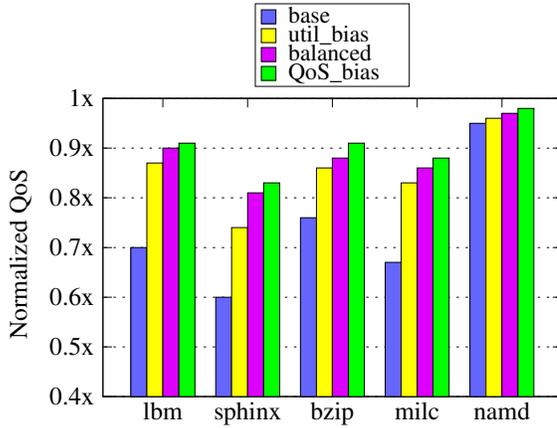


Figure 6.5: QoS of each benchmark co-running with sledge, normalized to solo QoS. (simple)

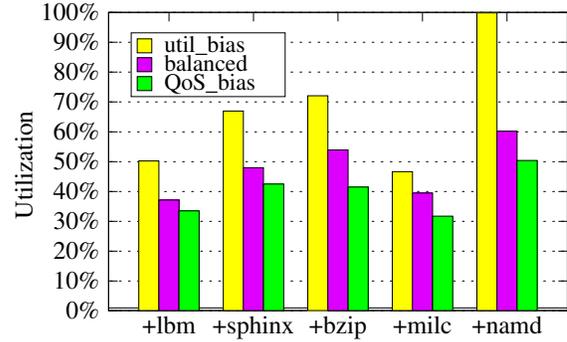


Figure 6.6: Utilization of sledge with each configuration. (simple)

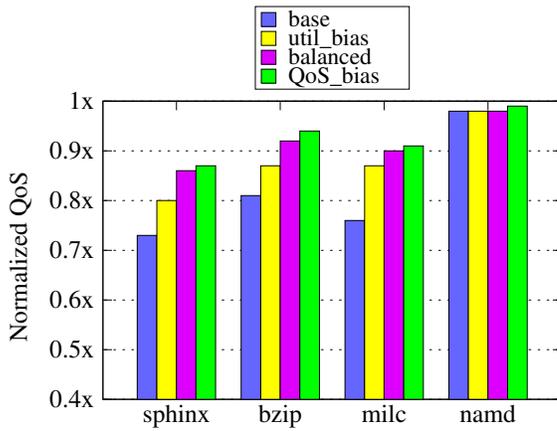


Figure 6.7: QoS of each benchmark co-running with lbm. (simple)

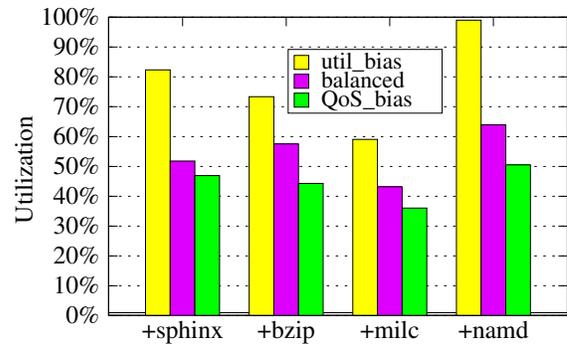


Figure 6.8: Utilization of lbm with each configuration. (simple)

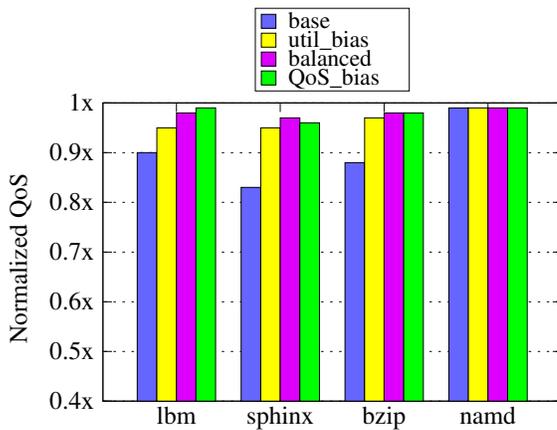


Figure 6.9: QoS of each benchmark co-running with milc. (simple)

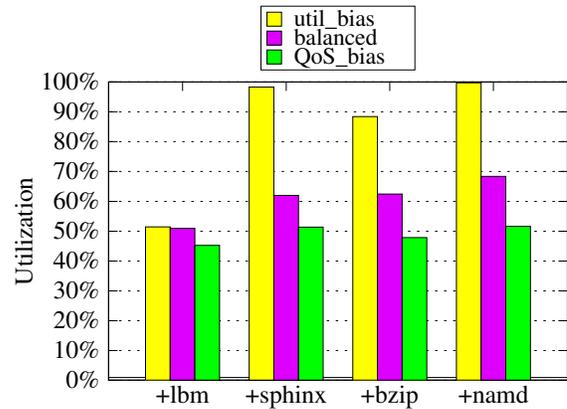


Figure 6.10: Utilization of milc with each configuration. (simple)

balanced, and `QoS.bias`, representing an emphasis on higher utilization, a balance between utilization and QoS, and higher QoS respectively. `Threshold_low`, `threshold_high` and `nap_ratio` are parameters for Algorithm 4 to control, respectively, the binning of monitored instructions-per-cycle (IPC) of the high-priority application and the nap duration of the low-priority application. In general the longer the nap ratio, the more throttling down the heuristic applies to the low priority applications, and the more biased the heuristic is towards the QoS of high priority applications.

Figures 6.5, 6.7 and 6.9 present the QoS of the high-priority application when we apply Reactive-Niceness to a low priority application with three configurations of `simple` heuristic. In each of these figures, the x-axis shows the high-priority applications and the y-axis shows their QoS when each of them is co-running with a low-priority application, normalized to its QoS performance when running alone on the machine. For each high-priority application, a cluster of four bars demonstrates four settings for the corunning low priority application. The first bar shows the QoS of the high priority application when it is corunning with the original low-priority application without the Reactive-Niceness. The rest of the three bars show its QoS when we apply RN with three configurations of the `simple` heuristic to the low-priority application. Each of three low-priority applications, `sledge`, `lbn`, and `milc` is used in Figures 6.5, 6.7, 6.9 respectively.

Figures 6.6, 6.8 and 6.10 show the corresponding utilization gained for each of the low-priority applications. Note that we are measuring the utilization of the computing resources used by the low-priority application.

