Rethinking the Architecture of 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

Jason Mars

May 2012

Approvals

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

---

Jason Mars

Approved:

Mary Lou Soffa (Advisor)

Kim Hazelwood (Chair)

Jack Davidson

David August

John Lach

Accepted by the School of Engineering and Applied Science:

Richard W. Miksad (Dean)

April 2012

Abstract

As the worlds computation continues to move into the massive datacenter infrastructures recently coined as "warehouse-scale computers" (WSCs), developing highly efficient systems for these computing platforms is increasingly critical.

The architecture of modern WSCs remain in their relative infancy. WSC architects have started with commodity off-the-shelf components including commodity processors and open source system software components, that are then stitched together to design a simple and cost effective WSC. This approach has been effective for producing systems that are functional, and that can scale the delivery of web-services as demand increases. However, efficiency has suffered, as these components have not been designed and refined with the unique characteristics of WSCs in mind. These characteristics may be critical for a highly efficient WSC design, and as such, we must rethink the architecture of modern WSCs.

This dissertation argues that one such characteristic has been overlooked: the diversity in *execution environments* in modern WSCs. We define a given task's execution environment as the coupling of the machine configuration, and the co-running tasks simultaneously executing alongside the given task. At any given time in a WSC, we have a high degree of diversity across these execution environments. This dissertation argues that acknowledging, exploiting, and adapting to, the diversity in execution environments are crit-

ical for the design of a highly efficient WSC. When ignoring this diversity, three critical design problems arise, including 1) *the homogeneous assumption*, where all machines and cores in a WSC are assumed to be equal and managed accordingly, 2) *the rigidness of applications*, where application binaries can not adapt to changes across and within execution environments, and 3) *the oblivion of interference*, where interference between tasks within an execution environment can not be measured or managed.

This dissertation addresses each of these three design problems. First, we address *the homogeneous assumption* at the cluster level by redesigning the task manager in the WSC to learn which execution environments tasks prefer, and map them accordingly. Second, we address *the rigidness of applications* at the machine level by providing a mechanism to allow applications to adapt to their execution environment, and leverage this mechanism to solve pressing problems in WSCs. Lastly, we address *the oblivion of interference* at both the cluster and machine levels by providing novel metrics and techniques for measuring and managing interference to improve the utilization of WSCs.

By incorporating an awareness of the diversity in execution environments in these three key design areas, we produce a WSC design that is significantly more efficient in both the performance of the applications that live in this domain, and the utilization of compute resources in the WSC. By improving efficiency for these two metrics, we effectively require a smaller WSC from some fixed workload, which has implications on reducing not only the cost of these systems, but also their environmental footprint.

Acknowledgements

My life has been impacted by a number of extraordinary people. Where does one begin?

First and foremost, I would like to thank Mary Lou Soffa for being one of the most important people in my life. Her contribution to me as a person goes beyond our professional relationship. She has been a second mother, wise teacher, and dear friend to me through the 10 years I have known her. She was the first mentor of mine to believe in me, and see real talent and potential, when most would have ruled me out. I could never thank her enough for the impact she's had on my life.

I'd also like to thank Robert Hundt, whom I met during my first internship at Google. Robert is one of the most amazing and rare people I know. He has been a close mentor and friend of mine. I have absorbed a great amount of wisdom and knowledge from him. Not just technical knowledge, but also how to challenge myself to have to courage to persue goals that others may talk themselves out of. I look forward to a life-long friendship with him.

Naveen Kumar was my first research colleague. Now he is one of my closest friends. I met him as a sophomore immediately after joining Mary Lou Soffa's group at the University of Pittsburgh. I thank him for taking an interest in me early on, and teaching me the ropes. I also thank him for being there for me every moment since meeting him, and look forward to

an epic future of collaborating on world class research.

I also would like to thank my wife, Lingjia Tang, who has been a positive and extraordinary force in my life in every way imaginable. When we are together, we are not additive, we are multiplicative. No, we are exponential... power set of R... second order power set of R cross R... you get my point. Like the song says, "I'm a movement by myself, but I'm a force when we're together..."

Other major influences in my life I'd like to thank includes but is not limited to Dick Sites, Neil Vachharajani, Vijay Reddi, Kevin Skadron, Kim Hazelwood, Sudhanva Gurumurthi, Jing Yang, Davidson Young, all of my graduate colleagues I chat with on a regular basis, and many others. I would also like to thank those mystery people at Google who has supported my career along the way. THANK YOU ALL!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

The landscape of computing is changing. Traditionally, the notion of computing held by users were of desktops in their homes that are used to accomplish some task, work or play, and return to their daily lives. However, in the recent decade, there has been a tectonic shift in the way end users view computing. With the evolution of the internet, and the emergence of mobile devices such as smartphones, tablets, and highly portable lap-

tops, we are now always connected, and interface computing continuously throughout the day. Much of the computation cycles consumed by this emerging market lives in massive datacenter infrastructures recently coined as "warehouse-scale computers" (WSCs) [50,79]. The distinction made with this new term is in viewing the entire warehouse itself, not as a collection of many computers, but as a single, massively parallel computer, where the programs run by this computer are large-scale web-services. This emerging computing domain is rapidly expanding. As noted by Forrester Research, cloud computing was a $40 billion market in 2011, and will grow six fold to $241 billion market by 2020 [101].

WSCs are massive computers with tens to hunderds of thousands of cores and petabytes of main memory, spread across thousands of machines. The underlying hardware components that comprise the WSC are primarily composed of commodity parts including processors, memory, disk, network, etc. These components are stitched together to compose a warehouse of interconnected machines. The architecture of such a system is largely defined by the middleware and system software stack. These software components provide a single programmatic view of the WSC, and organizes the underlying hardware infrastructure for efficient and scalable operation. Internet service companies such as Google, Amazon, Yahoo, Microsoft, and Apple spend tens to hundreds of millions of dollars on WSCs to provide web-services such as search, mail, maps, docs, video, voice recognition, etc [1,9,20,58]. This large cost stems from the machines themselves, networking equipment, power distribution and cooling, the power itself, and other infrastructure [35, 45]. Improving the efficiency of WSCs have been identified as one of the top priorities of web-service companies as it improves the overall total cost of ownership (TCO) of WSCs, and as noted by the Environmental Protection Agency (EPA) [35], improving efficiency is not only important for the cost

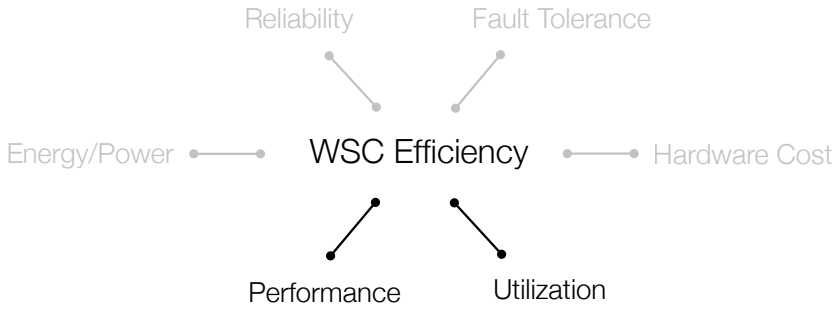


Figure 1.1: The Facets of WSC Efficiency. This Work Focuses on Performance and Utilization.

to companies, but for the environmental footprint of these WSCs as this computing domain rapidly expands.

However, inefficiency remains rampant in modern WSCs.

1.1 Rampant Inefficiency in WSCs

When considering the efficiency of WSCs there are a number of design objectives and metrics to consider. These *facets* of WSC efficiency are illustrated in Figure 1.1. In this work, we focus on software **performance** and machine **utilization** for improving the efficiency of WSCs. In addition to these objectives being identified as critical for efficiency in WSCs [50], improvements in both the overall performance of applications running in the WSCs and the utilization of the compute resources in WSCs have beneficial implications across many of the other design objectives.

To understand why WSCs are currently inefficient, its important to reflect on the design of these systems in recent history. Modern WSCs have been evolving for only about a decade now. During this time, the architects of these systems has had a functionality first, efficiency second approach. The first order objective has been to design a WSC system that can deliver web-services and products to the massive user base in such a way that

these infrastructures can scale with demand. To build a simple and cost effective system, WSC architects have used commodity off-the-shelf components including commodity x86 processors and open source system software components such as Linux, GCC, and the JVM. These general purpose components are then stitched together to design a simple and cost effective WSC system. As a second order objective, system architects then refine and improve upon these components for efficiency.

The problem with this approach is these commodity components were not built with WSCs in mind, and these massive WSCs do not resemble the traditional computing environments for which many of these commodity components have been initially designed and refined. When starting with these components to design the WSC architecture, system architects may produce a design that overlooks the unique characteristics and properties of WSCs. Thus, we must rethink the architecture of WSCs.

In this dissertation, we argue that one such characteristic is critical for designing a highly efficient system: the diversity of *execution environments* in WSCs.

1.2 Diversity of Execution Environment in WSCs

Given a running task in a WSC, its *execution environment* can be broadly defined as the set of elements in its context that impacts the performance and overall functioning of that task. Throughout the scope of this work, we focus on two of these elements: the machine, and its load. As such, the specific definition of *execution environment* used herein is the coupling of the machine configuration and co-running tasks simultaneously executing alongside the given task. Figure 1.2 provides an illustration of an execution environment. The regions highlighted in red show the components that

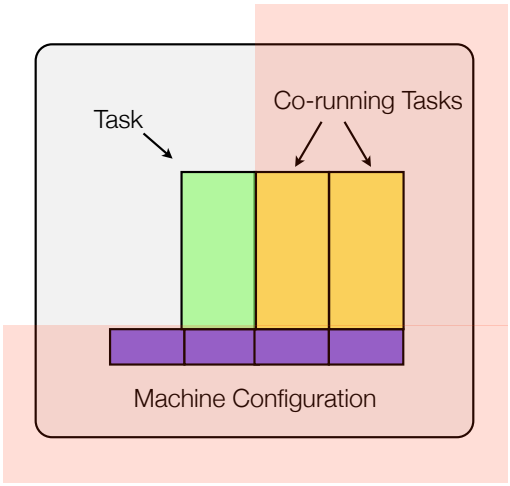


Figure 1.2: Execution Environment of a Task

comprise the execution environment of a task.

Within modern WSCs we have a high level of diversity across execution environments. Figure 1.3 provides a simplified illustration of the diversity that is commonly found in production WSCs. Across the machines shown in this WSC, we observe diverse microarchitectural configurations. In the Figure, we have an older Xeon architecture with only two cores, a newer Xeon architecture with four cores, and an Opteron architecture with four cores. These machines, spanning several generations of Intel and AMD architectures, differ not only in their number of course, but also in their microarchitectural design. In addition to diverse machine configurations, at any time during the lifetime of a WSC, each machine is loaded with different types, and a different number, of tasks. This co-runner diversity is also illustrated in Figure 1.3. Although we see a significant amount of diversity across various execution environments, modern WSC design does not acknowledge this diversity.

This dissertation argues for a WSC design that acknowledges the diversity in execution environments. We claim that incorporating an awareness of, and enabling adaptation to, this diversity is critical for highly efficient

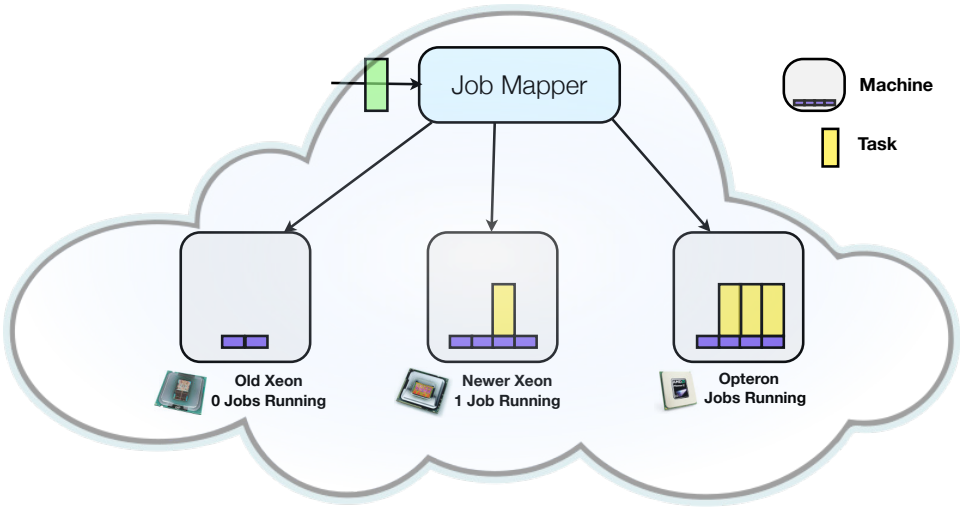


Figure 1.3: Diverse Execution Environments in WSCs

WSCs. In this dissertation, we address three design problems that results from overlooking the diversity across execution environments in WSCs.

1.3 Three Design Problems

There are three core problems that arise due to a WSC design that is oblivious of EE diversity:

1. **[Cluster Level]** *The homogeneity assumption.* The heterogeneity of machines that compose a WSC cluster is ignored. Tasks are not placed where they run best.

2. **[Machine Level]** *The rigidness of applications.* Application tasks do not adapt to changes across and within execution environments. Tasks run inefficiently.

3. **[Cluster and Machine Level]** *The oblivion of interference.* Interference between the tasks running within an execution environment is not measured or managed. The over-provisioning of compute resources is used for performance isolation leading to lower utilization.

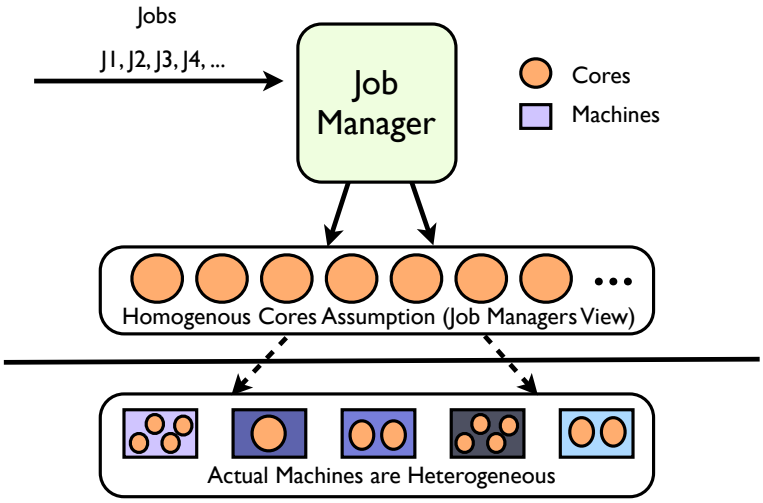


Figure 1.4: The Homogeneous Assumption - The Job Manager's View of a WSC

1.3.1 The Homogeneous Assumption

At the cluster level, *the homogenous assumption* is a source of inefficiency. WSCs have been embraced as homogeneous computing environments [9,50]. However, as previously discussed, this homogeneity is not the case in practice. WSCs are typically composed of cheap and replaceable commodity components. As machines are replaced in these WSCs, new generations of hardware are deployed while older generations continue to operate. This leads to a WSC that is composed of a mix of machine platforms, e.g. a *heterogeneous* WSC. Table 1.1 shows the amount of distinct machine con-

D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
4	3	2	3	2	3	2	5	2	2

Table 1.1: Number of Machine Types in Production WSCs

figurations found in 10 randomly selected anonymized Google WSCs in operation. As shown in the figure, these 10 WSCs house as few as two and as many as five different microarchitectural configurations spanning Intel and AMD servers from several consecutive generations. Yet, the assumption of homogeneity has been a core design philosophy behind the job management

subsystems of modern WSCs [50]. As Figure 1.4 shows, the job manager views the WSC as a collection of tens to hundreds of thousands of cores with the assumption of homogeneity. Each task in the WSC is configured with the number of cores and amount of memory it requires. The job manager then arbitrarily selects a machine with the required cores and memory available to assign the task.

Some types of tasks are very sensitive to changes in execution environment, while others are less affected. However, the diversity across the underlying microarchitectures and application co-runners in the WSC is not explicitly considered by the job management subsystem. As we show in this dissertation, ignoring this diversity in execution environments leads to inefficient execution of applications in WSCs. We also present an enhanced mapping system that exploits this diversity to improve the performance of the WSC.

1.3.2 Rigidness of Applications

At the machine level, *the rigidness of applications* is a source of inefficiency. Traditionally, an application program is written by a programmer, then statically compiled to a binary executable file composed of instructions from a targeted instruction set architecture (ISA). This binary can then be run on a range of micro-architectures that conforms to that ISA. The structure and layout of the binary code is determined statically, and consequently, it remains ridged across inputs, micro-architectures and execution environments. It is well known that semantically equivalent variations in the code structure and layout can cause a wide range of variance in its performance and other properties [39, 131]. As a result, programmers and optimizing compilers are faced with the task of statically determining the optimal code layout and structure for application binaries. However, these code structure and layout

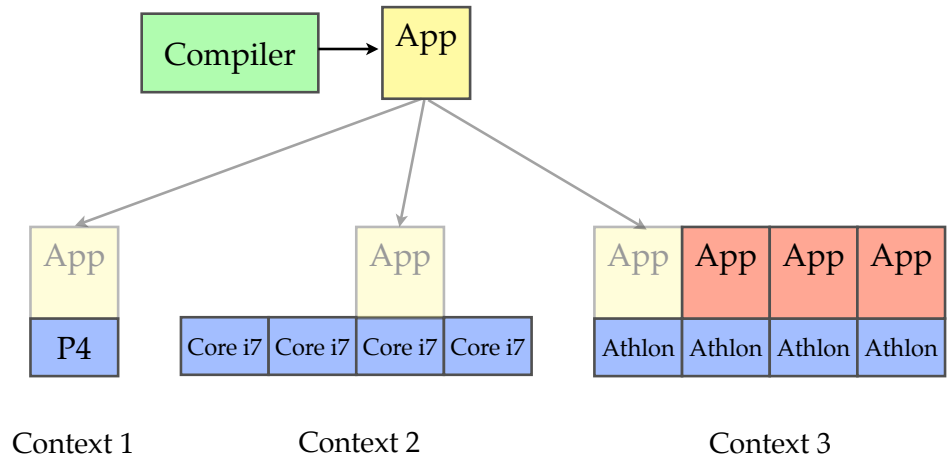


Figure 1.5: A Single *Rigid* Binary Executing in Three Contexts

decisions are impacted by changes across, and within, execution environments. These changes occur across application runs, and indeed during a single run.

Figure 1.5 shows three execution environments of a single application: in the first context the application runs alone on a single core machine, in the second context the same binary runs on a multicore machine, and in the third context the application's execution environment is includes three other co-running processes on a quad core machine. The applications optimal code layout and execution behavior may vary across these contexts. There are a class of problems that require the dynamic response to events as they occur during execution. To allow this flexibility, *online adaptation* is required. However, we lack a lightweight, deployable mechanism to enable online adaptation in modern WSCs. In this dissertation, we present a new paradigm for online adaptation that allows the adaptation of an application to its environment, and the environment to the application. We then leverage this approach to address two pressing problems in WSCs.

1.3.3 The Oblivion of Interference

At the cluster and machine levels, *the oblivion of interference* is a source of inefficiency. The cost of construction and operation of WSCs ranges from tens to hundreds of millions of dollars. As more computing moves into the cloud, it is becoming exceedingly important to utilize all the resources in WSCs as efficiently as possible. However, the utilization of the computing resources in modern WSCs remains low, often not exceeding 20% [72].

Each machine in the WSC house numerous cores, often 4 to 8 cores per socket, and 2 to 4 sockets per machine. However, in light of the significant potential for parallelism on a single machine, there are a number of resources shared among cores. This sharing can result in performance interference across cores, negatively and unpredictably impacting the performance of user-facing and latency-sensitive application threads [117]. Strictly defined performance requirements can also be refered to as *quality of service* (QoS) requirements. Interference negatively, and unpredictably, degrades QoS. To avoid the potential for interference on applications with strict QoS requirements, co-location is disallowed for latency-sensitive applications. This policy leaves cores idle, results in the overprovisioning of compute resources, and ultimately leads to lower utilization in the WSC.

This overprovisioning is often unnecessary, as co-locations may or may not result in significant performance interference. Figure 1.6 demonstrates the uncertainty of interference in modern WSCs. In this figure, we show the performance ($\frac{1}{latency}$), of a key user-facing component of Google's web-search when co-located with other Google workloads on a single socket, normalized to solo execution. The horizontal line shows the maximum allowable performance interference. The co-location of some workloads does not violate this QoS threshold (light bars), while others violate the threshold (dark bars).

The inability to quantify and predict the performance impact of inter-

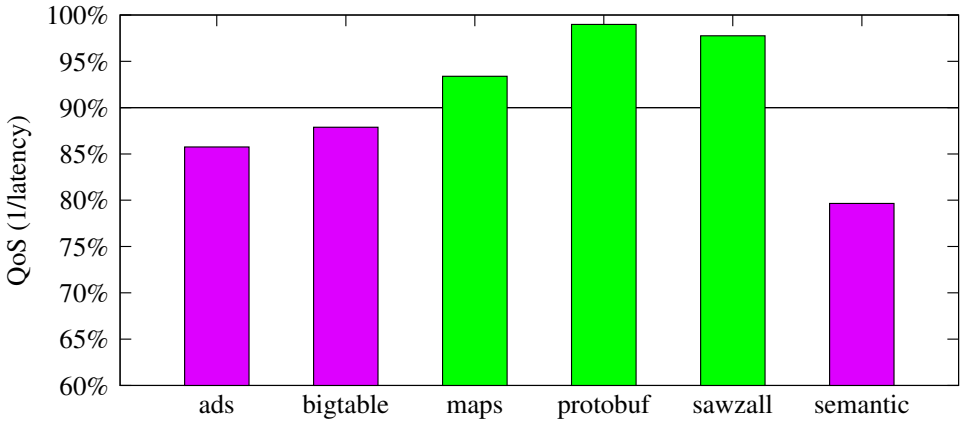


Figure 1.6: Some co-locations violate `web-search`'s 90% QoS threshold. The inability to precisely predict this performance interference leads to disallowing co-location for `web-search` and consequently, low machine utilization.

ference between tasks within the execution environment leads to the heavy handed solution of simply disallowing co-location. In this dissertation, we show it is indeed possible to perform this precise prediction using a novel mechanism and provide new metrics for characterizing an application's sensitivity to interference, as well as its aggressiveness.

1.4 Summary of Contributions

This dissertation investigates, and exploits, the diversity found in execution environments within production WSCs by first analyzing the diversity found in WSCs, and then addressing the three design challenges discussed.

1.4.1 Analyzing EE Diversity in Production WSCs

We first perform a study of the performance impact of the diverse execution environments (EE) across machines from Google's production fleet, and Google's large-scale commercial WSC workloads. We also replicate this study using an experimental testbed composed of benchmark workloads to

provide repeatable experimentation without the need of a commercial production environment. Chapter 3 examines:

- **Variability Across Execution Environments:** We investigate the performance variability for large-scale web-service applications caused by diversity in *machine configuration* and *application co-runners*. This is the first work to investigate this variability in a production commercial WSC. (Section 3.1)
- **Opportunity Factor:** We introduce a novel metric, the *opportunity factor*, that quantifies the sensitivity of an application to diversity in machine configuration and application co-runners. This metric also indicates an applications potential performance improvement when mapped to execution environments where they run best. (Section 3.2)

1.4.2 Intelligently Mapping Jobs at the Cluster Level

At the cluster level, we provide an intelligent mapping approach where tasks in the WSC are placed in execution environments where they run best. Chapter 4 examines:

- **SmartyMap:** We present SmartyMap, an extension to the current WSC architecture to exploit the heterogeneity in WSCs. SmartyMap intelligently maps jobs to machines to improve the overall performance of a WSC. A required component of such an approach is the ability to score and rank job-to-machine maps. We design map scoring approaches that take advantage of the live monitoring services in modern WSCs. We also provide four map scoring policies and discuss the key trade offs between them. (Section 4.1)
- **Benefit of Intelligent Mapping:** We investigate the effectiveness of SmartyMap and demonstrate the performance opportunity of exploit-

ing the heterogeneity in those production WSCs. In addition to production datacenters, our study also includes thorough experimentation on a Google testbed composed of 9 large-scale production web-service applications and 3 types of production machines, as well as an experimental testbed composed of benchmark applications. (Section 4.2)

- **Factors Affecting EE Diversity:** We perform a careful study of how varying the diversity in applications and machine types in a WSC affects how "homogenous" or "heterogeneous" the WSC is. We find that even a slight amount of diversity in these factors can present a significant performance opportunity. Based on our findings, we then discuss the tradeoffs for server purchase decisions and show that heterogeneous WSCs may be more cost-efficient than homogeneous WSCs. (Section 4.3)

1.4.3 Online Adaptation at the Machine Level

At the machine level, we provide a mechanism to allow tasks to adapt to the execution environment in which it runs. Chapter 5 examines:

- **Lightweight Online Adaptation:** The design of the *Lightweight Online Adaptation Framework* (Loaf), a novel *lightweight* online adaptation framework for native binary applications that is both able to adapt an application to its execution environment, and also the execution environment to the application. Loaf is composed of three core mechanisms: *periodic probing* for lightweight introspection, *scenario based multiversioning* for online code restructuring, and *cross core application cooperation* for coordinated adaptation across cores. (Section 5.1)

- **Online Adaptation for Aggressive Optimizations:** Using our Loaf framework, we provide an approach that enables applications to self-select compiler optimizations based on the execution environment in which it runs. This approach, *Scenario Based Optimization*, dynamically applies aggressive optimizations (known to either improve or degrade performance) only when these optimizations are detected to beneficial. (Section 5.2)

- **Online Adaptation for Contention Detection and Response:** We present a Loaf based runtime, the *Contention Aware Execution Runtime* (CAER) environment, that is capable of instantaneous contention detection and response on real commodity machines. CAER uses two contention detection heuristics: *Burst Shutter*, and *Rule Based* techniques. When contention is detected, CAER dynamically adapts low priority applications to minimize the interference caused to higher priority applications. (Section 5.3)

1.4.4 Mitigating Interference at Cluster and Machine Levels

At both the cluster and machine levels, we provide novel capabilities in measuring and managing interference between tasks within an execution environment. Chapter 6 examines:

- **Precise Interference Prediction:** We present the design of *Bubble-Up*, a general characterization methodology that enables the *precise prediction* of the performance degradation that results from contention for shared resources in the memory subsystem. A *precise prediction* is one that provides an expected amount of performance lost when co-located. With this information, co-locations that do not violate the

QoS threshold of an application can be allowed, resulting in improved utilization in the WSC. (Section 6.1)

- **Improving WSC Utilization with Bubble Up:** Using 17 production Google workloads and production machines we demonstrate how using Bubble-Up to steer co-location decisions can significantly improve the utilization of WSCs. (Section 6.2)

- **Direct Methodology to Quantify Interference Sensitivity:** We present a general *direct* measurement technique and metric for quantifying cross-core interference sensitivity at both application and phase levels, and the design of *CiPE*, a framework that employs this measurement technique along with contention synthesis to characterize application sensitivity to cross-core interference. We also design and present four contention synthesis kernels and the core algorithms for each. (Section 6.3)

Chapter 2

Background and Related Work

Contents

In this chapter, we provide background on how tasks are placed in WSCs, the importance of quality of service (QoS) requirements in WSCs, and the implications of co-locating tasks on machines in modern WSCs. We also present a summary of the related work of this dissertation.

2.1 Task Placement in Modern WSCs

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. A *task* is composed of an 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 co-locations with other tasks.

Task placement is conducted by a cluster-level manager that is responsible for a number of servers. Based on the resource requirement, the cluster manager uses an algorithm similar to bin-packing to place each task in a cluster of machines [82]. After a task is assigned a machine, a machine level manager (in the form of a deamon running in user-mode) uses *resource containers* [8] to allocate and manage the resources belonging to the task. For the remainder of this work, we use the term *application* to represent the program binary for a given component of a web-service, and *application task* as this binary coupled with its execution configuration file. The term *job* may also be used interchangeably with *task*.

2.2 Quality of Service

As multicores become widely adopted in datacenters, the cluster manager consolidates multiple disparate tasks on a single server to improve the machine utilization. However, various application tasks in a datacenter often have different *quality-of-service* (QoS) priorities. User-facing applications for 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 requirements are not strictly defined. We define the QoS of a latency-sensitive 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, in contrast to Bing's [58, 65], which uses the quality of search results provided.

2.3 Co-location

Tasks are *co-located* when they run simultaneously on a single machine. Managing performance interference and providing task level performance isolation on the machines housed by WSCs has been a challenge due to contention for resources shared across the numerous cores on a single machine. When co-locating a latency-sensitive application with other applications on the same server, the latency-sensitive application is at risk of suffering significant QoS degradations when the co-running applications are aggressively contending for resources such as shared cache space or memory bandwidth. Due to the inability of predicting the amount of this performance interference, the current placement policy in current WSCs is to disallows any co-location of a latency-sensitive task with other tasks on the same machine to guarantee its QoS. This ad-hoc approach is a major contributor to the low utilization we find in modern WSCs.

2.4 QoS Flexibility

There is a trade-off between the QoS performance of latency-sensitive applications and the machine utilization in WSCs. When equipped with mechanisms to measure and manage interference, a small amount of QoS degradation can be allowed to increase co-locations and improve the machine utilization. If a co-location is predicted to cause less QoS degradation than a specified threshold, the cluster manager can allow the co-location. We specify the tolerable amount of QoS degradation in a *QoS policy*. For example, a 95% QoS policy indicates that we are willing to sacrifice 5% of the QoS performance to improve machine utilization. To enforce these QoS policies, the precise prediction for QoS degradation due to co-location is needed. These QoS issues, and challenges affecting machine utilization, are

revisited later in this dissertation (Chapters 5 and 6), the earlier parts of this dissertation addresses software performance.

2.5 Related Work

Warehouse-Scale Computing

Much of the related datacenter research have focused specifically on improving energy and power efficiency [1,3,29,46,84,91]. There has also been work on scheduling in the datacenter [51], enabling QoS-aware control in the datacenter [85], and programming the datacenter [14]. Other works develop tools and support for interacting and developing for the datacenter [10,114]. There has been some important work studying web search in WSCs, including the work by Barroso et al. [9], which presents an insightful look at the design and layout of an industry strength web-search datacenter. More recently, Reddi et al. [58] presented the impact of running an industry-strength search engine on a datacenter composed of Atom chips.

There are prior works that have acknowledged heterogeneity in datacenters [13, 128]. While these works focus on datacenters that provide utility computing and improving MapReduce, our work focuses on the global design and optimization of the emerging space of WSCs. In this work, we focus on WSC efficiency via improving performance and utilization. There has been a significant amount of research effort applied to the domain of heterogeneous multicore architecture. Some of the most seminal works in this area were those by Kumar et al. [66, 67]. In these work, Kumar used simulation to investigated the performance benefit of exploiting heterogeneous architectures by evaluating heuristics to dynamically schedule thread workloads based on a speedup factor. This work lead to a number of other works on scheduling for the heterogeneous multicore [64, 68, 69, 102, 107]. The work by Winter et

al. [121] is related to the numerical optimization techniques used in this dissertation. This work investigated the task of scheduling for "unpredictably" heterogeneous multicore processors due to process variation.

PROPHET provides a goal-oriented provisioning infrastructure tunes the datacenter to satisfy the needs of particular end users [122]. The work by Kazempour et al. was the first to implement changes to the hypervisor scheduler to incorporate asymmetry-awareness [61]. Another related work discusses a scheduling policy that uses a linear programming approach that maximizes system capacity to map an application across a desktop grid [4]. This work focuses on distributed desktop computers and does not consider the interaction between microarchitectural and co-location diversity.

Online Adaptation in Managed Runtimes

Current online dynamic optimization approaches can be separated into two categories: those that deal with managed run-time systems targeting bytecode and those that apply to native application binaries. The majority of online optimization frameworks that target bytecode work at the function granularity [5, 90, 111]. These optimizers detect frequently executed methods and identify them as hot. These hot methods are then *just-in-time* compiled and recompiled at higher levels of optimization, depending on how often they are executed. Other bytecode online optimization and adaptation approaches [119, 120] address memory and other issues. However this work is concerned more with online adaptation of arbitrary binaries.

Online Adaptation at the Binary Level

This work deals with the class of online optimizers and optimization frameworks that deal with native binaries directly such as Dynamo [7], DynamoRIO [12], and Strata [106]. These current dynamic optimization tech-

niques have had limited success. One of the seminal works that has inspired many future projects was the work by Bala et al. [7] on Dynamo. Dynamo is a binary to binary translator and dynamic optimizer that works at the basic block and trace levels. Dynamo was the only online optimizer of its class to achieve consistent performance gains. This has mostly been attributed to the intricacies of the PA-RISC platform for which it was implemented. Attempts have been made to reproduce this performance benefit on other architectures but have been largely unsuccessful. Bruening et al. reimplemented the Dynamo infrastructure for x86 with the DynamoRio project [12] and was unable to achieve significant improvement. A similar effort was made with the Strata [106] infrastructure and was also unable to achieve performance gains. One major challenge these three approaches face is the added overhead from virtualizing the application and maintaining control of the executing binary. In fact there has been much work focused on optimizing the dynamic optimizer itself, in particular the handling of indirect branches [49].

Research attention has also been paid to online optimization approaches using multicore architecture and novel hardware techniques. The Adore infrastructure has been used by Lu et al. [74] to achieve dynamic software prefetching via the use of helper threads and performance monitoring hardware. A similar technique was also later applied to SUN's UltraSparc Architecture [75]. Zhang et al. proposed Trident [129, 130], a new dynamic optimizer framework that requires hardware support. This work proposes that trace selection occurs entirely in hardware and uses a number of hardware extensions. This work shows promising potential, but currently cannot be applied as it depends on novel micro-architectural features to be developed.

Extracting Run-time Information

The usefulness of information about an application's run-time behavior and dynamic micro-architectural impact has also shown to be quite important. Profiling has become the cornerstone for understanding an application's behavior and can play an important part in compiler optimizations as shown in the work by Chang et al. [23]. This seminal work introduces compiler support for profile feedback directed compiler optimizations. The compiler executes the application on a number of canned inputs, profiles it, and recompiles the application using this information. Using profiling information has lead to many new kinds of optimizations [44,92,97]. However these compiler optimizations remain rigid and thus tends to be applied conservatively.

Performance counters have shown to be a great tool to enable low overhead profiling of micro-architectural events. Moreover, these hardware structures are becoming more complex as is seen in the work by Dean et al. [32]. Azimi et al. presents a technique to use limited performance counters to simultaneously profile numerous events via sampling [6]. In recent work by Cavazos et al. [18] performance counters and machine learning are used together to find better compiler optimization settings for applications. These performance counters are also being used for more than just profiling. In the works by Chen et al. [24] and our prior work [78] performance monitoring hardware are used to form dynamic hot traces without slowing down the running application. We also see performance counters used in Java VMs and JITs to steer optimization in the works by Schneider et al. [105] and Adl-Tabatabai et al. [2]

Function Cloning and Versioning

Function cloning and Multiversioning is an inter-procedural code transformation that is used by a number of optimizations dating back to the earliest

works on compiler optimization [15–17, 30, 31, 33, 118]. It was originally conceived for classic optimizations such as inter-procedural constant propagation (IPCP) [16]. It has also been been used by Carini et al. for flow insensitive IPCP [17] and Cierniak et al. for inter-procedural array remapping [30].

Multiversioning approaches have also been used by Diniz et al. [33] and Voss et al. [118]. In the work by Diniz et al. multiversioning is used in the context of a parallelizing compiler for object-based languages to provide a mechanism to dynamically switch the implementation of a particular synchronization mechanism online. Although the concept of dynamic feedback is discussed in this work, a general mechanism to achieve online code adaptation using multiversioning is not explored. In addition, the mechanisms used in this work to gather information to steer version switching is significantly limited in comparison to the *scenario based multiversioning* approach presented in this work. The multiversioning approach provided by Diniz's *dynamic feedback* relies entirely on a timing approach, only allowing for variants of the *sampling/production* phase heuristic presented in their work. However our *scenario based multiversioning* provides much more general monitoring capabilities in that our approach is guided by the ability to identify scenarios based on a collection of the information available through our *lightweight introspection* interface. In addition the idea of having a number of co-running application adapting in cooperation is also not explored.

The work by Voss et al. [118] discusses the idea of switching regions of executing code dynamically, however multiversioning is not performed statically. Versions are generated continuously by compiler tools and optimizers running on sockets and machines separate to the executing applications. In addition users of their system must learn a new domain specific language,

and the flexibility of potential adaptation policies is limited to what can be expressed in this language. Requiring this new language presents a significant amount of complexity, which is contrary to the goal of Loaf. Also the fact that a new language must be learned to use their system may further deter users from adopting this approach.

More recently mutliversioning has been used in a number of works by Fursin et al. as a mechanism to provide dynamic machine-learning testbeds for evaluating optimization configurations and performing online optimization space pruning [39, 40]. In this work we take advantage of function cloning and multiversioning to provide a general, flexible and lightweight approach to enable online code adaptation.

Contention and Interference

When two application are running on neighboring cores, contention for the shared cache can affect application *Quality of Service* (QoS) and can negatively affect overall throughput and scheduling fairness. QoS and Fairness techniques have received much research attention [48,54,55,62,83,88,89,108]. These works propose QoS and fairness models, as well as hardware and platform improvement to enable QoS and fairness be enforced. Rafique et al. investigates micro-architectural extensions to support the OS for cache management [94]. There has been a number of works aimed at better understanding and modeling cache contention [11, 19] and job co-scheduling [26, 59]. Other hardware techniques to enable cache management have also received research attention [22,52,98,113]. Suhendra [113] proposes partitioning and locking mechanisms to minimize unpredictable cache contention. Cache reconfiguration [98] has also been proposed as a mechanism to enable cache partitioning. Although these works show promising future directions for hardware and system designers to take when addressing these problems,

unfortunately current commodity micro-architectures cannot support these solutions as they do not meet the micro-architectural assumptions made these works. Another very promising direction based on what is likely to be future hardware capabilities, is to leverage core specific dynamic voltage scaling as is presented by Herdirch, Illikkal, Iyer, et al [48].

While there has been a lot of work on mitigating the performance interference due to resource contention on multicore, not much work is directly applicable to the datacenter co-location problem. Perhaps the closest related work is the Quarta work by Govindan et al. [41]; however this work requires access to physical memory addresses which can only be attained via custom changes to the OS, and as the authors themselves mention, such an approach is not feasible at user-level. One direction that has attracted much research attention is the management of shared cache and bandwidth through techniques such as resource partitioning [22,71,86,87,93,95,96,109,110,112,124], throttling [34] and adaptive cache replacement policies [57]. Previous work has also investigated providing QoS management for different applications on multicore [43,48,56,83]. While demonstrating promising results, the previous work on QoS management and resource partitioning typically requires changes to the hardware design, which is not applicable to deployed servers. Software resource partitioning has also been proposed [28,70,115]. However, most software partitioning techniques focus on shared cache, while ignoring memory bandwidth contention, which is another main cause of performance interference. In general, our work is complementary to the above resource management research. While previous work focuses on providing resource management for performance isolation or performance optimization for co-running applications, our work focus on predicting which applications can be co-run with a given application without degrading its QoS beyond a certain threshold.

Previous work on scheduling to mitigate contention and to improve cache sharing is closely related to our work [25, 38, 60, 63, 127]. For an application, different co-runners may cause different amounts of performance interference on a CMP. The intuition of many contention-aware scheduling is to classify applications based on how aggressively they are for shared memory resources and intelligently matches highly aggressive applications with not aggressive applications to minimize the performance degradation [60, 63, 132]. However, most previous work focuses on maximizing the overall throughput or maintaining performance fairness. The approaches cannot address challenges when applications have different priorities and a subset of the applications have strict requirements in terms of the tolerable QoS degradation. The challenge for scheduling to provide such QoS guarantee is that the scheduler needs to accurately predict the potential performance degradation for co-running applications. Current classifiers in contention-aware schedulers only indirectly classify or rank applications in terms of their levels of aggressiveness [63, 123, 132] or predict their potential cache misses [19, 60, 76], but cannot provide direct accurate prediction in terms of performance degradation.

Chapter 3

Execution Environments in WSCs

Contents

In this chapter, we perform a study of execution environment (EE) diversity as it exists in commercial production WSCs and its impact on large-scale commercial web-service workloads. We also replicate this study using benchmark workloads on machines spanning multiple commodity machine configurations to provide repeatable experimentation. Finally, we introduce a metric, *opportunity factor*, that, given the application mix and machine mix in a WSC, quantifies an application's sensitivity to, and potential performance improvement from, exploiting the heterogeneity in that WSC.