From the figures, we observe that when applying Reactive-Niceness with our `simple` heuristic, the QoS of each high-priority application is significantly improved (by up to 26%) relative to the configuration of allowing the co-location of both applications without Reactive-Niceness (first bar in Figures 6.5, 6.7 and 6.9). Recall that without Reactive-Niceness, such colocation of low- and high-priority applications would be disallowed due to the possible QoS degradation. Compared to such a baseline of disallowing co-location, we gain a significant amount of utilization when allowing co-location with Reactive-Niceness, often more than 50%. Various configurations in `simple` heuristic also provide a wide range of options for balancing QoS and utilization. In this experiment, the utilization-biased

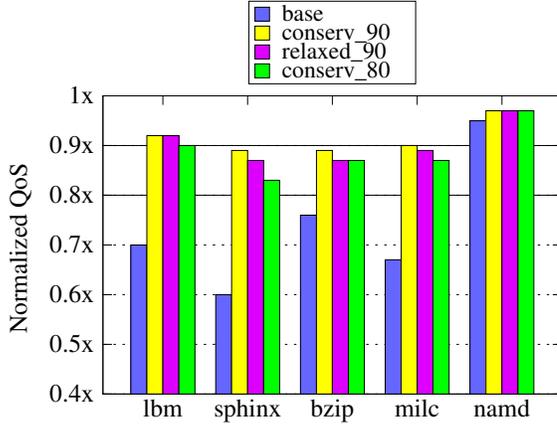


Figure 6.11: QoS of each benchmark co-running with `sledge`, normalized to solo QoS. (targeted)

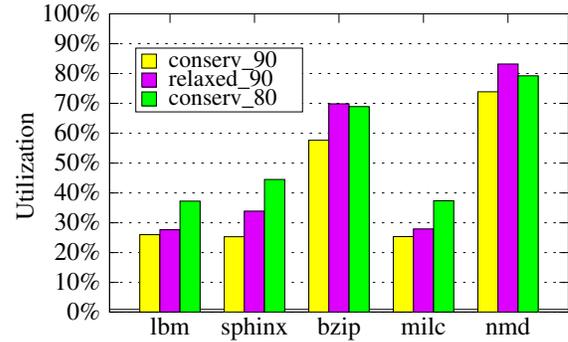


Figure 6.12: Utilization of `sledge` with each configuration. (targeted)

configuration achieves significantly higher utilization than other two configurations, and the QoS of each high-priority application only slightly degrades. This demonstrates that with parameter tuning, the `simple` heuristic can be effective in improving QoS while gaining a significantly amount of processor utilization.

6.4.3 Effectiveness of Reactive-Niceness: Targeted Heuristic

Our more sophisticated `targeted` heuristic enables a more precise enforcement to achieve the desired QoS requirements. This heuristic has effectively three “knobs,” one for the specific QoS threshold to enforce, and the other two for how conservatively (strictly) this QoS threshold must be enforced (parameters `QoS_goal`, `conservative_factor` and `execution_period` in Algorithm 5). With more conservative parameters, the application QoS is less likely to drop below the specified threshold; however, a larger amount utilization may be sacrificed. We explore this tradeoff in our evaluation.

Figures 6.11 – 6.16 are similar to those presented above. For this set of graphs we use our `targeted` heuristic with the three configurations presented in Table 6.2. The configurations `conserv_90`, `relaxed_90`, and `conserv_80` represent a conservative setting at a 90% QoS threshold, a relaxed setting at 90%, and a conserve setting at an 80% QoS threshold, respectively. Figures 6.11, 6.13 and 6.15 show the effect of using our `targeted` heuristic on the QoS of the high-priority applications. Note that two horizontal lines are drawn in each graph denoting the 90% and 80% QoS thresholds. Figures 6.12, 6.14 and 6.16 show

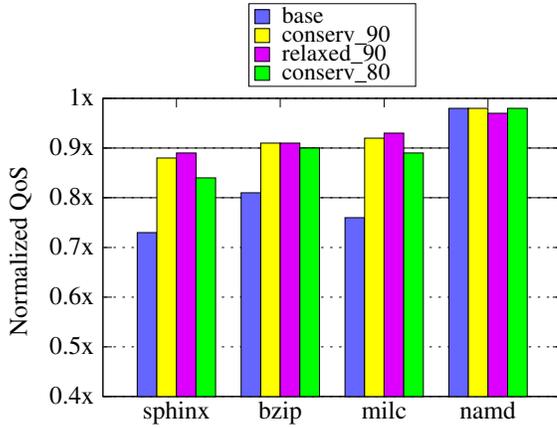


Figure 6.13: QoS of each benchmark co-running with lbm. (targeted)

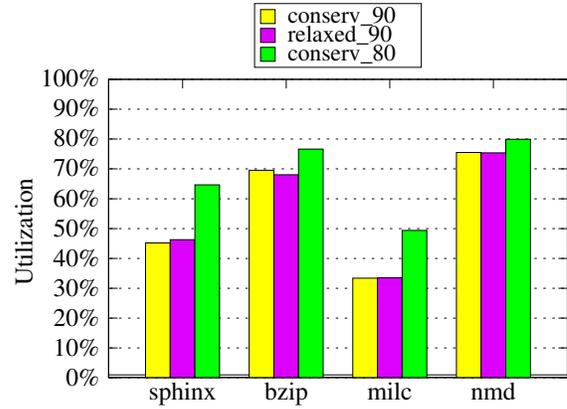


Figure 6.14: Utilization of lbm with each configuration. (targeted)

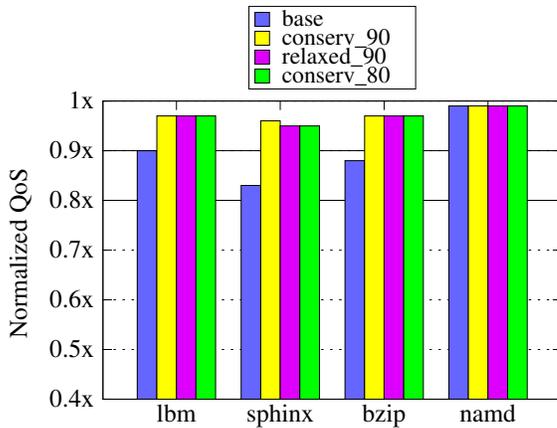


Figure 6.15: QoS of each benchmark co-running with milc. (targeted)

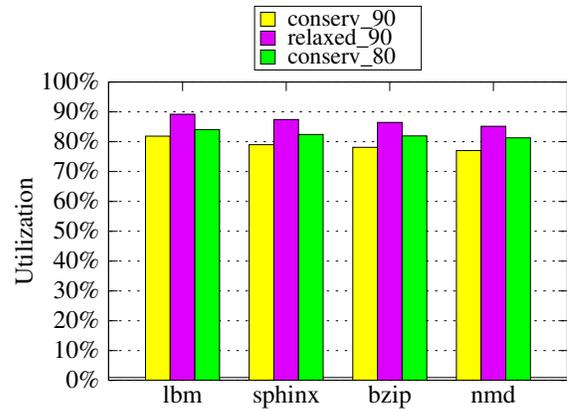


Figure 6.16: Utilization of milc with each configuration. (targeted)

Configurations	ex_period(ms)	conserv_factor	QoS_goal
conservative_90	6	0.4	90%
relaxed_90	12	1.0	90%
conservative_80	9	0.4	80%

Table 6.2: Three configurations of `targeted` heuristic

the corresponding processor utilization gained for each configuration.

As shown in these figures, the `targeted` heuristic is quite effective in bringing the QoS of the high priority applications to the desired QoS threshold, beating it in many cases and coming very close in the worst cases with our conservative settings. When using a relaxed setting, we observe a bump in the utilization, and our QoS target is often met. The decision as to how conservative or relaxed the QoS target is depends on the objectives and discretion of the application service provider and whether higher utilization is desired or stricter QoS polices are specified.

[Simple vs. Targeted] Our `simple` and `targeted` heuristics offer two options to application service providers: one allowing the tuning of the tradeoff between utilization and QoS when a specific QoS target is not specified, the other when the specific QoS degradation threshold is known. When configured appropriately, the `simple` heuristic can perform quite well. However, it may require a significant amount of parameter tuning to search for the appropriate configuration. The appropriate configuration may also change when the co-running applications change. The `targeted` does not require such parameter tweaking because it is self tuning and feedback directed. More comparison between `simple` and `targeted` is presented in the following section.

6.4.4 Effectiveness of Reactive-Niceness: Phase Level Behavior

We further evaluate the phase-level effectiveness of RN-Runtime in improving the QoS of high-priority applications.

Figure 6.17 presents the IPC of `sphinx` when it is running with the original `sledge`, comparing to its IPC when running with `sledge` on RN-Runtime using the `simple` heuristic. The IPC samples are normalized to `sphinx`'s IPC profile when running alone to demonstrate the IPC degradation due to contention. In this experiment, `simple` heuristic is using

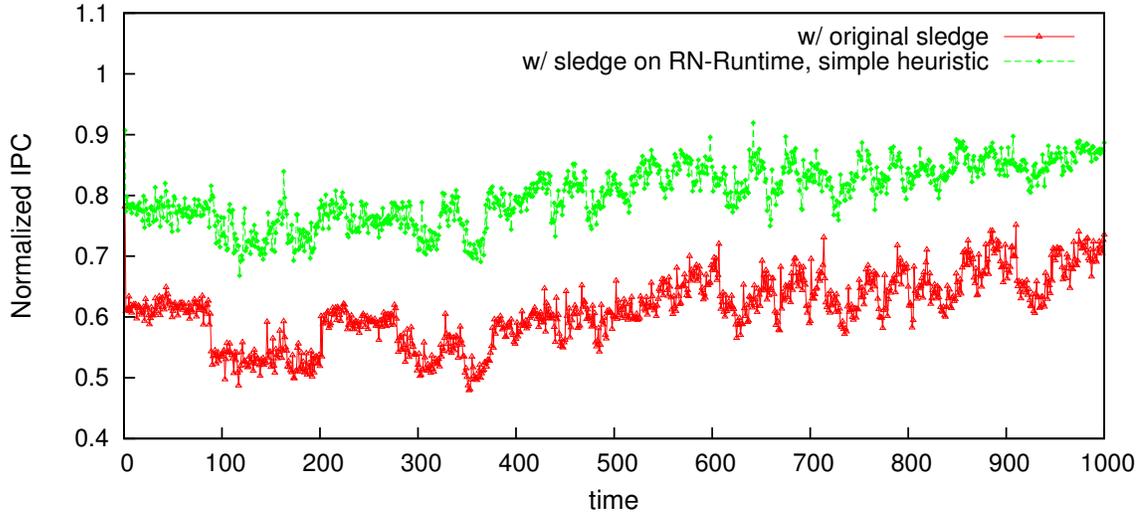


Figure 6.17: Sphinx normalized IPC with original `sledge` and with `sledge` with RN_H1

balance configuration and `sphinx` is using `ref` input. To calculate the normalized IPC, we collect the IPC profiles of `sphinx` when it is running alone (solo) and running with `sledge`. IPC is sampled every 1 ms and all profiles of the entire execution of `sphinx` are down sampled to 1000 data points. The normalized IPC at point i is calculated as $\frac{IPC_{corun,i}}{IPC_{solo,i}}$. Therefore, the closer the normalized IPC to 1, the less the degradation. In Figure 6.17, the line denoting the original `sledge` shows phase-level changes of the IPC degradation due to contention. For example, around samples 100 to 200, and 300 to 400, there are noticeable phases of degradation increase. Also the degradation is less significant during the later half of the execution. Figure 6.17 also clearly demonstrates the IPC improvement achieved by RN-Runtime along the entire execution of `sphinx`. Instead of around 60%-70% of the normalized IPC when running with the original `sledge`, Reactive-Niceness improves the normalized IPC to above 80% through most of the execution.

Similar to Figure 6.17, Figure 6.18 presents `sphinx`'s normalized IPC when it is running with `sledge` using `targeted` heuristics (`conservative_90` configuration), also comparing to its normalized IPC when running with the original `sledge`. Despite the distinctive phases of varying levels of degradation when running with the original `sledge` as discussed previously (for example, samples 100-200 and 300-400), `targeted` heuristic consistently guarantees around 90% IPC for `sphinx` through the entire execution. This is different from the `simple` heuristic shown Figure 6.17 where the improved normalized IPC fluctuates be-