CPU	GHz	Cores	L2/L3	Nickname
Clovertown Xeon E5345	2.33ghz	6	8mb	**Clover**
Istanbul Opteron 8431	2.4ghz	6	6mb	**Istan**
Westmere Xeon X5660	2.8ghz	6	12mb	**West**

Table 3.1: Production Microarchitecture Mix

3.1 Diversity in Execution Environments

The potential benefit of exploiting, and adapting to, execution environment (EE) diversity within WSCs can be illustrated by the amount of performance variability suffered by applications due to varying their execution environment. Specifically, this variation includes changes in machine configurations and the set of possible co-running applications. In this section, we investigate this performance variability for large-scale commercial web-services. We focus not only on how sensitive each application's performance is to the EE diversity, but also the variance of this sensitivity across a set of applications. To investigate how our findings generalize to other applications, and to provide repeatable experimentation, we also present results using an experimental testbed composed of benchmark applications in addition to the production study.

3.1.1 WSC Test Platform

We first conducted our experiments across the three production platform types presented in Table 3.1. These three platforms are commonly found coexisting in a single WSC in Googles production fleet. The applications we use in the study are described in Table 3.2. These applications cover nine large industry-strength workloads that are responsible for a significant portion of the cycles consumed in arguably the largest web-service WSC infrastructure in the world. Table 3.2 also presents a description for each application. Each application corresponds to an actual binary that is run

workload	description	type
bigtable	A distributed storage system for managing petabytes of structured data	user-facing
ads-servlet	Ads sever responsible for selecting and placing targeted ads on syndication partners sites	user-facing
maps-detect-face	Face detection for streetview automatic face blurring	batch
search-render	Web-search frontend server, collect results from many backends and assembles html for user.	user-facing
search-scoring	Web-search scoring and retrieval	user-facing
protobuf	Protocol Buffer, a mechanism for describing extensible communication protocols and on-disk structures. One of the most commonly-used programming abstractions at Google.	user-facing
docs-analyzer	Unsupervised Bayesian clustering tool to take keywords or text documents and "explain" them with meaningful clusters.	both
saw-countw	Sawzall scripting language interpreter benchmark	both
youtube-x264yt	x264yt video encoding.	batch

Table 3.2: Production WSC Applications

in the WSC. These applications are part of a test infrastructure developed internally at Google composed of a host of Google workloads and machine clusters that have been both laboriously configured by a team of engineers for performance analysis and optimization testing across Google. Each application shown in the table operates on a repeatable log of thousands of queries from actual user activity from production. The number of cores used by each application is configured to three for both solo and co-location runs. We also use this test infrastructure in Chapter 4.

3.1.2 Microarchitectural Diversity

We first characterize the performance variability that arises due to microarchitectural diversity in WSCs. In addition to quantifying the magnitude of the performance variability, our study also aims to investigate firstly, whether microarchitectures consistently outperforms others for all applications; and secondly, the differences in how sensitive the performance of each application is to varying platform types. As we discuss later in this section,

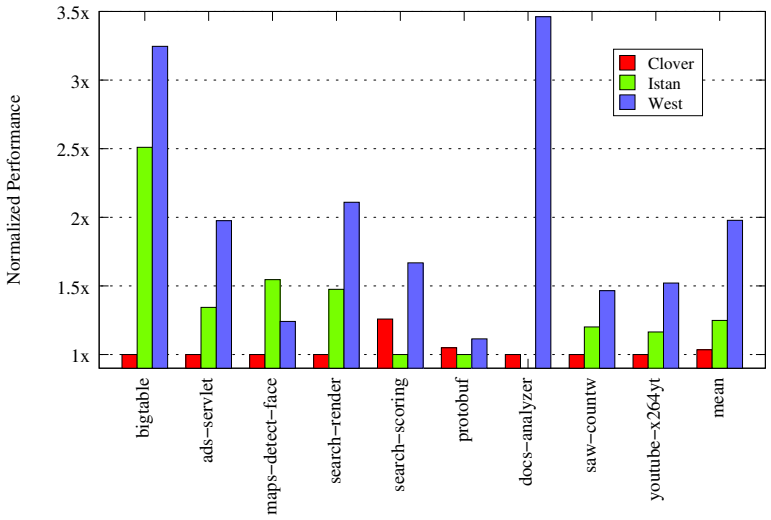


Figure 3.1: Performance comparison of key Google applications across three microarchitectures. Each cluster is normalized to poorest performing architecture (the higher the better)

the variance in performance sensitivity across all applications in a workload is an important indicator or the potential benefit from exploiting, and adapting to, EE diversity.

Figure 3.1 presents the experimental results for our Google testbed with 9 key Google applications running on 3 types of production machines.[1] The y-axis shows the performance (average instructions per second) of each application on three types of machines, normalized by the worst performance among the three for each application. As Figure 3.1 shows, even among three architectures that are from competing generations, there is a significant performance variability for Google applications. More interestingly, no platform is consistently better than the others in this experiment. Although the Westmere Xeon outperforms the other platforms for most applications, `maps-detect-face` running on the Istanbul Opteron outperforms the Westmere Xeon by around 25%. On the other hand, the Clovertown Xeon and Istanbul Opteron compete much more closely.

[1] `Docs-analyzer`'s data on Istanbul is missing because it is not configured for that particular platform.

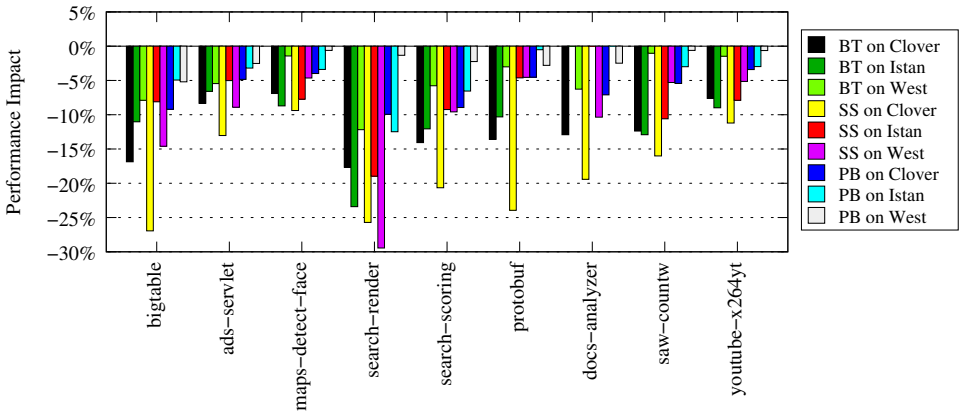


Figure 3.2: Google application performance when co-located with `bigtable` (BT), `search-scoring` (SS), and `protobuf` (PB). Negative indicates slowdown

It is also important to note that even though the Westmere Xeon platform is almost always better than the other two, the performance sensitivity to platform types vary significantly across applications, ranging from gaining only 10% speedup for `protobuf` when switching from the worst platform (Opteron) to the best (Westmere Xeon), to as large as 3.5x speedup for `docs-analyzer`. From this experiment, we conclude that diversity in *machine configurations* present in modern WSCs has a significant impact on application performance.

3.1.3 Co-Runner Diversity

Figure 3.2 illustrates the performance variability due to co-location interference for Google applications. This figure shows the performance interference of each of the 9 Google applications when co-locating with another application: `bigtable` (BT), `search-scoring` (SS) and `protobuf` (PB). The y-axis shows the performance degradation of each benchmark when co-located on each platform. We calculate this degradation using the application's execution rate when co-located normalized to the execution rate when it is running alone on that platform. The lower the bar, the more severe the

performance penalty. We observe that the same co-runner causes varying performance penalties to different applications. We observe a performance degradation ranging from close to no penalty, 2% or less in some cases, to almost 30% .

More interestingly, the variability in co-location penalty is not an isolated factor. It is also complicated by the diversity in microarchitectures. For each application shown in the figure, a single co-running application may cause varying performance penalties as the underlying microarchitecture changes. It is important to note that microarchitectural diversity on average has a more significant performance impact than co-location diversity. The performance variability due to microarchitectural heterogeneity is up to 3.5x; while the there is generally less than 30% performance degradation due to co-location. However, the relative impact of the two depends on applications. For some applications (e.g `protobuf`), co-running diveristy has a greater impact than machine diversity. From this experiment, we conclude that diversity in *co-running tasks* present in modern WSCs has a significant impact on application performance.

3.1.4 Motivation for Intelligent Mapping

As we demonstrate in Chapter 4, this variability in performance sensitivity (various speedup ratios) impacts how job placement decisions should be made to maximize the overall performance. This variability is indeed indicative of the amount of potential performance improvement achievable by intelligent mapping. For a WSC composed of limited number of each microarchitecture, the overall performance can be maximized by a smart job manager that prioritizes mapping applications with higher speedup ratios to faster machines. For example, as shown in Figure 3.1, to achieve the best overall performance, `docs-analyzer` or `big-table` should be priori-

CPU	GHz	Cores	L2/L3	Memory
Core i7 920	2.67ghz	4	8mb	4gb
Core 2 Q8300	2.5ghz	4	4mb	3gb
Phenom X4 910	2.6ghz	4	6mb	4gb

Table 3.3: Experimental Microarchitecture Mix

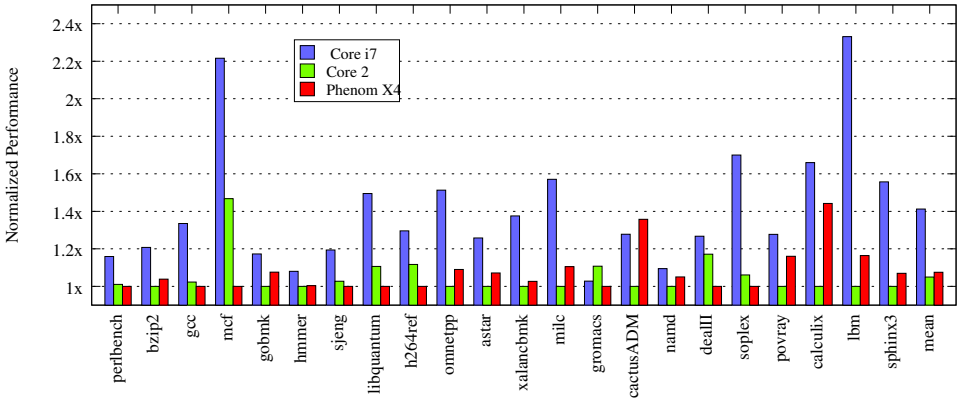


Figure 3.3: Performance comparison of benchmark workloads across three microarchitectures.

tized to use the Westmere Xeon over `protobuf`. Furthermore, as we show in Figure 3.2, when exploiting EE diversity in WSCs to perform better job-to-machine mapping, there may be a compounding benefit to consider both machine and co-runner diversity simultaneously. In Chapter 4, we provide a technique to exploit this opportunity, and delve into more details as to the causes of this performance variability.

3.1.5 Benchmark Testbed

To investigate how our findings using Google's infrastructure generalize to other application sets, and to provide experimental results that are repeatable without requiring access to Google's internal infrastructure, we replicate our study in an experimental benchmark testbed. In our experimental infrastructure we use a spectrum of 22 SPEC CPU2006 benchmarks on their `ref` input as our application types and three state-of-the-art microarchitectures

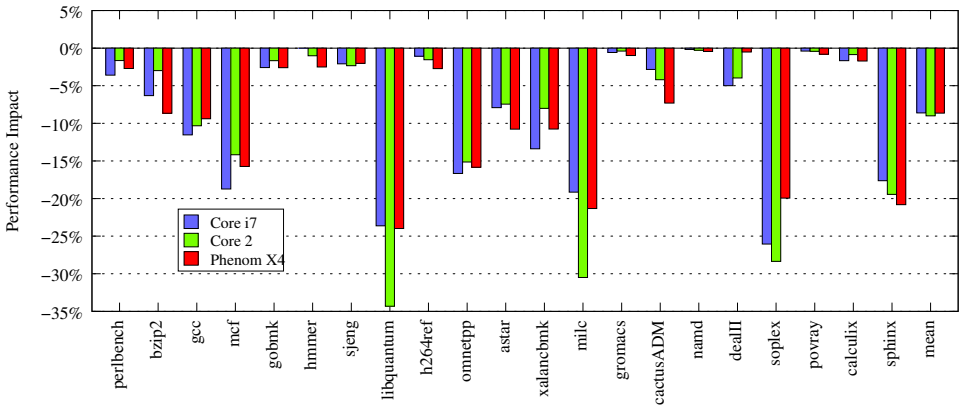


Figure 3.4: Benchmark slowdown when co-located with `lbm`.

as our machine types running Linux 2.6.29. The underlying microarchitectures of these three machine types are presented in Table 3.3. All application types are compiled with GCC 4.5 with O3 optimization. The results are presented in Figures 3.3 and 3.4. Similar to Figures 3.1 and 3.2, Figures 3.3 and 3.4 present each benchmark's normalized performance when running on varying hardwares, and its performance degradation when co-located with `lbm`, respectively. These two figures demonstrate that the observations of performance variability of Google applications can be generalized to SPEC benchmarks on the machine types used in our benchmark testbed. In addition, applications present various levels of performance sensitivity to such EE diversity. This indicates that the homogeneity assumption may leave a large performance opportunity untapped within our benchmark testbed also.

3.1.6 Implications on WSC Design

The findings of the studies presented in this section shows that EE diversity has a significant impact on the performance of tasks as they run in modern WSCs. However, currently, this EE diversity is ignored by the system design of WSCs. Tasks are not placed within the EE where they run best, tasks can

not adapt to its EE, and current systems can not measure and manage EE events such as the interference between tasks. This dissertation argues that exploiting, and adapting to, the EE diversity in modern WSCs is critical for a highly efficient WSC design.

3.2 An Opportunity Metric for EE Diversity

An important concept arises from the previous section. Depending on how "immune" or "sensitive" an application is to microarchitectural and co-runner variation, each application would benefit differently from a job mapping policy that takes advantage of EE diversity. We introduce a metric, *opportunity factor*, that approximates a given application's potential performance improvement opportunity relative to all other applications, given a particular mix of applications and machine types. The higher the opportunity factor, the more sensitive an application to diversity in the WSC. Note that this opportunity factor can be calculated only when the application mix and the machine mix are known.

For a given WSC, we can denote the application of type i as A_i, and the microarchitecture of type j as M_j. We define the speedup factor for A_i as:

$$SF_{A_i} = \frac{max_{j,k}\{IPS_{A_i,M_j,C_k}\} - min_{j,k}\{IPS_{A_i,M_j,C_k}\}}{min_{j,k}\{IPS_{A_i,M_j,C_k}\}}, \qquad (3.1)$$

where IPS_{A_i,M_j,C_k} is application A_i's IPS (instruction per second) when it is running on machine M_j with a set of co-runners C_k. The SF_{A_i} is essentially the amount of performance variability of A_i in all possible configurations of the execution environment, composed of the cross product of all machine options and co-runner options. Using SF_{A_i}, we can define the *Opportunity*

Factor (OF) for A_i as:

$$OF_{A_i} = \frac{SF_{A_i}}{\sum_j SF_{A_j}} \tag{3.2}$$

OF_{A_i} represents the sensitivity of each application type to the overall diversity of a given application mix, relative to all other applications. This metric allows datacenter designers, operators and reliability engineers to reason about the performance improvement potential of various applications in the datacenter and identify applications that are most likely to benefit from intelligent job mapping. We present and discuss OF results for both Google and benchmark testbeds in Section 4.2.4.

Chapter 4

Mapping Jobs to Diverse Execution Environments

Contents

In this chapter, we address *the homogeneous assumption* at the cluster level by providing a mechanism to automatically and continuously learn the execution environments tasks prefer, and intelligently map tasks where they run best. We then evaluate this approach in both our commercial production environment, and our benchmark testbed as presented in Chapter 3. Finally, we perform an investigative analysis of the key factors impacting the performance opportunity of exploiting EE diversity in WSCs, and the implications on machine purchasing strategies for populating WSCs.

4.1 Opportunistic Mapping with SmartyMap

In this section, we discuss how heterogeneity in WSCs can be exploited to improve the cost efficiency of WSCs by improving overall performance. We first present the overview of our approach. We then present the formulation of the problem of mapping a set of jobs to the heterogeneous WSC as an optimization problem. Finally, we describe our mapping approach, *SmartyMap*, and present four mapping policies.

4.1.1 Overview of SmartyMap

SmartyMap utilizes continuous profiling information provided by the monitoring service commonly found in WSCs, such as the Google Wide Profiler (GWP) [100], to intelligently map jobs to machines. Figure 4.1 illustrates how *SmartyMap* is integrated in a WSC system to exploit EE diversity. We formulate the problem of mapping jobs to machines as a combinatorial optimization problem and thus the main component of SmartyMap is an optimization solver (Section 4.1.2).

A key requirement for *SmartyMap* and especially for the optimization solver is the evaluation and comparison of mapping decisions. As illustrated

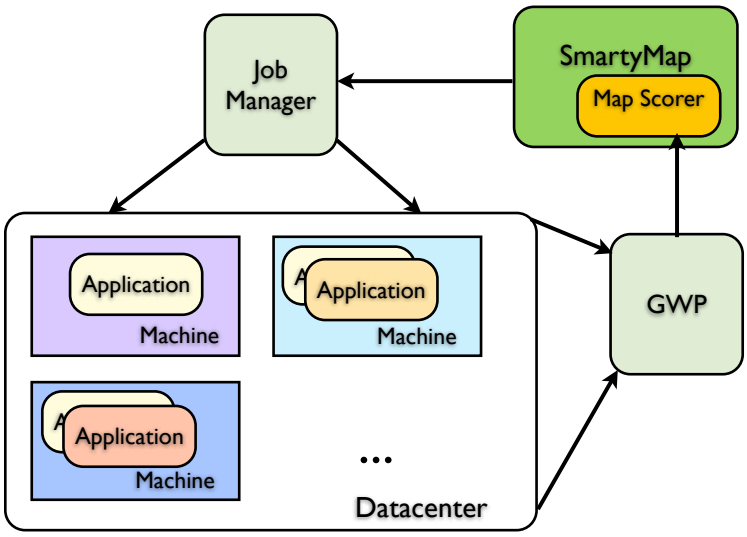


Figure 4.1: The Overview of SmartyMap

in Figure 4.1, *Map Scorer* utilizes GWP to perform such evaluation. GWP continuously monitors and profiles jobs as they run in production WSCs, and archives the profiles in a database. *Map Scorer* extracts information from the GWP database to build and continuously increment an internal representation of profiles, keeping track of the performance of each application in various execution environment. Using these profiles, *Map Scorer* compares various mapping decisions, and identifies better mappings based on the historical data describing how well a job performs in a given environment. It is important to note that, during the optimization process, instead of actually mapping jobs to various execution environments to measure its performance variability and identify the best mapping (which would be extremely costly), our approach utilizes the historical profiling data stored by GWP to simulate and score mappings. The presence of a continuous monitoring service, in this case GWP, is the key to making *SmartyMap* feasible in a production.

It is important to make the distinction between the costs associated with populating the GWP database (referred to later as profiling complexity),

which occurs continuously through the lifetime of operation of the WSC, versus the cost of utilizing the information in GWP's database to search for the optimal mapping, which is often in the order of minutes for a typical scale of thousands of machines and dozens of application types. There are tradeoffs between the amount of profiling and the accuracy of estimating and comparing mapping decisions based on the profiling. These trade-offs are discussed in detail in Section 4.1.3.

4.1.2 Mapping an Optimization Problem

As mentioned earlier, we formulate the problem of mapping jobs of different types and characteristics to a set of diverse execution environments as a combinatorial optimization problem. The optimization objective is to maximize the overall performance of the entire WSC, e.g. the aggregated instruction-per-second (IPS) of all jobs. Focusing on job mappings that improve overall performance by formulating it as an optimization problem is especially suitable for modern WSCs due to the unique properties of this computing domain: the set of important web-service applications are known and fairly stable; the main web-service jobs are often long running jobs; and migrations of jobs rarely happens because of the high cost.

The core algorithm (Algorithm 1) used by *SmartyMap* to solve the optimization problem is based on traditional iterative optimization techniques [99, 103, 121]. We use a stochastic hill climbing algorithm. First a random mapping is generated, then for each iteration that mapping is perturbed with a random swapping between two job placements. If the new mapping is better (determined by a scoring function) this mapping is kept, otherwise the jobs are "unswapped." This search process continues for a given period of time, controlled by the parameter *optimization timer*. Remember, these mappings are simulated internally by SmartyMap using

Algorithm 1: Core Optimization Algorithm

```
   Input: set of free machines and available jobs
   Output: an optimized mapping
 1 while free machines and available jobs do
 2 |   map random job to random machine;
 3 end
 4 set last_score to the score of current map;
 5 while optimization timer not exceeded do
 6 |   foreach machine do
 7 |   |   foreach job on that machine do
 8 |   |   |   swap job with random job on random machine;
 9 |   |   |   set cur_score to the score of current map;
10 |   |   |   if mapping score is better then
11 |   |   |   |   set last_score to cur_score;
12 |   |   |   else
13 |   |   |   |   swap jobs back to original placements;
14 |   |   |   end
15 |   |   end
16 |   end
17 end
```

scores (Section 4.1.3) calculated from GWP information. When the simulated mappings converges to optimal the optimized mapping is then used to steer the actual mapping of jobs to machines.

4.1.3 Map Scoring

A *score* of a particular placement of a job to a machine is used to measure how good the placement of a job is. To score an entire map of jobs to machines we use the sum of all of the placement scores. The higher the score, the better the map. The scoring policy is an essential part in *SmartyMap*. It is used in each optimization iteration to compare mappings. In this work, we present and evaluate a number of scoring policies that vary in the required profiling necessary to generate the score. Table 4.1 shows the descriptions and profiling complexities for our map scoring policies, where $|A|$ corresponds to the number of application types, $|M|$ corresponds to the

Approach	Description	Complexity
Smarty-C	**Co-location Score:** This score is based only on co-location penalty and only requires profiling the co-location penalty on any type of machine. Once a co-location profile is collected it is then used to score that co-location regardless of the underlying microarchitecture.	$\|A\|^n$
Smarty-Cs	**Co-location Score (Smart):** This score is based on co-location penalty with microarchitecture specific information. Information about co-location penalty must be collected for all platforms of interest.	$\|A\|^n \times \|M\|$
Smarty-M	**Microarchitectural Affinity Score:** This score is based on microarchitectural affinity and captures only the speedup of running each application on one microarchitecture over another.	$\|A\| \times \|M\|$
Smarty-MCs	**Microarchitectural Affinity and Co-location Score:** This scoring method includes both microarchitectural affinity and microarchitecture specific co-location penalty. This scoring technique has the heaviest profiling requirements.	$\|A\|^{n+1} \times \|M\|$

Table 4.1: Mapping Scoring Policies

number of machine types, and n corresponds to the number of co-runners allowed on each machine. The profiling complexity indicates the amount of profiling the *Map Scorer* needs from GWP. For example, among all policies, Smarty-M requires the smallest amount of profiling, $|A|$ x $|M|$, indicating that the scorer only needs performance profiles of each application type on each machine type from GWP, without the need of knowing the application's co-runners when the profiling was conducted. In a practical setting of a WSC, $|A|$ is in the order of magnitude of 10s – 100s, $|M|$ is often less than 10 and n is often only 1 or 2 as typically only one or two major web-service jobs, in addition to several low-overhead background processes such as logsaver, are co-located on a given machine in a WSC.

The accuracy of the scoring policy determines the mapping quality of *SmartyMap*. Smarty-MCs has the complete information thus could provide

the best result. Meanwhile, Smarty-M, Smarty-C and Smarty-Cs require less time for GWP to collect all needed information. However, they are also less accurate, thus may lead to suboptimal results. The trade off is between the amount of available profiling information and maximizing the performance gain. In addition, the landscape of diversity present in the WSC has a significant impact on the usefulness of some profiling information. In Section 4.3 we further investigate the factors that impact the diversity and discuss the selection of the appropriate map scoring policies.

The complexity of *SmartyMap* is decided by both the profiling complexity of scoring policies and the computation complexity of the optimization solver. However, as we mentioned before, GWP continuously profiles in the background through the lifetime of a WSC and its cost is thus hidden from *SmartyMap*. *SmartyMap* simply utilizes the profiling information available at any given time and keeps on updating its performance profiles based on the newly accumulated profiling data collected by GWP to continuously improve its scoring and thus the mapping decisions. On the other hand, the complexity of using our optimization solver based on the map scores to search for the optimal mapping is relatively low, typically in the order of minutes.

4.2 The Benefit of SmartyMap

We first evaluate *SmartyMap* using our production and benchmark testbeds and investigate the amount of performance improvement gained by exploiting the EE diversity in a given WSC over the status-quo. In addition, we compare four different scoring policies of *SmartyMap* to gain insights on the impacting factors of the potential performance benefit.

4.2.1 Goals and Methodology

The goal of this evaluation is to quantify and demonstrate the performance opportunity when taking advantage of the EE diversity in a WSC. We accomplish this by measuring the performance improvement when using our diversity-aware approach, *SmartyMap*, over the diversity-oblivious mapping. We also evaluate the performance of four different scoring policies discussed in Section 4.1.3. Comparing various scoring policies allows us to identify the performance improvement provided by considering microarchitecture or co-location in isolation, which sheds light on the important factors that impact the amount of EE diversity. In addition to the overall performance of an entire WSC, we present the application-level performance achieved by *SmartyMap* and compare it with the estimation provided by the *opportunity factor* discussed in Chapter 3.2.

We conduct thorough investigation and evaluation in three domains, using Google and benchmark testbeds as well as production data running live web-services in the field. For experimentation using Google and benchmark testbeds, we use platform types previously presented in Tables 3.1 and 3.3 along with the 9 Google key applications and 22 SPEC CPU2006 benchmarks, respectively.

For our testbed evaluation, we construct an oracle based on comprehensive runs on real machines. Given a map of jobs to a set of machines this oracle reports the performance of that mapping. To construct this oracle, we run all combinations of co-locations on all machine platforms and collect performance information. This performance information is in the form of instructions per second (IPS) for each application in every execution environment. Using this information we construct a *knowledge bank* that is used as a reference for the performance of a particular event in a given WSC. We have validated that all runs are repeatable in our Google and Benchmark

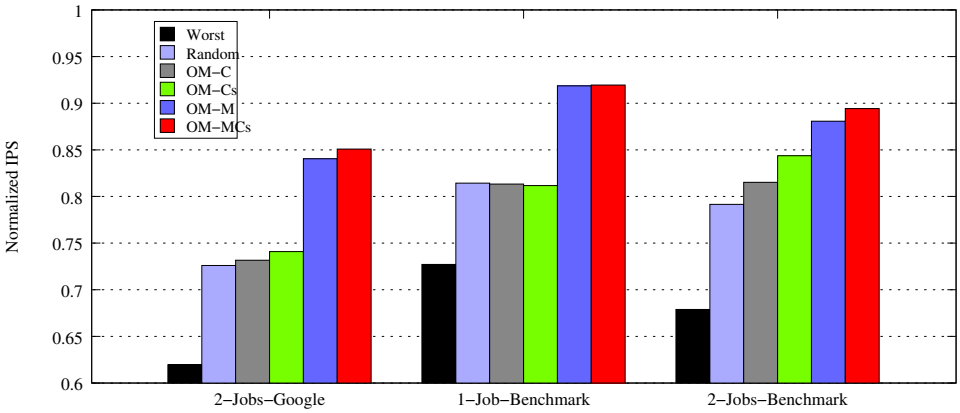


Figure 4.2: Opportunistic Mapper compared to random and worst cases. (higher is better)

testbeds (performance variation of 1% on average). Given a job-to-machine mapping, we use the knowledge bank as the oracle to calculate the aggregate performance of the entire WSC composed of various machine types. The knowledge bank is used in modeling GWP and provides partial profiling information that the map scorer needs. In this case, depending on the scoring policy, partial information (such as only machine diversity or co-location diversity) is calculated from the knowledge bank using sample runs to populate GWP's databased at various levels of profiling complexity.

4.2.2 Overall IPS

In Figure 4.2 we compare our *SmartyMap*, the random mapper and the worse case mapper for overall performance of a WSC. We first use the aggregated instructions per second (IPS) of the entire WSC as our performance metric. The experiments shown in this figure are conducted on the Google testbed (1st cluster of bars) and the benchmark testbed (2nd and 3rd clusters of bars). The y-axis shows the normalized overall performance (IPS) of a WSC when using various job-to-machine mapping policies. To calculate the normalized IPS performance of an entire WSC for a given job-to-machine mapping, we aggregate the average IPS of all jobs. The normalization base-

line for each cluster of bars is the sum of the average IPS of each job when it is run alone on its best performing machine type. The higher the bar is, the better the IPS performance.

The first cluster of bars presents results for the Google testbed. In this experiment, the testbed WSC is composed of 500 machines with 1000 jobs running; two jobs are co-located on each machine. We choose the 2-Jobs scenario because typically only one or two major web-service jobs are co-located on a given machine in a WSC. The machine composition and workloads of the WSC are randomly generated from the three machine types shown in Table 3.1 and 9 key Google applications shown in Table 3.2. Each bar in the cluster presents the performance for the worst mapping, random mapping, as well as *SmartyMap* using four varying scoring policies as discussed in Section 4.1.3. Similarly, the second and third clusters present results for benchmark testbed. For the 1-Job scenario, there are 500 jobs running in a WSC composed of 500 machines, with only one job running on each machine; while the 2-Jobs scenario has 1000 jobs running on 500 machines. The machine composition and workloads of the WSC are also randomly generated using the three machine types from our benchmark testbed (Table 3.3) and SPEC CPU2006 suite.

In Figure 4.2 we observe a significant benefit from using *SmartyMap* for the Google testbed experiment. Among the four scoring policies of *SmartyMap*, we achieve the best performance when considering both machine and co-location diversity (Smarty-MCs), which improves the overall normalized IPS of the entire WSC by 18% over the random mapping (from 0.72x to 0.85x) and 37% over the worst case mapping. Also, in this experiment, Smarty-M performs comparably well as Smarty-MCs. This indicates that there is a significant performance benefit to consider the machine diversity. Meanwhile, when only co-location effects are considered to score maps

(Smarty-C and Smarty-Cs), we observe less overall performance gains. It is within 1-2% of the random mapping result. However, note that the random mapping already greatly improves the IPS over the worst case, by around 17%. This is because Smarty-C and Smarty-Cs focuses only on the performance impact of resource contention between co-located applications on the same machine. When the workload is a fairly balanced mix of contentious (memory-intensive) applications and non-contentious (CPU intensive) applications, randomizing the mapping can effectively decrease the chance of co-locating two contentious applications, and in turn improve over the worst case by reducing a significant amount of co-location penalties. These results indicate that for the Google workloads and production machine mix in our testbed, exploiting the machine diversity may have a bigger impact than considering co-location diversity alone. However the relative importance of machine and co-location diversity depends on the machine/workload mix. We explore those impacting factors in greater detail in Section 4.3.1.

The results for the benchmark testbed, shown as the second and third clusters of bars, are in general consistent with the Google testbed results. For the 1-Job scenario in the benchmark testbed, as we expect, Smarty-C and Smarty-Cs do not improve performance over random mapping. And Smarty-M and Smarty-MCs perform equally well. This is because there is no co-location in a 1-Job scenario. The performance improvement of *SmartyMap* using Smarty-MCs over the worst case mapping is 26% and close to 14% over random mapping. For the 2-Jobs scenario (the 3rd cluster) we observe that scoring policies that only consider co-location diversity (Smarty-C, Smarty-Cs) are quite effective, generating up to an 8% improvement over random mapping. This is better than the performance of Smarty-C in 2-Jobs scenarios for Google applications, demonstrating that the effectiveness of Smarty-C depends on the machine/workload mix. Only consider-

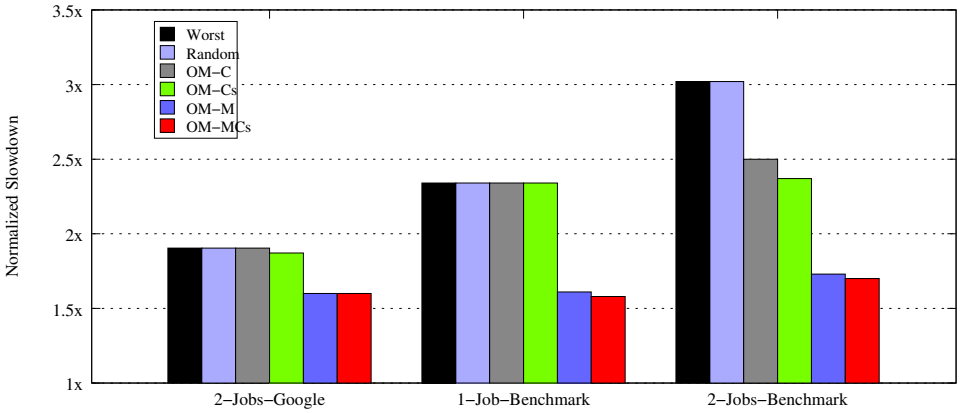


Figure 4.3: Mapping policy's impact on latency to complete all jobs. (lower is better)

ing microarchitectural diversity without considering co-location (Smarty-M) can produce a 12% performance benefit over the random mapping, higher than Smarty-C. When *SmartyMap* combines both machine diversity and co-location penalty diversity (Smarty-MCs), the performance improvement is increased to about 16%.

4.2.3 Latency

In addition to the aggregated IPS, we also compare the latency of all jobs in a WSC, defined as the execution time of the longest-running job under a given job-to-machine mapping. Figure 4.3 shows the latency of various mapping policies, normalized to the latency when all jobs run alone on their best performing machine type. Interestingly, although the random mapping can improve the average IPS performance, it performs equally as poorly as the worst mapping for improving latency. In this experiment our *SmartyMap* improves the job placement of the slowest job resulting in lower overall latency. Again, Smarty-MCs performs the best, and Smarty-M performs comparably well.

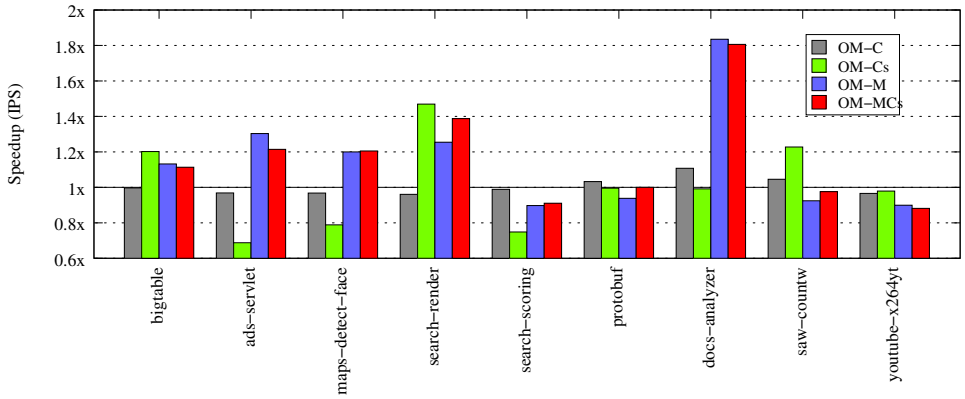


Figure 4.4: Speedup at the application level. (Google)

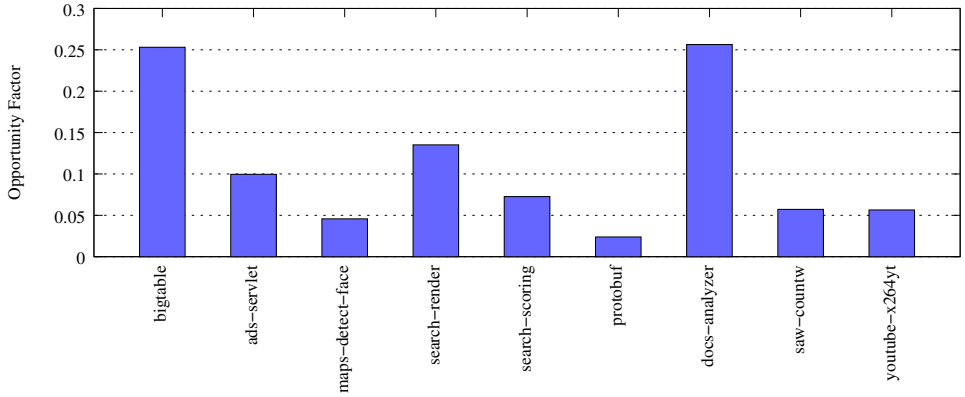


Figure 4.5: Opportunity factor of each application. (Google)

4.2.4 Impact at the Application-level

We further examine the performance improvement of each application and compare it with the estimation by the *opportunistic factor* (Section 3.2). Figure 4.4 presents the performance improvement at the application level from the 2-Jobs scenario with 500 machines and 1000 applications for Google testbed as in Figures 4.2 and 4.3. The y-axis shows each application type's performance using *SmartyMap*, normalized to each type's average performance in a random mapping. This figure demonstrates that application types have varying amounts of performance benefit from *SmartyMap*. For example, while there is a 16%-19% performance improvement overall, `docs-analyzer`, which is sensitive to both microarchitectural and co-location diversity, achieves a 80% performance improvement over random mapping. There are also applications that suffer performance degradation. However, as shown in the figure, the performance improvement greatly outweighs these degradations. Figure 4.5 presents the opportunity factor (OF) of each application, calculated using Equation 3.1 and Equation 3.2 in Section 3.2. As the corresponding Figures 4.4 and 4.5 show, OF correctly identifies the top applications that benefit from *SmartyMap* including `docs-analyzer` and `search-render`. However not all of the application-level opportunity is realized. Remember that mapping to exploit EE diversity in WSCs is a constraint optimization problem. As a result, not all applications can be mapped to their individual optimal situations to achieve the maximum performance improvement. For example, `docs-analyzer` has a slightly better OF than `bigtable` and they both prefer Westmere platform, so as the "preferred" Westmeres in a WSC are consumed by `docs-analyzer`, `bigtable`'s mapping options are reduced. As shown in Figures 4.6 and 4.7, we observe similar results for the applications in our benchmark testbed.

Both *SmartyMap* and the *opportunistic factor* (OF) rely on GWP pro-

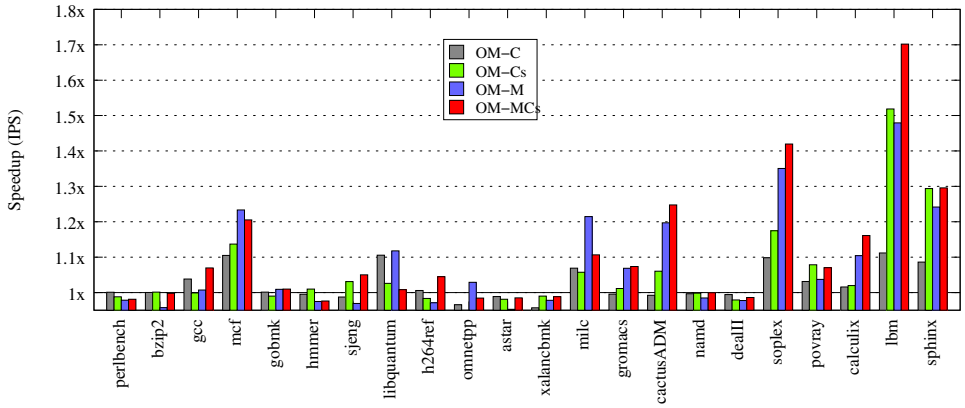


Figure 4.6: Speedup at the application level. (Benchmark)

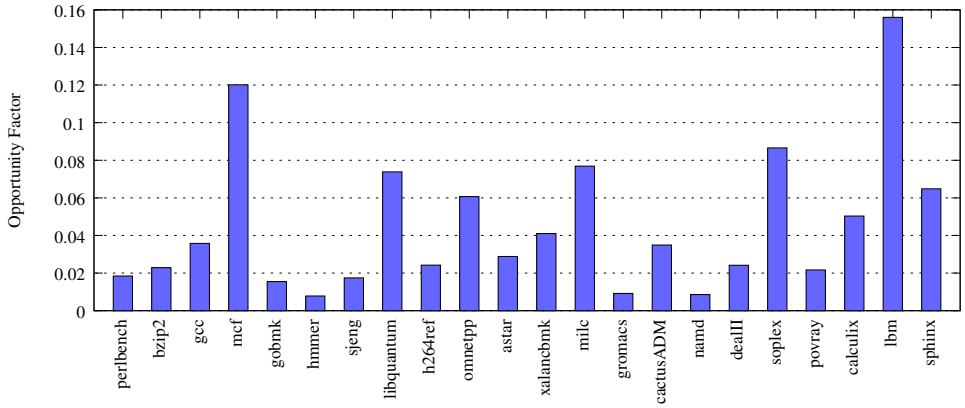


Figure 4.7: Opportunity factor of each application. (Benchmark)

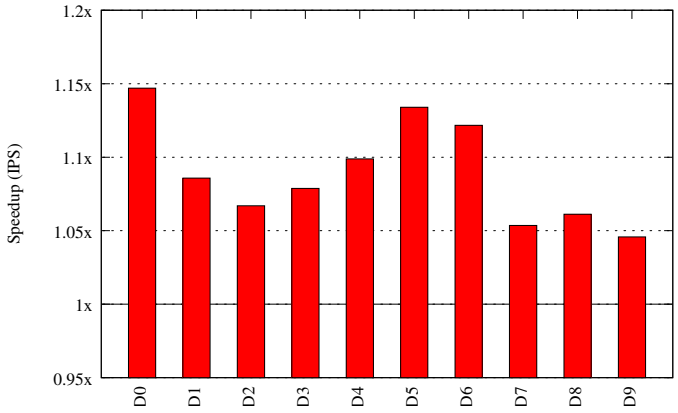


Figure 4.8: Performance improvement from *SmartyMap* over the currently deployed mapper in production

filing information. And both of their accuracy is determined by the amount of profiling. However they have different focuses. OF focuses on each individual application. It summarizes the amount of performance variability an application has due to EE diversity, relative to the rest of the workloads; and it indicates the maximum amount of performance improvement the application can benefit from intelligent mapping. On the other hand, *SmartyMap* often focuses on improving the overall performance of a WSC. OF helps identify applications that can have significant performance improvement and can be used by WSC operators to make mapping decisions on top of *SmartyMap*. For example, operators may realize the potential of `bigtable` using OF, and thus prioritize its mapping by inserting special rules in *SmartyMap*.