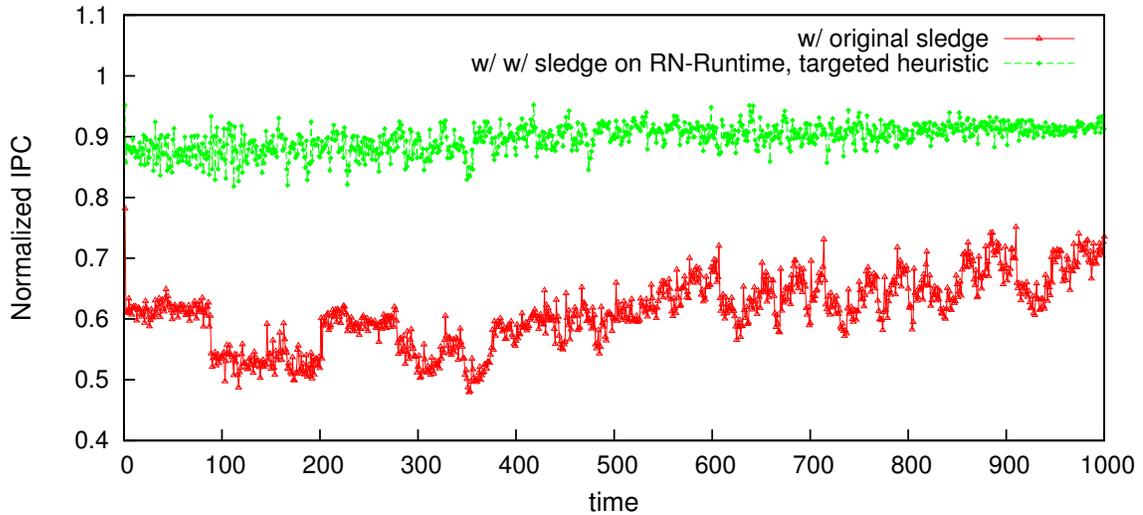


Figure 6.18: Sphinx normalized IPC with original `sledge` and with `sledge` with `RN_H2`

tween 70% and 90%. This comparison highlights the difference between the `simple` and `targeted` heuristics. While `simple` is effective in improving the QoS, `targeted` heuristic is effective in adapting, achieving and maintaining a stable QoS level as specified.

Similar to Figures 6.17 and 6.18, Figures 6.19 and 6.20 present the normalized IPC of `sphinx` when it is running with the original `milc`, as well as `milc` with Reactive-Niceness. In Figures 6.19 and 6.20, the RN-Runtime for `milc` uses the `simple` heuristic and `targeted` heuristic respectively. The IPC of `sphinx` when running with the original `milc` demonstrates the varying levels of contention and degradation. For example, during samples 600 to 800, the degradation is significantly smaller (normalized IPC close to 1) than the rest of the execution. A few samples with normalized IPC higher than 1 are due to aliasing of down-sampling. In this set of experiments, `targeted` heuristic is configured as `conservative_90`, meaning RN-Runtime aims at less than 10% of the QoS degradation for `sphinx`. `Simple` heuristic is configured with `QoS_biased` configuration to achieve the similar QoS goal as `targeted`. Figures 6.17 and 6.18 demonstrate the effectiveness of Reactive-Niceness. The QoS of `sphinx` is significantly improved after applying RN to corunning `milc`; the normalized IPC of `sphinx` is stable and between 0.9 and 1.

Figure 6.21 presents the corresponding average nap duration of `milc` for every 2 ms' execution, decided dynamically by RN-Runtime based on dynamic contention detection. The longer the nap duration, the lower the utilization. Figure 6.21 shows that `simple` heuristic

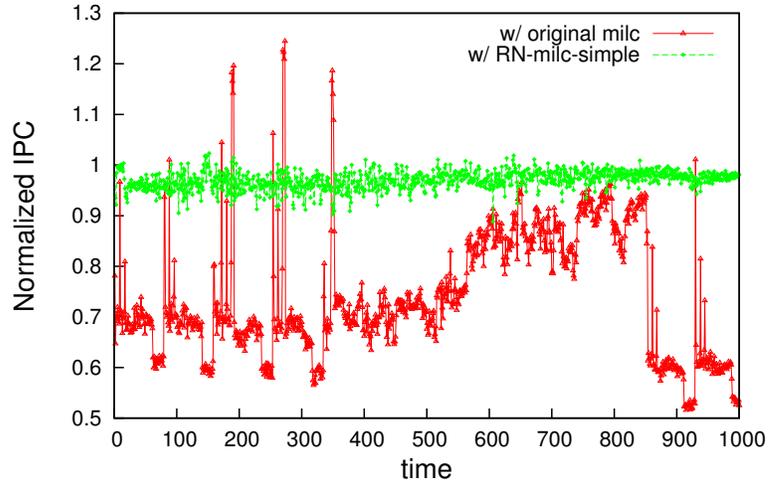


Figure 6.19: Sphinx normalized IPC with original milc and with simple milc

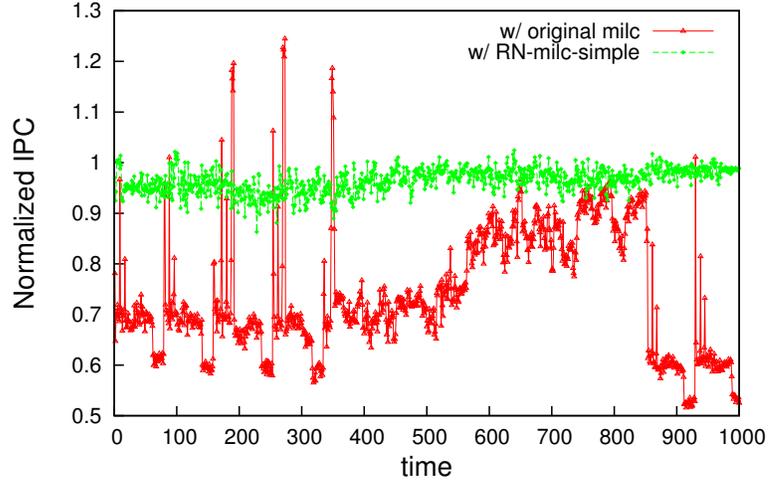


Figure 6.20: Sphinx normalized IPC with original milc and with targeted milc

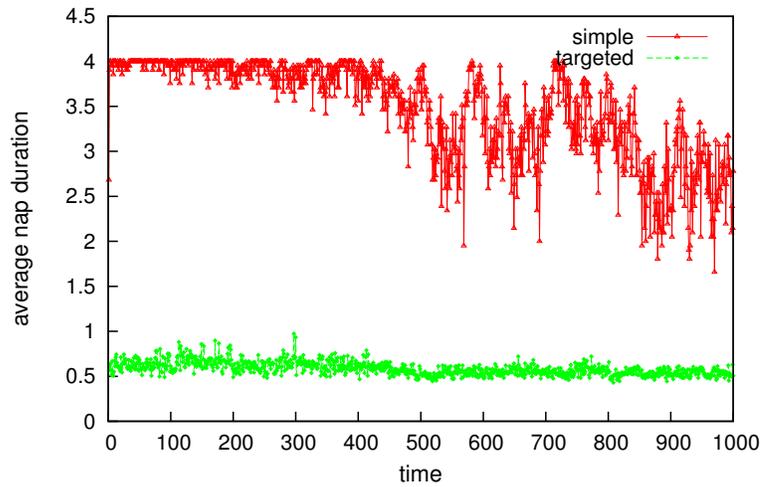


Figure 6.21: Average nap duration for milc with simple vs. milc with targeted

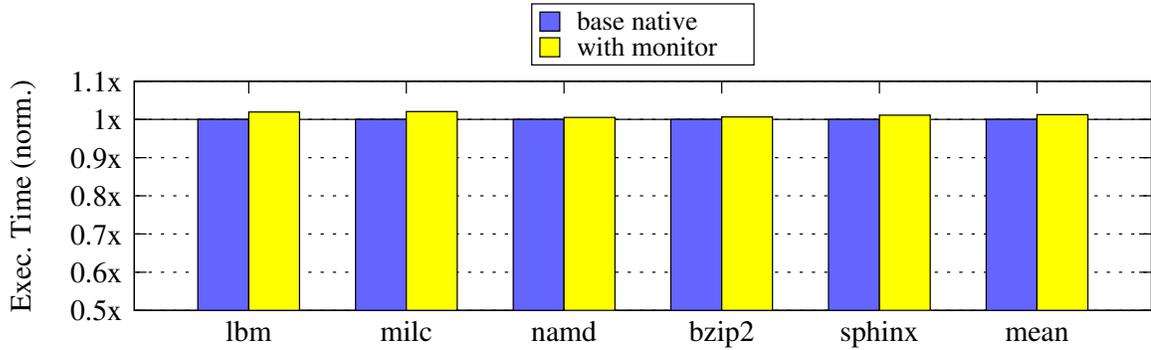


Figure 6.22: Overhead of monitoring for high-priority application.

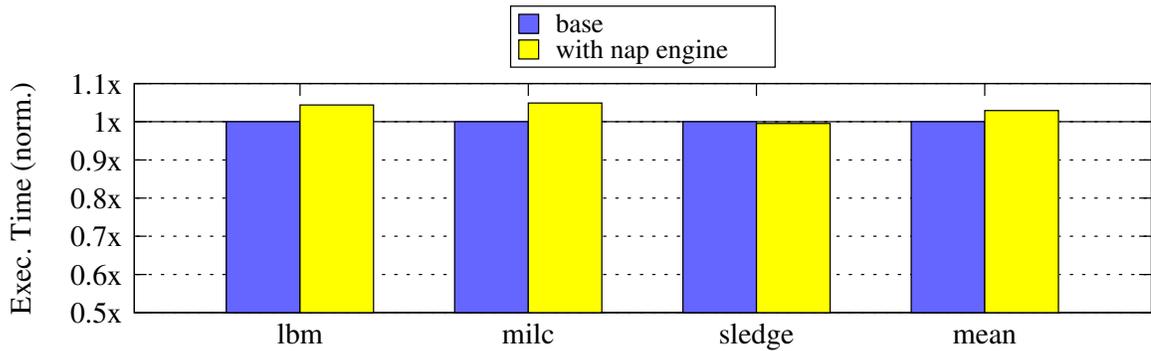


Figure 6.23: Overhead of nap engine for low-priority application.

demonstrates certain adaptability. After sample 600 when the contention is not as significant, naps become shorter. However, in general, the nap duration is significantly shorter using the `targeted` heuristic, while achieving similar QoS improvement as the `simple` heuristic. This is because `targeted` heuristic can estimate the amount of QoS degradation and the necessary amount of nap/throttling for achieving the QoS goal, and adaptively adjust the nap based on the estimation; while `simple` heuristic may over-conservatively throttle down the low-priority application, especially when it is configured to bias towards QoS.

6.4.5 Overhead

Figure 6.22 presents the performance costs of the monitoring the QoS of the high-priority application. The overhead is minimal. The overhead suffered by high-priority applications is less than 1% on average with a max of 2% in the case of `milc`.

Figure 6.23 shows the performance overhead of invoking the Nap Engine to throttle

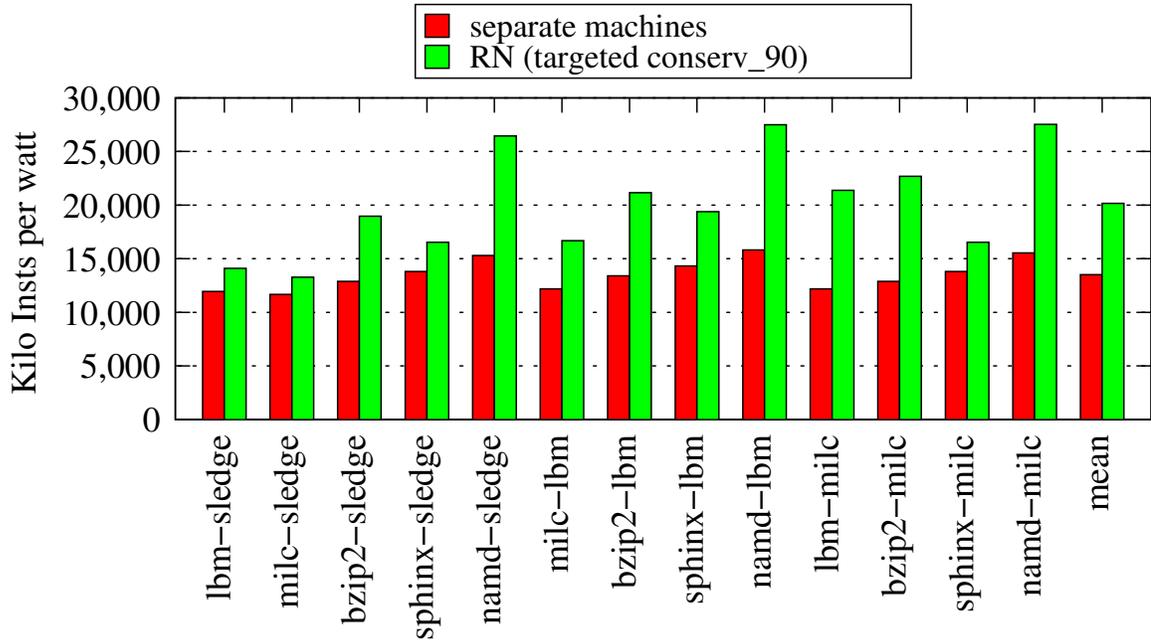


Figure 6.24: Efficiency of allowing co-location with Reactive-Niceness vs over-provisioning. (targeted)

down low-priority applications. The overhead of probing the Nap Engine is slightly more costly, approaching 5% for milc. However, the Nap Engine is only causing overhead to the low-priority application, and the performance cost is not as important.