4.2.5 SmartyMap in the Wild

D0	D1	D2	D3	D4	D5	D6	D7	D8	D9
4	3	2	3	2	3	2	5	2	2

Table 4.2: Number of Machine Types in Production WSCs

Lastly, we employ *SmartyMap* on live production GWP profiles to study the potential performance improvement when exploiting EE diversity in the

wild for live production WSCs. We conducted our evaluation in 10 randomly selected WSCs in Google's production fleet. These WSCs present various levels of machine diversity. We have collected detailed Google-Wide Profiling (GWP) [100] profiles of around 100 job types running across these WSCs consisting of numerous machines in the wild. These jobs span most of Google's main products, including web-search. Using the GWP profiles, we conducted a postmortem *SmartyMap* analysis to re-map jobs to machines and calculate the expected performance improvement. To do that, firstly, instructions per cycle (IPC) samples are derived from GWP profiles. We use *cycle* and *instruction* samples collected using hardware performance counters by GWP over a fixed period of time, aggregated per job and per machine type. IPS (instructions per second) for each application on each machine type is then computed by normalizing the IPC by the clock rate. These IPS samples are used for map scoring. Here we use Smarty-M policy, considering only microarchitectural diversity.

Using *SmartyMap*, we produce an intelligent job-to-machine mapping in a matter of seconds at the scale of thousands of machines of multiple types, over a hundred job types and the performance profiles of over the course of a month of operation. Figure 4.8 shows the calculated performance improvement when using *SmartyMap* over the currently deployed diversity-oblivious mapping in 10 anonymized Googles active WSCs. Although some major applications are already mapped to their best platforms through manual assignment, we have measured significant potential improvement of up to 15% when intelligently placing the remaining jobs. The performance opportunity calculation based on this section is now an integral part of Google's WSC monitoring infrastructure. Each day the number of 'wasted cycles' due to inefficiently mapping jobs to the WSC is calculated and reported across each of Google's WSCs world wide.

It is important to note that each of these 10 WSCs have varying machine and application mixes. As shown in Table 4.2, the amount of performance opportunity does not simply depend on the number of machine types in the WSC. For example, D6 has 12% performance improvement although it is composed of only 2 machine types. However, while D7 is composed of 5 machine types, its performance improvement is only around 5%. Instead, it is the diversity along both machine and application mixes that drastically impacts the performance potential. In the next section, we further explore how these factors impact the amount of diversity and the resulting performance opportunity in WSCs.

4.3 Factors Impacting EE Diversity

The rationale behind the homogeneity assumption stems from a lack of understanding on how the gradual introduction of diversity in a WSC impacts performance variability. In this section, we perform a study of how varying the diversity in a WSC affects the performance opportunity from the diversity available in the WSC along two dimensions, application mix and machine platform mix. We then present insights into how these two factors affect server purchase options as well as the selection of the appropriate map scoring policies.

4.3.1 Impact of Workload Mix on EE Diversity

In this section, we evaluate a variety of workload mixes to investigate how workload mixes impact the performance improvement when exploiting EE diversity. We partitioned our 9 Google applications into two types, memory intensive (and thus likely to be contentious) and CPU intensive. We also selected the top 8 memory intensive applications and the top 8 CPU inten-

Workload	Application Types
Google Mostly Mem	bigtable, ads-servlet, search-render, docs-analyzer
Google Mostly CPU	maps-detect-face, search-scoring, protobuf, saw-countw, youtube-x264yt
Memory	lbm, libquantum, mcf, milc, omnetpp, soplex, sphinx, xalancbmk
CPU	hmmer, namd, povray, h264ref, gobmk, dealII, sjeng, perlbench
Mix ($\frac{1}{2}$Mem/ $\frac{1}{2}$CPU)	lbm, libquantum, mcf, milc, hmmer, namd, povray, h264ref
Mostly Mem ($\frac{3}{4}$Mem/$\frac{1}{4}$CPU)	lbm, libquantum, mcf, milc, omnetpp, soplex, hmmer, namd
Mostly CPU ($\frac{3}{4}$CPU/$\frac{1}{4}$Mem)	hmmer, namd, povray, h264ref, gobmk, dealII, lbm, libquantum

Table 4.3: Workload Mixes

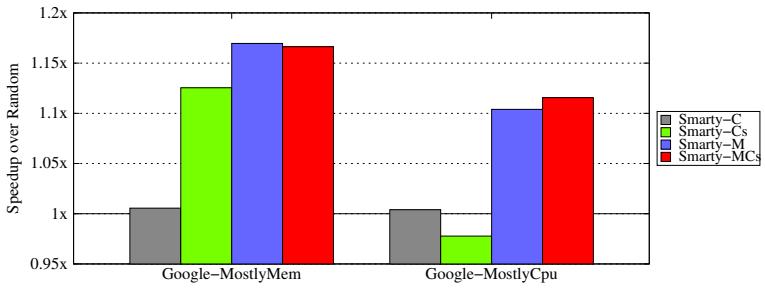


Figure 4.9: Impact of varying workload mix on available EE diversity for Google testbed. Performance is normalized to random mapping. (higher is better)

sive applications from SPEC 2006. As shown in Table 4.3, we constructed 7 types of workloads using our classification. We then conducted various job mapping experiments on these workloads to investigate the performance benefit of using *SmartyMap* over the random mapping. All experiments on the Google testbed use a WSC of 500 machines evenly distributed from 3 machine types listed in Table 3.1 (166 Clovertown Xeon, 166 Istanbul Opteron, 168 Westmere Xeon). Similarly, the benchmark testbed experiments use 400 machines composed of 3 types of microarchitectures listed in Table 3.3 (133 Core i7s, 133 Core 2s, 134 Phenom X4s).

Figures 4.9 and 4.10 present our experimental results for the Google and benchmark testbeds, respectively. We conducted a 2-jobs-per-machine

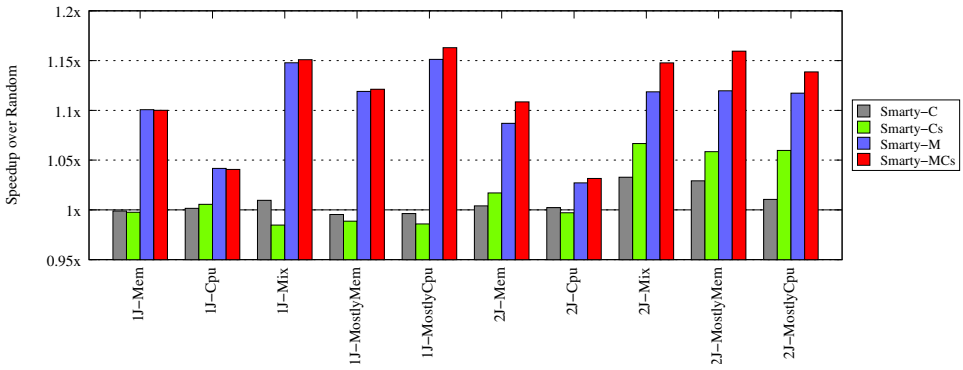


Figure 4.10: Impact of varying workload mix on available EE diversity for SPEC benchmark testbed. Performance is normalized to random mapping.

experiment using the Google testbed and both 1-Job and 2-Jobs scenarios for the benchmark testbed. In each figure, the x-axis shows each experiment's configurations. For example, in Figure 4.10, the notation 1J-MostlyCPU indicates the 1-job-per-machine scenario and the workload is composed of $\frac{3}{4}$ CPU intensive benchmarks and $\frac{1}{4}$ memory intensive benchmarks. The y-axis shows SmartyMap's performance improvement over the random mapping. The performance metric is the overall aggregated IPS of all machines. As the figures show, the amount of performance benefit of using *SmartyMap* to take advantage of EE diversity varies when the workload mix varies. Specifically, we have the following observations and insights.

1. The performance benefit potential is smaller for CPU intensive workloads than memory intensive workloads or mixed workloads. Figure 4.9 shows that for Google experiments, the workload of mostly CPU intensive applications achieves a little over 10% improvement over the random mapping, as opposed to close to 15% for memory intensive workloads. In Figure 4.10, both 1J-CPU and 2J-CPU experiments have relatively low performance improvement (less than 5%). This indicates that for CPU intensive benchmarks, the microarchitectural diversity is smaller. For our workloads and the 2 sets of microarchitec-

tures (Tables 3.1 and 3.3), **much of the performance variability and opportunity are in the diversity of memory subsystem design**.

2. In general, more diverse workloads, such as workloads composed of both CPU and memory intensive benchmarks, have higher performance improvement potential for using *Smartymap* than workloads composed of pure CPU or pure memory intensive benchmarks. For example, in Figure 4.10, for the 1-Job scenarios (left half of the figure), Smarty-MCs has more performance improvement over the random mapping for 1J-mix (15%) than 1J-CPU (3%) or 1J-Memory (10%). Similarly, for the 2-Jobs scenarios (right half of Figure 4.10), when the workload is composed of only CPU intensive benchmarks (2J-CPU), the performance improvement is much smaller (4%) than that for 2J-mix (14%), which has a more diverse workloads.

3. Considering machine diversity only (Smarty-M) is fairly competitive with considering both machine and co-location heterogeneity (Smarty-MCs) in most scenarios. On the other hand, considering co-location only (Smarty-Cs) does not outperform considering machine diversity only (Smarty-M) in any scenario. One reason is that for both our Google and benchmark testbeds, the performance variability due to microarchitectural diversity is as high as 3.5x and 2x, respectively, while the performance variability due to the penalty of co-locating two jobs is only around 30% (Figures 3.1 and 3.2). However, in the next section we will further investigate the performance difference between Smarty-MCs and Smarty-M when the amount of microarchitectural diversity changes.

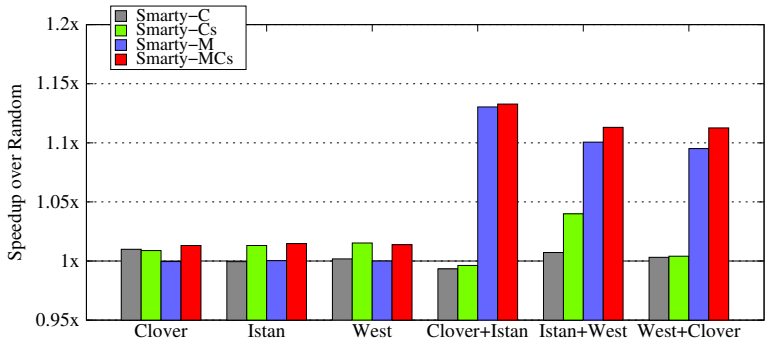


Figure 4.11: Impact of varying machine mix on EE diversity for Google testbed. Performance is normalized to random mapping. (higher is better)

4.3.2 Impact of Machine Mix on EE Diversity

In addition to the workload mix, microarchicture mix also has a significant impact on the amount of EE diversity in a WSC. In this section we study the impact of varying microarchitecture mix on the performance improvement of *SmartyMap*. We conducted experiments using 6 types of machine mixes for Google testbeds. The 6 types include: an entire WSC composed of all Clovertown Xeon, all Istanbul Opteron, all Westmere Xeon, $\frac{1}{2}$ Clovertown + $\frac{1}{2}$ Istanbul, $\frac{1}{2}$ Instanbul+ $\frac{1}{2}$ Westmere and $\frac{1}{2}$ Clovertown + $\frac{1}{2}$ Westmere. The workload for Google testbed is composed of all 9 key Google applications (Table 3.2). We also conducted similar experiments on the benchmark testbed, using a a workload composed of mostly memory intensive applications (Table 4.3).

Figures 4.11 and 4.12 present the results for Google and benchmark testbed, respectively. Similar to previous figures in Section 4.3.1, in each figure, the y-axis shows the performance improvement of *SmartyMap* using four different scoring policies over the random mapping for different machine mixes.

The first observation from these two figures is that even mixes of machines from a similar generation present a significant performance opportunity for exploiting EE diversity. In Figure 4.11, even for machine mixes

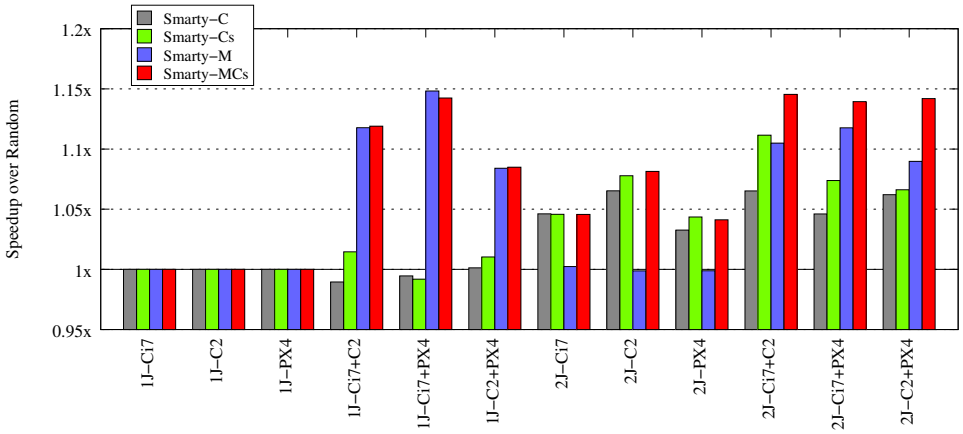


Figure 4.12: Impact of varying machine mix on EE diversity for SPEC benchmark testbed. Performance is normalized to random mapping.

composed of only 2 types of machines, *SmartyMap* generates significant performance improvement over random mapping. For Clovertown and Istanbul, which have similar average performance (Figure 3.1), the performance improvement of their mix is also significant (more than 10%). Similar observation can be made for the benchmark testbed as shown in Figure 4.12.

It is also important to note that for some machine mixes, the benefit of using Smarty-MCs (considering both microarchitecture and co-location) over Smarty-M (considering only microarchitectural diversity) is significant. For example, in the 2J-Core 2+Phenom X4 scenario shown in Figure 4.12 (the last cluster of bars), Smarty-MCs's performance improvement over the random mapping is 14%, significantly higher than the Smarty-M's 8% improvement. This is different from the observations we made in Section 4.3.1 that often Smarty-M performs similarly with Smarty-MCs. The reason for this difference is that there is less microarchitectural diversity (only 2 types of machines in the mix) in these experiments than those in Section 4.3.1 and thus the co-location diversity becomes more important. This observation demonstrates that although the microachitectural diversity is generally dominantly important, the amount of additional performance benefit when

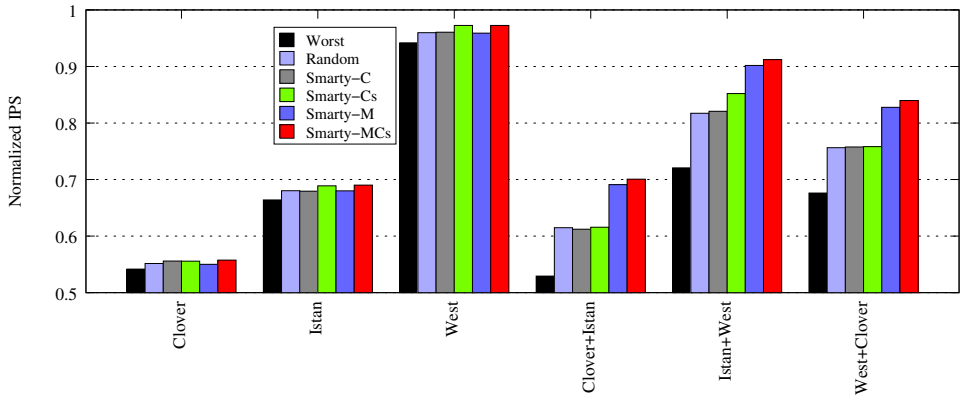


Figure 4.13: Normalized performance of various options of WSC machine composition (higher is better)

considering co-location is largely determined by the workloads mix and the machine mix. We discuss more on this topic in Section 4.3.4.

4.3.3 Which Servers to Purchase?

An important question arises when making server purchasing decisions. Is machine diversity in a WSC desirable or not? Should we try to increase or decrease it?

The study of EE diversity in this chapter indicates that when making such "heterogeneous" vs "homogeneous" decisions, simply comparing servers' average performance for the workload is insufficient and may be misleading. Instead, we advocate using *SmartyMap* to estimate the performance of WSC with various machine mixes. In fact, Smartymap makes the "heterogeneous" WSC a potentially more cost effective (better performance/dollar) option than purely homogeneous WSCs.

To illustrate that, Figure 4.13 shows the performance of several WSCs composed of various machine mixes. The experiments are conducted using the same workload as in Figure 4.11, composed of all 9 Google applications. In contrast to Figure 4.11, the performance of all experiments here is normalized to a single baseline, the aggregate IPS of all applications, each running

alone on its best performing platform. The baseline thus is the optimal (not necessarily achievable) performance. This facilitates the comparison of relative performance between WSCs.

Figure 4.13 shows that for homogeneous WSCs (WSCs composed of only 1 type of machine), Westmere is significantly better than the other two machine types, whose normalized performance is close to 1x. The WSCs composed of all Clovertown or Istanbul perform poorly, achieving only 55% and less than 70% normalized IPS, respectively. However, Westmere is rather pricey compared to the other two options. On the other hand, when using *SmartyMap*, the WSC composed of half Istanbul and half Westmere can achieve more than 90% of the performance as the WSC composed of all Westmere. And the WSC composed of half Clovertown and half Westmere performs 80% as well as all Westmere. The two heterogeneous WSCs are more cost-effective than the homogeneous Westmere WSC because half Westmere and half Istanbul (or Clovertown) would be drastically cheaper than a WSC of all Westmere, while achieving 90% of the performance (or 80% for Clovertown/Westmere mix). This makes the heterogeneous WSC a potentially better purchase candidate. However, without *SmartyMap*, simply looking at the average performance of these three types of machines may generate the misleading conclusion that the performance of heterogeneous WSCs is significantly worse than the homogeneous WSCs composed of elite machines. As shown in Figure 3.1 (the last cluster of bars), there is a drastic 2x performance difference between Clovertown and Westmere, and around 60% for Istanbul and Westmere, indicating their mix may not deliver similar performance as a WSC full of Westmere. When only looking at the performance of the random mapping, the performance of half Istanbul and half Westmere is only 80% compared to 90% when using *SmartyMap*. For half Clovertown and half Westmere, the performance of random mapping is 75%

compared to 82% when using *SmartyMap*. In summary, using *SmartyMap*, a heterogeneous WSC may be a more cost-effective option than homogeneous WSCs.

4.3.4 Revisiting Map Scoring

In addition to the findings discussed in the above sections, this study also leads to a number of insights on how to select the scoring policy:

1) **No free lunch**. Among all four scoring policies, Smarty-MCs always delivers the best performance improvement. However, it also requires the most amount of profiling to be effective.

2) **Smarty-M: big bang for your buck**. As this section shows, in most settings, Smarty-M generates significant performance improvement over random mapping with a very small amount of profiling. The profiling complexity is only $|A|\text{x}|M|$ as shown in Table 4.1. This indicates that Smarty-M can be adopted as an easy and effective first step for *SmartyMap* and can be triggered as soon as GWP finishes profiling the basic machine diversity information.

3) **Smarty-MCs: gradually improve over Smarty-M.** As Section 4.3.2 shows, depending on the workload and machine mixes, Smarty-MCs may also improve over Smarty-M significantly, delivering extra performance benefit, especially when there is much co-location penalty variability. Therefore, Smarty-MCs can be used to gradually improve over the mapping of Smarty-M, as the GWP accumulates more information regarding co-location.

4) It is important to remember that although the profiling complexity of Smarty-MCs appears high (Table 4.1), GWP runs continuously throughout the lifetime of the WSC probing each machine once every minute. As the scale of the WSC increases to thousands of machines the rate at which the

profiling information becomes robust also increases.

Chapter 5

Adaptation For Diverse Execution Environments

Contents

In this chapter, we address *the rigidness of applications* at the machine level by providing a mechanism that enables applications to add to its execution environment, and also the environment to the application. There is a class of problems whose solutions require this *online adaptation*. Two of these problems are pressing challenges in modern WSCs: the adaptive application of aggressive compiler optimizations, and the dynamic detection and response to contention. After presenting a novel *lightweight* approach to online adaptation, we demonstrate the utility of this mechanism by designing novel solutions to both of these problems.

5.1 A Mechanism for Online Adaptation in WSCs

To perform online adaptation, information that is only available at run-time must be used to restructure the application's execution, its environment, or both. *Online adaptation* is composed of two key tasks. First, the application's execution or execution environment must be monitored as it executes. Second, when a particular run-time characteristic or event is observed, the application's execution or its environment is then restructured or adapted in some way to accommodate this behavior. To perform these two key tasks, *online adaptation* approaches require a run-time component to be present and executed in tandem with the host application.

However achieving *online adaptation* for native applications has proved quite challenging. The runtime layer necessary to perform the monitoring and dynamic restructuring of the binary application increases the amount of work required to execute the application. The benefit of adding run-time online optimization or adaptation must outweigh the penalty suffered from the added complexity. Effectively achieving online adaptation at the binary level has proved difficult and has, by-in-large, not been adopted for practical

use in current industry and commercial domains. Current techniques fall into two categories: heavyweight dynamic binary translation approaches that provide too little benefit for the added complexity, and approaches that propose novel hardware changes and are unable to be realized on current chip architectures. Both of these approaches are not suitable for modern WSCs. A new approach is needed.

In this section, we present a new paradigm for achieving online adaptation at the binary level that uses what we call *lightweight introspection*. In contrast to a *heavyweight* online adaptation technique that requires either instrumentation of the host application to enable monitoring or the dynamic translation of the applications binary instructions, a *lightweight* online adaptation technique uses no instrumentation and performs no binary to binary online translation. Our *lightweight introspection* approach takes full advantage of the performance monitoring hardware features that are ubiquitous in current micro-architectural design [53] to perform all online monitoring with negligible overhead and minimal added software complexity. Using a technique we call *periodic probing*, monitoring is performed by taking snapshots of the hardware performance monitors using timer interrupts. In this work, we take advantage of this *lightweight* online adaptation methodology to design **Loaf**, the *Lightweight Online Adaptation Framework*. To effectively achieve online optimization on current microarchitectures, Loaf enables 1) the monitoring of the application and execution environment, 2) the dynamic restructuring of application code, and 3) the cooperative adaptation of the co-runners in an applications execution environment.

5.1.1 An Overview of Loaf

In order to effectively enable online adaptation, Loaf must provide the capability to monitor online events and adapt the application or its environment

to these events. To achieve these capabilities we have three key functionality requirements for Loaf which includes:

1. An efficient mechanism for the online monitoring of the application or its environment's behavior.

2. An efficient mechanism to allow the dynamic restructuring of application code. One of the key methods used to adapt application behavior is to allow code restructuring in response to dynamic events.

3. An efficient mechanism to enable the adaptation of multiple co-running applications and threads in an application's execution environment. Multicore architectures are ubiquitous in today's computing environment, and an application can be affected by its simultaneously executing co-runners.

The underlying philosophy of our online adaptation approach is to achieve efficiency by remaining as *lightweight* as possible. An approach that is lightweight is critical for deployability in modern WSCs. Therefore, to achieve the tasks of online adaptation, observation and adaptation, with minimal application interference we use the following approaches to the three design goals mentioned above:

1. To achieve online monitoring, we use the *lightweight* approach of periodically probing the hardware performance monitors on current microarchitectures. We call this approach **lightweight introspection**.

2. To achieve the dynamic restructuring of application code, we use the *lightweight* approach of statically providing multiple code versions for regions of interest and allowing dynamic switching based on online monitoring. We call this approach **scenario based multiversioning**.

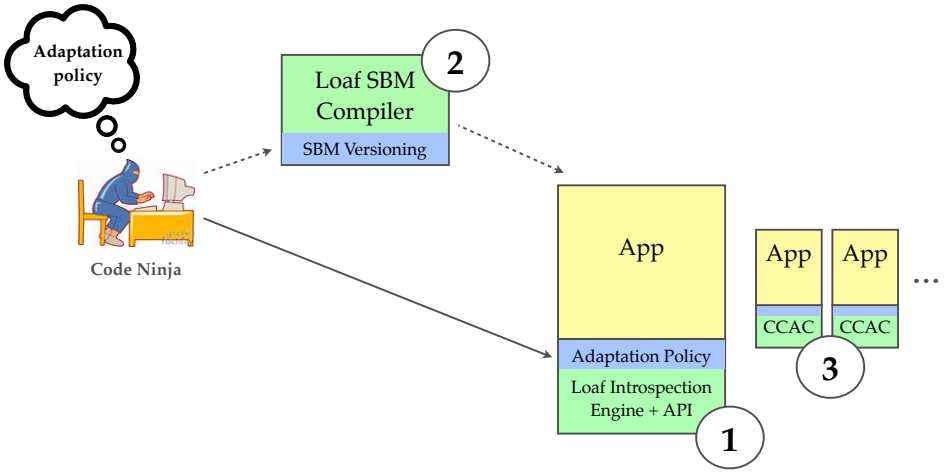


Figure 5.1: Loaf Overview. (1) Lightweight Introspection (2) Scenario Based Multiversioning (3) Cross-Core Application Cooperation

3. To accommodate adaptation of the application and its environment based on events that occur due to simultaneous co-scheduling on current multicore architectures, we use the *lightweight* approach of sharing dynamic monitoring information across cores using a shared communication table, allowing multiple threads to *cooperate* during online adaptation. We call this approach **cross-core application cooperation**.

Figure 5.1 illustrates how a user of the Loaf infrastructure interacts with Loaf. Each of the three components mentioned above corresponds to the numbers in Figure 5.1 respectively. The blue sections of each component denotes the locations a user must touch to implement the desired *adaptation policy*. An adaptation policy is a specification of a desired response to some dynamic event or set of events. With a particular adaptation policy in mind, the user can leverage Loafs API in each of these components to enact the policy.

Algorithm 2: Loaf LIE Initialization

```
1 events_of_interest ← user_defined_events;
2 probe_interval ← user_defined_interval
3 foreach e in events_of_interest do
4 |   active_counters ←PMUConfigure(e)
5 end
6 foreach c in active_counters do
7 |   PMUBeginCounting(c)
8 end
9 IssueTimerInterrupt(probe_interval)
```

Algorithm 3: Loaf Periodic Probes

```
1 PMUStopCounters();
2 foreach e in events_of_interest do
3 |   e.value ←PMUReadCounter(e)
4 end
5 DoAdaptationAnalysis();
6 Adapt();
7 if new_events_of_interest then
8 |   events_of_interest ← new_events_of_interest
9 end
10 if new_probe_interval then
11 |   probe_interval ← user_defined_interval
12 end
13 foreach e in events_of_interest do
14 |   active_counters ←PMUConfigure(e)
15 end
16 foreach c in active_counters do
17 |   PMUBeginCounting(c)
18 end
19 IssueTimerInterrupt(probe_interval)
```

5.1.2 Online Monitoring

To achieve the necessary task of efficient online monitoring we use *lightweight introspection* as shown in Figure 5.1(1). The core intuition of this approach is to remain lightweight by leveraging *periodic probing* with the usage of hardware performance monitors. These hardware performance monitors provide realtime micro-architectural information about the applications currently running on chip. As the counters record this information, the program executes uninterrupted, and thus recording this online profiling information presents no instrumentation overhead. These capabilities can be leveraged with one of the many software APIs, such as PAPI [73] or Perfmon2 [36]. In this work, we use Perfmon2 as it is one of the most robust and flexible PMU interfaces, and supports a wide range of micro-architectures.

The self introspection run-time employs a *periodic probing* approach, meaning information is gathered and analyzed intermittently. Using a timer interrupt the environment will periodically read the performance monitoring hardware, reset the timer, and restart the performance monitoring counters. The algorithms that comprise our lightweight introspection engine is shown in Algorithms 2 and 3. Periodic probing is an efficient method for collecting information from hardware performance monitors. The overhead of this technique is determined by two factors: the frequency of probes (e.g. interrupts), and the complexity of the analysis and adaptation work done during those interrupts. These two factors are impacted by the nature of the desired adaptation policy. For the policies implemented in our case studies, the *probe_interval* used is one every millisecond and the resulting overhead is negligible.

Its important to note that hardware performance counters are a ubiquitous hardware feature and has been used in prior work in various ways. There has been work discussing and using performance counters for partic-

ular applications such as selecting optimizations [18], enhancing operating systems [63], and in Java virtual machines [104], among others. In this work, we present a general online adaptation framework for native binaries that leverages hardware performance monitors exclusively to provide a lightweight monitoring and introspection runtime.

Our Loaf lightweight introspection runtime serves as the core mechanism for the monitoring of application behavior, and the execution environment. The runtime can be attached to a host application in a number of ways including statically linking the run-time module into the application binary, dynamically linking in as a module, or as a third party virtual application host such as `gdb`. In this work we staticly link the runtime into the binary itself. Further implementation details for online monitoring can be found in Appendix A.

5.1.3 Adapting the Application

To achieve the dynamic restructuring of application code as execution occurs we use a multiversioning technique we call *scenario based multiversioning (SBM)* as shown in Figure 5.1(2). Traditionally, static code layout and structure is rigid regardless of changes in its execution environment or application phase, this is exactly what the scenario based multiversioning is equipped to address. The key insight of this *scenario based multiversioning* is to enable compiler writers to enable adaptation by statically anticipating a variety of dynamic scenarios and situations and specializing code regions accordingly. SBM is truly a static/dynamic hybrid framework. Using SBM, compiler writers can apply various optimizations and code layouts across multiple instances (or versions) of a code region, each specialized to particular dynamic situations and events that can be identified dynamically by the introspection runtime. Execution can then be rerouted dynamically by the

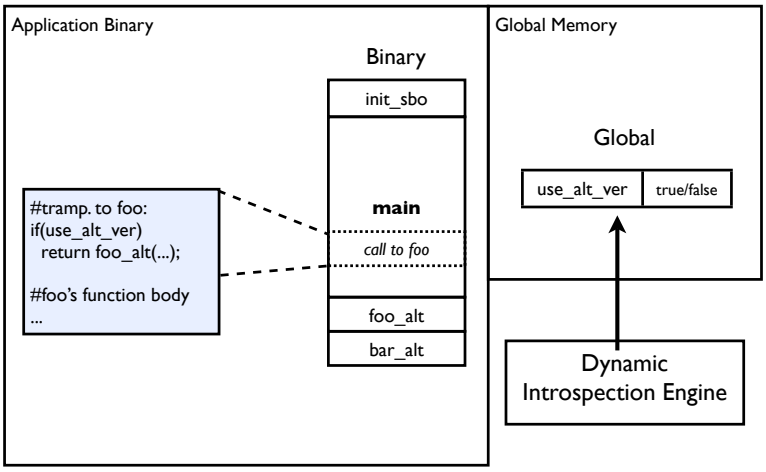


Figure 5.2: Alternate Versioning Scheme

runtime to execute the desired code regions. This capability is the key distinction between traditional multiversioning approaches and SBM. Instead of adding checks and conditionals to the application binary to select and switch versions, which necessitates the execution of checks during each execution of the multiversioned code, SBM allows for the unobtrusive rerouting of the application binary without explicit checks. With SBM, this rerouting can occur anytime during execution, and in parallel with execution. While allowing flexible runtime adaptability, SBM retains all of the capabilities and advantages of static compilation, such as the availability of high-level source information.

The SBM approach used in this dissertation performs its multiversioning at the function level allowing the generation of specialized versions of a function to target different scenarios. For SBM we provide an interface between the static binary and a lightweight introspection run-time component in the form of a dispatch table. This interface allows the introspection engine to hook into the executing binary and reroute the execution via reseting the active versions of the functions. To accomplish this we have two designs.

We call the first design the *alternate versioning scheme* and the second

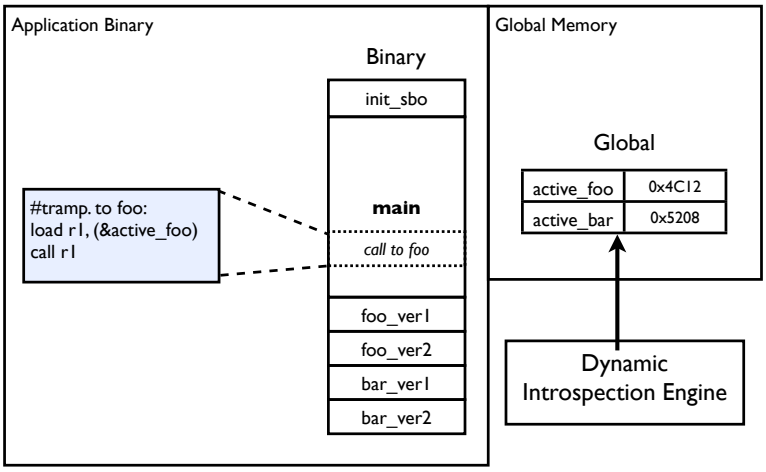


Figure 5.3: N-Version Versioning Scheme

the *n-version versioning scheme*. While both techniques requires the use of a trampoline as the multiplexing mechanism, there are differences. Figure 5.2 shows the alternate version scheme. For this scheme, we have a default and alternate version of particular functions. With the alternate version scheme there is a single global switch that the dynamic component interfaces to control which version the application uses. With this scheme the entire binary will either execute the default versions for all multiversioned functions or the alternative version. This provides a simple abstraction that a compiler writer can use to design SBM based techniques that do not require too much complexity.

Figure 5.3 shows the design of the *n-version versioning scheme*. This scheme allows for any number of versions for any function and individual version switching. For this scheme, we maintain a global mapping table in memory for each function. Instead of a global switch, each call to a multiversioned function is transformed to an indirect call. During execution, the target address of the call is controlled by the dynamic component and any combination of versions can be active at anytime. This allows for much more complex SBM techniques where multiple scenarios can occur at the

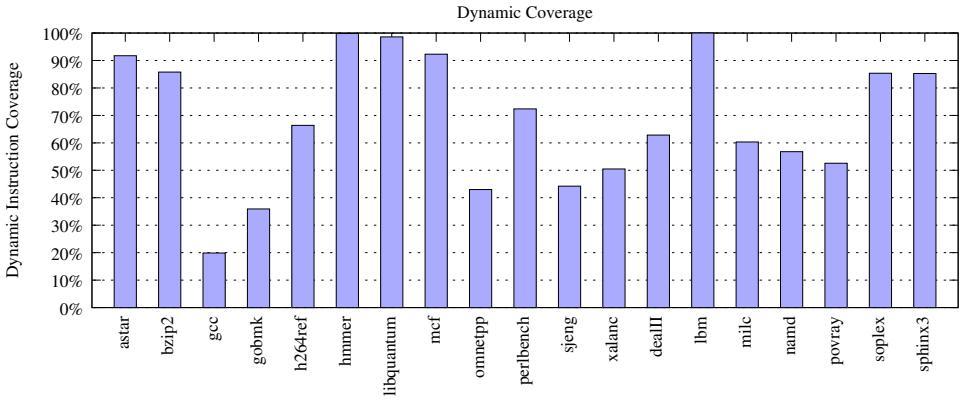


Figure 5.4: This graph shows the percent of execution time spent executing of the Top 5 hottest functions across SPEC2006 benchmarks.

same time.

One important consideration is that we cannot have multiple versions of every function in our application binary. This would cause an unacceptable amount of code growth, which would limit the applicability of SBM and ultimately have a negative impact on application performance. Therefore we limit the number of functions we multiversion to only the hottest functions in the application. To efficiently multiversion our application we take advantage of some basic profiling that has proven useful for determining the hottest code in an application [24, 74, 75]. SBM can use the simple profiling provided by GCC's GProf to identify the hottest functions of the application. We know from prior work that the top 2 to 8 functions most often covers the vast majority of the dynamically executed instructions across the SPEC 2006 benchmarks. In Figure 5.4 we show the dynamic instruction coverage of the top 5 functions in the SPEC2006 benchmark suite. This data was collected using GProf [42]. As the graph shows, just the top 5 hottest functions can cover a significant portion of an applications execution, many times over 90%. Multiversioning these top functions leads to a very slight amount of code growth, for the SPEC2006 benchmarks less than 2% on average.

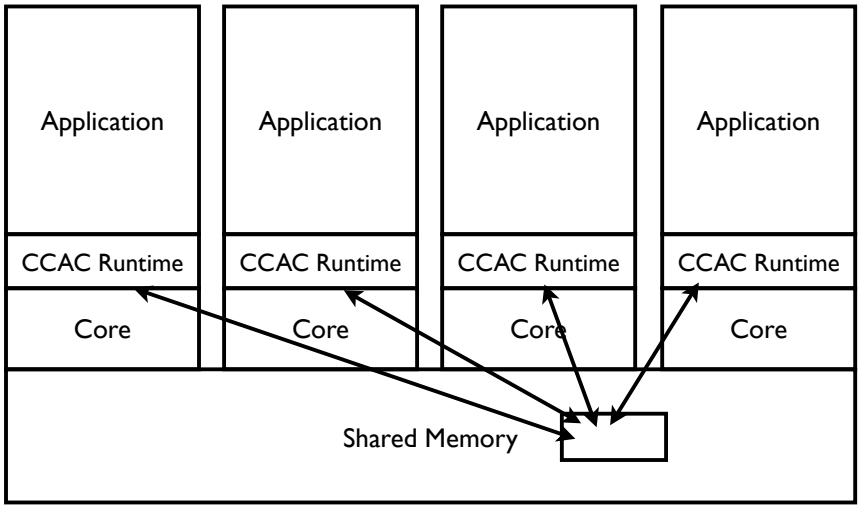


Figure 5.5: Cross-Core Application Cooperation Run-time

5.1.4 Adapting the Environment

To accommodate the adaptation of an application and its environment based on events that occur due to simultaneous co-scheduling on current multicore architectures, we use a *cross-core application cooperation* (CCAC) approach as shown in Figure 5.1(3). This approach is designed for problems that require the coordination of a number of processes or threads for a particular goal. The design of the CCAC enabled Loaf run-time environment is presented in Figure 5.5. In the scenario presented in the diagram, we have four applications running simultaneously on a quad core machine. In order to monitor and collect thread/core specific performance information on current hardware, we collect performance monitoring information on each core hosting the applications of interest and use a *shared communication table* to provide this information to other Loaf runtimes. Also, adaptation directives can be issued from one core to another through this shared communication table. We use the table to allow multiple CCAC enabled Loaf run-times to cooperate, respond, and adapt to each other.

We use shared memory to achieve this cross-core application coopera-

tion (shown in Figure 5.5 as arrows pointing into the table). Performance information is gathered and added to the communication table intermittently using the *lightweight introspection* provided by Loaf. It is also useful to record a window of multiple samples of performance information in the table as keeping a window of recent activity will allow us to observe trends in application behavior. To accommodate this communication protocol, we also develop an abstract primitive for each table entry which is supplied by our API.

5.1.5 Leveraging Loaf

With a lightweight framework for online adaptation in place, we can address the two problems previously discussed. In the remainder of this chapter, we present two case studies showing how the Loaf infrastructure is used to design and construct practical lightweight adaptive solutions to two pressing problems in our field. The first problem is that of *aggressive optimizations*. These are optimizations that are risky, as they can significantly improve or degrade performance. As shown in previous work [39, 131], the effect of applying this class of optimizations cannot be predicted statically, as it may depend on input size, micro-architectural events, and execution environment. In this work we show how the Loaf infrastructure is used to enact an adaptation policy to dynamically apply aggressive optimization only when there is benefit.