The low cost of our runtime approach is due to the fact that we only invoke the runtime system at the 1 ms granularity for both low and high-priority applications. The overhead of reading and recording performance counters is also minimal. The cost is slightly higher for the low-priority application because we add a lightweight check at the point of every compiler-inserted marker. Coarsening the granularity can further reduce these overheads; however the tradeoff must be made between a lower overhead and a higher penalty for potential delays in detecting contention as it occurs.

6.4.6 Energy Efficiency of using Reactive-Niceness

Figure 6.24 presents the improved energy efficiency when allowing co-location with Reactive-Niceness. These experiments were performed using a P3 International Kill A Watt® power meter connected to our Quad Core Intel Nehalem machine to measure whole system watt consumption during execution. For each cluster of bars in the figure, the energy efficiency

is calculated by the instructions processed per watt for a three minute time period after the machine wattage stabilizes during each run. The higher the bar, the more energy efficient. The x-axis shows the workloads, the high priority and low priority application pairs. The first bar for each workload shows the energy efficiency when using separate machines for low and high priority applications; the second bar shows the energy efficiency of co-locating both high and low priority applications using Reactive-Niceness with the `targeted` policy and the `conserv_90` configuration shown in Table 6.2. We observe a significant energy efficiency improvement for many workloads. Application pairs that include less contentious applications, such as `namd`, produce a greater benefit as there is less napping occurring. Meanwhile, highly contentious pairs, such as `sphinx-lbm`, show a more modest benefit. On average there is a 42% improvement of using Reactive-Niceness to allow co-location over using two separate machines for low and high priority applications.

6.4.7 Varying Architecture

To investigate the effectiveness of Reactive-Niceness across architectures, we performed experiments on a 2.6GHzs Quad Core AMD Phenom X4 system with 6MB last level cache and 3GB of main memory. This machine is also running Linux 2.6.29.6 and our customized GCC 4.4.6.

Figures 6.25 and 6.26 show the results for our `targeted` heuristic using the same configurations shown in Table 6.2. As shown in these figures, Reactive-Niceness is also quite effective on this platform. For both `lbm` and `milc` we achieve 80% to 90% utilization while significantly reducing the performance interference on our high-priority applications. The contentiousness of `sledge` is severe on this processor. For the `lbm-sledge` pair, we observe that when lowering the QoS threshold to 80% from 90% we more than double the utilization. Overall, our conservative settings meet and exceed our QoS requirements for each of the experiments shown in Figure 6.25, and our relaxed configuration satisfies the QoS constraint in for majority of the applications.

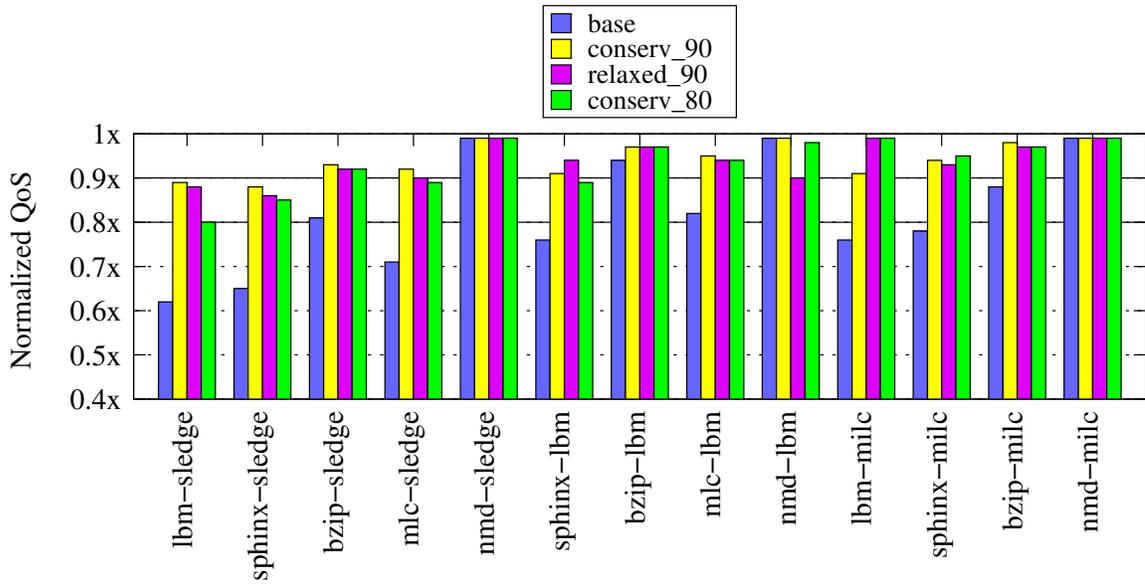


Figure 6.25: QoS of each benchmark co-running with sledge, lbm, and milc. (targeted)

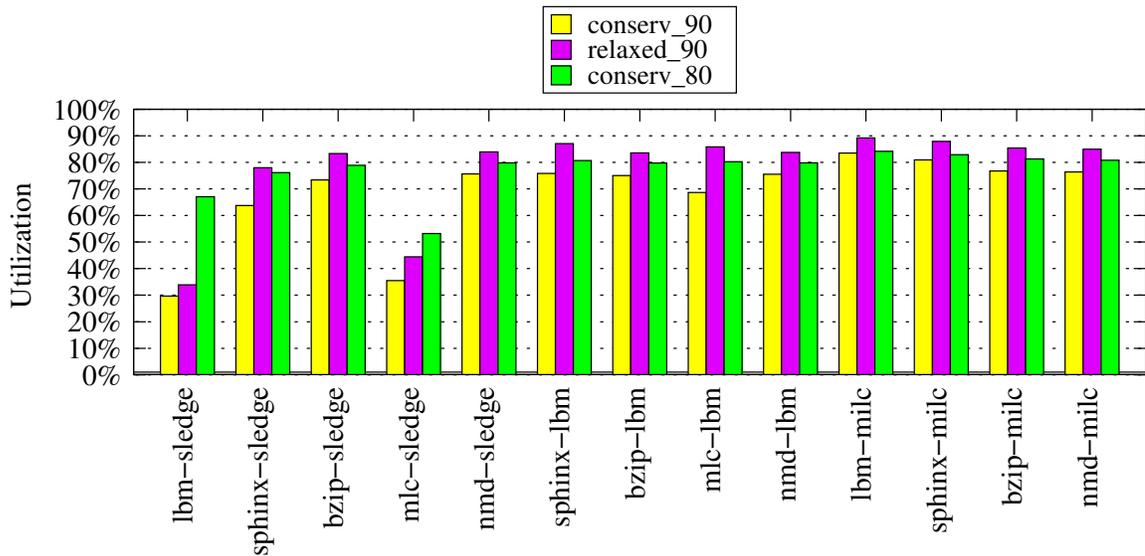


Figure 6.26: Utilization of sledge, lbm and milc with each configuration. (targeted)