The second case study illustrates how Loaf can be used to adapt the environment to the application. The problem addressed in this case study is that of cross-core application interference. Contention for shared resources and cross-core application *interference* due to contention, pose a significant challenge to providing application level quality of service (QoS) guarantees on commodity multicore micro-architectures. The commonly used solution

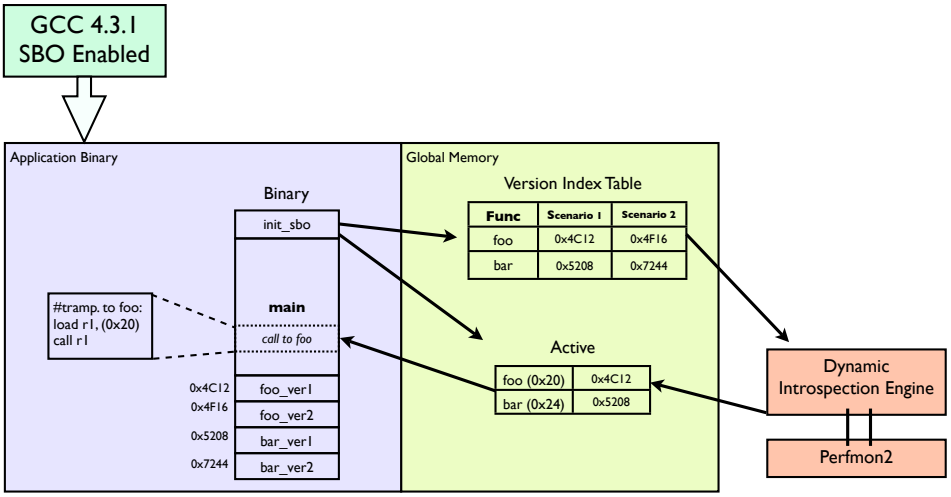


Figure 5.6: Overview of Scenario Based Optimizations.

is to simply disallow the co-location of latency-sensitive applications and throughput-oriented batch applications on a single chip, sacrificing utilization. In this work, we show how to use Loaf's ability to cooperatively adapt co-running applications to design an agnostic contention aware execution environment that will adapt an application's environment to minimize cross-core interference due to contention, while maximizing chip utilization.

5.2 Adapting the Application: Aggressive Optimization

In this section, we demonstrate how Loaf can be used to adapt the application to its environment by addressing a pressing problem in modern WSCs: the adaptive application of aggressive compiler optimizations. We call this approach, *Scenario Based Optimizations* (SBO). Figure 5.6 shows a more detailed diagram of our SBO technique.

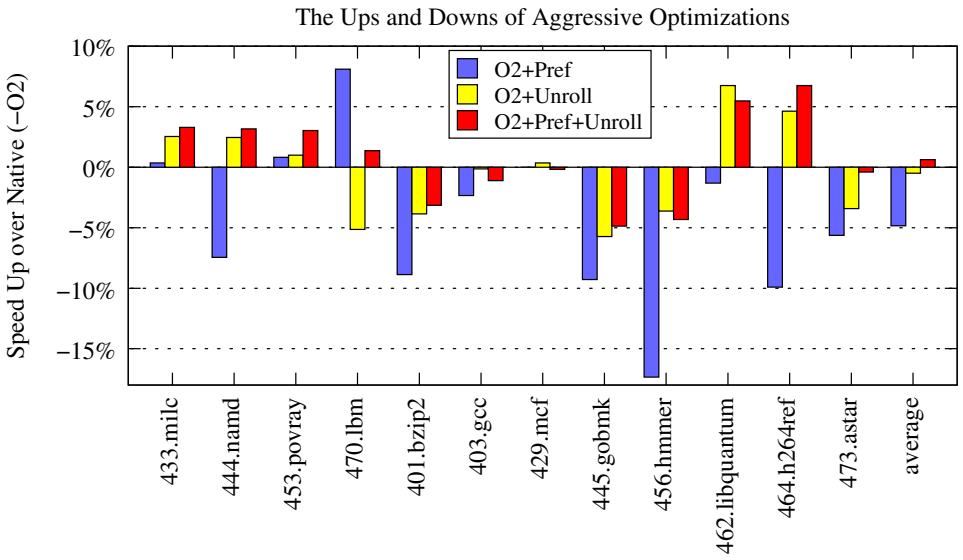


Figure 5.7: This graph shows the speedup in execution time when *aggressive* optimizations are applied. Note that sometimes there is a benefit other times we see a degradation.

5.2.1 Motivation: Win Some, Loose Some

Aggressive optimization may increase performance in some contexts and decrease performance in others. For our SBO approach we have identified two such optimizations, software cache prefetching and loop unrolling. These optimization heuristics, both found in GCC 4.3.1 as optional optimizations, both improve performance in some cases and degrade performance in others.

Figure 5.7 shows the impact these optimizations have on performance for 12 of the SPEC2006 benchmarks. These experiments were run on the Core 2 Quad 6600 running Linux 2.6.25. This graph shows the speed up that results from applying either or both of these optimizations.

With notable exception of `lbm`, in most cases, software prefetching has a negative impact on performance. A negative interaction with the already present hardware prefetching structures on the Core 2 is possible. If the hardware prefetcher is already doing the work, having the explicit prefetch instructions simply adds an extra burden to the architecture. Additionally,

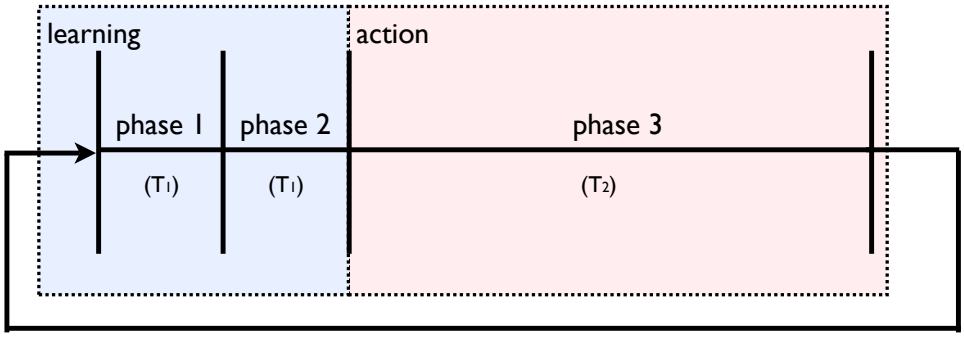


Figure 5.8: This represents the three phase execution approach of SBO.

if the prefetch accuracy is low, the prefetch instructions may be only serving to pollute the caches. As the figure shows, software prefetching is an aggressive optimization. While software prefetching improves `lbm's` performance by 8%, in the case of `hmmer`, the degradation due to adding the software prefetching is over 15%.

Loop unrolling is also an aggressive optimization, as we see a performance improvement in some cases, and a degradation in others. This observation brings us to the primary motivation of our approach. Using the dynamic introspection engine, coupled with Scenario Based Optimization we are able to detect the scenarios when aggressive optimizations are improving or degrading performance, and then reroute execution accordingly. However the question arises as to how we achieve this adaptation using the Loaf framework.

5.2.2 Three Phase Execution

For the design and implementation of SBO, we use the *alternate versioning* SBM scheme mentioned previously, with an online three phase approach. Statically, we generate code for two scenarios. First, we generate code without software prefetching or loop unrolling for the scenario that aggressive optimizations would degrade performance; we call this the non-aggressive

version. Then, for the scenario that aggressive optimizations would improve performance, we generate code for that same function that has software prefetching and loop unrolling; we call this the aggressive version.

The dynamic component of our SBO approach has three phases, as shown in Figure 5.8. The first two phases compose the learning and monitoring part of SBO, and the third phase composes the action part of SBO. During execution these phases continually loop until the host application terminates.

Our design of the three phase approach is as follows:

1. During the first phase we set the active version for the binary to non-aggressive. The dynamic engine then starts the counters to look at the absolute number of instructions retired. The application then executes for T_1 time. The number of instructions that successfully executed are then recorded.

2. During the second phase we set the active version for the binary to aggressive. We then do the same; we record the number of instructions that executed for this T_1 time. Before the third phase begins, we compare the number of instructions retired for both phases 1 and 2.

3. We select the version with the highest number of instructions retired to be executed in the third phase which lasts for T_2 time. Essentially, we select the version that has exhibited the lower average CPI for T_1 time. This ad-hoc performance metric conveys whether the scenario is well suited for aggressive optimization.

4. After T_2 seconds of executing the winning version we enter phase one and restart the process.

We use shorter time periods for T_1 in phases 1 and 2 and select longer time periods for T_2 in phase 3. A longer value for T_2 is used because our

assumption is that there are steady phases, however phase changes do occur, so we should retest to keep our versioning decision relevant.

5.2.3 The Effectiveness of SBO

In this section, we present the results of our experimentation evaluating the effectiveness of Scenario Based Optimization. The goal of SBO is to eliminate the degradations of aggressive optimization while reaping the benefits. We also hypothesized that we would be able to exceed the potential benefits of applying and using aggressive optimizations statically.

All of our experiments were performed on a machine with the Intel Core 2 Quad 6600 architecture and 2gb of ram. We used benchmarks from the SPEC2006 v1.1 suite and ran them on their reference inputs to completion. We used the GCC 4.3.1 compiler to compile these benchmarks. The benchmarks were all compiled with optimization level -O2, and tuned to the Core 2 architecture (compiler option -march=core2). All experiments were run on Ubuntu Linux Kernel 2.6.25 patched with Perfmon2.

Execution Time

Figure 5.9 shows the impact on execution time when applying aggressive optimizations with, and without, SBO. The data shown in this graph has been normalized to the baseline, optimization level -O2. Anything greater than 100% marker shows a degradation anything lower than this marker shows a speedup.

One of the major goals of SBO is to eliminate the degradations incurred by aggressive optimizations. As the data in figure 5.9 shows, only when using SBO do we observe performance improvements in all cases, with exception of `gobmk`. Degradations are effectively eliminated. In addition to eliminating the degradations and leaving only performance improvement,

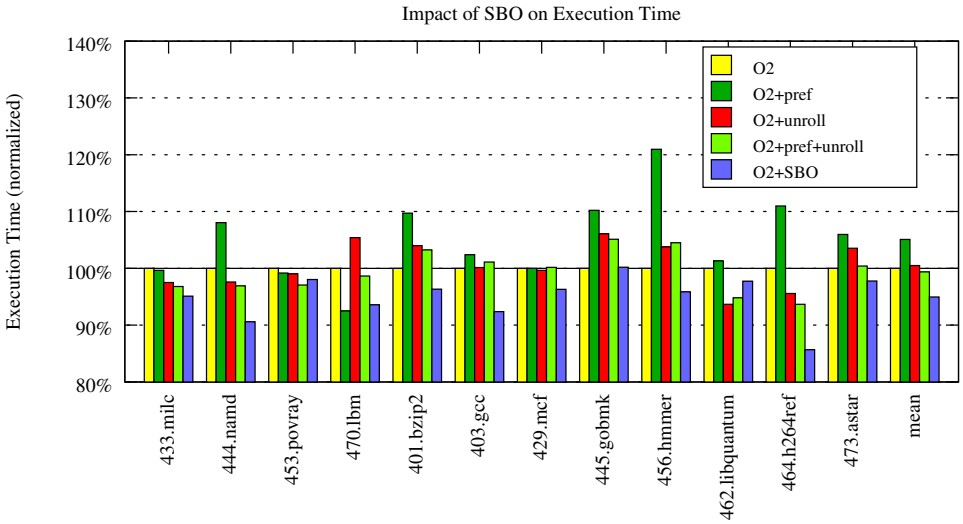


Figure 5.9: This is the execution time after applying the aggressive optimizations statically compared to applying the same optimizations using SBO. (lower is better)

for the large majority of the benchmarks, the performance improvements significantly exceeds those produced by any combination of aggressive optimization without SBO. In 9 out of the 12 benchmarks presented, SBO exceeds the benefit of all combinations of aggressive optimizations applied statically, in most cases more than doubling the performance boost.

Effect of Dynamic Switching

One important question that arises, is whether there is much switching occuring dynamically. If there is not much dynamic switching occuring, there may be no need to continually probe the counters and redo analysis. We address this question in Figure 5.10. Here, we show the speedup of SBO adaptively switching the active version between aggressive and non-aggressive, compared to only having one version execute for the duration of application execution.

In Figure 5.10, the first bar shows SBO over only having the non-aggressive version, and the second bar shows having SBO over having only the aggressive version. In this figure, we highlight the fact that only for

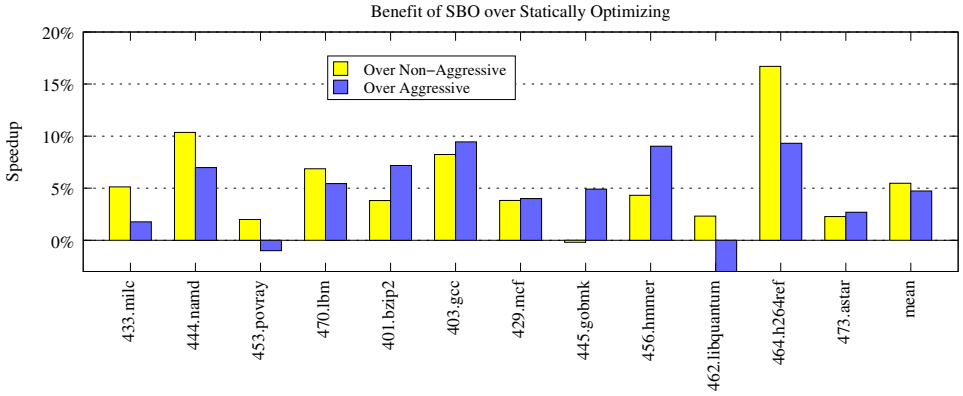


Figure 5.10: Here we show the benefit of using SBO to dynamically select the right version for a scenario versus using only the code for either scenario for the entire run.

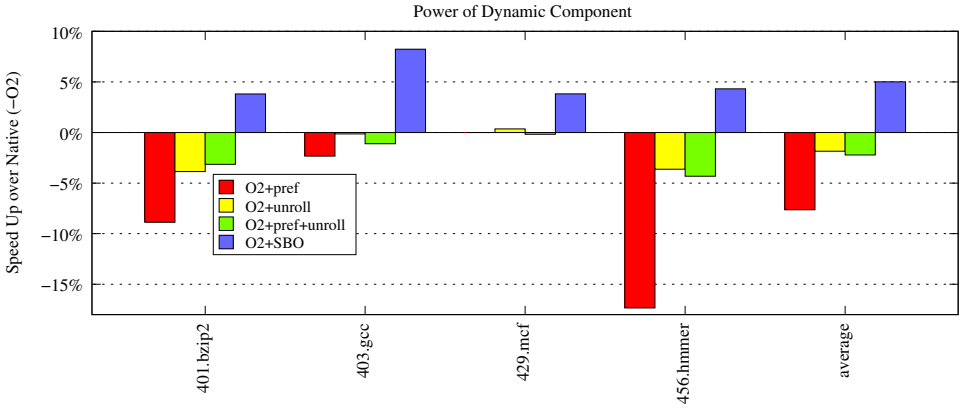


Figure 5.11: This graph highlights the power of a Scenario Based dynamic approach. These benchmarks all degrade or show no benefit when applying aggressive optimizations statically.

two benchmarks, `povray` and `libquantum`, it is better to have the statically assigned aggressive version.

Degradation Reversal

In Figure 5.11, we highlight one of the brightest contributions of SBO. That is the fact that SBO actually makes loop unrolling and software prefetching show benefit where it would otherwise not. In the benchmarks presented in this figure, software prefetching and loop unrolling simply does not work without SBO. They both show degradations regardless of whether they are

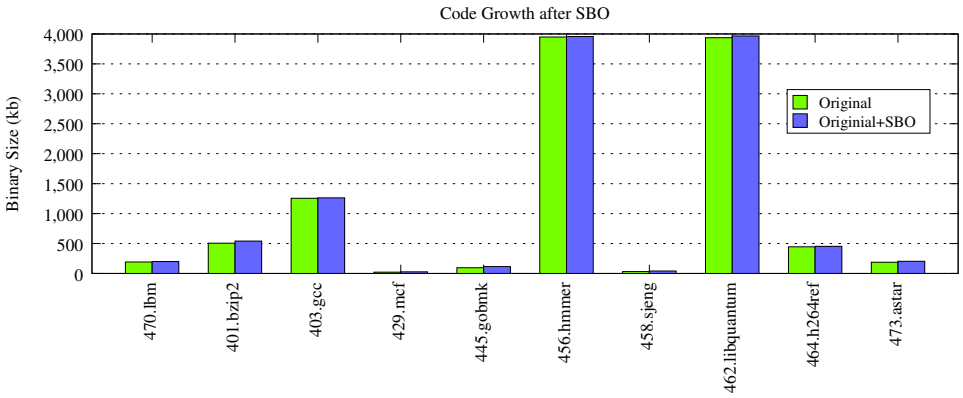


Figure 5.12: Here we show what percentage of the binary is occupied by code added by prefetching and unrolling in addition to that added by SBO.

applied individually or simultaneously. However, when governed by SBO these aggressive optimizations show significant performance improvements.

Code Growth

As we designed our SBO framework, careful attention was paid to other types of overhead, such as the impact on code size. Scenario Based Optimizations requires the duplication of functions, however it is not necessary to multiversion every function. As previously mentioned, we only multiversion the top 5 hottest functions.

Figure 5.12 shows the code growth due to the SBO framework. The sizes of the added dynamic runtime component that is statically linked into each binary are included in these measurements. The first bar shows the size of the original binary, and the second bar shows the size of the binary compiled with 2 versions of its top 5 hottest functions and the SBO dynamic component linked in. We see that the final code size of the binary is largely unaffected by SBO. This is due to the fact that the increase in code size ranges from 3kb, to a mere 12kb. For many benchmarks the absolute sizes of the binary are in the hundreds and thousands of kilobytes.

5.3 Adapting the Environment: Contention Detection

In this section we demonstrate how Loaf can be used to adapt the environment to an application by addressing another pressing problem in modern WSCs: the dynamic detection and response to contention.

5.3.1 Challenge of Interference in WSCs

Multicore architectures are ubiquitous and have become the norm in computing systems today. These architectures dominate in many domains, including WSCs where quality of service (QoS) and low latency requirements dominate. Multicore architectures are composed of a number of processing cores, each with a private cache(s), and typically larger caches that are shared among many cores [53]. Other shared system resources include the bus, main memory, disk, and other I/O devices. When processes and threads are executing in parallel on a single multicore CPU we say they are *co-located*. Co-located processes and threads place varying amounts of demand on these resources; this demand can often lead to *contention* for these resources. Resource contention directly impacts application performance. When an application's performance is negatively affected by another application executing on a separate core, we call this *cross-core interference*.

Application priority and quality of service requirements often cannot withstand unexpected cross-core interference. For example, applications commonly found in the web-service data center domain such as search, maps, image search, email and other user facing web applications are *latency-sensitive* [21, 27]. These applications must respond to the user with minimal latency, as having high latency displeases the user. Data centers for web-services classify applications as either being *latency-sensitive* or as

throughput-oriented *batch* applications, where latency is not important [21]. To avoid cross-core interference between latency-sensitive and batch applications, web-service companies simply disallow the co-location of these applications on a single multicore CPU. Using this solution may leave the CPU severely underutilized, and is a contributing reason to the server utilization of these data centers often being 15% or less [72]. Low utilization results in wasted power, and lost cost saving opportunities.

5.3.2 Motivation: Cross-Core Interference

Intel's Core 2 Duo architecture has 2 cores, each with a private L1 cache and a single L2 of 4mb shared between the two cores. Intel's new Core i7 (Nehalem) architecture has 4 cores, each with private L1 and L2 caches and a single 8mb shared L3 cache for all 4 cores [53]. These types of shared memory multicore architectures are common in modern WSCs. When the workload of the individual application processes and threads executing on these multicore processors fits neatly into private caches, there is no cross-core interference (assuming coherence traffic is at a minimum). When the size of an application's working set exceeds the size of the private cache, the working set spills over into the larger shared caches. The shared last level cache presents the first level of possible contention. Much of the contention in these levels manifest themselves as traffic off-chip and thus show up as misses in the last level cache on the chip.

Our strategy is to leverage Loaf to monitor activity in the last level of cache to detect contention and focus on minimizing contention in shared caches and bandwidth to memory. When more than one application is using the shared last level of cache heavily, and the data is not shared, contention occurs. One way to address this problem is to increase the size and associativity of the cache. However, although cache sizes have been increasing

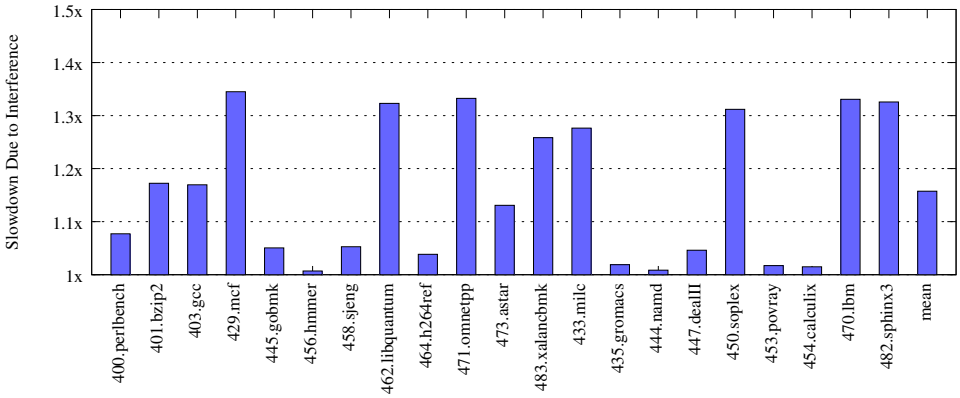


Figure 5.13: Performance degradation due to contention for shared last level cache on Core i7 (Nehalem) while running alongside lbm.

with every generation of processors, we are still far behind the demands of today's application workloads. Figure 5.13 shows the degradation in performance of a set of applications due to cross-core interference caused by cache contention. This experiment was run on a state of the art general purpose processor (Intel Core i7 920 Quad Core), and demonstrates the impact of just two applications contending on a multicore chip for a large 8mb, 16way associative, shared, last level cache. The applications shown come from the SPEC2006 benchmark suite. Each application was first run alone on the quad core chip, then with the `lbm` benchmark running alongside on a neighboring core. The bars in Figure 5.13 shows the slowdown of each benchmark running alongside `lbm`. `Lbm` is an example of an application with aggressive cache usage. An application that is more affected by `lbm` implies that that application is also aggressive with its cache usage. Remember this data shows just two applications running on a quad core machine with a large cache designed to handle the load from four cores simultaneously doing work. In many cases we see a performance degradation exceeding 30%.

Figure 5.14 shows the increase in last level cache misses when running with a contender. It is important to notice the delta in cache misses between

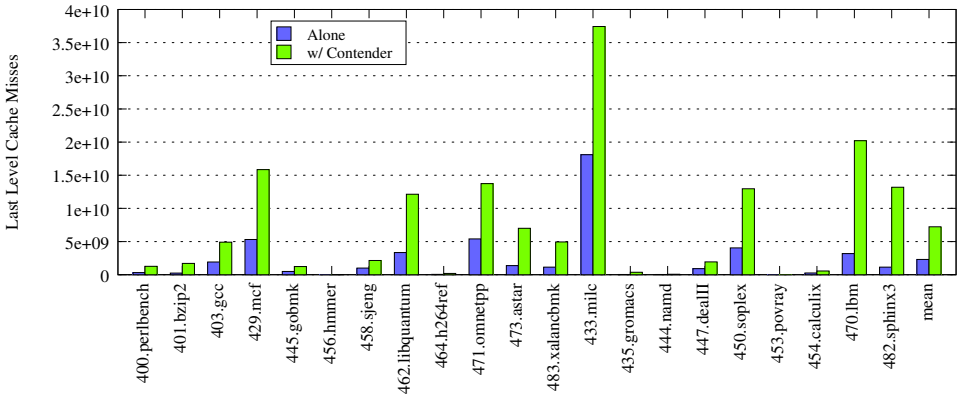


Figure 5.14: Increase in last level cache misses when running with contender.

the application running alone and when it is running with the contender. It is also important to get a sense of the absolute number of misses for each and how that impacts its sensitivity to contention. Having a 150% increase in cache misses impacts performance much less as the absolute number of misses goes down. From this graph it is clear that the more last level cache misses an application experiences, the more sensitive it is to cross-core interference.

We define the *utilization* of a multicore processor as

$$U = \frac{\sum_{i=1}^{N} \frac{R_i}{R_i + I_i}}{N} \tag{5.1}$$

for some time, where N is the number of cores on the chip, R_i is the amount of time spent running on core i, and I_i is the amount of time idle on core i.

5.3.3 A Solution with CAER

Our goal is to leverage Loaf to address the contention in the shared caches of current multicore chip design by minimizing the cross-core interference penalty on latency-sensitive applications while maximizing chip utilization. To do so we use Loaf to design a runtime, CAER, the *contention aware*

execution runtime.

Inferring Contention

The basic premise of our solution is that information from PMUs can be used in a low/no overhead way to infer contention. In this work we focus on the shared last level cache (LLC) miss behavior. Last level cache misses directly (and negatively) impact the instruction retirement rate (i.e. IPC). Figure 5.15 illustrates this phenomenon with two SPEC2006 benchmarks that exhibit clear LLC miss phases. These benchmarks were run on their `ref` inputs to completion. The x-axis represents time from beginning of the application run to the end in all four of the graphs presented. Figure 5.15 shows two pairs of graphs, each pair correlating the LLC miss rate over time to the instruction retirement rate over time. We can see clear and compelling evidence of the inverse relationship between the number of LLC misses and the retirement rate.

CAER is based on the hypothesis that if two or more applications are simultaneously missing heavily in the last level shared cache of the micro architecture, they are both making heavy usage of the cache and probably evicting each others data (i.e. contending). This contention then leads to increased cache misses in both applications, which is evident in Figure 5.14. We believe that if we can dynamically monitor and analyze the chip wide information about thread/core specific impact on the last level cache misses we should be able to detect contention and thusly respond to this contention.

Architecture of CAER

The design and architecture of the CAER execution environment is presented in Figure 5.16. To the left of the diagram we present the overall design vision of the CAER environment, and to the right we present the ac-

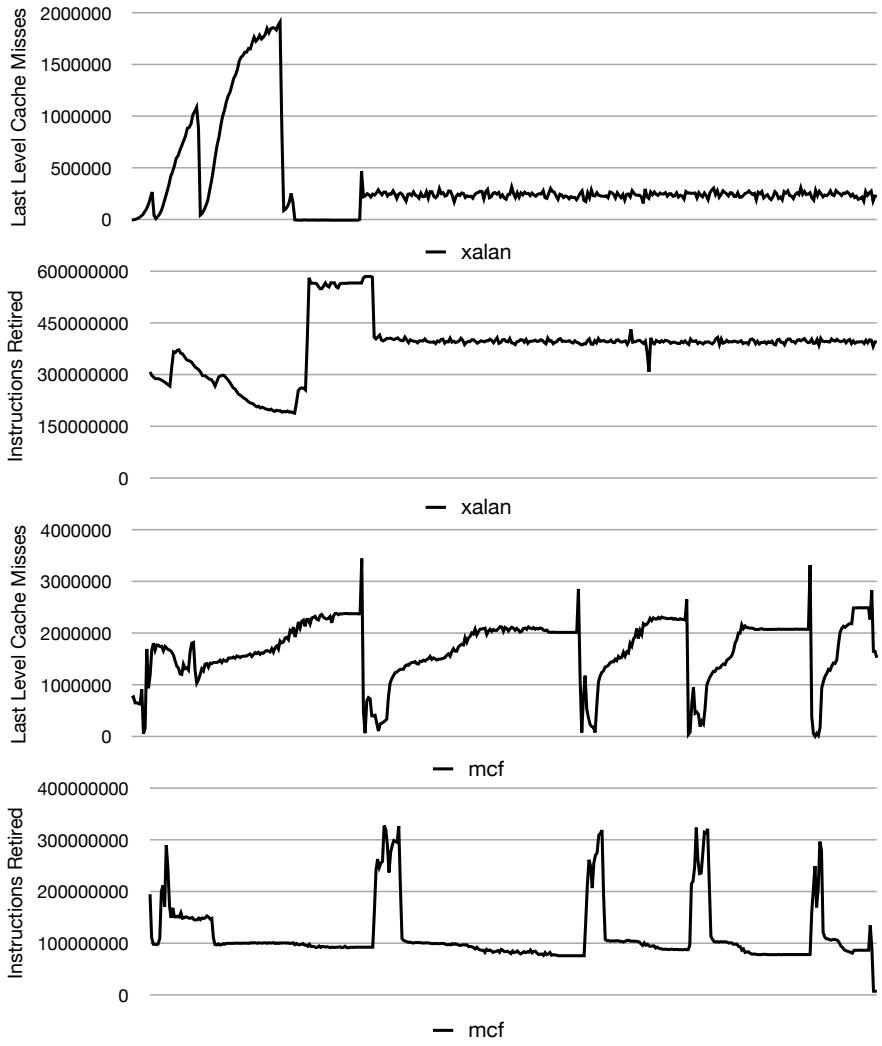


Figure 5.15: Correlating last level shared cache misses, and reduction in instruction retirement rate.

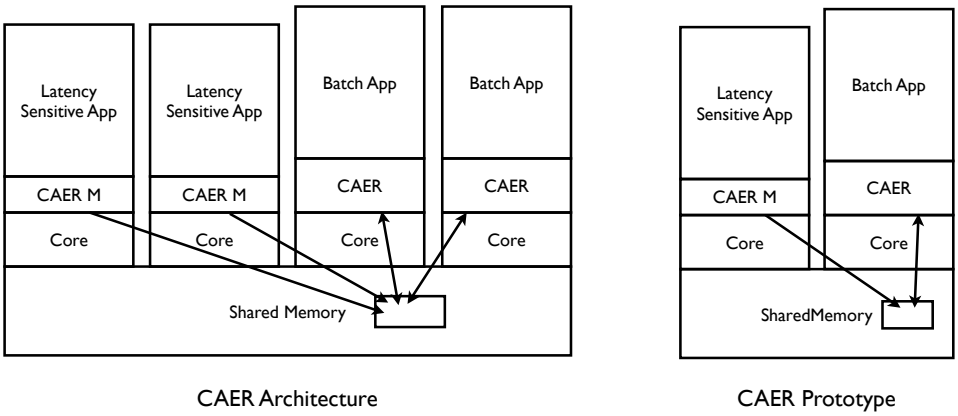


Figure 5.16: Architecture of our Contention Aware Execution Runtime

tual working prototype we have implemented for this study. In the scenario presented on the left of the diagram we have two latency-sensitive applications or threads, and two batch applications or threads. In order to monitor and collect thread/core specific performance information on current hardware, we must issue the performance monitoring unit (PMU) configuration and collection directives on the particular core hosting the application of interest. For this reason a Loaf runtime must be present beneath all application threads of interest. These CAER runtimes are cooperative and must share information, respond, and adapt to each other.

CAER's cooperation is accomplished via shared memory using the communication table discussed in Section 5.1 and also shown in Figure 5.16 (arrows pointing into the table). Notice that the runtime layer (CAER M) beneath the latency-sensitive applications appear thinner in Figure 5.16. These (monitor) runtime layers are more light weight than the main CAER engines and only are responsible for collecting PMU data and placing this data in the communication table. The main CAER engines that lie underneath the throughput-oriented batch applications processes this information and perform the contention detection and response heuristics. CAER only applies any dynamic adaption or modifications on the batch application.

The latency applications always remain untouched.

The CAER runtime employs the *periodic probing* approach described in Section 5.1. Using a timer interrupt the environment periodically reads and restarts the PMU counters.

CAER runtime uses a period of one millisecond. Every millisecond each CAER runtime probes their relevant performance monitoring units and reports last level cache information to the communication table. This table records a window of sample points, which allows us to observe trends of many samples. The main CAER engines that lie under the batch processes detect and react to contention. Note that all of the batch processes/threads must react together. Reaction directives are also recorded in the table, and all batch processes must adhere to the reaction directives. In the current design of CAER, these directive include pausing and staggering execution.

Our prototype is shown to the right of Figure 5.16. This instance of CAER supports two applications, one running atop CAER M, and the other on the main CAER engine. The CAER runtime is statically linked into the binary. Our prototype is fully functional and, as we show later, effective on real commodity hardware.

The diagram in Figure 5.17 shows the contention detection and response phases used in the CAER runtime that lies under the batch applications. Throughout execution CAER resides in one of these states and continually transitions among these states. After CAER performs its contention detection heuristic, either contention, or the absence of contention is asserted, and we enter into the relevant response state as shown with the `yes` and `no` transitions in Figure 5.17. We call the state where contention is asserted the *c-positive* response, and the state where the absence contention is detected the *c-negative* response. The next section explores the heuristics and methods by which we detect contention corresponding to the left side of Fig-

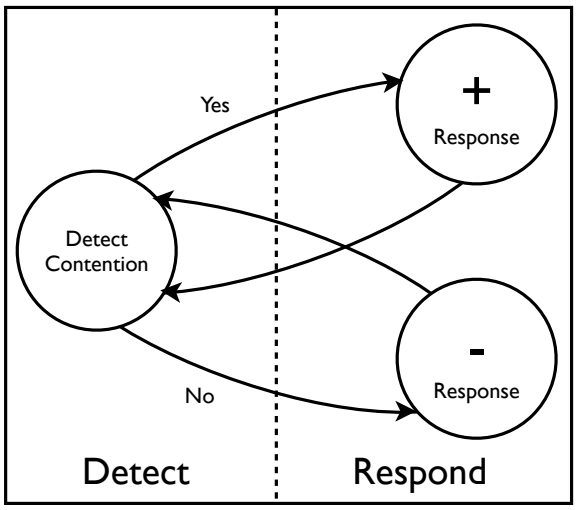


Figure 5.17: Basic Detection Response

ure 5.17, and the subsequent section explores CAER's contention responses corresponding to the right side.

5.3.4 Detecting Contention with CAER

Before CAER can react to contention in the shared cache, it must first detect that the applications are indeed contending. We have developed two heuristics for this task: a *burst shutter* approach and a *rule based* approach. These heuristics run continuously throughout the lifetime of CAER to detect and respond to contention.

Burst-Shutter Approach

If our batch application's execution is going to increase the last level cache misses in the neighboring latency-sensitive application, we should be able to see that spike in misses when the batch application has a burst of execution. That is, if the latency-sensitive application is running alone while the batch application is halted, when the batch application then has a burst of exe-

Algorithm 4: CAER Shutter Burst Algorithm

```
1  footnotesize
   Description: This main loop is executed throughout the lifetime of
        the host application. (pause_self is used to signal whether to
        pause execution for the next period)
2  count ← 0;
3  while application running do
4      update l_window with llc_misses;
5      update r_window with neighbors_llc_misses;
6      count++;
7      pause_self ← true;
8      if count equals switch_point then
9          foreach e in r_window until switch_point do
10             steady_average ←
               steady_average + (e/(Size(r_window) − switch_point)
11         end
12         pause_self ← false;
13     end
14     if (count > switch_point) and (count < end_point) then
15         pause_self ← false;
16     end
17     if count equals end_point then
18         foreach e in r_window from switch_point to end_point do
19             burst_average ←
               burst_average + (e/(end_point − switch_point)
20         end
21         if ((burst_average − steady_average) >
           noise_thresh)and(burst_average >
           (steady_average ∗ (1 + impact_factor))) then
22             contending ← true;
23         end
24         else
25             contending ← false;
26         end
27     end
28 end
```

cution, we should see a sharp increase in the last level cache misses of the latency-sensitive application. We perform this analysis online as follows:

1. We have a number of periods where we halt the execution of the batch application and collect samples of the last level cache misses of the

latency-sensitive application.

2. We then record the average last level cache miss rate.

3. We then have a number of periods where we execute the batch at full force (i.e. burst) and record the misses of the latency-sensitive application.

4. We calculate the average miss rate for these periods.

5. If the number of cache misses are significantly higher in the burst case, we assert the batch application is impacting the miss rate of the latency-sensitive application and report contention, else we report no contention.

The corresponding algorithm is presented in Algorithm 4. There are a number of parameters that can be tuned. We must determine how long (as in how many periods) we would like to halt the batch process's execution, how long the burst should last, and how high the sharp increase should be before asserting contention. In Algorithm 4 these parameters correspond to setting the `switch_point`, `end_point` and `impact_factor`.

Rule-Based Approach

Our rule based approach is more closely based on the premise of our hypothesis. Remember our hypothesis is that if two or more applications are simultaneously missing heavily in the last level shared cache of the micro architecture, they are both making heavy usage of the cache and probably evicting each others data (i.e. contending). The rule based heuristic tries to test this directly. The basic intuition says, if the latency-sensitive application is not missing in the cache heavily, it is probably not suffering from cache contention, and also if the batch application is not missing

Algorithm 5: CAER Rule Based Algorithm

```
1  footnotesize
2  while application running do
3      update l_window with llc_misses;
4      update r_window with neighbors_llc_misses;
5      contending ← true;
6      foreach e in l_window do
7          average ← average + (e/Size(l_window))
8      end
9      if average < usage_thresh then
10         contending ← false
11     end
12     average ← 0;
13     foreach e in r_window do
14         average ← average + (e/Size(r_window))
15     end
16     if average < usage_thresh then
17         contending ← false
18     end
19 end
```

heavily in the cache, it is probably not using or at least not contending in the cache very much. This heuristic works by maintaining a running average of the last level cache miss windows for both the latency-sensitive, and batch applications. When this average for either application dips below a particular threshold, we assert that we are not contending, otherwise we report contention. Algorithm 5 presents the corresponding algorithm. In this heuristic the parameters include the size of the window and defining what missing heavily means. In the algorithm these correspond to `window` and `usage_thresh`.

Responding to Contention with CAER

As Figure 5.17 shows, after detecting contention we transition into one of the response states, either *c-negative* or *c-positive*. In these states the CAER runtime environment can respond by dynamically modifying and adapting

the batch application under which it runs. In this work CAER reacts to contention by enforcing a fine grained throttling of the execution of the batch application to relieve pressure in the shared cache.

Our CAER runtime environment currently employs two throttling based dynamic contention response mechanisms: a *red-light green-light* approach, and a *soft locking* approach. Our red-light green-light approach, as the name implies stops or allows execution for a fixed or adaptive number of periods, based on the outcome of our contention detection phase. The red-light part of this response technique correlates to the c-positive result, the green-light correlates to the c-negative result. An adaptive approach can be applied, increasing the length if the detection phase is consistently producing the same result. In our CAER runtime environment we use this *red-light green-light* response with our burst shutter approach.

Our soft locking response technique applies a *soft lock* on the shared last level cache until the cache is no longer being used heavily by the latency-sensitive application. The amount of pressure placed on the cache by the latency-sensitive application is measured using the same performance monitoring information used for the contention detection phase. The batch application is allowed to fully resume execution when the pressure on the cache subsides. In our CAER runtime environment we use this response technique with our rule based approach.

5.3.5 The Effectiveness of CAER

The goal of our contention aware execution runtime is to dynamically detect and respond to contention on real commodity multicore processors. We aim to minimize the cross-core interference penalty (overhead of the latency-sensitive application due to contention) and maximize the utilization of the chip. We demonstrate the effectiveness of our CAER environment by show-

ing a considerable reduction in this cross-core interference penalty when allowing co-location, while achieving a significant increase of chip utilization compared to disallowing co-location.

Experimental Design

Our CAER prototype supports two applications, one deemed latency-sensitive and the other a throughput-oriented batch application. We use the SPEC2006 benchmark (C/C++ only) and run all programs to completion using their reference inputs. We use the Intel Core i7 (Nehalem) 920 Quad Core architecture to perform our experimentation. This processor has three levels of cache, the first two private to each core, the third shared across all cores. The sizes of the L1 and L2 caches are 16kb and 256kb respectively. The L3 cache is 8mb and inclusive to the L1 and L2. The system used has 4gb of main memory, and runs Linux 2.6.29.

In the experiments shown here, the `lbm` benchmark served as our batch application and was co-located on a neighboring core. The main benchmark is assumed to be the latency-sensitive application. `Lbm` was chosen as our batch application because it presents an interesting adversary as it makes heavy usage of the L3 cache. We have performed complete runs using other benchmarks such as `libquantum` and `milc` and produced very similar results. Note that adversaries that make light usage of the L3 cache present more trivial scenarios; contention occurs when two or more applications are making heavy usage of the last level cache. As presented shortly, our experimentation covers cases where the latency-sensitive application make both light and heavy usage of the shared cache.

We have scripted our SPEC runs to launch the latency-sensitive application shortly after the batch is launched. As our applications run, CAER logs the decisions it makes and wall clock execution time of our latency-

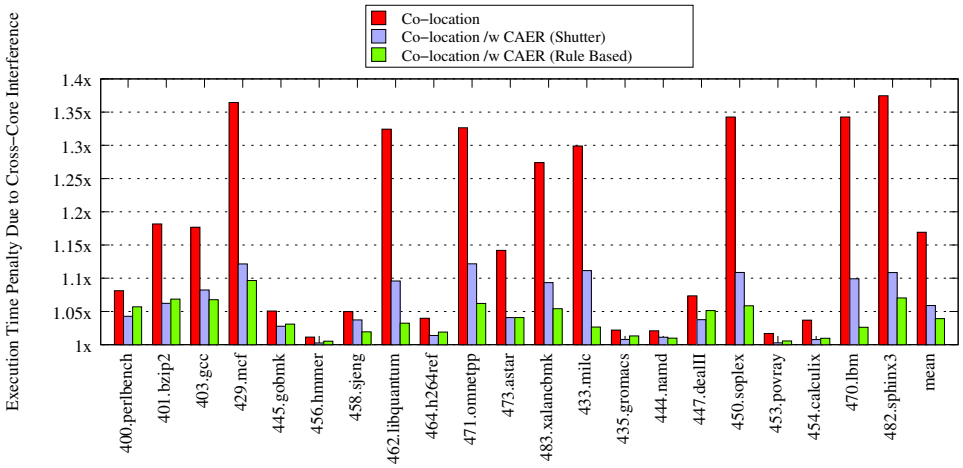


Figure 5.18: Investigating the reduction in cross-core interference penalty.

sensitive application running on CAER M. In the few cases the `lbm` (batch) benchmark completes before the latency-sensitive we automatically and immediately relaunch it and aggregate logs.