6.5 Summary

In this chapter, we combine static compilation and dynamic adaptation to address the challenge of cross-core interference on the QoS of high priority applications. We have presented Reactive-Niceness, a static/dynamic compilation approach to improving machine utilization in WSCs by enabling the adaptive manipulation of the contentiousness of low-priority applications to ensure the QoS of high-priority co-runners. Reactive-Niceness consists of a profile guided compilation technique that identifies and inserts markers in contentious code regions, and a lightweight runtime that monitors the QoS of high-priority applications and reactively triggers short naps of low-priority applications when cross-core interference is detected. Our evaluation shows that Reactive-Niceness is able to improve utilization by more than 70% in many cases, and more than 50% on average, while enforcing a 90% QoS threshold.

Finally let us compare the static approach, QoS-Compile, presented in Chapter 5, with the hybrid approach Reactive-Niceness presented in this chapter.

- QoS-Compile, as a static compilation solution, is more simplistic, lightweight and does not require deploying a runtime system. Since QoS-Compile only throttles down the contentious code regions, it avoids unnecessary throttling down and utilization loss when contentious code regions are not executing.
- Reactive-Niceness, taking advantage of the QoS-Compile's prediction model, dynamically throttles down low-priority applications based on the amount of contention and QoS degradation detected. Because of the flexibility, it achieves better server utilization, especially when the co-running high priority application is not sensitive and not affected by the low-priority applications. Another advantage of Reactive-Niceness is more accurate QoS management, especially when using the **targeted** heuristic.

Chapter 7

Conclusions and Future Directions

Contents

7.1 Summary of Themes and Results	111
7.2 Future Direction	113
7.2.1 Managed Runtime for QoS and utilization in WSCs	113
7.2.2 Runtime systems and research infrastructure for WSCs	113

This dissertation comprehensively investigates the impact of memory resource sharing on industry-strength large-scale datacenter workloads and shows that, contrary to conclusions from recent work [61], memory resource sharing has a significant performance impact on emerging large-scale web-service applications in modern warehouse scale computers. This dissertation also presents two complementary software strategies to mitigate memory resource contention for improving performance and server utilization of WSCs. We design a heuristic based system and a runtime system to intelligently map application threads to cores to promote positive resource sharing and mitigate resource contention to improve application performance. We design novel compilation techniques and runtime systems that statically and dynamically manipulate applications' contentious nature to enable the co-location of applications with varying QoS requirements, and as a result, greatly improve server utilization in WSCs.

7.1 Summary of Themes and Results

[**The impact of memory resource contention for WSC workloads**] Our investigation studies both the impact of memory resource sharing among threads from a single application and among threads from different co-running applications.

- **Intra-application Sharing** Our investigations demonstrate that, across several key datacenter applications including websearch, the impact of sharing the last level cache among threads 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.
- **Inter-application Sharing** Contention between multiple applications for the shared caches and memory bandwidth can often cause significant performance degradation for emerging WSC workloads. As a result, an application’s performance swings between its best and worst thread-to-core mapping can be significant (up to 40%).
- **Optimal Thread-to-Core Mapping** The best thread-to-core mapping for a given application does not only depends on the application’s sharing and memory characteristics, it is also impacted dynamically by the characteristics of other applications that are co-running on the same machine simultaneously.

[**Intelligent thread-to-core mapping**] We design intelligent TTC mapping approaches to mitigating memory resource contention to improve performance, including a heuristic-based approach and an adaptive approach.

- **Heuristic-based TTC Mapping** We show 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. We identify the application characteristics that impact performance in the various thread-to-core mapping scenarios, and our algorithm can accurately predict the optimal TTC map-

ping in most cases. In other cases, its predictions generate no more than 2% worse results than the optimal.

- **Adaptive TTC Mapping** We conclude that our online adaptive learning approach is a preferable approach for arriving at good thread to core mappings in the datacenter as it is more flexible and portable. It arrives at near optimal decisions and is agnostic to applications' sharing characteristics. By employing the adaptive thread-to-core mapper, **AtoM**, the performance of the datacenter applications is improved by up to 22% over status quo thread-to-core mapping and performs within 3% of optimal.

[**Static/Dynamic Compilation for QoS and Utilization**] We design both static and dynamic approaches to mitigating memory resource contention to improve server utilization. Our approaches manipulate low-priority applications' contentious nature to improve the corunning latency-sensitive applications' QoS. By providing such QoS management on multicores, our approaches enable the co-location of applications with varying QoS requirements and thus greatly improve server utilization in WSCs.

- **Identify Code Regions** We demonstrate that contentiousness is a consistent characteristic of an application and a code region. We then design a performance counter based prediction model that can accurately identify code regions that are contentious in nature. The linear correlation efficient of our model is 0.91, showing high prediction accuracy.
- **QoS-Compile** We design two novel compilation transformations *padding* and *nap insertion* and demonstrate their effectiveness in reducing the memory request rate of a code region and the interference the code region can cause to co-running applications. Finally, our experiments show that by combining our code region identification and compilation transformations, QoS-Compile improves applications' QoS performance by 21% and machine utilization 36% on average for both SPEC benchmarks and key Google applications on state-of-the-art server machines.
- **Reactive-Niceness** Our experiments demonstrate that Reactive-Niceness is able to improve server utilization by more than 70% in many cases, and more than 50%

on average, while enforcing a 90% QoS threshold. We are also able to improve the energy efficiency of modern multicore machines by 47% on average over the policy of disallowing co-locations that is commonly used.

7.2 Future Direction

There are a number of other important and promising research directions for improving the efficiency of WSCs. In this section, we discuss managed runtimes for QoS and utilization, and other runtime systems and infrastructure that is critical for researching WSCs.

7.2.1 Managed Runtime for QoS and utilization in WSCs

Managed runtimes, such as the Java VM, are commonly used in WSCs. For example, Gmail arguably constitutes one of the largest java codebases in the world. However, it is unclear how to mitigate the negative impact of memory resource contention for VM workloads. I plan to extend my research in static/dynamic compilation to design a QoS-Aware VM. The VM provides a potentially broader design space than native runtimes and thus more opportunities to address the challenges of memory resource contention and QoS in a datacenter. In addition to restructuring code layouts to reduce contention, novel language constructs can be designed to annotate information of a code region such as its QoS target, priorities and its sensitivity to performance interference. Based on this information, dynamic runtime can decide accordingly if necessary QoS management such as throttling down other applications is needed. I also plan to investigate how to use the virtual execution and garbage collection capabilities of managed runtimes to provide a harness for novel dynamic memory re-layout techniques to reduce memory resource contention.