Minimizing Contention and Maximizing Utilization

First we evaluate the reduction in cross-core interference penalty due to contention when running on our CAER environment. In Figure 5.18 we show the slowdown in execution time due to contention when we co-locate the latency-sensitive and batch applications. The first bars show the cross-core interference penalty when co-locating the native applications directly on multicore chip. The second bars shows the cross-core interference penalty when co-locating the native applications on CAER with the *burst shutter* heuristic. The last bars show this co-location on CAER with the *rule based* approach.

As Figure 5.18 shows we significantly reduce the cross-core interference penalty for the wide range of SPEC2006 benchmarks. Our *burst shutter* contention detection technique uses the *red-light green-light* response with a

response length of 10 periods. The `impact threshold` in Algorithm 4 for the burst shutter detection is set to 5%, meaning if the batch application burst causes a spike of 5% or more in last level cache misses of the latency-sensitive application we assert contention. Using this approach CAER brings the overhead due to contention from 17% down to 6% on average, while gaining close to 60% more utilization of the processor over running the latency-sensitive application alone, which can be seen in Figure 5.19.

Our *rule based* contention detection technique uses the *soft locking* response and the `usage threshold` found in Algorithm 5 is set to `1500`, meaning we have to see an average of 1500 or more last level cache misses per period (1 ms) to assert heavy usage of the cache. Using this approach CAER brings the overhead due to contention from 17% down to 4% on average, while gaining 58% more utilization of the processor over running the latency-sensitive application alone, as show in Figure 5.19.

Our *rule based* CAER contention detection approach slightly outperforms our *shutter based* approach on average. However, the *shutter based* approach has a critical capability that our *rule based* approach lacks.. The *burst shutter* approach is highly tunable to the QoS requirements of the application. The impact threshold determines how much cross-core interference the latency application is willing to withstand; this provides a "knob" which intuitively sets the sensitivity of detection. Here we use "sensitivity" to mean the amount of impact needed to trigger a *c-positive* response. Although the *rule based* approach is also tunable as to how conservative or liberal the definition of "heavy usage" of the cache is, it provides a less useful abstraction as the number of cache misses alone does not imply contention. As the goal of this evaluation is to demonstrate the effectiveness of our CAER runtime environment and its applicability to current multicore architecture, we reserve further investigation of the heuristic tuning space

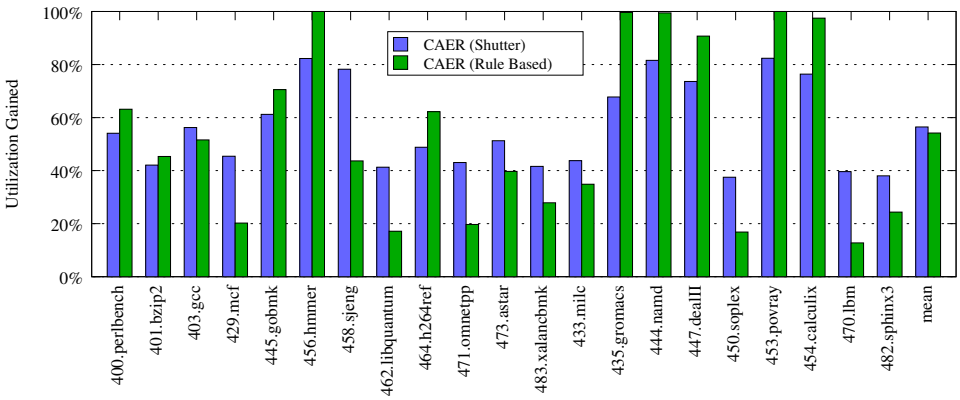


Figure 5.19: Maximizing Utilization (Higher is Better)

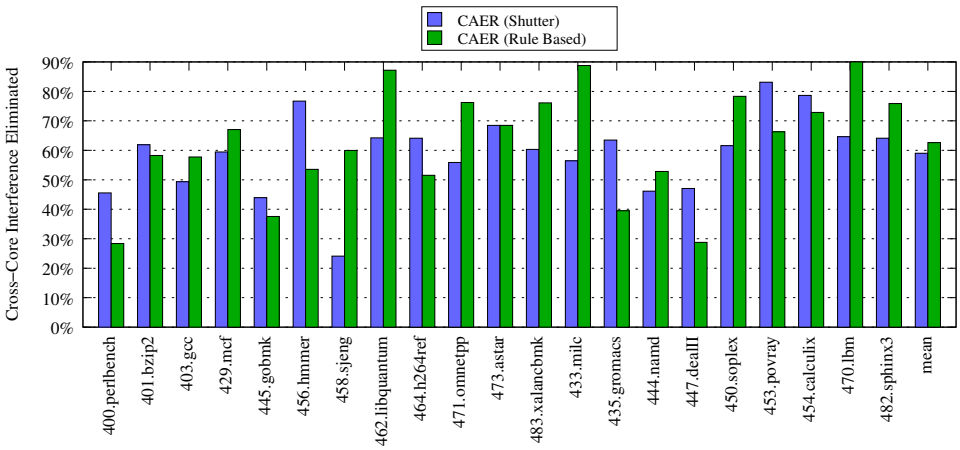


Figure 5.20: Minimizing Cross-Core Interference (Slowdown Eliminated, Higher is Better)

for future work.

Figures 5.19 and 5.20 further illustrate CAER's effectiveness. As mentioned before, Figure 5.19 shows the utilization gained on the multicore processor when co-locating the latency-sensitive and batch applications using CAER. Figure 5.20 is another way to represent the decrease in cross-core interference penalty shown in Figure 5.18, showing the percentage of the cross-core interference penalty eliminated. For both of these Figures, higher is better. Running the latency-sensitive application alone will provide 100% cross-core interference elimination but 0% utilization gained. Running the

applications together will provide 0% cross-core interference elimination but will have 100% utilization gained. Our goal is to maximize both while running both application on our CAER framework. It is important to note that utilization gained and cross-core interference eliminated are two separate units of measurement, so 50% cross-core interference eliminated for 50% more utilization can be a great result depending on the *cross-core interference sensitivity* of the latency-sensitive application. We provide a deeper exploration of *cross-core interference sensitivity* in the following section.

Adapting to Cross-Core Interference Sensitivity

The amount of performance impact an application can experience due to contention for shared resources differs from application to application. We call this application characteristic its *cross-core interference sensitivity.* This characteristic can also be determined by the amount of reliance an application puts on a shared resource. Applications whose working set fits in its core-specific private caches are *cross-core interference insensitive.* Applications whose working set uses shared cache, memory, etc, are *cross-core interference sensitive.*

When performing contention detection and response the handling of cross-core interference insensitive and cross-core interference sensitive applications should be different. More concretely, the amount of utilization that is sacrificed to reduce contention of a cross-core interference sensitive application should be higher than the cross-core interference insensitive application. For example, an application a is 50% slower when experiencing contention x, while another application b is 4% slower when contending with x. We say application a is more cross-core interference sensitive than b. To eliminate half of the cross-core interference penalty of a is more valuable than b, meaning the benefit gained, a 20% increase in speed, with a is better

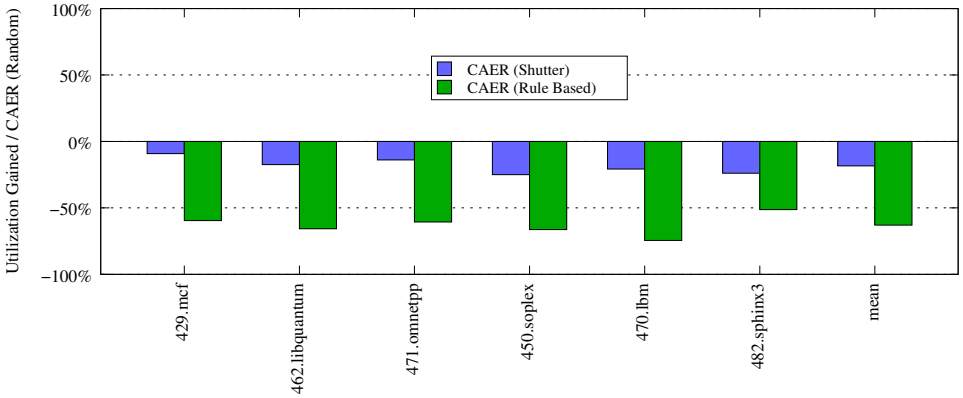


Figure 5.21: Utilization gained relative to random for 6 most cross-core interference sensitive applications.

than the 1.9% speed up in b. Thus, we should be willing to sacrifice more utilization to eliminate 50% of the cross-core interference penalty for a than b since a is more cross-core interference sensitive.

Lets take `mcf` as application a and `namd` as b. As shown previously in Figure 5.18 `mcf` suffers a 36% slowdown when contending with `lbm`, `namd` only suffers a 2% performance degradation. Clearly `mcf` is more latency-sensitive than `namd`, therefor a good contention detection and response approach will be able to detect these different cross-core interference sensitivities and sacrifice more utilization for the former case. CAER does exactly this. For `mcf` CAER *burst shutter* approach sacrifices 36% more utilization to accommodate `mcf`'s cross-core interference penalty, and CAER *rule based* sacrifices 80% more utilization.

Contention Detection Accuracy

When detecting contention it is possible to have both false positives and false negatives. A false positive occurs when contention is detected where there is none. A false negative occurs when no contention is detected where there is contention. To evaluate a heuristic's ability to accurately detect

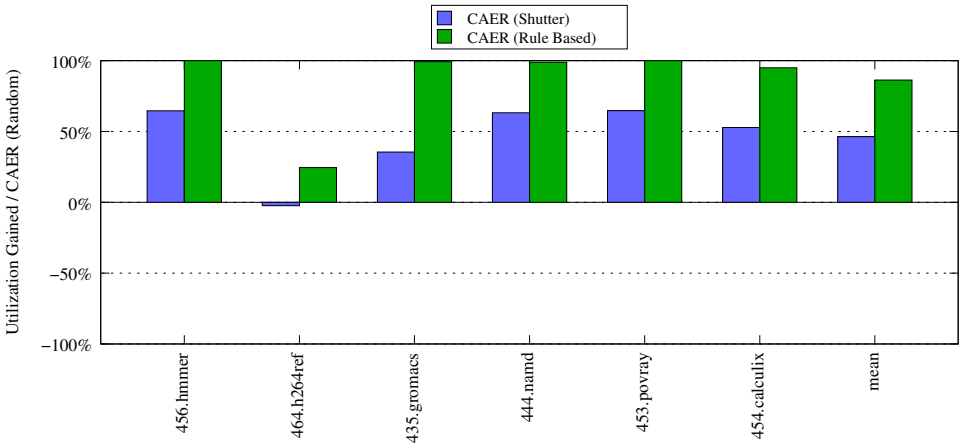


Figure 5.22: Utilization gained relative to random for 6 least cross-core interference sensitive applications.

contention we have developed a baseline random heuristic. This heuristic reports contention with probability P and no contention with probability $1-P$. In our experiments P equals 0.5. To respond to contention this heuristic uses the *red-light green-light* with a length of 1 period. To illustrate a CAER heuristic's ability to detect contention accurately we use the following

$$A = \frac{U_h}{U_r} - 1 \tag{5.2}$$

where U_h is the utilization gained from a heuristic h, and U_r is the utilization gain with the random heuristic. Figures 5.21 and 5.22 demonstrates the contention detection accuracy of the *burst shutter* and *rule based* heuristics for the six most, and six least cross-core interference sensitive benchmarks respectively. The y-axis corresponds with the calculation of A from the equation. Figure 5.21 shows that, for cross-core interference sensitive benchmarks, our CAER heuristics sacrifices more utilization than the random technique, indicating that our detection is correctly responding to these applications as high contenders (i.e. cross-core interference sensitive). Figure 5.22 shows the opposite for cross-core interference insensitive

benchmarks. The heuristics gain much more utilization than the random heuristics, indicating we are correctly responding to these workloads as low contenders.

Also note that any inversion in this response to cross-core interference penalty indicates inaccurate contention detection. Gaining more utilization for a cross-core interference sensitive application than the random heuristic represents a false negative (asserting no contention where there is contention). And contrarily, gaining less utilization for cross-core interference insensitive applications represents a false positive (asserting contention where there is none).

Chapter 6

Mitigating Interference in WSCs with Precision

Contents

In this chapter, we address *the oblivion of interference* at the cluster and machine levels by providing novel mechanisms for precisely measuring and managing interference within execution environments in WSCs. First,

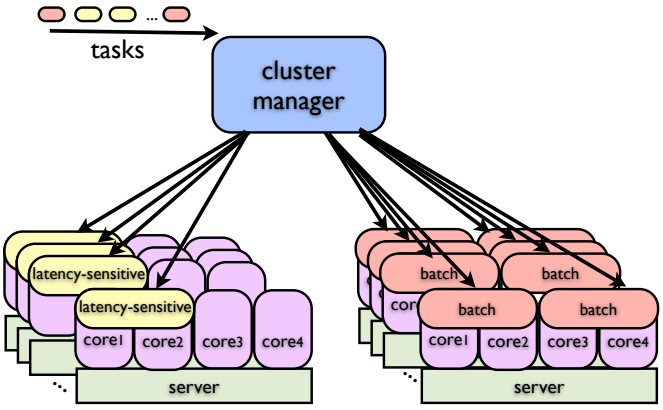


Figure 6.1: Task placement in a cluster. The cluster manager does not co-locate latency-sensitive applications with others to protect their QoS from performance interference, causing low machine utilization.

we present a general characterization methodology, *Bubble-Up*, that enables the *precise prediction* of the performance degradation that results from contention for shared resources in the memory subsystem. We then demonstrate how Bubble-Up can be used to steer co-location decisions at the cluster level using 17 production Google workloads and production machines to significantly improve the utilization of WSCs. Finally, we provide a general metric and*direct* measurement technique for quantifying and characterizing an application's interference sensitivity that captures contentious phases and code regions.

6.1 Precisely Predicting CCI Performance Penalty

As discussed in previous chapters, there is a trade-off between the QoS performance of latency-sensitive applications and the machine utilization in modern WSCs. As shown in Figure 6.1, in modern WSC systems, co-location between latency-sensitive jobs and other jobs is disallowed due to the inability to precisely predict how the performance of latency-sensitive

jobs will be affected by co-running jobs. However, this overprovisioning is often unnecessary as co-locations may or may not result in significant performance interference. It is the inability to *precisely* predict the performance impact for a given co-location that leads to the heavy handed solution of simply disallowing co-location. On the other hand, without prediction, the brute-force approach of profiling all possible co-locations' performance interference to guide co-location decisions is prohibitively expensive. The profiling complexity for all pairwise co-locations is $O(N^2)$ (N as the number of applications). With hundreds to thousands of applications running in a WSC, and the frequent development and updating of these applications, a brute-force profiling approach is not practical. A linear approach is need.

The goal of this work is to provide a linear approach for the *precise* prediction of the performance degradation that results from contention for shared resources in the memory subsystem. A *precise* prediction is one that provides an expected amount of performance lost when co-located. With this information, co-locations that do not violate the QoS threshold of an application can be allowed, resulting in improved utilization in the WSC.

This is a challenging problem as effects such as contention is not explicitly visible or manageable through the software interface to commodity microarchitectures. The most relevant related work aims to classify applications based on how aggressive they are for the shared memory resources and identify co-locations to reduce contention based on the classification [19, 60, 63, 76, 80, 81, 125, 126, 132]. However, prior work has not presented a solution to precisely predict the amount of performance degradation suffered by each application due to co-location, which is essential for co-location decisions of latency-sensitive applications in WSCs. In this section, we present such a solution: The **Bubble-Up** methodology.

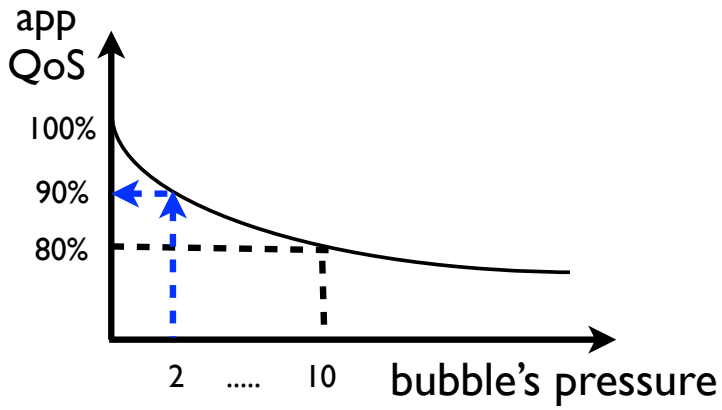


Figure 6.2: Example sensitivity curve for A. Assuming B's pressure score is 2 we can predict A will be performing at 90% of full performance.

6.1.1 The Bubble-Up Methodology

The key insight of Bubble-Up is that predicting the performance interference of co-running applications can be decoupled into 1) measuring the pressure on the memory subsystem an application generates, and 2) measuring how much an application suffers from different levels of pressure. The underlying hypothesis is both pressure and sensitivity can be quantified using a common pressure metric. Having such a metric reduces the complexity of co-location analysis. As opposed to the brute force approach of profiling and characterizing every possible pairwise co-location, Bubble-Up only requires characterizing each application once to produce precise pairwise interference predictions (e.g. $O(N)$).

Bubble-Up is a two-step characterization process. First, each application is tested against an expanding *bubble* to produce a *sensitivity curve*. The *bubble* is a carefully designed stress test for the memory subsystem that provides a "dial" for the amount of pressure applied to the entire memory subsystem. This bubble is run along with the host application being characterized. As this dial is increased automatically (expanding the bubble), the impact on the host application is recorded, producing a sensitivity curve

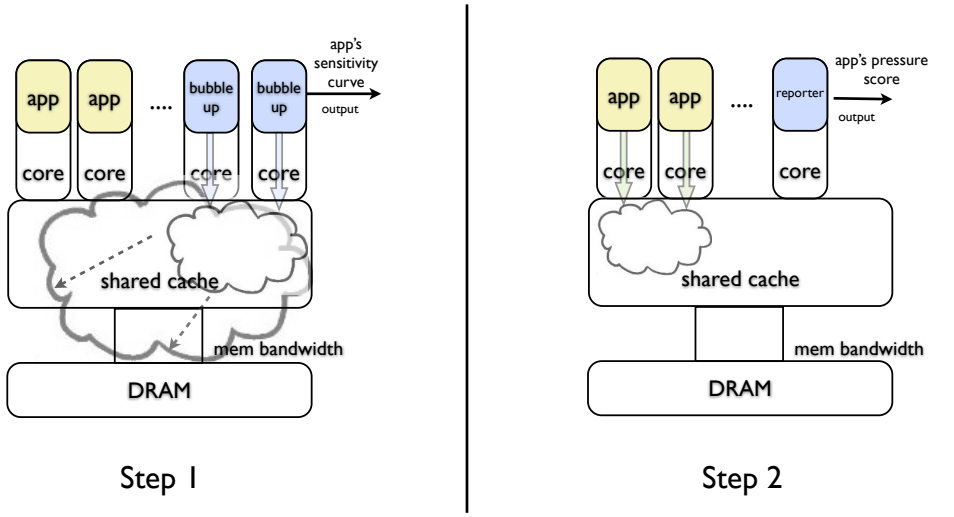


Figure 6.3: Bubble-Up Methodology. In Step 1, we characterize the *sensitivity* of the host application task to pressure in the memory subsystem using a *bubble*. In Step 2, we characterize the *contentiousness* of the host application in terms of the amount of pressure it causes on a *reporter*.

for the host application such as the one illustrated in Figure 6.2. On the y-axis, we have the normalized QoS performance of the application (latency, throughput, etc), and the x-axis shows the bubble pressure. In the second step, we identify a pressure score for the application using a *bubble pressure score* reporter. After these two steps of the Bubble-Up methodology is applied to each application, we have a sensitivity curve and a pressure score for each application. Given two applications A and B, we can then predict the performance impact of application A when co-located with application B by using A's sensitivity curve to look up the relative performance of A, at B's pressure score. In the example shown in Figure 6.2, B has a pressure score of 2, and as we can see from A's sensitivity curve, A's predicted QoS with that co-location is 90%.

Two Step Methodology

Figure 6.3 illustrates the Bubble-Up methodology. The two steps to Bubble-Up are,

1. In Step 1, we characterize the *sensitivity* of each application task to pressure in the memory subsystem. In this step, we use a carefully designed stress test we call the *bubble* to iteratively increase the amount of pressure applied to the memory subsystem (e.g. bubble up in the subsystem). As we incrementally increase this pressure "dial", we produce what we call a *sensitivity curve* for the application with QoS on the y-axis and pressure on the x-axis. This sensitivity curve shows how each application's QoS degrades as pressure increases.

2. In Step 2, we characterize the *contentiousness* of each application task in terms of its pressure on the memory subsystem. We call this measure of contentiousness a *bubble score*. To identify the bubble score of an application, we use a *reporter* which observes how its own performance is affected by the application to generate a score for the application.

With the sensitivity curves and bubble scores of each application we are able to precisely predict the performance degradation from arbitrary co-locations. It is important to note that step one needs only to be applied to applications whose QoS needs to be enforced. Step two only needs to be applied to the applications that may threaten an application's QoS.

Modeling Bubble-Up and Error

In this section, we present the formal modeling of our Bubble-Up methodology and the source of errors. We first provide a general model the performance degradation an application A suffers when co-running with other applications as,

$$Deg_{A_C} = \sum_{i}^{N} (S_{AR_i} \times P_{CR_i}) \tag{6.1}$$

where A is the application, C is the co-runner or set of co-runners, Deg_{A_C} is the A's degradation when running with C, R_i is a shared memory resource component such as shared cache, memory bandwidth or memory controller, P_{CR_i} is the pressure C generates on the shared resource R_i, and S_{AR_i} is A's performance sensitivity to the pressure on the shared resource R_i. The total degradation is the sum of the stalled cycles caused by contention for each shared resource.

The main component of the Bubble-Up methodology is an expandable bubble, which functions as a "dial" for pressure on the memory system. We denote the bubble dial levels as a set $B : \{B_0, B_1, ..., B_M\}$ where M is the number of dial levels for the bubble. As we dial up, the bubble generates an increasing amount of pressure on each shared resource R_i, denoted as $P_{B_j R_i}$. In step 1 of Bubble-Up, we increase the bubble size and at each bubble dial j, we measure the degradation of application A when co-running with the bubble B_j. The result is a set of A's degradations at varying bubble sizes: $\left\{Deg_{A_{B_0}}, Deg_{A_{B_1}}, ..., Deg_{A_{B_M}}\right\}$. This set is the discretized sensitivity curve of application A.

When the bubble reporter provides a pressure score of a co-running application C, it reports this pressure in the form of a bubble dial level, B_K, that generates the closest amount of pressure as C (for details see Section 6.1.1). To approximate application A's degradation when co-running with C (Deg_{A_C}), Bubble-Up then uses A's degradation when running with bubble B_K ($Deg_{A_{B_K}}$).

Because the pressure score is using the bubble's pressure to approximate an application's pressure, Bubble-Up introduces a small amount of error when predicting the degradation of A. This error stems from the mismatch of the relative pressures applied to the various individual shared resources. As we discuss in Section 6.1.1, this mismatch is minimized by designing

the bubble to prioritize those resources that act as first-order effects in the degradation to A. By substituting Deg_{A_C} and $Deg_{A_{B_K}}$ using Equation 6.1, we can formally model the prediction error as:

$$Error = \left| Deg_{A_C} - \widetilde{Deg_{A_C}} \right| \tag{6.2}$$

$$= \left| Deg_{A_C} - Deg_{A_{B_K}} \right| \tag{6.3}$$

$$= \sum_{i}^{N} |S_{AR_i} \times P_{B_K R_i} - S_{AR_i} \times P_{CR_i}| \tag{6.4}$$

Step One: Characterizing Sensitivity

As described in Section 2.1, each task is configured to use a prescribed number of cores for the cluster level bin packing algorithm used to assign tasks to machines in the WSC. This number is usually less than the number of cores available on a socket. The bubble is run on the remaining cores. It is important to understand that there is no *correct* design for the bubble. Each design needs only to approximate varying levels of pressure, and there may be many good designs. In this section, we present one such design, and show that our single bubble design is effective across myriad application workloads and architectures. Keep in mind that there are a number of assumptions made about the architectures for which this type of bubble design is applicable. Most importantly we assume the microarchitecture uses shared last level caches, memory controller, and bandwidth to memory.

The Art of Bubble Design Although there may be many ways to design a bubble, to arrive at a good design that is not prone to error and imprecision, there is a set of key requirements and guidelines that apply generally to bubble design.

1. ***Monotonic Curves*** - As the bubble's pressure increases (turning the

pressure "dial" up) the amount of performance interference should also increase monotonically. Assuming the host application task is sensitive to cross-core interference, higher amounts of pressure should result in worse performance.

2. ***Wide Dial Range*** - The pressure dial should have a range that captures the contentiousness of all the application tasks of interest. It should start from essential no pressure, and incrementally increase pressure to a point close to the maximum possible pressure, or at least worse than the most contentious application task in the set.

3. ***Broad Impact*** - The bubble should be designed to apply pressure to the memory subsystem as a whole, not stressing a single component in the memory subsystem. However, keep in mind that, as mentioned in Section 6.1.1, error is introduced in the difference in component pressure relative to the host task's sensitivity. This error is generally minimized if first-order effects are prioritized.

Designing the Bubble The design principle of our bubble is to use working set size as our measure of pressure. For a given working set size, we perform memory operations in software to excercise that working set as aggressively as possible.

Our bubble is a multithreaded kernel that generates memory traffic using both loads and stores with a mixture of random and streaming accesses. The number of threads spawned is based on the configuration file of the task being characterized. The pressure "dial" we use is the working set size on which our kernel works. For example, a pressure size of 1 means our bubble will continuously smash a 1MB chunk of memory. As we increase the pressure, we increase our kernel's working set size. This increases the amount of data

```
//Super cheap rand using a linear feedback shift register
unsigned lfsr;
#define MASK 0xd0000001u
#define rand (lfsr = (lfsr >> 1) ^ (unsigned int) \
    (0 - (lfsr & 1u) & MASK))

unsigned int footprint_size=0; //Size of footprint
unsigned int dump[100]; //Dumps (manual ssa)

#define r (rand%footprint_size)
```

Figure 6.4: Bubble's LFSR number generator.

```
while(1)
{
  dump[0]+=data_chunk[r]++;
  dump[1]+=data_chunk[r]++;
  dump[2]+=data_chunk[r]++;
  ...
  dump[98]+=data_chunk[r]++;
  dump[99]+=data_chunk[r]++;
}
```

Figure 6.5: Manual SSA for no dependencies.

being pumped through the memory subsystem, as computation is not the bottleneck. Figures 6.4 – 6.6 show the key design points for our bubble.

As shown in Figure 6.4, we use a *linear shift feedback register* (LFSR) based psuedo random number generator as opposed the `rand` function provided by the `C` standard library. Minimizing the amount of computation in between memory accesses is critical to maximize the activity applied to a particular footprint. Using the standard library incurs a significant amount of computation between random numbers. However, an LFSR implementation requires only a few cycles to arrive at the next random number on modern processors. In our LFSR implementation, we use a mask of `0xd0000001u`, which produces a *period* of 2^{32} random numbers.

```
while(1)
{
  double *mid=bw_data+(bw_stream_size/2);
  for(int i=0; i<bw_stream_size/2; i++)
  {
    bw_data[i]=scalar*mid[i];
  }
  for(int i=0; i<bw_stream_size/2; i++)
  {
    mid[i]=scalar*bw_data[i];
  }
}
```

Figure 6.6: Streaming access for bandwidth.

Figure 6.5 shows the random memory accesses used in our bubble. Here we have manually constructed a basic block of 100 memory operations that are in *single static assignment* form such that there are no dependencies in between operations. This basic block has a high level of instruction level parallelism to maximize the number of operations applied to the footprint of the kernel.

As Figure 6.6 shows, we also use a streaming access pattern in our bubble. This implementation is based on the scalar portion of the STREAM bandwidth stressing benchmark. Using this access pattern further stresses the bandwidth to memory, and also triggers the prefetcher, another shared resource at the level of the shared last level cache.

Step Two: Bubble Scoring

The second step of Bubble-Up is to produce a pressure score for an application task that describes what size of bubble is representative of that task. To detect this score we use a bubble score *reporter*. The reporter is a carefully designed single threaded workload that is sensitive to contention. Like the bubble, the reporter is only designed once and then can be used with myriad applications and architectures.

Designing the Reporter The reporter's own sensitivity to performance interference is used as a basis for reporting how its own performance has been affected by the pressure generated by a host application. The impact *felt* by the reporter is then translated in terms of the predicted bubble score of the host application. The only guideline to designing a good reporter is to have a *broad sensitivity*, e.g. it should be sensitive to the memory subsystem holistically.

[**Designing the Reporter**] Like the bubble, there is also no *correct*

design for the reporter. However, unlike the bubble, there is more flexibility in designing the reporter. This flexibility comes from the fact that the reporter is trained, and the sensitivity curve serves as a rubric for score reporting, no matter the shape. To implement the reporter, we use a mixture of random accesses and streaming accesses similar to those used for the bubble itself, without the high ILP. The working set of the reporter is about 20MB, thus it uses the last level cache, memory bandwidth and prefetcher.

Before the reporter can be used, it must first be trained using the bubble on the architecture for which it will be reporting. This training involves running the reporter against the bubble on the architecture of interest, and collecting the sensitivity curve of the reporter. This needs to be done only once. The reporter can then use its own sensitivity curve to translate a performance degradation it suffers to the corresponding bubble score. The curve is essentially used in reverse. Instead of using scores to predict QoS, we use the QoS of the reporter to ascertain the score of the co-located application.

6.1.2 Large-Scale WSC Workloads

In this section, we present the large-scale WSC application workloads used in this work, and characterize their susceptibility to performance interference due to contention for the shared memory sybsystem resources. We also introduce **SmashBench**, our in house benchmark suite for the characterization of performance interference.

Large-Scale Data Intensive Workloads

While a large portion of the world's computation is housed in the cloud, little is known about the application workloads that live in this computing domain. The characteristics of the tasks that compose a large scale

workload	description	type	metric
bigtable	A distributed storage system for managing petabytes of structured data	latency-sensitive	user time (secs)
ads-servlet	Ads sever responsible for selecting and placing targeted ads on syndication partners sites	latency-sensitive	cpu latency (ms)
maps-detect-face	Face detection for streetview automatic face blurring	batch	user time (secs)
maps-detect-lp	OCR and text extraction from streetview	batch	user time (secs)
maps-stitch	Image stitching for streetview	batch	user time (secs)
search-render	Web-search frontend server, collect results from many backends and assembles html for user.	latency-sensitive	user time (secs)
search-scoring	Web-search scoring and retrieval (traditional)	latency-sensitive	queries per sec
nlp-mt-train	Language translation	latency-sensitive	user time (secs)
openssl	Secure Sockets Layer performance stress test.	latency-sensitive	user time (secs)
protobuf	Protocol Buffer, a mechanism for describing extensible communication protocols and on-disk structures. One of the most commonly-used programming abstractions at Google.	latency-sensitive	aggregated
docs-analyzer	Unsupervised Bayesian clustering tool to take keywords or text documents and "explain" them with meaningful clusters.	both	throughput
docs-keywords	Unsupervised Bayesian clustering tool to take keywords or text documents and "explain" them with meaningful clusters.	both	throughput
rpc-bench	Google rpc call benchmark	both	throughput
saw-countw	Sawzall scripting language interpreter benchmark	both	user time (secs)
goog-retrieval	Web indexing	batch	ms per query
youtube-x264yt	x264yt video encoding.	batch	user time (secs)
zippy-test	A lightweight compression engine designed for speed over space.	both	user time (secs)

Table 6.1: Production WSC Applications

web-service vary significantly. In addition to data retrieval tasks, there are compute-intensive tasks for the analysis, organization, scoring, and preparation of information for applications such as search, maps, ad serving, etc.

Table 6.1 presents a number of key application tasks housed in Google's production WSCs. These application tasks comprise a majority of the CPU cycles in arguably the largest WSC infrastructure in the world. In addition to each application task's name, Table 6.1 shows the description, priority class, and key optimization metric for each workload. Each application task

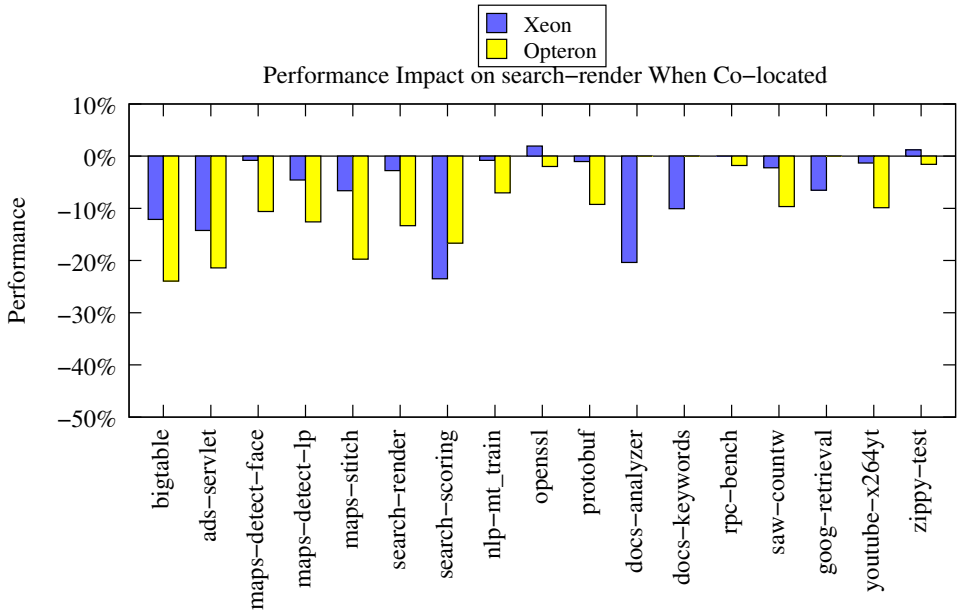


Figure 6.7: The performance degradation suffered by `search-render` when co-located with each of the other WSC applications on Xeon and Opteron

corresponds to an actual binary that is run in the WSC. Application tasks that are user-facing, both directly and indirectly, are classified as *latency-sensitive* as the response time is paramount. Throughput oriented tasks that are not user-facing are classified as *batch*. Notice that some tasks may be used in both roles, and are denoted as *Both* in the table. The column marked *metric* shows the key metric for each application. In the context of this work, each task's QoS is defined to be its performance along this metric.

We have highlighted `search-render` in Table 6.1. This task is responsible for assembling the final view of the search process for the user, which includes assembling scored search results (including web, image, and video), relevent ads from the ads-servlet, etc. This task is highly latency-sensitive and presents a compelling case that we use throughout this work to illustrate the necessity and value of Bubble-Up.

Figure 6.7 shows the performance degradation of `search-render` when co-located with the other application tasks shown in Table 6.1. This figure

kernel	access pattern	stress point	w.s. size	type
blockie-small	Dense 3D Matrix Transformations	cpu-bound	7KB	streaming
blockie-medium	Dense 3D Matrix Transformations	cache	10MB	streaming
blockie-large	Dense 3D Matrix Transformations	bandwidth	46MB	streaming
bst-small	Binary Search Tree Traversal	cache (partial)	4MB	structured
bst-medium	Binary Search Tree Traversal	cache	8MB	structured
bst-large	Binary Search Tree Traversal	bandwidth	50MB	structured
lfsr-small	Linear Shift Feedback Register Random Access	cache (partial)	4MB	random
lfsr-medium	Linear Shift Feedback Register Random Access	cache	8MB	random
lfsr-large	Linear Shift Feedback Register Random Access	bandwidth	50MB	random
naive-small	STL Random Access	cache (partial)	4MB	random
naive-medium	STL Random Access	cache	8MB	random
naive-large	STL Random Access	bandwidth	50MB	random
sledge1	Streaming Sparse Matrix Operations	cache	7MB	streaming
sledge2	Streaming Sparse Matrix Operations	bandwidth	42MB	streaming
sledge3	Streaming Sparse Matrix Operations	bandwidth	399MB	streaming

Table 6.2: SmashBench Suite (stress point assumes a last level cache size of 6MB to 12MB)

shows this performance interference on a production six-core Nehalem-based Xeon and a production six-core K10-based Opteron respectively. Each task in the co-location is configured to use three out of the six cores on the same socket of each platform. Note that `docs-analyzer, docs-keywords` and `goog-retrieval` do not have a data point for the Opteron. These particular workloads must be run on the Xeon platform as they have been specially configured for this platform. As shown in the figure, we observe a significant amount of cross-core interference. This has led to a policy to disallow the co-location of `search-render` and other tasks on the same machine. However, some co-locations result in minimal to no interference. With the ability to precisely predict the performance interference suffered, those co-locations can be identified as safe, and the utilization of idle cores can be reclaimed.

SmashBench: Performance Interference Benchmark Suite

It is also important to note that various applications generate contention with various access patterns and working set sizes, which results in differing amounts of pressure across shared resources such as on-chip caches and buses to main memory. To characterize the sensitivity of our large-scale WSC workloads to a spectrum of contention types, and to properly evaluate the effectiveness of Bubble-Up for predicting the performance interference that results from these various types of contention, we have created an in-house benchmark suite of contentious kernels, which we call **SmashBench.**

Table 6.2 shows the various kernels in our SmashBench suite. This suite of contentious workloads were designed specifically to exercise the resources that lie between the cores of a multicore processor and main memory in a spectrum of access patterns and working set sizes. As shown in the figure, SmashBench spans five access patterns. Each pattern continuously performs memory operation on a chunk of memory, which is of the size denoted *w.s. size*. The *stress point* of each application is either the shared on-chip caches or the bandwidth to memory, which is primarily a function of its working set size. Note that this instance of the suite assumes a last level cache size of approximately 6MB to 12MB. Current production WSC deployments house processors with these specifications. It is important to note that while this version of the suite was designed for the platforms used in current production WSCs, it can be easily extended for future processor designs by modifying the working set size.

A good characterization methodology for the precise prediction of performance interference must be sensitive to various types of contention. To highlight this point we perform a simple experiment. Figures 6.8 and 6.9 illustrates the effect of only changing the access pattern while keeping the working set constant, and only changing the working set while keeping the

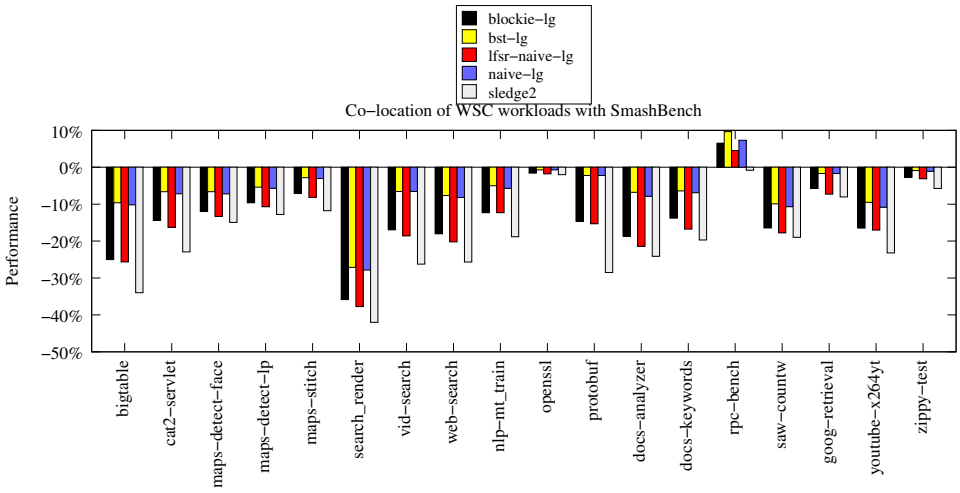


Figure 6.8: For a given working set size, we observe a different amount of interference when varying access pattern.

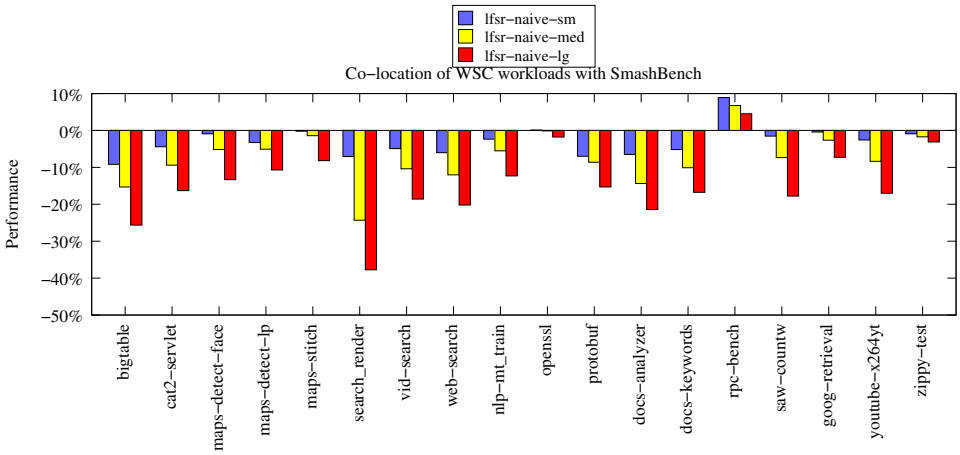


Figure 6.9: For a given access patter, we observe a different amount of interference when varying working set size.

access pattern constant. In Figure 6.8, each of our WSC applications are co-located with our SmashBench benchmarks with a working set size of 50MB. As shown in the figure, even a slight change in the random number generator used (when comparing `lfsr-naive` with `naive`) can result in a large impact on the contentiousness of an application. In addition, as Figure 6.9 shows, changing the working set size when keeping the access pattern the same results in also varing amounts of performance interference.