7.2.2 Runtime systems and research infrastructure for WSCs

Research on architecting large-scale datacenters is still in its relative infancy. There are still numerous questions and opportunities as how to design an effective software stack especially for WSCs. I plan to conduct further research on designing large-scale cross-layer runtime systems that intelligently orchestrate and manage resources at on-chip level,

machine-level and cluster-level, for improving performance, QoS and reducing cost in WSCs. The runtime system will integrate the performance and information of applications as well as the underlying hardware resource monitoring. It also needs to break the communication boundaries between various management levels existent in current systems. During the process of this research, I will also design methodologies, simulations, benchmarks and mini-cloud environment that can facilitate the community to conduct datacenter research without accessing the production workloads and datacenters. The lack of access has always been an obstacle for the community. I believe that my experience of close collaboration with the industry will help me provide such validated research environment.

Bibliography

- [1] Intel 64 and ia-32 architectures software developer's manual volume 2b: Instruction set reference, m-z.
- [2] Latent semantic analysis. http://en.wikipedia.org/wiki/Latent_semantic_analysis.
- [3] Protocol buffer. <http://code.google.com/p/protobuf/>.
- [4] M. Banikazemi, D. Poff, and B. Abali. Pam: a novel performance/power aware meta-scheduler for multi-core systems. *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 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] L. Barroso and U. Holzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, Jan 2009.
- [7] M. Bhaduria and S. McKee. An approach to resource-aware co-scheduling for cmps. *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, Jun 2010.
- [8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. *11th International Symposium on High-Performance Computer Architecture*, 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] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2006.
- [12] 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. *ASPLOS '10: Proceedings of Architectural support for programming languages and operating systems*, Mar 2010.
- [13] EPA. Epa report to congress on server and data center energy efficiency. Technical report, U.S. Environmental Protection Agency, 2007.
- [14] S. Eranian. What can performance counters do for memory subsystem analysis? *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with ASPLOS'08*, pages 26–30, 2008.
- [15] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques 2007*, Sep 2007.
- [16] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–355, 2007.
- [17] J. Hamilton. Internet-scale service infrastructure efficiency. *SIGARCH Comput. Archit. News*, 37(3):232–232, 2009.
- [18] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, Jun 2009.

- [19] L. Hsu, S. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, Sep 2006.
- [20] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao: An extensible micro-architectural optimizer. In *2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, april 2011.
- [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 '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Jun 2007.
- [22] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proceedings of the 38th annual international symposium on Computer architecture (ISCA '10)*, New York, NY, USA, 2010. ACM.
- [23] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, Oct 2008.
- [24] 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.
- [25] M. Kandemir, S. Muralidhara, S. Narayanan, Y. Zhang, and O. Ozturk. Optimizing shared cache behavior of chip multiprocessors. *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 505–516, 2009.
- [26] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. Irwin, and Y. Zhnag. Cache topology aware computation mapping for multicores. *PLDI '10: Proceedings of*

the 2010 ACM SIGPLAN conference on Programming language design and implementation, Jun 2010.

- [27] 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. *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–11, 2010.
- [28] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, Sep 2004.
- [29] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.
- [30] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30, July 2010.
- [31] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. *IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, 2008.
- [32] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on DOI - 10.1109/HPCA.2010.5416655*, pages 1–12, 2010.
- [33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05*, pages 190–200, New York, NY, USA, 2005. ACM.

- [34] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 7th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '09, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.
- [35] J. Mars and M. L. Soffa. Synthesizing Contention. In *Workshop on Binary Instrumentation and Applications*, 2009.
- [36] J. Mars, L. Tang, and R. Hundt. Heterogeneity in “homogeneous” warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 2011.
- [37] 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 *MICRO '11: Proceedings of The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011. ACM.
- [38] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Increasing utilization in warehouse scale computers using bubble-up! *Special Issue: IEEE Micro's Top Picks from 2011 Computer Architecture Conferences*, 2012.
- [39] 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.
- [40] 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.
- [41] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. <http://www.cs.virginia.edu/stream/>, February 2005.
- [42] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of Architectural support for programming languages and operating systems*, ASPLOS '09, pages 205–216, New York, NY, USA, 2009. ACM.

- [43] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37(4):34–41, Mar. 2010.
- [44] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. *Micro, IEEE DOI - 10.1109/MM.2008.48*, 28(3):6 – 16, 2008.
- [45] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2006.
- [46] P. Ranganathan and N. Jouppi. Enterprise it trends and implications for architecture research. *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on DOI - 10.1109/HPCA.2005.14*, pages 253– 256, 2005.
- [47] 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*, 30:65–79, 2010.
- [48] A. Sandberg and D. Eklöv. . . . Reducing cache pollution through detection and elimination of non-temporal memory accesses. *SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010.
- [49] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 258 – 269, 2008.
- [50] S. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, Feb 2009.
- [51] S. Srikantaiah, M. Kandemir, and M. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. *ASPLOS XIII: Proceedings of the 13th international*

- conference on Architectural support for programming languages and operating systems*, Mar 2008.
- [52] S. Srikantaiah, M. Kandemir, and Q. Wang. Sharp control: controlled shared cache management in chip multiprocessors. *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2009.
- [53] R. Szeliski. Image alignment and stitching: a tutorial. *Found. Trends. Comput. Graph. Vis.*, 2(1):1–104, 2006.
- [54] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58, New York, NY, USA, 2007. ACM.
- [55] L. Tang, J. Mars, and M. L. Soffa. Reactive niceness: Static/dynamic compilation for qos in warehouse scale computers. *In submission*.
- [56] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, colocated with PLDI '11, pages 12–21, New York, NY, USA, 2011. ACM.
- [57] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *CGO '12: Proceedings of the 10th annual IEEE/ACM international symposium on Code generation and optimization*, 2012.
- [58] 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 38th annual international symposium on Computer architecture*, ISCA '11, pages 283–294, New York, NY, USA, 2011. ACM.

- [59] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *The 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.
- [60] D. Xu, C. Wu, and P.-C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, Sep 2010.
- [61] E. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? *PPoPP 2010*, pages 203–212, 2010.
- [62] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Sep 2007.
- [63] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, Mar 2010.