6.1.3 The Effectiveness of Bubble-Up

In this section, we evaluate the accuracy of Bubble-Up in precisely predicting performance degradations due to interference.

The primary platform used in our evaluation is a six-core Nehalem-based Xeon. The performance metric used to describe the QoS of each Google application is the internal metric as presented in Table 6.1. Each application task is configured to use three cores on the six-core machines and two cores on the quad-core machines. As previously described, during the characterization phase, the bubble runs on the remaining cores.

Sensitivity Curves of WSC Workloads

We first present the sensitivity curves for Google benchmarks generated by our Bubble-Up methodology. The goal is to 1) examine our Bubble-Up design through analyzing the resulted sensitivity curves and 2) to further improve our understanding of how pressure in the shared resources affects the QoS of Google's applications. To generate sensitivity curves, we adjust the pressure Bubble-Up generates and measure an application's QoS under each given pressure. Figures 6.10a to 6.10i present the sensitivity curve of each Google application. For each figure, the x-axis shows the pressure on the shared memory system generated using Bubble-Up's bubble. The

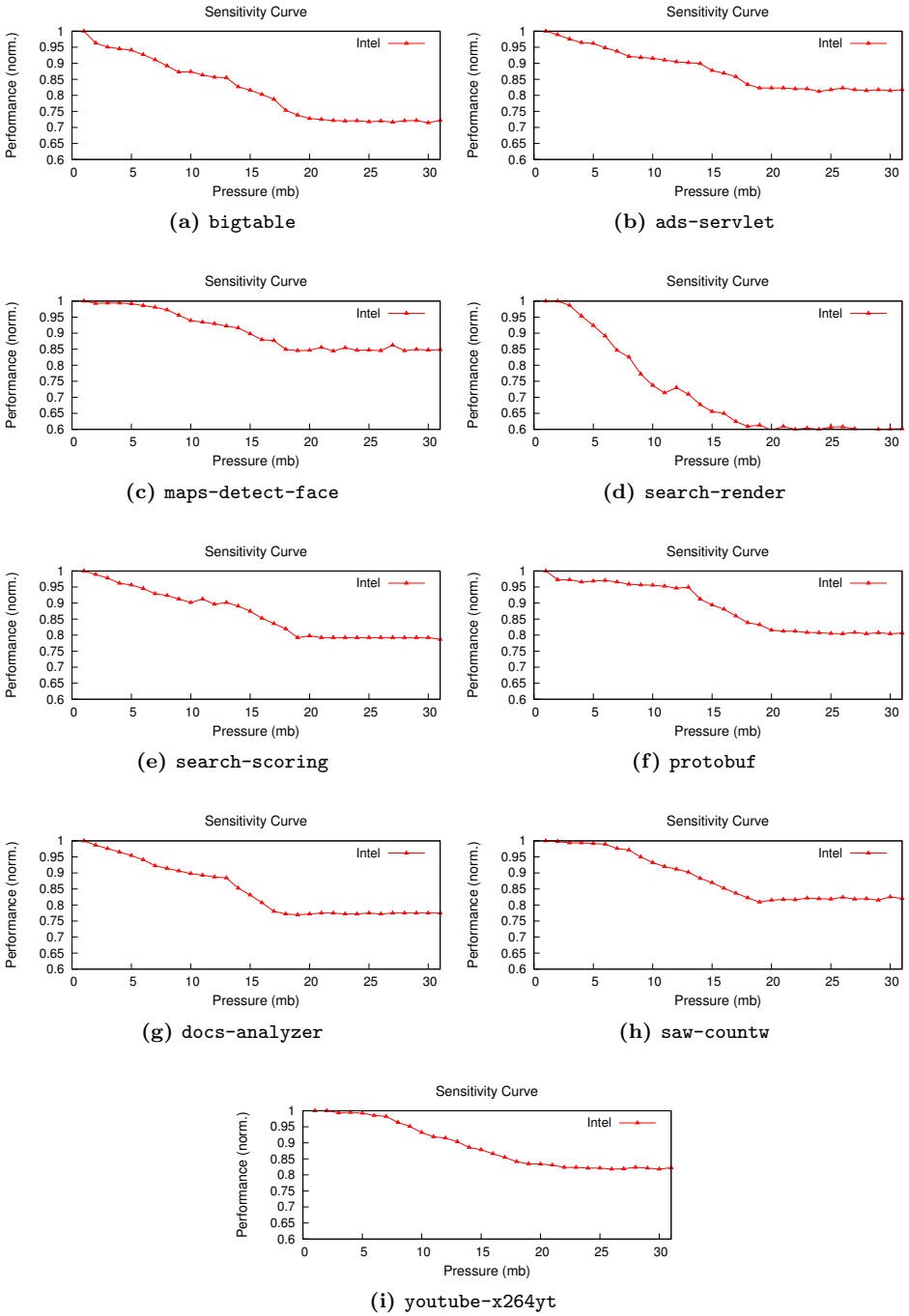


Figure 6.10: Bubble-Up Sensitivity Curves for 9 Highly Sensitive Google Workloads.

y-axis shows the QoS performance of each application, normalized by its performance when it is running alone on the platform.

Bubble Design Figures 6.10a to 6.10i illustrate that our bubble design does indeed have the three properties described in Section 6.1.1. We observe *Monotonic Curves*. In general, each application's QoS is decreasing as the pressure increases. This confirms our hypothesis that we can create an aggregate pressure "dial" in software that negatively and monotonically impacts an application's QoS. We also observe *Wide Dial Range*. The sensitivity curves generally flatten after the Bubble-Up pressure goes beyond 20MB. At this point, the pressure on the shared cache and memory bandwidth saturates and further increase of the pressure would not have much more impact on an application's QoS. Finally, we observe *Broad Impact*. The monotonic trend beyond 12MB shows that we are not only saturating the cache, but also the bandwidth to memory. The pressure generated by the bubble stresses the caches, bandwidth and prefetchers (due to streaming behavior described in Section 6.1.1).

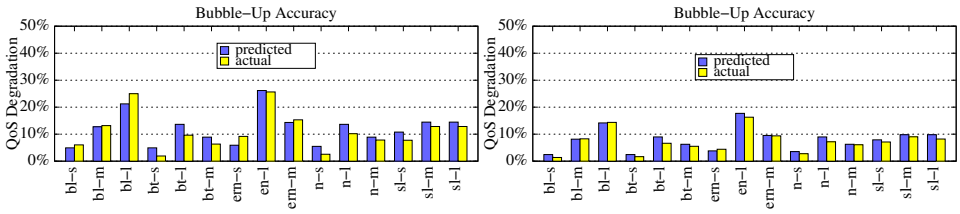
Workload Characteristics Some curves (e.g., `search-render`) are decreasing more sharply than others (e.g., `protobuf`). Also, at a given pressure point, each application suffers a different amount of QoS degradation. For example, at pressure point 10, `search-render`'s normalized QoS is only 0.7, while `protobuf`'s QoS is still around 0.95. When the curve flattens, each application's plateau QoS is also different, ranging from 0.6 to 0.85. This shows that Google applications' QoS have different levels of sensitivity to the pressure in the shared resources. Also, each application's sensitivity to various resources may be different. Since the experimental platform has a 12MB last level cache, the pressure before 12MB is only applied on the shared cache, and after 12MB, the pressure is applied on both shared cache and memory bandwidth. Comparing the gradients of an application's curve before and after 12 MB can help gain insights on its sensitivity to various

resources. For example, `protobuf`'s curve is fairly flat before 12 MB but has a steep dip after 12MB. This indicates that `protobuf`'s QoS may be more sensitive to the pressure on the memory bandwidth than the shared cache.

Bubble Up Prediction Accuracy

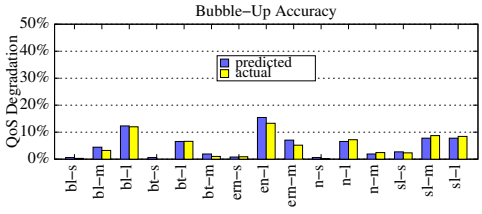
In this section, we evaluate Bubble-Up's accuracy when predicting the QoS degradation of the applications due to performance interference.

Co-locating Google with SmashBench We first evaluate the effectiveness of Bubble-Up in predicting the impact of our SmashBench workloads on Google's applications. In this experiment, we apply step one of our methodology to 9 memory intensive Google applications, and step two to our 15 SmashBench workloads. As previously mentioned, step one needs only to be applied to applications whose QoS needs to be enforce. Step two only needs to be applied to the applications that may threaten an application's QoS. Figure 6.11a to 6.11c present the results for each of the 9 Google applications. For each figure, the x-axis shows each of the 15 SmashBench benchmarks. The y-axis shows the Google application's QoS degradation. For each benchmark on the x-axis, the first bar shows the Google application's predicted degradation when co-located with the benchmark; the second bar is its measured degradation. The closer the two bars are, the more accurate the Bubble-Up prediction is. Each figure's caption also documents the average prediction error for each Google application, calculated using the absolute difference between the prediction and the measured value. In general, Bubble-Up's prediction error is quite small. For the nine Google applications, the prediction error is 2.2% or less. SmashBench exhibits a wide range of memory access patterns, stress points and working set sizes. The fact that a single Bubble-Up design can predict accurately the QoS degrada-

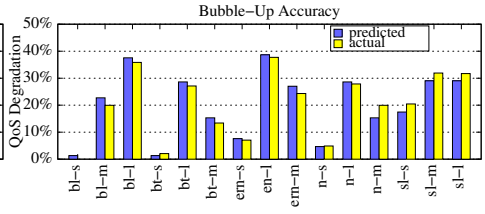


(a) `bigtable`, avg. error - 2.2%

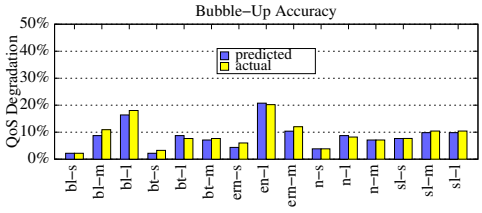
(b) `ads-servlet`, avg. error - 0.8%



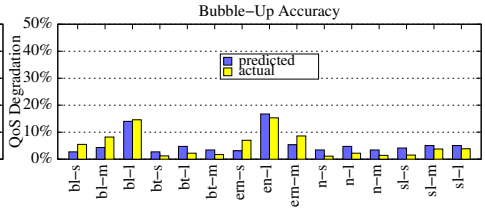
(c) `maps-detect-face`, avg. error - 0.7%



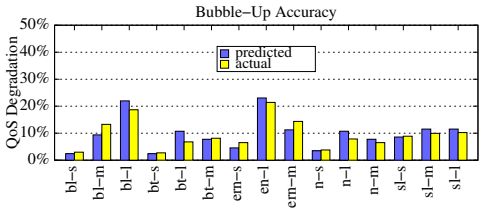
(d) `search-renderer`, avg. error - 1.8%



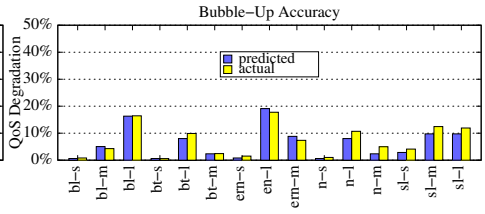
(e) `search-scoring`, avg. error - 0.8%



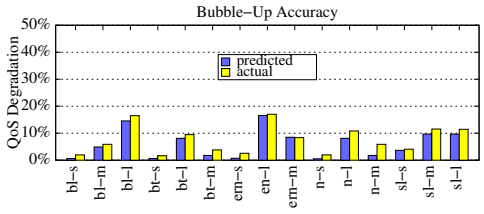
(f) `protobuf`, avg. error - 2.2%



(g) `docs-analyzer`, avg. error - 1.7%



(h) `saw-count2`, avg. error - 1.2%



(i) `youtube-x264yt`, avg. error - 1.5%

Figure 6.11: The Accuracy of Bubble-Up in Predicting QoS Impact for 9 Highly Sensitive Google Workloads.

tion caused by SmashBench demonstrates the generality of the Bubble-Up methodology.

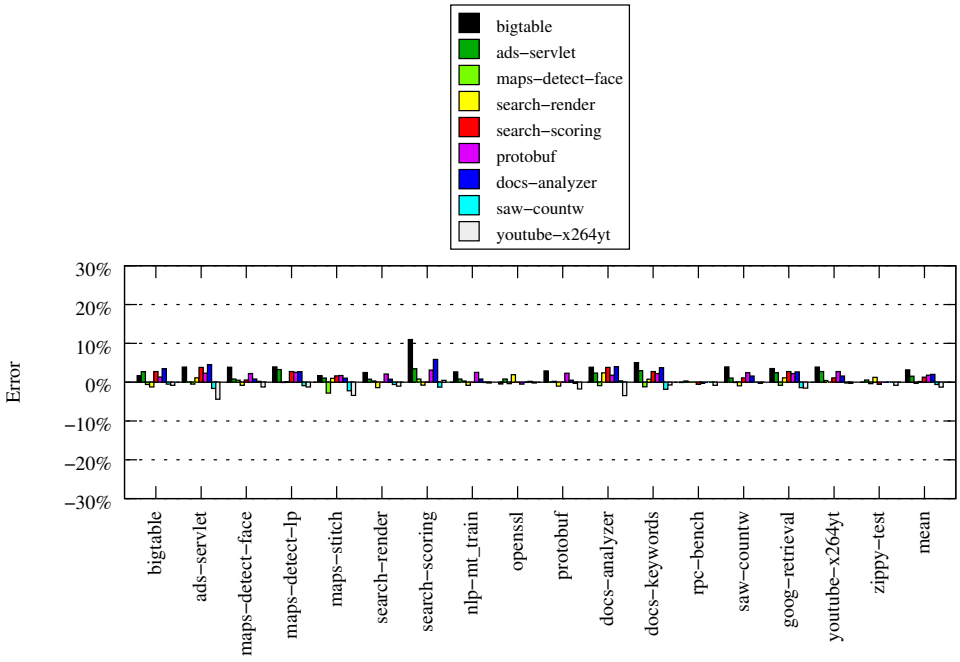


Figure 6.12: Bubble-Up's predication accuracy for pairwise co-locations of Google applications.

Pairwise Google Co-location Figure 6.12 summarizes the prediction accuracy of Bubble-Up for pairwise co-locations with nine of the most sensitive Google applications with the complete set of Google applications (shown in the x-axis). Each bar shows the error (delta) between the performance degradation predicted by Bubble-Up and the actual measured performance degradation in the co-location. Errors in the negative direction imply that the actual QoS degradation is worse (more) than predicted; errors in the positive direction implies that the actual QoS degradation is better (less) than predicted. Only errors in the negative direction can result in a violation of a QoS policy. As the figure shows, Bubble-Up's prediction error is fairly small across all Google pairwise co-locations.

6.2 Improving WSC Utilization with Bubble-Up

In this section, we present an investigation of how Bubble-Up's QoS prediction can be leveraged to steer the cluster manager to increase co-locations in a WSC environment, and ultimately improve utilization.

6.2.1 Applying Bubble-Up in WSCs

To predict the performance degradation on an application, A, when co-located with a corruner, B, we use B's bubble score to index into A's sensitivity curve. To improve machine utilization, we allow latency-sensitive applications to have a small amount of QoS degradation. The tolerable degradation threshold is specified in a QoS policy as described in Chapter 2. Using Bubble-Up, we can predict the QoS degradation and allow co-location of latency-sensitive applications with other applications when the predicted QoS degradation is within the specified threshold. To evaluate the effectiveness of Bubble-Up, we constructed a scenario where we evaluate 1) the machine utilization improvement when using Bubble-Up; and 2) the success of Bubble-Up's prediction in satisfying a QoS policy without violating the specified QoS threshold.

For the production scenario presented in this section, we conducted our evaluation using a cluster that is composed of 500 machines, described in Section 6.1.3. In this experiment, we focused on `search-render` as our main latency-sensitive application whose QoS degradation must be limited within a small amount. In this cluster, there are 500 instances of `search-render`, each placed on a single machine. There are 500 other Google applications, evenly distributed across 15 application types shown in Table 6.1. Every application uses three cores. Our evaluation baseline is the currently deployed cluster management that disallows co-location of `search-render` with any

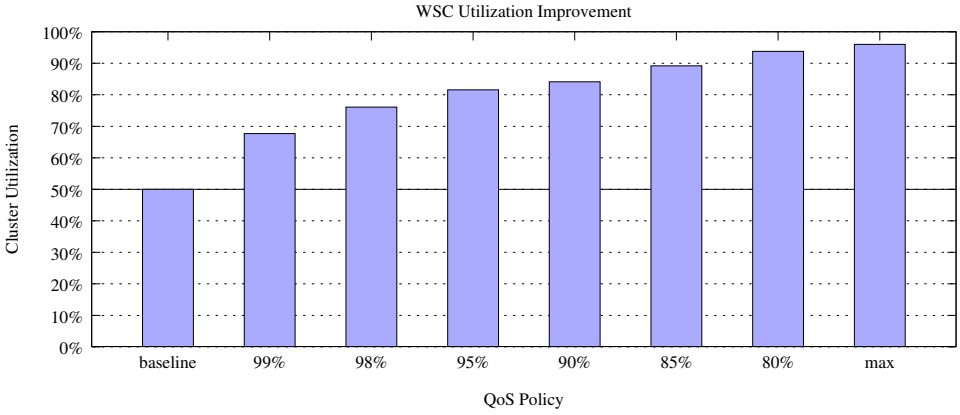


Figure 6.13: Improvement in cluster utilization when allowing Bubble-Up co-locations with `search-render` under each QoS policy.

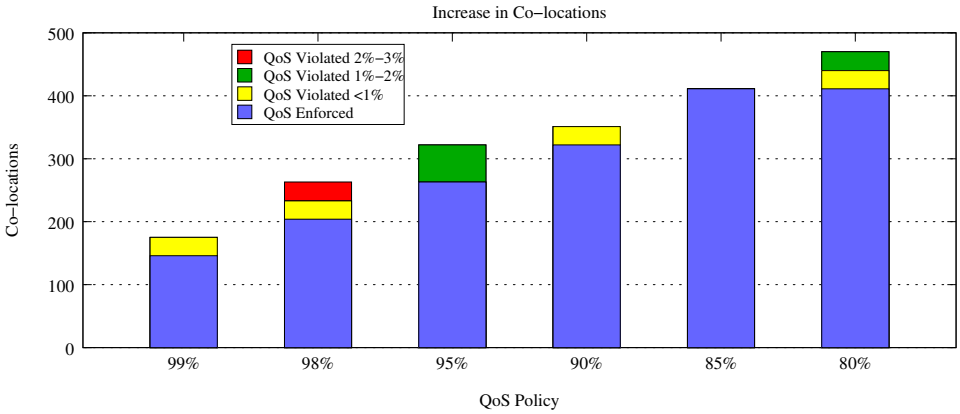


Figure 6.14: Number of Bubble-Up co-locations under each QoS policy.

other applications. In this experiment, we investigated the potential co-location and utilization gained using Bubble-Up predictions under varying QoS policies.

Figure 6.13 presents the cluster's utilization achieved by Bubble-Up prediction under various QoS policies. The baseline is the utilization of the cluster when co-location is disallowed and each instance of `search-render` is occupying three out of the six cores on a single machine, and thus at 50% cluster utilization. The max utilization is achieved by allowing all co-locations; placing each of all 500 other Google applications to co-run with a `search-render` on every machine, regardless of `search-render`'s QoS

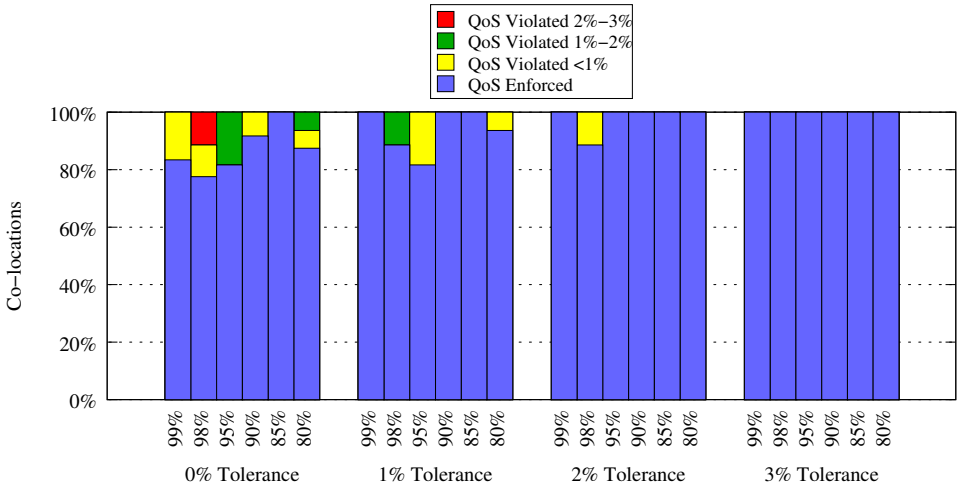


Figure 6.15: Reduction in QoS violations when applying a *tolerance* to each QoS policy. Having a tolerance of just a few percent results in no violations.

degradation. The max utilization is not 100% because we define a machine's utilization as the aggregate performance of all applications running on the machine, normalized by their solo performance. For example, application A and B are co-locating and occupying all six cores on a machine, but due to cross-core interference, their performance is only 90% of that when running alone occupying three cores on a machine. Then the machine utilization when co-located is only 90% instead of 100%.

As Figure 6.13 demonstrates, Bubble-Up prediction greatly improves machine utilization. Even under 99% of QoS policy (when the tolerable QoS degradation is only 1%), the utilization is improved from 50% to close to 70%. Allowing a more relaxed QoS policy improves the utilization even more. Under 80% QoS policy, the utilization improvement is close to 80%, showing great potential benefit of adopting Bubble-Up in WSCs.

Figure 6.14 presents the total number of co-locations allowed by the cluster manager according to Bubble-Up prediction under each QoS policy. Similar to utilization, the number of co-location increases as the allowed QoS degradation increases. The baseline co-location is 0. With 99% QoS

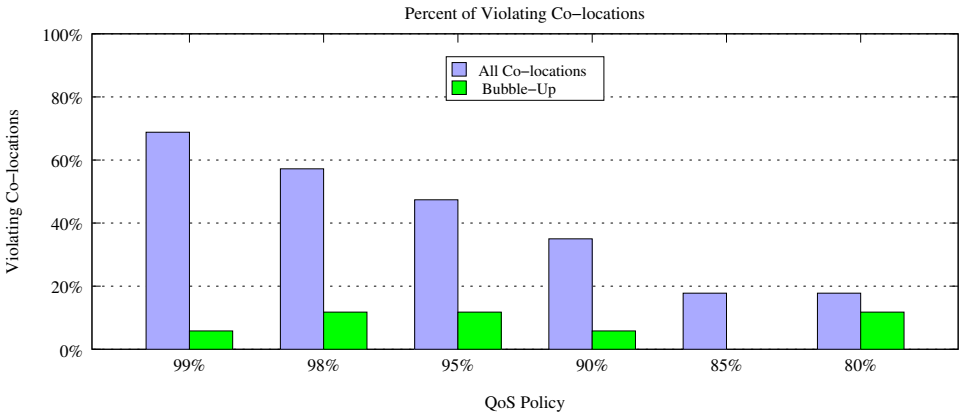


Figure 6.16: QoS violations when allowing all 500 co-locations with `search-render` under each QoS policy. (Random Assignment)

policy, the co-location is close to 200. With 80% QoS policy, the allowed co-locations increase to 400. However, because of Bubble-Up's prediction error, there may be co-locations that violate the QoS threshold specified in the policy. Both the number of co-locations that satisfy the QoS policy and the number of violations are presented in stack bars. The violations are broken down into three categories: violations that cause less than 1% extra degradation beyond the QoS policy, 1-2% extra degradation and 2-3% degradation. For example, as shown in the figure, under 99% QoS policy, around 10% of the co-locations violate the policy. However, all of these violations only cause less than 1% extra QoS degradation beyond the policy, meaning their QoS is within a 98% QoS policy. Figure 6.15 shows the effect of updating the QoS policy to include an error *tolerance*. As shown in the figure, increasing error tolerance at each QoS policy reduces the number of violations. Note that most of the violations cause only less than 2% of extra QoS degradation beyond the QoS policy.

Figure 6.16 shows the percentage of violating co-locations when allowing all co-locations for each QoS policy.

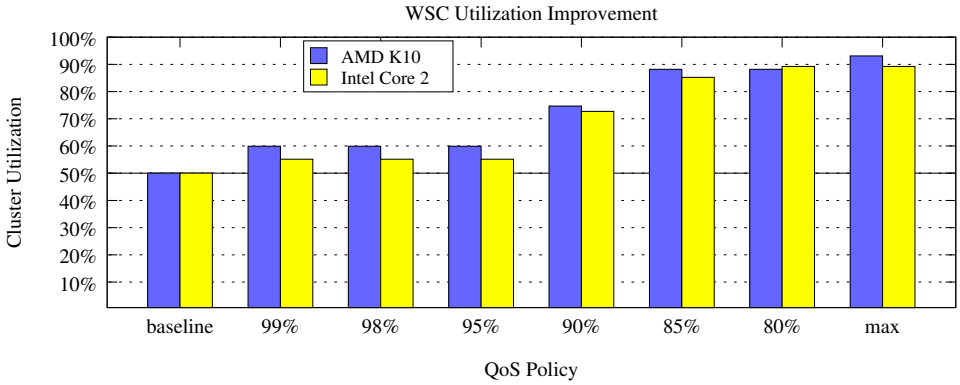


Figure 6.17: Improved utilization in a clusters composed of AMD K10 Opteron servers and Intel Core 2 Xeon servers.

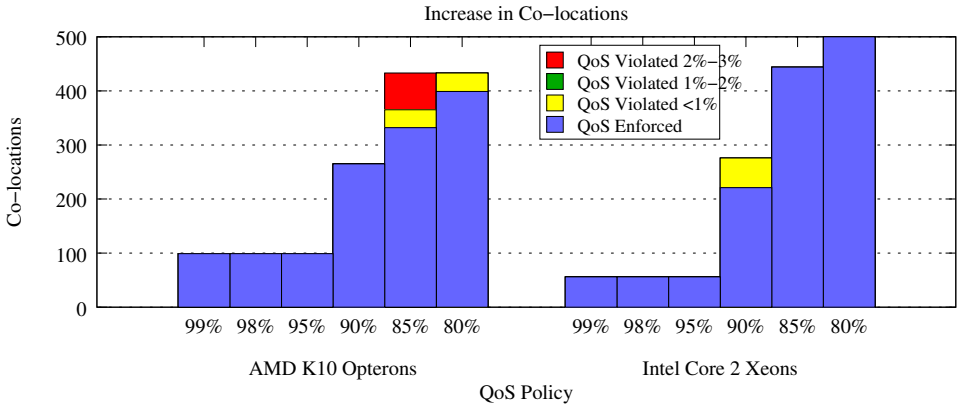


Figure 6.18: Co-locations allowed in Opteron and Xeon clusters.

6.2.2 Impact of Varying Architecture

To evaluate the generality of our Bubble-Up across microarchitectures, we conducted similar experiments on two additional platforms (a six-core K10-based Opteron and a four-core Core2-based Xeon) with the same bubble and reporter used on the Nehalem processor. The experimental setup is similar as Section 6.2.1.

Figure 6.17 demonstrates the utilization improvement for the cluster when using Bubble-Up prediction for a cluster composed of Opteron and a cluster composed of Core2-based Xeons.

As Figure 6.18 shows, Bubble-Up can effectively increase the number of

co-locations with a small amount of error on both platforms. The platforms presented here both have a smaller shared cache sizes and lower bandwidth than the Nehalem processor. This leads to a higher degree of contention on these processors. As a result, we observe fewer co-locations at higher QoS policy thresholds. At the 90% threshold the number of co-locations allowed increases dramatically.

6.3 Directly Quantifying CCI Sensitivity

While prior work has demonstrated the importance of characterizing an application's cross-core interference sensitivity, current techniques use *indirect* methods [19, 37, 63, 125, 132]. An indirect analysis is one that infers an application's cross core interference sensitivity. An example of an indirect analysis is the usage of an application's last level cache missrate to predict its cross-core interference sensitivity [63]. A *direct* analysis on the other hand, is one that characterizes the impact on application performance when contention is actually occurring in comparison to when no contention is present.

In this section, we present the **C**ross-core **i**nterference **P**rofiling **E**nvironment, **CiPE**, the first *direct* methodology and framework for the characterization of an application's sensitivity to cross-core performance interference. This approach is also the first to identify and characterize contentious phases of execution and regions of source code, in contrast to Bubble-Up's sensitivity curves that only characterizes the entire application.

The key insight and observation motivating the design of our cross-core profiling methodology is the fact that contention for shared cache and memory resources is an intrinsically dynamic property of the application's memory behavior, coupled with the intricacies of the particular microar-

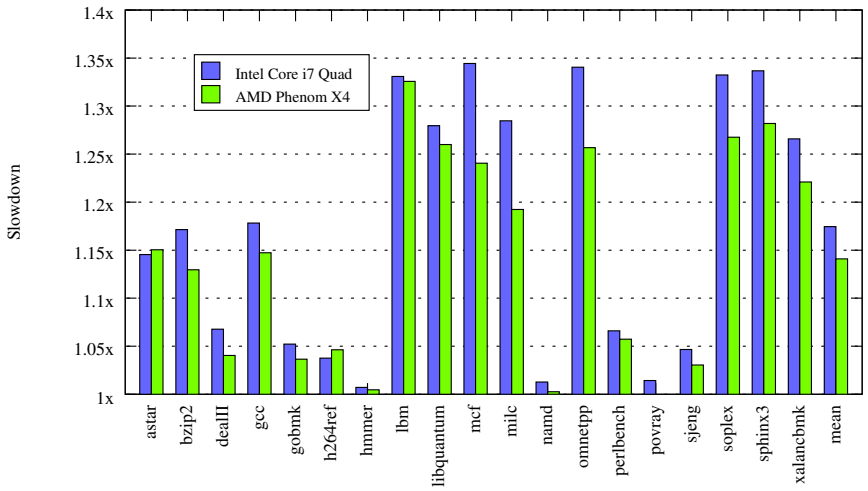


Figure 6.19: Performance impact due to contention from co-location with LBM.

chitectural and memory subsystem design. Therefore, we employ a direct, empirical, online characterization approach.

As an application executes on our CiPE environment, a carefully designed *contention synthesis engine* is spawned on a neighboring core to run alongside the application. This contention synthesis engine is dynamically manipulated by CiPE, and the resulting impact on the host application is analyzed.

6.3.1 Revisiting the Problem of Cross-Core Interference

Two representative examples of the state of the art multicore chip designs are the Intel Core i7 Quad Core chip and AMD's Phenom X4 Quad Core. Intel's Core i7 has four processing cores sharing a large 8mb L3 cache. AMD's Phenom X4 also has four cores and shares a 6mb L3 cache. These chips were designed to accommodate 4 simultaneous streams of execution. However, as we have seen through experimentation, their shared caches and memory subsystem often cannot efficiently accommodate even 2 co-running processes.

Figure 6.19 compares the potential *cross-core interference* that can occur

when multiple co-running applications are executing on the Core i7 with the Phenom X4 architectures described above. Similar to Section 5.3, in this experiment we study the cross-core performance interference suffered by each of the SPEC2006 benchmarks when co-running with `lbm`, one of the SPEC2006 benchmarks known be an especially heavy user of the on-chip memory subsystem. Figure 6.19 shows the slowdown of each benchmark due to the cross-core interference from `lbm`. Each application was executed to completion on their `ref` inputs. On the y-axis we show the execution time of the application while co-running with `lbm` normalized to the execution-time of the application running alone on the system. The first bar in Figure 6.19 presents this data for the Core i7 architecture and the second bar for the Phenom X4. As this graph shows, on both architectures, there are severe performance degradations due to cross-core interference for many of the Spec benchmarks. The large last level on-chip caches of these two architectures do little to accommodate many of these co-running applications. On a number of benchmarks including `lbm`, `mcf`, `omnetpp`, and `sphinx`, this degradation approaches 35%. However, not all applications are effected by the contention properties of their co-runners. Applications such as `hmmer`, `namd`, and `povray` appear to be immune to `lbm`'s cross core interference, demonstrating that cross-core interference sensitivity varies substantially across applications.

It is clear from Figures 6.19 that knowledge of an application's sensitivity to cross-core performance interference is critical to understanding the dynamic interaction and the resulting performance implications of co-running applications on current commodity multicore architecture. In this work, we aim to characterize this sensitivity at three levels: entire applications, their individual phases, and their source level code regions.

6.3.2 Overview of CiPE

CiPE is a profiling analysis approach capable of characterizing an applications *cross-core interference sensitivity*. To perform this characterization, an application is run only once on our CiPE framework. This characterization produces application-level, phase-level, and source-level information that can be subsequently used for a range of purposes such as contention-conscious scheduling, performance analysis and debugging, server consolidation in the WSC, and a host of other uses.

It is important to remember, however, that since the design of the underlying architecture and memory subsystem determine the potential for cross-core interference, each CiPE profile represents the application's cross-core interference sensitivity on the underlying architecture on which the profile was collected. For example a multicore chip with core-private L1 caches big enough to contain the working set of `lbm` could run multiple instances of `lbm` with no cross-core interference. On this architecture `lbm` is not *sensitive* to *cross-core interference*. This is not the case on other chips whose core-private caches cannot accommodate `lbm`, such as the Core i7 or Phenom X4. This insight about the nature of cross-core interference further motivates having a general and portable *direct* approach like CiPE. While the profiles produced by CiPE are representative of a particular underlying architecture and those that are similar, the characterization methodology, and CiPE itself, is portable from chip generation to chip generation.

Profiling Environment

Figure 6.20 provides an overview of **CiPE** running on a multicore architecture with two separate cores sharing an on-chip cache and memory subsystem. The shaded boxes represent our CiPE profiling framework, which is composed of the CiPE runtime and a *contention synthesis engine* (CSE).

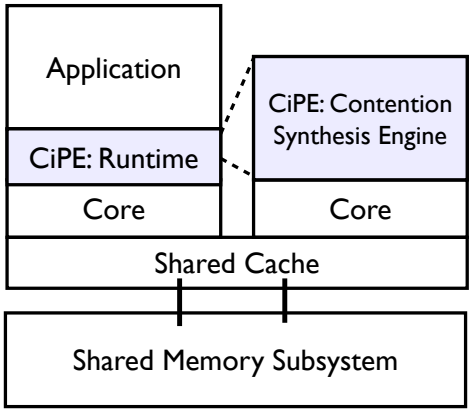


Figure 6.20: The CiPE Framework

As shown on the left side of Figure 6.20, the host application is monitored throughout its execution by the CiPE runtime. Before the execution of the host application, the CiPE runtime spawns the CSE on a neighboring core, as shown to the right of the figure. This CSE shares the cache and memory subsystem of the host application. As the application executes, the CSE aggressively accesses memory causing as much cross-core interference as possible. The CiPE runtime manipulates the execution of the CSE allowing bursts of execution to occur by turning the CSE on and off. Slowdowns in the application's instruction retirement rate that result from this bursty execution are monitored using the *hardware performance monitoring* (HPM) information [53] and are used to characterize its sensitivity. This intermittent control of the CSE and monitoring of the HPM are achieved using a *periodic probing* approach [77]. A timer interrupt is used to periodically execute the monitoring and profiling directives. Periodic probing has been shown to be a very low overhead approach for the dynamic monitoring and analysis of applications.

The core algorithm of our CiPE runtime is presented in Algorithm 6. In this algorithm, *CSE_Status* (line 3) is a flag used to denote whether the CSE engine is executing (active) or sleeping (dormant). *CSE_active_ir* (line 7)

Algorithm 6: CiPE Core Algorithm

Description: This main loop is executed throughout the lifetime of the host application.

```
1  Initialize_CSE();
2  CSE_Off();
3  CSE_Satus ← dormant;
4  while application running do
5      Let_App_Run(probe_time);
6      if CSE_Status equals active then
7          CSE_active_ir ← Read_PMU(instructions_retired);
8          Characterize_CIS(CSE_dormant_ir,CSE_active_ir);
9          Record_Profile();
10         CSE_Satus ← dormant;
11         CSE_Off();
12     end
13     else if CSE_Status equals dormant then
14         CSE_dormant_ir ← Read_PMU(instructions_retired);
15         CSE_Satus ← active;
16         CSE_On();
17     end
18 end
```

and *CSE_dormant_ir* (line 14) records the value of the *instructions_retired* performance counter available in most current microarchitectures. The periodic probing interval is set with *probe_time*. `CSEOn()` and `CSEOff()` (lines 11 and 16) turn the contention synthesis engine on and off (described in Section 6.3.4). `Characterize_CIS()` calculates a cross-core interference sensitivity score (described in Section 6.3.3). Our CiPE Algorithm runs continuously for the duration of the host application's execution.

As the algorithm shows, the performance monitors are read at every *probe_time* interval. A sample of the *instructions_retired* is collected when the CSE is *active*, and another is collected when the CSE is *dormant*. Both samples are passed as input to the cross-core sensitivity characterization routine `Characterize_CIS()`, and finally recorded by `Record_Profile()`.

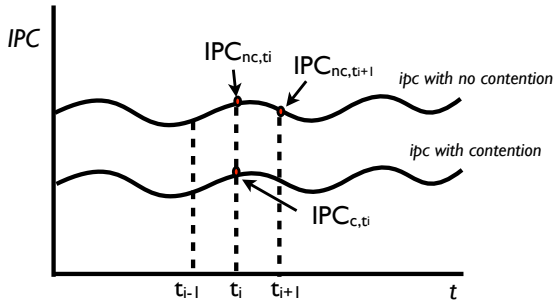


Figure 6.21: IPC Curves

6.3.3 Quantifying Sensitivity

In this section, we present the metrics and measurements used to quantify *cross-core interference sensitivity* (CIS). This analytical model is used for the `Characterize_CIS()` function presented in Algorithm 6.

Defining Sensitivity (CIS)

We define an application's *cross-core interference sensitivity* in terms of performance degradation. Our *direct* metric to characterize an application's cross-core interference sensitivity is the normalized difference between an application's IPC (instruction per cycle) in the presence of contention and its IPC when it is running alone.

CIS can be studied at a fine granularity continuously during an application's execution. The resulting CIS report would reveal dynamic phases of the application's sensitivity characteristics. We use the following formula to define CIS at any time point t_i during an execution:

$$CIS_{t_i} = \frac{IPC_{no\_c,t_i} - IPC_{c,t_i}}{IPC_{no\_c,t_i}} \tag{6.5}$$

where IPC_{no_c,t_i} is the application's IPC with no contention at time t_i, and IPC_{c,t_i} is the application's IPC in the presence of contention at time t_i. The intuition of the formula is shown in Figure 6.21. Here we present an illustra-

tive diagram of an application's two IPC curves of its two executions, with and without contention respectively. At time t_i, CIS is represented by the distance between two points at t_i on two IPC curves, IPC_{no_c,t_i} and IPC_{c,t_i}, then normalized by the IPC with no contention IPC_{no_c,t_i}. Compared to using other metrics such as cache misses to indirectly infer the application's sensitivity, CIS directly measures sensitivity using the percentage of an application's performance (IPC) that is lost due to cross-core interference. Notice that executions with and without contention may vary in time to finish so one IPC curve may need to be normalized for the sake of the calculation.

CIS can also be studied at an application level to characterize the application's general intrinsic sensitivity to cross-core interference. This metric can be viewed as the average distance between two curves, and can be graphically interpreted as the area between the two curves normalized to the area under the curve for IPC with no contention as shown in Figure 6.21 and the formula:

$$CIS_{avg} = \frac{\int_{t_s}^{t_e} IPC_{no\_c} - \int_{t_s}^{t_e} IPC_c}{\int_{t_s}^{t_e} IPC_{no\_c} \times (t_e - t_s)} \quad (6.6)$$

$$= (IPC_{no\_c} - IPC_c)_{avg} \quad (6.7)$$

where t_s and t_e is the start and end point of the time period we are characterizing. This can also be calculated simply as the average CIS throughout the execution, as shown in the formula.

CIS Sampling Methodology

Given the above CIS definition, our CiPE system provides an approach to calculate, monitor and profile CIS through execution. The CiPE system invokes the contention synthesis engine to inject cross-core interference at

regular frequent intervals, which facilitates direct measurement and profiling of contention's impact on an application's performance. To measure the difference between IPCs with and without contention, CiPE system uses a bursty sampling methodology. The CiPE system first turns off the synthesis engine for a sample interval, collects the host application's IPC without contention, then in the immediately-following sample interval, turns on the synthesis engine to collect the application's IPC under the synthesized contention. CiPE uses these two adjacent samples to approximate Formula 6.5 by approximating IPC_{no_c,t_i} using $IPC_{no\_c,t_{i+1}}$, as shown in the following formula:

$$CIS_{t_i} \approx \frac{IPC_{no\_c,t_{i+1}} - IPC_{c,t_i}}{IPC_{no\_c,t_{i+1}}} \tag{6.8}$$

where $t_{i+1} - t_i$ is the sample interval. Notice that the interval from t_{i-1} to t_i is when CiPE has the synthesis engine on to generate contention, and at t_i, IPC_{c,t_i} is collected. From t_i to t_{i+1} the synthesis engine is off and $IPC_{no\_c,t_{i+1}}$ is sampled at t_{i+1}. We call the characterization using this formula, the CIS score.

The above formula is used to generate CIS score at every sample interval along the application's execution. Also, we can define the application's average CIS score using the following formula to calculate the average of all CIS scores during the execution to approximate formula 6.6 where n is the number of samples.

$$CIS_{avg} \approx \frac{\sum_{i=0}^{n} CIS_{t_i}}{n} \tag{6.9}$$

6.3.4 Contention Synthesis

In this section, we discuss the challenge and task of synthesizing contention.

Challenge of Contention Synthesis

The type of data access pattern and the way that data is mapped into the cache is very important to consider when constructing the CSE. Structures such as hardware cache prefetchers and victim caches can avert poor and contentious cache behavior even when the working set of the application is very large. The features and functionality of these hardware techniques are difficult to anticipate as vendors keep these details closely guarded.

With these advances in microarchitectural design, simply accessing a large amount of data does not necessarily cause high pressure on cache and memory performance. For example, access patterns that exhibit a large amount of spatial or temporal locality can easily be prefetched into the earlier and later levels of cache, and prefetch buffers can be used. An important question that arises is, on sophisticated modern architectures, whether an application's sensitivity to contention depends on the manner in which the contention is synthesized.

Designing Contention Synthesis Kernels

To design our contention synthesis engine we explored and experimented with a number of common data access patterns. These designs consist of the random access of elements in a large array, the random traversal of large linked data structures, data movement in 3d object space commonly found in simulations and scientific computing, a real world fluid dynamics application (the `lbm` SPEC2006 benchmark) and finally, we reverse engineered *lbm*, found its most contentious code, and further tweaked it to construct a highly contentious synthesis engine which we call "The Sledgehammer." We present the core of each algorithm and provide full details in Appendix B.

Algorithm 7: Naive Contention Synthesis Kernel

Description: This kernel is executed throughout the lifetime of the host application.

```
1 data ← allocate size_of_LLC() bytes;
2 foreach byte in data do
3 |   byte ← random value;
4 end
5 while not paused and application running do
6 |   foreach byte in data do
7 |   |   byte ← random byte in data;
8 |   end
9 end
```

Naive The core algorithm for our naive contention synthesis kernel is presented in Algorithm 7. This kernel is designed to simply access a large array of memory (matching the size of the L3 cache) performing both loads and stores, while minimizing the computation and instruction level parallelism within the kernel. Our earliest designs traversed an array of memory matching the size of last level of on-chip cache in a number of clever ways that avoided calculating future indices within the kernel. However, the hardware prefetchers on both the Intel and AMD chips were able to cleverly prefetch these indices to early levels of cache. One example of an approach subverted by the hardware prefetchers was the caching of 10,000 random numbers to be used to access the elements of a large array. Although accesses to the data were random, the lookup for the random index is predictable, and modern prefetchers are clever enough to identify the lookup-access pattern. Ultimately, our naive design evolved to simply calculating the random index on the fly as shown in Algorithm 7. While the hardware prefetcher was unable to anticipate these memory accesses, the drawback of this approach, however, is the fact that each memory access is interleaved with the logic to calculate the random number, allowing for a high degree of instruction level parallelism.

Algorithm 8: Linked Data Structure (BST) Trample Function

Input: A tree node *bst_node*

```
1  return_value ← 0;
2  if bst_node ≠ null then
3      if bst.id mod 2 then
4          if bst_node.left ≠ null then
5              return_value ← trample(bst_node.left);
6          end
7          if bst_node.right ≠ null then
8              return_value ← trample(bst_node.right);
9          end
10         return_value ← return_value + bst_node.data[bst_node.id mod data_size]
           ;
11         bst_node.id ← bst_node.id + return_value;
12         bst_node.data[bst_node.id mod data_size] ← bst_node.id mod 256;
13     else
14         if bst_node.right ≠ null then
15             return_value ← trample(bst_node.right);
16         end
17         if bst_node.left ≠ null then
18             return_value ← trample(bst_node.left);
19         end
20         return_value ← return_value − bst_node.data[bst_node.id mod data_size]
           ;
21         bst_node.id ← bst_node.id + return_value;
22         bst_node.data[bst_node.id mod data_size] ← bst_node.id mod 256;
23     end
24     return ret;
25 end
```

Algorithm 9: Linked Data Structure (BST) Contention Synthesis Kernel

Description: This kernel is executed throughout the lifetime of the host application.

```
1  bst ← new binary search tree;
2  while size of bst < size of LLC do
3      bst_node.id ← random id;
4      bst_node.data ← allocate 128 bytes;
5      for i=1 to 128 do
6          bst_node.data ← random value;
7      end
8      insert bst_node into bst;
9  end
10 while not paused and application running do
11     trample(bst.root)
12 end
```

Linked Data Structure The core algorithms for our linked data traversal contention synthesis kernel is presented in Algorithms 8 and 9. This design for this contention synthesis kernel consists of the random construction and traversal of a binary search tree. There were also a number of steps taken to reverse optimize (optimize for high contention) this linked structure contention synthesis approach. For example, the `trample` function is a specialized traversal that recursively picks whether the left or right subtree is to be *trampled* first as shown in Algorithm 8. In the final design of this contention synthesis kernel, each tree node consists of an `id` and `data` payload. The payload consists of a 128 random bytes to have each node map into its own cache line. The contentious kernel of this approach uses the `trample` function to performed a random depth first search through the tree touching and changing the data alone the way by using the randomized `id` to permute both the `id` and the `data` payload of each node.

3D Data Movement The core algorithm for our 3D data movement contention synthesis kernel is presented in Algorithm 10. This 3D data movement micro benchmark consists of a number of large 3D arrays of `double` precision values that represent solid virtual cubes. We use the dimensionality of 30x30x30 for these cubes as this produces a total memory footprint of about 10mb which can fill most modern last level cache sizes. This contention synthesis kernel transposes the cells of each cube into the space of another cube. The cells of one cube is continuously copied to another.

LBM from SPEC2006 The implementation of the LBM benchmark can be found in the official SPEC2006 benchmarks suite [47]. LBM is an implementation of the "Lattice Boltzmann Method" (LBM). The Boltzmann Method is used to simulate incompressible fluids. We selected this benchmark as one of our synthesis mechanisms, as it proved to be one of the most

Algorithm 10: 3D Data Movement Contention Synthesis Kernel

Description: This kernel is executed throughout the lifetime of the host application.

```
1  nugget_size ← 128;
2  dim ← 30;
3  nugget ← bytes[nugget_size];
4  block1 ← nugget[dim][dim][dim];
5  block2 ← nugget[dim][dim][dim];
6  block3 ← nugget[dim][dim][dim];
7  for i=1 to dim do
8  |   for j=1 to dim do
9  |   |   for k=1 to dim do
10 |   |   |   for l=1 to nugget_size do
11 |   |   |   |   block1[i][j][k][l] ← random value;
12 |   |   |   |   block2[i][j][k][l] ← random value;
13 |   |   |   |   block3[i][j][k][l] ← random value;
14 |   |   |   end
15 |   |   end
16 |   end
17 end
18 while not paused and application running do
19 |   for i=1 to dim do
20 |   |   for j=1 to dim do
21 |   |   |   for k=1 to dim do
22 |   |   |   |   for l=1 to nugget_size do
23 |   |   |   |   |   block1[i][j][k][l] ← block2[j][k][i][l];
24 |   |   |   |   end
25 |   |   |   |   for l=1 to nugget_size do
26 |   |   |   |   |   block2[j][k][i][l] ← block3[i][j][k][l];
27 |   |   |   |   end
28 |   |   |   end
29 |   |   end
30 |   end
31 end
```

contentious of the SPEC2006 benchmark suite. For a complete description of LBM please refer to [47].

Algorithm 11: The Sledgehammer Contention Synthesis Kernel

Description: This kernel is executed throughout the lifetime of the host application.

```
1  margin ← 400 × 1000;
2  src_data ← allocate (2 × margin + 26 × 1000000 × size_of double);
3  dest_data ← allocate (2 × margin + 26 × 1000000 × size_of double);
4  while not paused and application running do
5      for i=margin to 26 × 1000000 + margin do
6          dest_data[i] = src_data[i];
7          dest_data[i − 1998] = src_data[(1) + i];
8          dest_data[i + 2001] = src_data[(2) + i];
9          dest_data[i − 16] = src_data[(3) + i];
10         dest_data[i + 23] = src_data[(4) + i];
11         dest_data[i − 199994] = src_data[(5) + i];
12         dest_data[i + 200005] = src_data[(6) + i];
13         dest_data[i − 2010] = src_data[(7) + i];
14         dest_data[i − 1971] = src_data[(8) + i];
15         dest_data[i + 1988] = src_data[(9) + i];
16         dest_data[i + 2027] = src_data[(10) + i];
17         dest_data[i − 201986] = src_data[(11) + i];
18         dest_data[i + 198013] = src_data[(12) + i];
19         dest_data[i − 197988] = src_data[(13) + i];
20         dest_data[i + 202011] = src_data[(14) + i];
21         dest_data[i − 200002] = src_data[(15) + i];
22         dest_data[i + 199997] = src_data[(16) + i];
23         dest_data[i − 199964] = src_data[(17) + i];
24         dest_data[i + 200035] = src_data[(18) + i];
25     end
26 end
```

"The Sledgehammer" The core algorithm for our sledgehammer contention synthesis kernel is presented in Algorithm 11. This design is the result of reverse engineering and investigating `lbm` to learn its contentious core nature. The name of this kernel is motivated by the fact that its behavior can be visualized as touching an element in a 1D or 2D array, and a number of sparsely surrounding elements are also effected. As shown in Algorithm 11, this contention synthesis kernel first allocates two large arrays and enters its contentious kernel which continuously copies data back and forth.

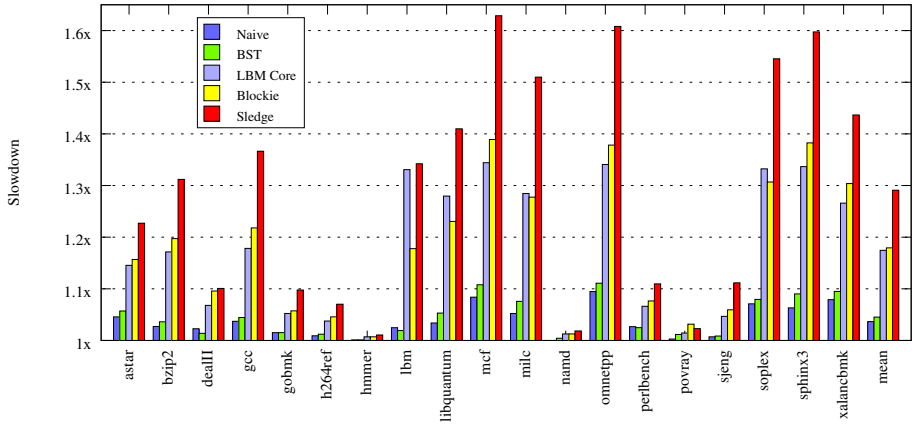


Figure 6.22: Slowdown caused by contention synthesis on Intel Core i7.

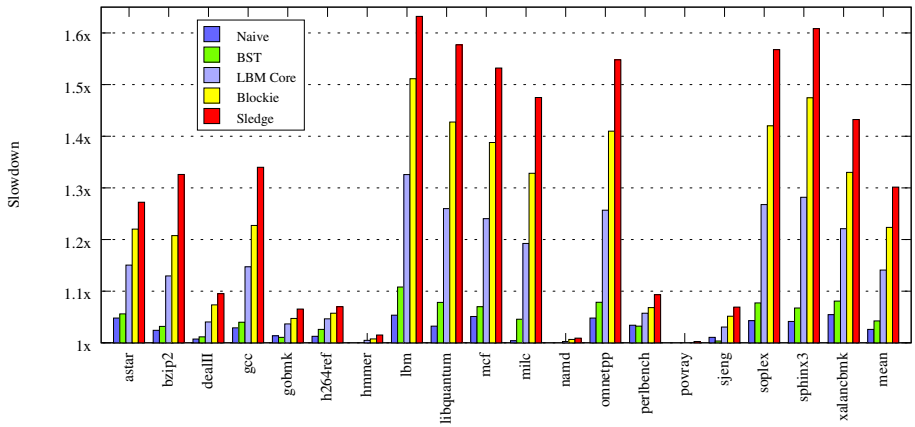


Figure 6.23: Slowdown caused by contention synthesis on AMD Phenom X4.

Kernel Performance Analysis

Goals of Experiment We seek to answer two questions with our evaluation of contention synthesis designs. The first is whether there is a drastic difference between the interactions of different applications to the different contention synthesis designs. We hypothesize that contention is agnostic to the nature of memory access. We seek to evaluate this very question. The other goal of this evaluation is to learn whether there exists a synthesis engine that consistently generates more contention than all others, and if so, identify it.

Experiments with 4 Designs Figures 6.22 and 6.23 show the performance impact of co-running each of the contention synthesis designs with each of the SPEC2006 benchmarks (C/C++ only) run to completion on `ref` inputs. Figure 6.22 shows the results when performing this co-location on Intel's Core i7 Quad architecture, and Figure 6.23 shows these results on AMD's Phenom X4 Quad. The bars show the slowdown when co-located with naive random access (naive), binary search tree (BST), the lbm benchmark (LBM Core), the 3d block data movement (Blockie), and our sledgehammer technique (Sledge), in that order. The `lbm` benchmark is used as a baseline to compare the synthetic engines. It is clear from the graphs that the `Naive` and `BST` approaches produce the smallest amount of contention. However note that they do an adequate job of identifying the applications that are most sensitive to cross-core interference. The contention produced by `Naive` and `BST` is low as there is computation performed between single memory accesses. `Blockie` and `Sledge` touch large amounts of data in a single pass and with less computation. Note that our `Blockie` and `Sledge` techniques are more effective than using the most contentious of the SPEC benchmarks.

Across the two architectures the general trend is similar, although we do see some differences. We see that applications that tend to be sensitive to contention tend to be uniformly so across these two representative architectures. We also see that the varying contention synthesis designs rank similarly on both architectures. This general trend supports our hypothesis that contention is agnostic across this class of commodity multicore architectures.

Although the general trend is the same, there are some clear differences. For example, the benchmark most sensitive to cross-core interference on the two architectures differs. On Intel's architecture `mcf` shows the most signif-

icant degradation in performance, while on AMD's architecture `lbm` has the most significant degradation. These variations are due to the idiosyncrasies of the microarchitectural design.

The key observation is the effectiveness of the contention synthesis designs are mostly uniform across the different benchmark workloads. This trend supports our hypothesis that in addition to being generally agnostic across this class of commodity multicore architectures, it is also agnostic across the varying workloads and memory access patterns present in SPEC.

For our CiPE framework we finalized the design of our main CSE with a implementation based on Sledge as it most vividly illustrates contention.

6.3.5 Applying the CiPE Methodology

In this section we first present the results of our CIS analysis. We then demonstrate the practicality and usefulness of CiPE by using it to address two problems. The first problem is the selection of *contention-conscious* co-schedules for a batch of jobs to dynamically minimize cross-core performance interference and maximize overall throughput and performance. The second problem is to-locate regions of code that, when executed dynamically, are highly sensitive to cross-core performance interference. We address this problem by designing a novel performance analysis and debugging tool using CiPE.

Characterizing Application Phase

Figures 6.24 to 6.29 show phase-level CIS scores calculated using our CiPE system for a representative selection of the SPEC 2006 benchmarks. CIS scores are calculated using samples collected at 1 ms interval along the application's complete execution on `ref` input. For each benchmark, the two lines in the graph indicate the CIS scores on the two different architectures

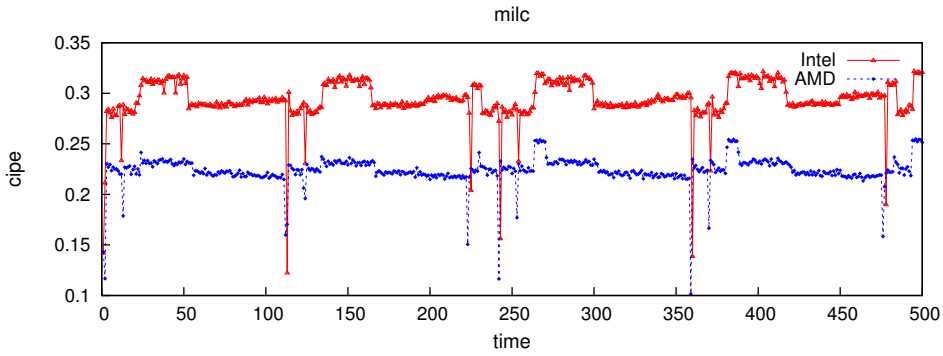


Figure 6.24: Contentious phases of milc

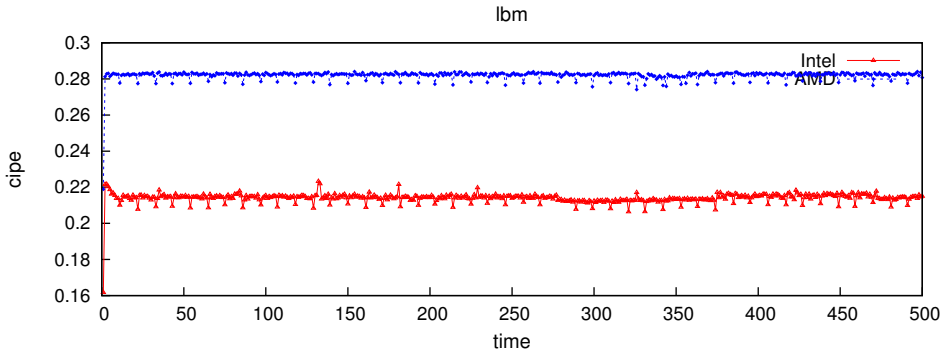


Figure 6.25: Contentious phases of lbm

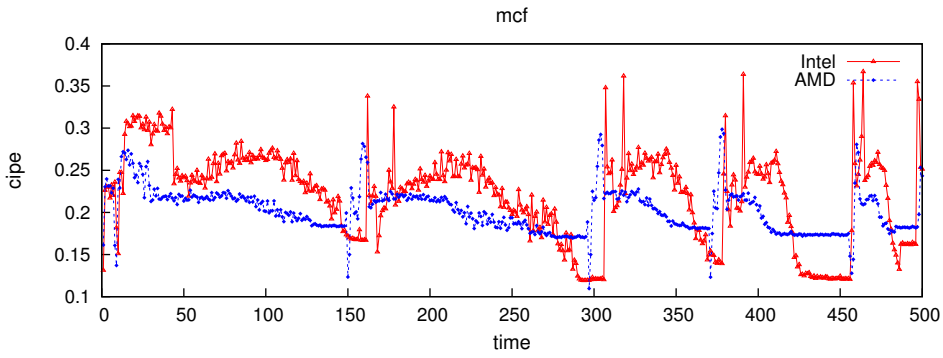


Figure 6.26: Contentious phases of mcf

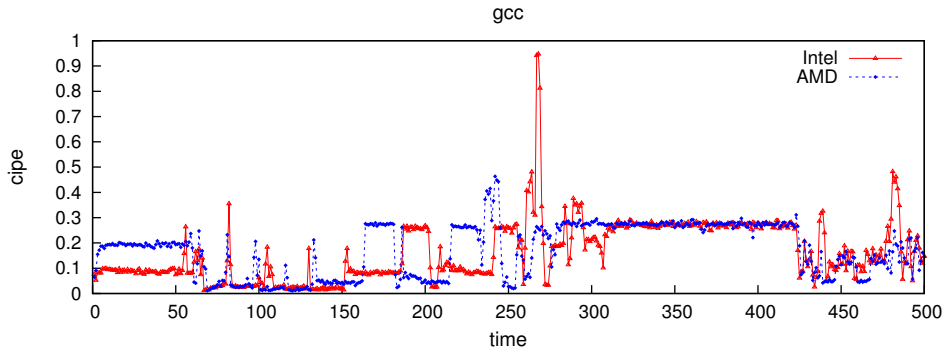


Figure 6.27: Contentious phases of gcc

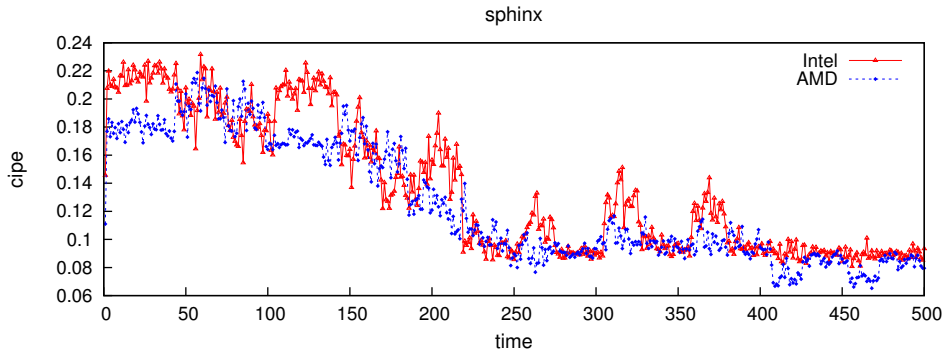


Figure 6.28: Contentious phases of sphinx

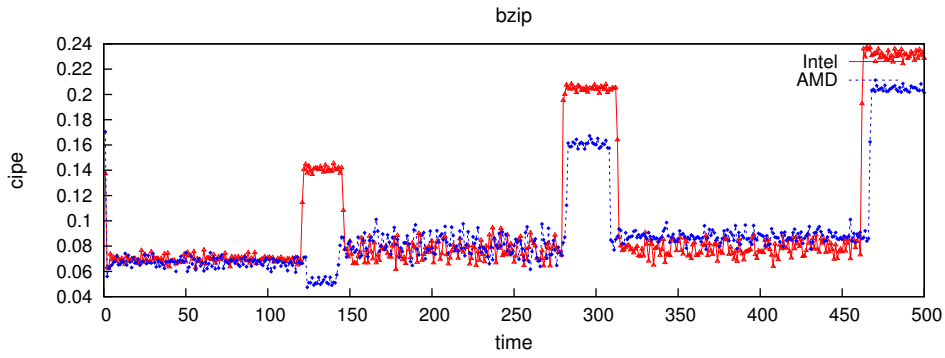


Figure 6.29: Contentious phases of bzip

Intel Core i7 and AMD Phenom X4. Both lines represent a complete execution and is smoothed to 500 data points. We selected representative key benchmarks from the SPEC2006 suite. The higher the CIS score is, the more sensitive the application is to cross-core interference. One important observation is that there are interesting clear phases in both some of the highly sensitive benchmarks (`milc`, `mcf`, `sphinx`) and in relatively insensitive benchmarks (`bzip`). There are also benchmarks that do not exhibit clear phases including both sensitive benchmarks (`lbm`) and insensitive benchmarks.

Profiling and discovering phase level characteristics of sensitivity is valuable for dynamic co-scheduling, whether the scheduling is done through a runtime system or OS. For example, as shown in Figure 6.28, `sphinx` is highly sensitive during the first half of the execution but later its sensitivity drops. Thus an intelligent dynamic scheduler equipped with phase information can foresee peaks of contention sensitivity and schedule the application wisely according to the phase. In addition, as shown later in this section, we have designed a performance analysis and debugging tool that associates these phases to source code regions. Using this, users can identify code regions that are highly sensitive and optimize accordingly.

Characterizing Whole Execution

Figures 6.30 and 6.31 show the average CIS scores calculated using Formula 6.9 for all C/C++ benchmarks in SPEC2006, compared against the performance degradation when each benchmark is co-running with `lbm`, on both Intel Core i7 and AMD Phenom X4. We also compare our CIS approach with average last level cache (LLC) miss rates, as this approach is currently believed to be one of the best known indicators of contention sensitivity [63, 132]. In Figures 6.30 and 6.31 we present the cache miss rate

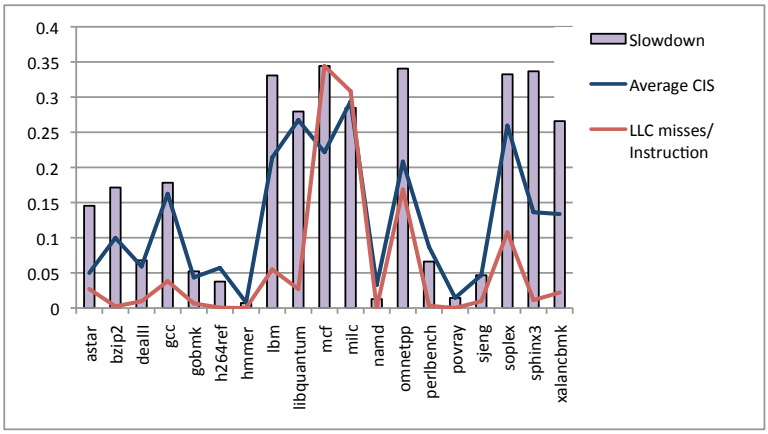


Figure 6.30: CIS score and last level cache misses compared to slowdown when contending with LBM (Intel Core i7)

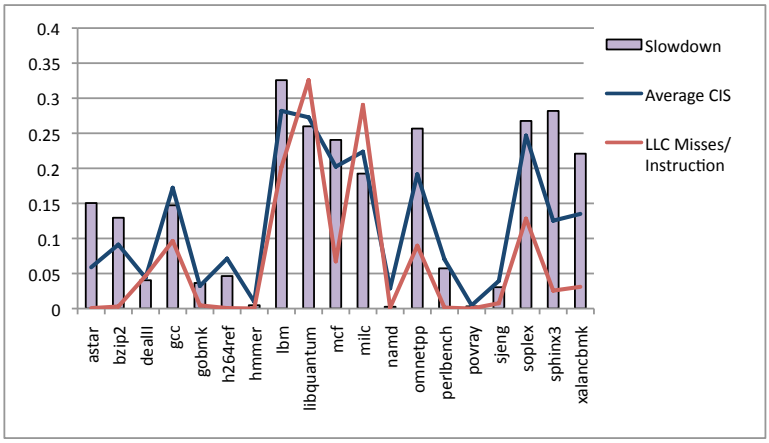


Figure 6.31: CIS score and last level cache misses compared to slowdown when contending with LBM (AMD Phenom X4)

using a line graph for each benchmark.

Our results show that generally, an application's average CIS score has a strong correlation with its performance degradation (e.g., a lower CIS scores indicate smaller degradations and vice versa). Although in a few cases our CIS scores is less representative of the actual degradation (`sphinx`, `xalan` and `astar`), we see that in general our CIS scores match the actual performance degradation much more closely than cache miss rates. These three benchmarks have more sporadic phases that we believe increased the inaccuracy on its average. However, notice that even in these cases, the CIS score significantly outperforms using last level cache miss rates. In addition, studying the phase level CIS scores for these types of applications would give more insight about their dynamic sensitivity.

Identifying Code Regions

We also applied our CiPE framework to identify contentious code regions. This source level information is also critical for emerging compiler technology that requires the identification of contentious code regions [116]. Software developers and system analyzers can also use this information for performance analysis, performance debugging, and to detect the most contention sensitive application phases and identify regions of source code responsible for this contention. We have developed a performance debugger using CiPE that points to contentious regions of code.

Our performance debugger functions as a post processor of the profiles generated by our CiPE profiling environment. For each CIS sample generated by the CiPE profiling framework while executing an application, a record of the number of instructions executed since the previous sample is recorded. This record represents a region of executed instructions. This dynamic instruction trace can then easily be linked back to source level code

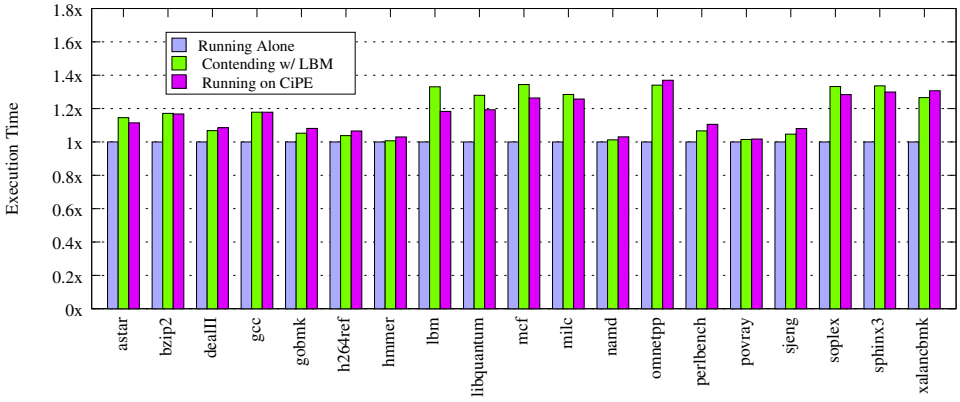


Figure 6.32: CiPE Overhead

using the standard `elf` debugging information.

More details on identifying code regions with CiPE and a case study showing our performance debugger in action can be found in Appendix D.

Profiling Overhead

Although CiPE profiles only need to be collected once, one nice attribute of our CiPE profiling approach is that it is highly efficient. Figure 6.32 shows the overhead of collecting CiPE profiles. CiPE requires only one pass of the application's execution. In this figure, for each benchmark, the execution time when it is running on top of our profiling system (the second bar) is normalized against the execution time when it is contending with `lbm` (the first bar). For the sake of comparison we also include the execution time when the application is running alone (the third bar), also normalized to the contending cases. For half of the benchmarks our profiling system's overhead is even smaller than the degradation caused by contention generated by `lbm`. For remaining benchmarks the overhead is less than 5% more than the degradation. Notice that the synthesis engine generally causes more degradation than our baseline contending case `lbm`. Therefore the overhead introduced by the profiling technique alone is fairly low.

Solving Real World Problems

We demonstrate the application of our CiPE methodology to two real world problems in Appendicies C and D.

Chapter 7

Conclusion and Future Directions

Contents

As the landscape of computing evolves, and much of our computation moves into the cloud, developing highly efficient WSC architectures becomes vitally important. This dissertation argues that a WSC design that is aware of, and exploits, the diversity of execution environments is critical in arriving at a highly efficient design. In this dissertation, we have shown how this diversity manifests itself statically, through the diverse machine configurations and microarchitectural designs housed in a WSC, and dynamically, through the various co-running tasks that a machine my host at any given time. We have illustrated three design problems that stem from remaining oblivious to the diversity in execution environments: *the homogeneous assumption* where all machines and cores in a WSC are assumed equal when they are in fact quite diverse, *the rigidness of applications* where application binaries can not adapt to changes across and within execution environments, and *the oblivion of interference* where interference between tasks within an execution environment can not be measured or managed.

This dissertation addresses each of these three issues to incorporate an awareness of execution environment diversity, and demonstrate a significant opportunity for improving efficiency. At the cluster level, we address *the homogeneous assumption* by enabling the WSC to continuously learn the execution environments tasks prefer, and map jobs accordingly. At the machine level, we address *the rigidness of applications* by providing a novel mechanism to allow applications to adapt to their execution environment, and leverage this mechanism to solve pressing problems in WSCs. At the cluster and machine levels, we address *the oblivion of interference* by providing novel metrics and techniques for measuring and managing interference to improve the utilization of WSCs.

With the work presented in this dissertation, we take a major leap forward in understanding how to build a highly efficient WSC, which has implications on reducing not only the cost of these systems, but also their environmental footprint.

7.1 Summary of Themes and Results

[EE Diversity is Rampant and Impactful] Through our comprehensive study of the performance impact of diversity in execution environments, we find that diverse execution environments are rampant, and significantly impacts the performance of tasks within a WSC.

- When studying the performance impact of the diversity in machine configurations across machines from Google's production fleet, we find a variability between 5% and 3.5x on Google's large-scale commercial WSC workloads.

- When performing the same study on the diversity in co-running tasks we find a variability in performance degradations of 0% to over 30%.

- We find these results also apply to benchmark workloads in a test suite.

- Using the novel metric, *opportunity factor*, we are able to quantify the sensitivity of an application to diversity in machine configuration and application co-runners.

[**Intelligent Job Mapping is Critical for Efficiency**] Using an intelligent mapping approach, tasks in the WSC are placed in execution environments where they run best, exploiting the significant performance potential that remains unrealized in current WSC systems.

- By extending current WSC systems to exploit the EE diversity using SmartyMap, we improve the overall performance of a production cluster environment by 12% to 16%.

- Web-service applications that are particularly sensitive to EE diversity, such as `docs-analyzer`, can be improved improved by up to 82%.

- Varying either the machine mix or the application mix has a significant impact on the performance opportunity given by EE diversity.

- When using an intelligent mapping approach such as SmartyMap, a cheaper WSC composed of an even mix of newer (faster) and older (slower) machines can achieve more than 95% of the performance as a more expensive WSC composed only of the newer machines.

[**Application Adaptation is Critical for Efficiency**] By providing a novel mechanism that allows tasks to dynamically adapt to the execution environment in which they run, two important problems facing WSCs can be addressed.

- Through the design of the *Lightweight Online Adaptation Framework* (Loaf), native binaries can employ adaptation approaches that allows

the application to adapt to its environment using *scenario based multiversioning*, and allow the co-runners in the environment to adapt to an application using *cross core application cooperation*.

- This mechanism for adaptation is extremely low overhead, incurring a performance impact of less than 1%.

- Using this framework, we provide an approach that enables applications to self-select compiler optimizations based on the execution environment in which it runs improving the performance of applications by 4% to 15%.

- Using this framework to design the *Contention Aware Execution Runtime* (CAER) environment, we provide the capability of instantaneous contention detection and response on real commodity machines. While reducing the performance degradation from 17% to 4% on average, we are able to increase the utilization of the neighboring core by 60% on average.

[**Mitigating Interferece is Critical for Efficiency**] Through providing novel capabilities in measuring and managing interference between tasks within the execution environment at both the cluster and machine levels, we are able to significantly increase the amount of co-locations in the WSC and improve utilization.

- Using the general characterization methodology *Bubble-Up*, we enable the *precise prediction* of the performance degradation that results from contention for shared resources in the memory subsystem with an error of often less than 1% and never exceeding 2.2%.

- In a cluster scenario composed of 17 production Google workloads on production machines we demonstrate that using Bubble-Up to steer

co-location decisions can significantly improve the utilization of WSCs from 50% to over 80% when using a 95% QoS policy.

- By slightly modifying how service level agreements are defined we can eliminate the already slight violations that is produced by prediction errors to 0.

- Using our general *direct* measurement technique for quantifying cross-core interference sensitivity, *CiPE*, we are able to not only characterize the sensitivity of entire applications, but also their phases of execution, and source code regions.

7.2 Future Directions

The architecture of modern WSCs remains in its infancy. Beyond the challenges addressed in this dissertation, there are a number of future directions for innovation with the design of WSCs to improve efficiency and reduce their environmental footprint. This section outlines just a few of these directions.

Architecting a Heterogeneous WSC

Currently, WSCs have been designed and constructed to be as homogenous as possible, in that the aim is to have all machine configurations be identical. The advantages of this homogeneous approach is that it simplifies the software stack and how operators interface the cloud. However, this may not be the right way to construct WSCs. Within the community there is currently a debate as to whether it is best to use many "wimpy" cores, or a few "brawny" cores in the machines that comprise large-scale datacenters. Both camps may be correct. Across the diverse set of workloads residing in the cloud, some work best on wimpy cores, while others depend heavily

on brawny cores for performance. Instead of choosing one design, why not break the homogenous assumption and use as many of each type of configuration as needed by the diverse set of workloads residing in the target cloud environment. This goes far beyond just "wimpy" and "brawny" cores, to an arbitrary set of configurations of machines specialized for various classes of workload types.

The Accelerated WSC

Beyond embracing a heterogeneous cloud, it would be highly advantageous to leverage acceleration units in WSCs. Web-service companies can design a SoC with specialized acceleration components for common algorithms and functions executed continuously in the cloud. While this type of system level integration has been prevalent in the embedded space, current WSCs and other general purpose domains do not leverage this potential. WSCs are particularly well suited for system level integration especially as it relates to acceleration units as the same set of workloads are executed for days and months at a massive scale. Accelerating the common processing tasks at the scale of millions of machines that are always running can result in massive economic and environmental savings.

A Language to Express Adaptation Policies

There is currently a lack of a general interface to specify behaviors and execution policies concerning the performance and efficiency of how jobs are executed in the cloud. For this research direction, the goal is to design a language to describe how the software platform in the cloud should react to various events and situations, and redesign the software stack to support this language. A natural language example of a policy for which we would like to express in this language include: *"when we detect a nasty co-runner*

C on machine M increases the latency of an important job J, also on machine M, let's throttle down, migrate, or kill C depending on the category and importance of job C." At the scale of 10s of webservices composed of hundreds of various job types, the challenge of enabling these policies by hand across the entire WSC proves prohibitive. In this direction, the goal is to provide a unified interface with a simple language to allow a confluence of policies to be expressed and enacted.

Appendix A

Technical Details of Loaf Framework

Contents

We have implemented our compile-time SBO infrastructure as a new pass in the GCC 4.3.1 compiler. The passes within GCC can be broken into four parts. First, there are the parsing passes where the text of the source code are processed. Second, we have the gimplification passes where GCC generates its Gimple intermediate representation on which optimizations can occur. Third, we have tree-SSA passes that optimize high level Gimple IR. Finally, we have the RTL passes where low level optimizations and code generation occurs.

Our new SBO pass has been placed right after GCC's earliest IR is generated as the first inter-procedural pass. This allows for maximum flexibility for compiler writers to design how the SBO function versions can be configured. For example, a function can be annotated to disable or enable any of the optimizations in later passes.

To specify which functions are to be multiversioned we have added a new command-line option to GCC, `-fmultiver_funcs=`. For example, if

the functions `foo` and `bar` are to be multiversioned, invoking gcc with the command `gcc -fmultiver_funcs=foo,bar test.c` accomplishes this.

Internally GCC provides a function and call graph cloning routine that is used or inter-procedural constant propagation. SBO uses this routine to clone the internal function data structures as many times as needed. We take the original function and rewrite its internals. This function now becomes a trampoline that the SBO dynamic component can manipulate via shared memory hooks. The way this new trampoline functions depends on whether we are using the alternate scheme or the n-version scheme.

For the alternate scheme we simply inject basic blocks into the function's head using GCC's internal basic block writing APIs. The logic of the injected Gimple basic blocks first checks a global, if it is set the calling parameters are then passed on to the alternate version and it is called using a direct call. Any values returned from the alternate version are then passed on to the original call site. If the global is not set we execute the default function code. The dynamic component controls this trampoline via this global.

For the n-version scheme we always trampoline out of the original function similarly to the case where the global is set in the alternate version. The primary difference is, with the n-version scheme, the function call is an indirect. The dynamic component controls this trampoline by writing the address of the target function in the address location the indirect call uses. The global variable and tables that are required to provide the interface to dynamic component are all injected into the binary through this SBO pass.

Finally, this pass injects one basic block into the head of the main function of the application. This basic block is composed of a single call to `init_sbo`. This call initializes and launches the dynamic component. The dynamic component is implemented as a library and contains the body to the `init_sbo` call. Any application compiled with SBO enabled must be

linked with SBO's dynamic component.

The dynamic component is responsible for monitoring the execution context of the application and detect when a scenario may have begun. If this occurs the dynamic component is responsible for re-routing execution to only include the code best suited for the detected scenario.

A.1 Performance Monitoring

We take advantage of performance monitoring hardware to continually identify the current execution context of our host application. By using performance monitoring hardware we are able to collect this information about the execution environment while incurring negligible overhead. There are a number of APIs available for taking advantage of performance monitoring hardware including OProfile, PAPI, and Perfmon among others.

We have chosen to use Perfmon2 [36] for the design and implementation of our dynamic introspection engine. The goal of the Perfmon2 project is to design and implement a general, standard Linux interface to architectural performance monitoring hardware. In addition to the kernel work, Stephane Eranian and the other Perfmon2 developers have also implemented user-level libraries and tools to facilitate development with Perfmon2. Perfmon2 supports most major architectures including core/core2, amd64, itanium, and powerpc. For these reasons we selected to build our dynamic infrastructure on Perfmon2.

A.2 Periodic Probing

One thing to keep in mind is the dynamic component is modular and flexible. SBO statically generates binaries with specialized versions of hot functions and provides hooks for the dynamic component. The dynamic component

```
//init_sbo is called at application startup
void init_sbo()
{
  //initialize performance counters
  set_up_performance_counters();

  //lauch the counters
  start_counting();

  //start the timer interrupt
  lauch_timer_interrupt();
}

//when the interrupt is thrown we handle it here
void interrupt_handler()
{
  //stop the counters, collect the information
  stop_counters();
  read_counters();

  //do the analysis required by
  //the scenario detection heuristic
  do_analysis();

  //switch the active versions of functions in
  //our application to match the detected scenario
  reroute_execution();

  //start the counters again after reseting them
  start_counters();

  //launch the timer
  lauch_timer_interrupt();
}
```

Figure A.1: This is pseudo code for the general dynamic introspection component of SBO.

can then use any heuristic to reroute execution via control through these hooks. How the dynamic component monitors execution is entirely up to the optimization designer and can vary in any way.

That being said our SBO infrastructure has a default design for the dynamic component. It is shown in Figure A.1. To detect whether a scenario is occurring, SBO's dynamic component uses a timer interrupt approach. The dynamic component includes an `init_sbo` routine that is called once when the host application begins. When the `init_sbo` routine is called, performance counters are setup and the timer interrupt is started. When the timer interrupt has triggered, the interrupt handler executes. As shown in Figure A.1 the counters are then stopped and read. Next, the scenario detection code executes. If a target scenario is detected, the dynamic engine will reconfigure the executing binary to execute the function versions tuned to that scenario. The counters are then reset and the timer launched again.

```
void catch_alarm(int sig_num)
{
  int i;
  //stop and read the counters
  pfm_stop(ctx_fd);
  pfm_read_pmds(ctx_fd, pd, inp.pfp_event_count);

  //execute the code for current phase
  //and move to the next phase
  if(phase==0) {
    phase=1;
    ver1_stat=ver2_stat=0;
    __mv_version_switch=0;
  }
  else if(phase==1) {
    phase=2;
    ver1_stat=pd[0].reg_value;
    __mv_version_switch=1;
  }
  else if(phase==2) {
     phase=0;
    ver2_stat=pd[0].reg_value;
    if(ver1_stat>ver2_stat)
      __mv_version_switch=0;
  }

  //clear and restart counters
  for (i=0; i < inp.pfp_event_count; i++) {
    pd[i].reg_value=0;
  }

  pfm_write_pmds(ctx_fd, pd, inp.pfp_event_count);
  pfm_start(ctx_fd, NULL);

  //launch the timer for next signal
  if(phase==0) alarm(10);
  else alarm(1);
  //renter executing application
}
```

Figure A.2: This is the core three phase code to the dynamic component of the SBO algorithm.

This interrupt driven periodic probing execution pattern executes continually as the application is running. The overhead of such a technique is determined by the frequency of the probing. The amount of runtime overhead incurred by our probing technique depends on two factors: the frequency of interrupts, and the complexity of the analysis due to those interrupts. These two factors are determined by the nature of the optimization hosted by our SBO framework. Using our default design for the Dynamic Introspection Engine, this overhead is negligible. For example the overhead of the optimization presented in the next section causes a slowdown of less than 0.5%.

Figure A.2 shows the pseudo code of our design.

Appendix B

Contention Synthesis Kernel Implementations

Contents

Here we present the C implementations of our four contention synthesis kernels

B.1 Naive.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

char *data;

main(int argc, char *argv[]) {
  srand(time(0)+getpid());
  if(argc<2) exit(1);
  int bytes=atoi(argv[1])*1024;
  data=(char*)malloc(bytes);
  for(int i=0; i<bytes; i++) data[i]=rand()%256;
```

```
  while(1) {
    for(int j=0; j<bytes-2; j++) {
      data[rand()%bytes]+=data[rand()%bytes];
    }
  }
  printf("%d\n",(int)data[rand()%bytes]);
}
```

B.2 BST.c

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

const int payload_size=128;

void gen_name(char *ret) {
  for(int i=0; i<payload_size; i++) {
    ret[i]=(char)rand()%256;
  }
}

struct tree_node {
  ~tree_node(){
  if(left) delete left; if(right) delete right;
  }
  tree_node* left;
  tree_node* right;
  int data;
  char text[payload_size];
};

class BST {
  private:
    tree_node* root;
  public:
    BST() {
      root = NULL;
    }
    bool isEmpty() const { return root==NULL; }
    void insert(int);
    void remove(int);
    void clear(){if(root){delete root;} root=NULL;}

    unsigned long trample();
    unsigned long trample(tree_node *p);
};

...[standard implementation of insert and remove]
```

```
unsigned long BST::trample(){return trample(root);}
unsigned long BST::trample(tree_node *p) {
  unsigned long ret=0;
  if(p != NULL) {
    //Using random traversal + 5%
    if(p->data%2) { //Using p->data instead of rand + 2%
      if(p->left) ret+=trample(p->left);
      if(p->right) ret+=trample(p->right);
      ret+=(unsigned long)p->text[p->data%payload_size];
      p->data+=ret; //Moding data + 6%
      p->text[p->data%payload_size]=p->data%256;
    }
    else {
      if(p->right) ret+=trample(p->right);
      if(p->left) ret+=trample(p->left);
      ret-=(unsigned long)p->text[p->data%payload_size];
      p->data+=ret; //Moding data + 6%
      p->text[p->data%payload_size]=p->data%256;
    }
  }
  return ret;
}

int main(int argc, char *argv[]) {
  int footprint=8192;

  BST b;
  srand(time(0)+getpid());

  unsigned int node_size=sizeof(tree_node)+sizeof(BST);

  for(int i=0; i<footprint*1024/node_size; i++) {
    b.insert(payload_size+(rand()-payload_size));
  }

  unsigned long long sum=0;
  while(1)
     sum+=b.trample()+b.trample();
}
```

B.3 Blockie.c

```
#include <iostream>
#include <cstdlib>

using namespace std;

const int nug_size=128;

class nugget {
```

```
  public:
    char n[nug_size];
    nugget(){
      for(int i=0; i<nug_size; i++){n[i]=rand()%256;}
    }
};

class block {
  public:
    nugget ***b;
    unsigned size;
    block(unsigned sz);
    ~block();
};

block::block(unsigned sz) {
  b=new nugget**[sz];
  for(int i=0; i<sz; i++) {
    b[i]=new nugget *[sz];
    for(int j=0; j<sz; j++)
      b[i][j]=new nugget[sz];
  }
  size=sz;
}

block::~block() {
  for(int i=0; i<size; i++) {
    for(int j=0; j<size; j++)
      delete [] b[i][j];
    delete [] b[i];
  }
  delete [] b;
}

int main() {
  const int size=30;
  block b1(size);
  block b2(size);
  block b3(size);

  cout << "smash" << endl;
  while(1)
    for(int i=0; i<size; i++)
      for(int j=0; j<size; j++)
        for(int k=0; k<size; k++)
          b1.b[i][j][k]=b2.b[j][k][i]=b3.b[i][j][k];
  return 0;
}
```

B.4 Sledge.c

```
#include <stdlib.h>

typedef double LBM_Grid[26000000];

static double *srcGrid,*dstGrid;
int main()
{
  const unsigned long margin = 400000,
    size = sizeof( LBM_Grid ) + 2*margin*sizeof( double );
  srcGrid = malloc( size );
  dstGrid = malloc( size );
  srcGrid += margin;
  dstGrid += margin;

  while(1)
  {
    int i;
    for( i = 0; i < 26000000; i += 20 ) {
      dstGrid[i] = srcGrid[i];
      dstGrid[i-1998] = srcGrid[(1)+i];
      dstGrid[i+2001] = srcGrid[(2)+i];
      dstGrid[i-16] = srcGrid[(3)+i];
      dstGrid[i+23] = srcGrid[(4)+i]
      dstGrid[i-199994] = srcGrid[(5)+i];
      dstGrid[i+200005] = srcGrid[(6)+i];
      dstGrid[i-2010] = srcGrid[(7)+i];
      dstGrid[i-1971] = srcGrid[(8)+i];
      dstGrid[i+1988] = srcGrid[(9)+i];
      dstGrid[i+2027] = srcGrid[(10)+i];
      dstGrid[i-201986] = srcGrid[(11)+i];
      dstGrid[i+198013] = srcGrid[(12)+i];
      dstGrid[i-197988] = srcGrid[(13)+i];
      dstGrid[i+202011] = srcGrid[(14)+i];
      dstGrid[i-200002] = srcGrid[(15)+i];
      dstGrid[i+199997] = srcGrid[(16)+i];
      dstGrid[i-199964] = srcGrid[(17)+i];
      dstGrid[i+200035] = srcGrid[(18)+i];
    }
  }
  return 0;
}
```

Appendix C

Contention Conscious Scheduling with CiPE

When two applications are co-scheduled on current commodity multicore architectures in a *contention oblivious* fashion, cross-core performance interference occurs, and ultimately system utilization and throughput suffer. This problem is especially worrisome in the data-center and cluster computing domains [**?**]. A CiPE based *contention-conscious* scheduling approach is especially well suited in these domains as the set of applications running on these systems are known, and system application scheduling plans and policies can be created offline. For example, Google, Microsoft, and Yahoo have a known set of applications that run in their data-centers, including Search, Maps, Mail, Video etc. An understanding of each application's sensitivity to cross-core interference can prove critical to improving throughput, responsiveness and even power and energy.

For our experimental setup we use the following scheduling model. We have a single batch of jobs to execute and two processing cores available. Jobs are selected to run concurrently with another job on the neighboring core. If any core becomes free, a job from the job queue is selected to run. For our experiment our queue consists of the 19 SPEC2006 benchmarks

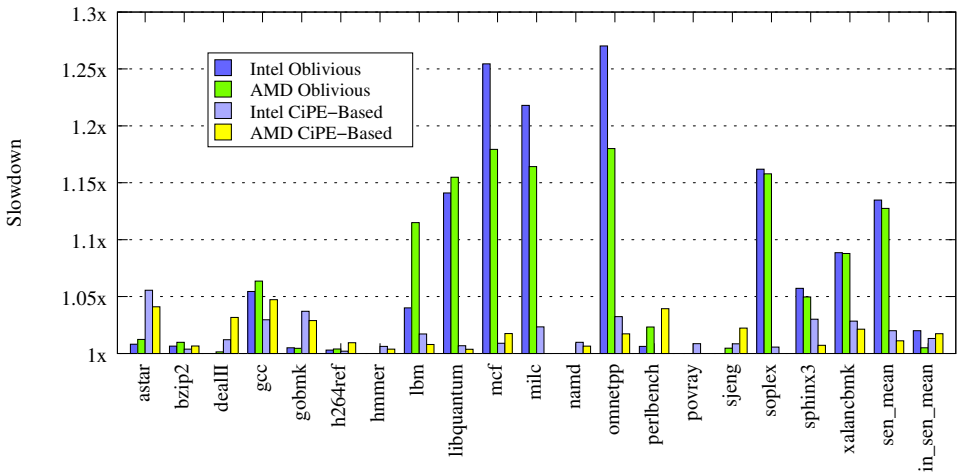


Figure C.1: Contention Oblivious vs CiPE-based scheduling (lower is better)

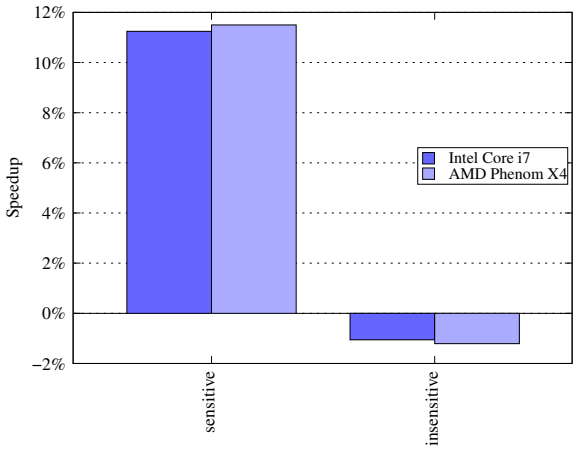


Figure C.2: Performance Impact due to Co-Scheduling (higher is better)

(C/C++). Each benchmark is run to completion on its `ref` inputs. We performed this experimentation on both Intel's Core i7 and AMD's Phenom X4 multicore architectures.

As a baseline we show the effects of *cache oblivious* scheduling. Our *contention oblivious* scheduling is a random schedule where contentious applications are naively co-scheduled. Our CiPE based *contention conscious* scheduling heuristically places applications with high sensitivity to cross-core interference with applications with a low sensitivity. Every time a job completes, the scheduler selects a job from the queue with either the highest or lowest CIS score. If the currently running job is sensitive, a job with the lowest CIS score is selected. If the currently running job is insensitive the job with the highest CIS score is selected.

Figures C.1 and C.2 show the results of our experimentation. Figure C.1 shows a significant reduction in the performance degradation that occurs due to cross-core performance interference. The bars show the execution time of each application while being co-scheduled over running alone on both the Core i7 and the Phenom X4. For each benchmark, the first and second bars show performance degradation when scheduled in a contention oblivious fashion and the third and forth bars show the degradation when scheduled based on CIS scores. As shown in the figure, for most benchmarks, the performance degradation dropped significantly using our CiPE-based approach.

Figure C.2 summarizes the overall improvement in throughput when using CiPE to select *contention-conscious* co-locations. Using our approach we were able to improve the performance of the contention sensitive SPEC2006 benchmarks by 12%, on average and up to 24% in the case of `mcf`. The remaining insensitive jobs only suffered a 1% performance impact from being co-scheduled with sensitive jobs.

Appendix D

Identifying Sensitive Code Regions with CiPE

Contents

Figures D.1 and D.2 show examples of the output of our CiPE performance debugging tool. In each figure the debuggers output from a single CIS sample is shown. As our debugger replays execution, a stream of these samples are printed to the screen or to a log file. For each CIS sample, our performance debugger points to the source-level basic blocks which were responsible for that sample's CIS score, and their dynamic coverage relative to each other. The number of blocks shown is a parameter set by the user; two or three is often covers more than 99% of the sample interval.

Our general CIS Analysis can be used for other performance debugger designs. For example, instead of replaying CiPE profiles, branch informa-

```
CIS Score: 0.21033
Rank: 1 Occupies: 68% File: lbm.c Lines: 187 - 206
Rank: 2 Occupies: 32% File: lbm.c Lines: 257 - 168
```

Figure D.1: lbm Profile Sample

```
CIS Score: 0.28453
Rank: 1 Occupies: 51% File: su3_proj.c Lines: 42 - 47
Rank: 2 Occupies: 49% File: s_m_a_mat.c Lines: 16 - 17
```

Figure D.2: milc Profile Sample

```
186   SWEEP_START( 0, 0, 0, 0, 0, SIZE_Z )
187     if( TEST_FLAG_SWEEP( srcGrid, OBSTACLE )) {
188       DST_C ( dstGrid ) = SRC_C ( srcGrid );
189       DST_S ( dstGrid ) = SRC_N ( srcGrid );
190       DST_N ( dstGrid ) = SRC_S ( srcGrid );
191       DST_W ( dstGrid ) = SRC_E ( srcGrid );
192       DST_E ( dstGrid ) = SRC_W ( srcGrid );
193       DST_B ( dstGrid ) = SRC_T ( srcGrid );
194       DST_T ( dstGrid ) = SRC_B ( srcGrid );
195       DST_SW( dstGrid ) = SRC_NE( srcGrid );
196       DST_SE( dstGrid ) = SRC_NW( srcGrid );
197       DST_NW( dstGrid ) = SRC_SE( srcGrid );
198       DST_NE( dstGrid ) = SRC_SW( srcGrid );
199       DST_SB( dstGrid ) = SRC_NT( srcGrid );
200       DST_ST( dstGrid ) = SRC_NB( srcGrid );
201       DST_NB( dstGrid ) = SRC_ST( srcGrid );
202       DST_NT( dstGrid ) = SRC_SB( srcGrid );
203       DST_WB( dstGrid ) = SRC_ET( srcGrid );
204       DST_WT( dstGrid ) = SRC_EB( srcGrid );
205       DST_EB( dstGrid ) = SRC_WT( srcGrid );
206       DST_ET( dstGrid ) = SRC_WB( srcGrid );
207       continue;
208     }
```

Figure D.3: Contention bottleneck in `lbm.c`

tion can be efficiently extracted online using structures such as Intel's last branch record (LBR), and then linked back to the source level during the CiPE profile generation. Enabling these such modifications are matters of engineering. However, once the CiPE profiles are gathered, our post processing replay debugger provides the same functionality and suffers only 20% overhead over native execution.

D.1 Contentious Code Regions of LBM and MILC

Figures D.3, D.5, and D.7 show the SPEC2006 code snippets that corresponds to the regions detected in Figures D.1 and D.2. As these figures show, for `lbm` and `milc`, the most contentious regions of code are composed of dense array and memory operations. Figures D.4, D.6, and D.8 provide a closer look at the architectural instruction stream seen by the processor.

As shown in Figure D.4, the most contentious bottleneck in `lbm` is densely

```
690:  31 d2                    xor    %edx,%edx
692:  f6 87 98 00 00 00 01     testb  $0x1,0x98(%rdi)
699:  0f 84 e5 00 00 00        je     784 <LBM_performStreamCollide+0xf4>
69f:  48 8b 07                 mov    (%rdi),%rax
6a2:  48 89 04 32              mov    %rax,(%rdx,%rsi,1)
6a6:  48 8b 47 08              mov    0x8(%rdi),%rax
6aa:  48 89 84 32 90 c1 ff     mov    %rax,-0x3e70(%rdx,%rsi,1) [6b1: ff]
6b2:  48 8b 47 10              mov    0x10(%rdi),%rax
6b6:  48 89 84 32 88 3e 00     mov    %rax,0x3e88(%rdx,%rsi,1) [6bd: 00]
6be:  48 8b 47 18              mov    0x18(%rdi),%rax
6c2:  48 89 44 32 80           mov    %rax,-0x80(%rdx,%rsi,1)
6c7:  48 8b 47 20              mov    0x20(%rdi),%rax
6cb:  48 89 84 32 b8 00 00     mov    %rax,0xb8(%rdx,%rsi,1) [6d2: 00]
6d3:  48 8b 47 28              mov    0x28(%rdi),%rax
6d7:  48 89 84 32 30 96 e7     mov    %rax,-0x1869d0(%rdx,%rsi,1) [6de: ff]
6df:  48 8b 47 30              mov    0x30(%rdi),%rax
6e3:  48 89 84 32 28 6a 18     mov    %rax,0x186a28(%rdx,%rsi,1) [6ea: 00]
6eb:  48 8b 47 38              mov    0x38(%rdi),%rax
6ef:  48 89 84 32 30 c1 ff     mov    %rax,-0x3ed0(%rdx,%rsi,1) [6f6: ff]
6f7:  48 8b 47 40              mov    0x40(%rdi),%rax
6fb:  48 89 84 32 68 c2 ff     mov    %rax,-0x3d98(%rdx,%rsi,1) [702: ff]
703:  48 8b 47 48              mov    0x48(%rdi),%rax
707:  48 89 84 32 20 3e 00     mov    %rax,0x3e20(%rdx,%rsi,1) [70e: 00]
70f:  48 8b 47 50              mov    0x50(%rdi),%rax
713:  48 89 84 32 58 3f 00     mov    %rax,0x3f58(%rdx,%rsi,1) [71a: 00]
71b:  48 8b 47 58              mov    0x58(%rdi),%rax
71f:  48 89 84 32 f0 57 e7     mov    %rax,-0x18a810(%rdx,%rsi,1) [726: ff]
727:  48 8b 47 60              mov    0x60(%rdi),%rax
72b:  48 89 84 32 e8 2b 18     mov    %rax,0x182be8(%rdx,%rsi,1) [732: 00]
733:  48 8b 47 68              mov    0x68(%rdi),%rax
737:  48 89 84 32 e0 d4 e7     mov    %rax,-0x182b20(%rdx,%rsi,1) [73e: ff]
73f:  48 8b 47 70              mov    0x70(%rdi),%rax
743:  48 89 84 32 d8 a8 18     mov    %rax,0x18a8d8(%rdx,%rsi,1) [74a: 00]
74b:  48 8b 47 78              mov    0x78(%rdi),%rax
74f:  48 89 84 32 f0 95 e7     mov    %rax,-0x186a10(%rdx,%rsi,1) [756: ff]
757:  48 8b 87 80 00 00 00     mov    0x80(%rdi),%rax
75e:  48 89 84 32 e8 69 18     mov    %rax,0x1869e8(%rdx,%rsi,1) [765: 00]
766:  48 8b 87 88 00 00 00     mov    0x88(%rdi),%rax
76d:  48 89 84 32 20 97 e7     mov    %rax,-0x1868e0(%rdx,%rsi,1) [774: ff]
775:  48 8b 87 90 00 00 00     mov    0x90(%rdi),%rax
77c:  48 89 84 32 18 6b 18     mov    %rax,0x186b18(%rdx,%rsi,1) [783: 00]
784:  48 81 c2 a0 00 00 00     add    $0xa0,%rdx
78b:  48 81 c7 a0 00 00 00     add    $0xa0,%rdi
792:  48 81 fa 00 d4 65 0c     cmp    $0xc65d400,%rdx
799:  0f 85 f3 fe ff ff        jne    692 <LBM_performStreamCollide+0x2>
```

Figure D.4: Architectural instructions for contention bottleneck in `lbm.c`

```
38 void su3_projector( su3_vector *a,
   su3_vector *b, su3_matrix *c ){
39 register int i,j;
40 register double tmp,tmp2;
41     for(i=0;i<3;i++)for(j=0;j<3;j++){
42   tmp2 = a->c[i].real * b->c[j].real;
43   tmp = a->c[i].imag * b->c[j].imag;
44   c->e[i][j].real = tmp + tmp2;
45   tmp2 = a->c[i].real * b->c[j].imag;
46   tmp = a->c[i].imag * b->c[j].real;
47   c->e[i][j].imag = tmp - tmp2;
48     }
49 }
```

Figure D.5: Contention bottleneck in su3_proj.c

```
0000000000000000 <su3_projector>:
   0:   31 c0                    xor    %eax,%eax
   2:   f2 0f 10 57 08           movsd  0x8(%rdi),%xmm2
   7:   f2 0f 10 07              movsd  (%rdi),%xmm0
   b:   66 0f 28 ca              movapd %xmm2,%xmm1
   f:   f2 0f 59 06              mulsd  (%rsi),%xmm0
  13:   f2 0f 59 4e 08           mulsd  0x8(%rsi),%xmm1
  18:   f2 0f 58 c1              addsd  %xmm1,%xmm0
  1c:   f2 0f 11 04 10           movsd  %xmm0,(%rax,%rdx,1)
  21:   f2 0f 10 0f              movsd  (%rdi),%xmm1
  25:   f2 0f 59 16              mulsd  (%rsi),%xmm2
  29:   66 0f 28 c1              movapd %xmm1,%xmm0
  2d:   f2 0f 59 4e 10           mulsd  0x10(%rsi),%xmm1
  32:   f2 0f 59 46 08           mulsd  0x8(%rsi),%xmm0
  37:   f2 0f 5c d0              subsd  %xmm0,%xmm2
  3b:   f2 0f 11 54 10 08        movsd  %xmm2,0x8(%rax,%rdx,1)
  41:   f2 0f 10 57 08           movsd  0x8(%rdi),%xmm2
  46:   66 0f 28 c2              movapd %xmm2,%xmm0
  4a:   f2 0f 59 46 18           mulsd  0x18(%rsi),%xmm0
  4f:   f2 0f 58 c8              addsd  %xmm0,%xmm1
  53:   f2 0f 11 4c 10 10        movsd  %xmm1,0x10(%rax,%rdx,1)
  59:   f2 0f 10 0f              movsd  (%rdi),%xmm1
  5d:   f2 0f 59 56 10           mulsd  0x10(%rsi),%xmm2
  62:   66 0f 28 c1              movapd %xmm1,%xmm0
  66:   f2 0f 59 4e 20           mulsd  0x20(%rsi),%xmm1
  6b:   f2 0f 59 46 18           mulsd  0x18(%rsi),%xmm0
  70:   f2 0f 5c d0              subsd  %xmm0,%xmm2
  74:   f2 0f 11 54 10 18        movsd  %xmm2,0x18(%rax,%rdx,1)
  7a:   f2 0f 10 57 08           movsd  0x8(%rdi),%xmm2
  7f:   66 0f 28 c2              movapd %xmm2,%xmm0
  83:   f2 0f 59 46 28           mulsd  0x28(%rsi),%xmm0
  88:   f2 0f 58 c8              addsd  %xmm0,%xmm1
  8c:   f2 0f 11 4c 10 20        movsd  %xmm1,0x20(%rax,%rdx,1)
  92:   f2 0f 10 07              movsd  (%rdi),%xmm0
  96:   48 83 c7 10              add    $0x10,%rdi
  9a:   f2 0f 59 56 20           mulsd  0x20(%rsi),%xmm2
  9f:   f2 0f 59 46 28           mulsd  0x28(%rsi),%xmm0
  a4:   f2 0f 5c d0              subsd  %xmm0,%xmm2
  a8:   f2 0f 11 54 10 28        movsd  %xmm2,0x28(%rax,%rdx,1)
  ae:   48 83 c0 30              add    $0x30,%rax
  b2:   48 3d 90 00 00 00        cmp    $0x90,%rax
  b8:   0f 85 44 ff ff ff        jne    2 <su3_projector+0x2>
  be:   f3 c3                    repz retq
```

Figure D.6: Architectural instructions for contention bottleneck in su3_proj.c

```
12 void scalar_mult_add_su3_matrix(su3_matrix *a,su3_matrix *b,double s,
13   su3_matrix *c){
14 register int i,j;
15     for(i=0;i<3;i++)for(j=0;j<3;j++){
16   c->e[i][j].real = a->e[i][j].real + s*b->e[i][j].real;
17   c->e[i][j].imag = a->e[i][j].imag + s*b->e[i][j].imag;
18     }
19 }
```

Figure D.7: Contention bottleneck in s_m_a_mat.c

```
0000000000000000 <scalar_mult_add_su3_matrix>:
   0:   66 0f 28 c8              movapd %xmm0,%xmm1
   4:   31 c0                    xor    %eax,%eax
   6:   66 0f 28 c1              movapd %xmm1,%xmm0
   a:   f2 0f 59 04 30           mulsd  (%rax,%rsi,1),%xmm0
   f:   f2 0f 58 04 38           addsd  (%rax,%rdi,1),%xmm0
  14:   f2 0f 11 04 10           movsd  %xmm0,(%rax,%rdx,1)
  19:   66 0f 28 c1              movapd %xmm1,%xmm0
  1d:   f2 0f 59 44 30 08        mulsd  0x8(%rax,%rsi,1),%xmm0
  23:   f2 0f 58 44 38 08        addsd  0x8(%rax,%rdi,1),%xmm0
  29:   f2 0f 11 44 10 08        movsd  %xmm0,0x8(%rax,%rdx,1)
  2f:   66 0f 28 c1              movapd %xmm1,%xmm0
  33:   f2 0f 59 44 30 10        mulsd  0x10(%rax,%rsi,1),%xmm0
  39:   f2 0f 58 44 38 10        addsd  0x10(%rax,%rdi,1),%xmm0
  3f:   f2 0f 11 44 10 10        movsd  %xmm0,0x10(%rax,%rdx,1)
  45:   66 0f 28 c1              movapd %xmm1,%xmm0
  49:   f2 0f 59 44 30 18        mulsd  0x18(%rax,%rsi,1),%xmm0
  4f:   f2 0f 58 44 38 18        addsd  0x18(%rax,%rdi,1),%xmm0
  55:   f2 0f 11 44 10 18        movsd  %xmm0,0x18(%rax,%rdx,1)
  5b:   66 0f 28 c1              movapd %xmm1,%xmm0
  5f:   f2 0f 59 44 30 20        mulsd  0x20(%rax,%rsi,1),%xmm0
  65:   f2 0f 58 44 38 20        addsd  0x20(%rax,%rdi,1),%xmm0
  6b:   f2 0f 11 44 10 20        movsd  %xmm0,0x20(%rax,%rdx,1)
  71:   66 0f 28 c1              movapd %xmm1,%xmm0
  75:   f2 0f 59 44 30 28        mulsd  0x28(%rax,%rsi,1),%xmm0
  7b:   f2 0f 58 44 38 28        addsd  0x28(%rax,%rdi,1),%xmm0
  81:   f2 0f 11 44 10 28        movsd  %xmm0,0x28(%rax,%rdx,1)
  87:   48 83 c0 30              add    $0x30,%rax
  8b:   48 3d 90 00 00 00        cmp    $0x90,%rax
  91:   0f 85 6f ff ff ff        jne    6 <scalar_mult_add_su3_matrix+0x6>
  97:   f3 c3                    repz retq
```

Figure D.8: Architectural instructions for contention bottleneck in s_m_a_mat.c

packed with mov instructions. Each of the 37 consequtive `mov` instructions are either moving data from memory into the `%rax` register, or from the `%rax` register back to memory. This memory access pattern composes the entire body of the loop less three logic operations. Each memory reference is also sparsely distributed as can be seen from the various offsets throughout the instruction stream. The lack of spacial locality in this memory access pattern further exacerbates the contentiousness of this code region since each memory reference that misses in the cache brings in an entire line.

The contentious code regions of the `milc` application is composed of streaming *singlie instruction, multiple data* (SIMD) operations. These SIMD operations use the `xmm` registers. These registers are 128 bits wide and can simultaneously perform a logic operation on either, four 32-bit single-precision floating point, two 64-bit double-precision floating point numbers, four 32-bit integer numbers, two 64-bit integers, or sixteen 8-bit bytes or characters. As Figures D.6 and D.8 shows, the contentious code regions of

`milc` has a high concentration of `moveapd` and `movesd` instructions intermixed with `addsd`, `subsd`, and `mulsd` instructions. These instructions are specialized for SIMD operations. The `moveapd` instructions represent moves of a double quadword containing two packed double-precision floating-point values, and the `movesd`, `addsd`, `subsd`, and `mulsd` instructions apply to the low double-precision floating-point value. Notice that in both of these contentious code regions, many of the logic operations are also referencing memory directly as well. This high concentration of memory intensive SIMD operations explains the contentiousness of these code regions.

An important observation that arises from inspecting these regions of code is that new low level optimizations and code transformations may be helpful in reducing contentiousness of these regions. One area of future work is to investigate the potential of compiler transformations that can either reduce the rate of memory operations, or restructure/relayout data to reduce the contentiousness of code regions that are densely packed with memory operations.

Bibliography

[1] D. Abts, M. Marty, P. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. *ISCA '10*, Jun 2010.

[2] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 267–276, New York, NY, USA, 2004. ACM.

[3] F. Ahmad and T. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. *ASPLOS '10*, Mar 2010.

[4] I. Al-Azzoni and D. Down. Dynamic scheduling for heterogeneous desktop grids. *GRID '08*, Sep 2008.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM.

[6] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance coun-

ters. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM.

[7] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000.

[8] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *OSDI '99*, Berkeley, CA, USA, 1999. USENIX Association.

[9] L. Barroso, J. Dean, and U. Holzle. Web-search for a planet: The Google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[10] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. Amaral. Shared memory programming for large scale machines. *PLDI '06*, May 2006.

[11] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.

[12] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[13] J. Burge, P. Ranganathan, and J. Wiener. Cost-aware scheduling for heterogeneous enterprise machines (cash'em). *CLUSTER '07*, Sep 2007.

[14] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: A framework for cloud computing. Technical Report MSR-TR-2010-159, Microsoft Research, November 2010.

[15] M. Byler, M. Wolfe, J. R. B. Davies, C. Huson, and B. Leasure. Multiple version loops. In *ICPP*, pages 312–318, 1987.

[16] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. *SIGPLAN Not.*, 39(4):155–166, 2004.

[17] P. R. Carini and M. Hind. Flow-sensitive interprocedural constant propagation. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 23–31, New York, NY, USA, 1995. ACM.

[18] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society.

[19] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[20] 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. *OSDI '06*, Nov 2006.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[22] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.

[23] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.

[24] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[25] S. Chen, P. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. *SPAA '07*, Jun 2007.

[26] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on cmps. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115, New York, NY, USA, 2007. ACM.

[27] L. Cherkasova, Y. Fu, W. Tang, and A. Vahdat. Measuring and characterizing end-to-end internet service performance. *ACM Trans. Internet Technol.*, 3(4):347–391, 2003.

[28] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. *MICRO 39*, Dec 2006.

[29] B.-G. Chun, G. Iannaccone, G. Iannaccone, R. Katz, G. Lee, and L. Niccolini. An energy case for hybrid datacenters. *SIGOPS Operating Systems Review*, 44(1), Mar 2010.

[30] M. Cierniak and W. Li. Interprocedural array remapping. In *PACT '97: Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, page 146, Washington, DC, USA, 1997. IEEE Computer Society.

[31] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Comput. Lang.*, 19(2):105–117, 1993.

[32] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[33] P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, May 1997.

[34] 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*, 2010.

[35] EPA. Epa report to congress on server and data center energy efficiency. Technical report, U.S. Protection Agency, 2007.

[36] S. Eranian. Perfmon2. http://perfmon2.sourceforge.net/.

[37] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 53(2), 2010.

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

[39] G. Fursin, A. Cohen, M. F. P. O'Boyle, and O. Temam. Quick and practical run-time evaluation of multiple program optimizations. *Trans. on High Performance Embedded Architectures and Compilers*, 1(1):13–31, Jan. 2007.

[40] G. Fursin and O. Temam. Collective optimization. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *SOCC '11*, Oct 2011.

[42] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39:49–57, April 2004.

[43] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. *MICRO 40*, Dec 2007.

[44] R. Gupta, D. A. Berson, and J. Z. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on*

Microarchitecture, pages 358–368, Washington, DC, USA, 1997. IEEE Computer Society.

[45] J. Hamilton. Internet-scale service infrastructure efficiency. *SIGARCH Comput. Archit. News*, 37(3):232–232, 2009.

[46] T. Heath, B. Diniz, E. V. Carrera, W. Meira, Jr., and R. Bianchini. Energy conservation in heterogeneous server clusters. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 186–195, New York, NY, USA, 2005. ACM.

[47] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[48] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 479–488, New York, NY, USA, 2009. ACM.

[49] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

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

[51] R. Huang, H. Casanova, and A. Chien. Automatic resource specification generation for resource selection. *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Nov 2007.

[52] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A nuca substrate for flexible cmp cache sharing. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 31–40, New York, NY, USA, 2005. ACM.

[53] Intel Corporation. *IA-64 Application Developer's Architecture Guide*. Intel Corporation, Santa Clara, CA, USA, 2009.

[54] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, New York, NY, USA, 2004. ACM.

[55] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 2007. ACM.

[56] 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*, Jun 2007.

[57] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr, and J. Emer. Adaptive insertion policies for managing shared caches. *PACT '08*, Oct 2008.

[58] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web-search using mobile cores: quantifying and mitigating the price of efficiency. In *ISCA '10*, New York, NY, USA, 2010. ACM.

[59] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[60] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. *HiPeac '10*, pages 201–215, 2010.

[61] V. Kazempour, A. Kamali, and A. Fedorova. Aash: an asymmetry-aware scheduler for hypervisors. *VEE '10: Proceedings of the 6th /SIGOPS international conference on Virtual execution environments*, Mar 2010.

[62] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

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

[64] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Apr 2010.

[65] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 30, July 2010.

[66] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. *MICRO 36: Proceedings of the 36th annual International Symposium on Microarchitecture*, Dec 2003.

[67] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *ISCA '04*, Jun 2004.

[68] N. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009.

[69] T. Li, D. Baumberger, D. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Nov 2007.

[70] 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. *HPCA '08*, pages 367–378, 2008.

[71] F. Liu, X. Jiang, and Y. Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. *HPCA '10*, pages 1–12, 2010.

[72] S. Lohr. Demand for data puts engineers in spotlight. *The New York Times*, 2008. Published June 17th.

[73] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using

hardware counters. In *14th Conference on Parallel and Distributed Computing Systems*, August 2001.

[74] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of run-time data cache prefetching in a dynamic optimization system. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.

[75] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[76] J. Machina and A. Sodan. Predicting cache needs and cache sensitivity for applications in cloud computing on cmp servers with configurable caches. *IPDPS 2009*, pages 1 – 8, 2009.

[77] J. Mars and R. Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[78] J. Mars and M. L. Soffa. Multicore adaptive trace selection. Appeared at STMCS '08: Third Workshop on Software Tools for MultiCore Systems, March 2008.

[79] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. Lou Soffa. Bubble-Up: increasing utilization in modern warehouse-scale computers via sensible co-locations. In *MICRO-44 '11: Proceedings of the 44th Annual*

IEEE/ACM International Symposium on Microarchitecture. ACM Request Permissions, Dec. 2011.

[80] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross-core interference through contention synthesis. In *HiPEAC '11*, pages 167–176, New York, NY, USA, 2011. ACM.

[81] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *CGO '10: Proceedings of the 2010 International Symposium on Code Generation and Optimization*, pages 257–265, New York, NY, USA, 2010. ACM.

[82] A. Mishra, J. Hellerstein, W. Cirne, and C. Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

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

[84] R. Nathuji, C. Isci, and E. Gorbatov. Exploiting platform heterogeneity for power efficient data centers. *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, Jun 2007.

[85] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Apr 2010.

[86] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. *MICRO-39*, pages 208 – 222, 2006.

[87] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. *Micro, IEEE*, 28(3):6 – 16, 2008.

[88] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Washington, DC, USA, 2006. IEEE Computer Society.

[89] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 57–68, New York, NY, USA, 2007. ACM.

[90] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 150–162, Washington, DC, USA, 2007. IEEE Computer Society.

[91] S. Pelley, D. Meisner, P. Zandevakili, T. Wenisch, and J. Underwood. Power routing: dynamic power provisioning in the data center. *ASPLOS '10*, Mar 2010.

[92] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM.

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

[94] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 2–12, New York, NY, USA, 2006. ACM.

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

[96] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. *PACT 2007*, pages 245 – 258, 2007.

[97] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-directed optimization of event-based programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 106–116, New York, NY, USA, 2002. ACM.

[98] R. Reddy and P. Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 198–207, New York, NY, USA, 2007. ACM.

[99] C. R. Reeves, editor. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[100] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *Micro, IEEE*, 30(4):65 –79, july-aug. 2010.

[101] S. Ried, H. Kisker, P. Matzke, A. Bartels, and M. Lisserman. Sizing the cloud, understand and quantifying the future of cloud computing. *Forrester Research, Inc.*, 2011.

[102] J. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Apr 2010.

[103] S. M. Sait and H. Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

[104] F. Schneider, M. Payer, and T. Gross. Online optimizations driven by hardware performance monitoring. *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, Jun 2007.

[105] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 373–382, New York, NY, USA, 2007. ACM.

[106] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[107] D. Shelepov, J. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous

multicore systems. *SIGOPS Operating Systems Review*, 43(2), Apr 2009.

[108] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 258–269, Washington, DC, USA, 2008. IEEE Computer Society.

[109] S. Srikantaiah, R. Das, A. Mishra, C. Das, and M. Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. *SC '09*, Nov 2009.

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

[111] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, 2005.

[112] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02*. IEEE Computer Society, 2002.

[113] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.

[114] N. Tallent, J. Mellor-Crummey, L. Adhianto, M. Fagan, and M. Krentel. Diagnosing performance bottlenecks in emerging petascale

applications. *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009.

[115] D. Tam, R. Azimi, L. Soares, and M. Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. *ASPLOS '09*, Feb 2009.

[116] 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 2012 International Symposium on Code Generation and Optimization*, New York, NY, USA, 2012. ACM.

[117] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA '11: Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 283–294, New York, NY, USA, 2011. ACM.

[118] M. Voss and R. Eigemann. High-level adaptive program optimization with adapt. *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, Jul 2001.

[119] M. Wegiel and C. Krintz. The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 91–102, New York, NY, USA, 2008. ACM.

[120] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed run-times. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages*

and operating systems, pages 289–300, New York, NY, USA, 2009. ACM.

[121] J. A. Winter and D. H. Albonesi. Scheduling algorithms for unpredictably heterogeneous cmp architectures. *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 42 – 51, 2008.

[122] D. Woo and H.-H. Lee. Prophet: goal-oriented provisioning for highly tunable multicore processors in cloud computing. *SIGOPS Operating Systems Review*, 43(2), Apr 2009.

[123] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in cmps. *Proc. of CMP-MSI, held in conjunction with ISCA-35*, 2008.

[124] Y. Xie and G. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. *ISCA '09*, Jun 2009.

[125] 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.

[126] C. Xu, X. Chen, R. Dick, and Z. Mao. Cache contention and application performance prediction for multi-core systems. In *ISPASS 2010*, march 2010.

[127] D. Xu, C. Wu, and P.-C. Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. *PACT '10*, Sep 2010.

[128] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. *OSDI'08*, Dec 2008.

[129] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[130] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 50–64, Washington, DC, USA, 2006. IEEE Computer Society.

[131] M. Zhao, B. R. Childers, and M. L. Soffa. An approach toward profit-driven optimization. *ACM Trans. Archit. Code Optim.*, 3(3):231–262, 2006.

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