ADDRESSING SHARED RESOURCE CONTENTION IN DATACENTER SERVERS

by

Sergey Blagodurov

Diploma with Honors, Moscow Engineering Physics Institute (State University), 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2013

APPROVAL

Name: Sergey Blagodurov

Degree: Doctor of Philosophy

Title of Thesis: Addressing Shared Resource Contention in Datacenter Servers

Examining Committee:

Dr. Greg Mori
Associate Professor & Associate Director
Chair

Dr. Alexandra Fedorova
Senior Supervisor
Associate Professor

Dr. Jian Pei
Supervisor
Professor

Dr. Arrvindh Shriraman
Internal Examiner
Assistant Professor

Dr. Diwakar Krishnamurthy
External Examiner
Associate Professor, Electrical and Computer Engineering
University of Calgary

Date Defended/Approved: August 6, 2013

Partial Copyright Licence

Abstract

Servers are major energy consumers in modern datacenters. Much of that energy is wasted because applications compete for shared resources and suffer severe performance penalties due to resource contention. Contention for shared resources remains an unsolved problem in existing datacenters despite significant research efforts dedicated to this problem in the past. The goal of this work is to investigate how and to what extent contention for shared resource can be mitigated via workload scheduling. Scheduling is an attractive tool, because it does not require extra hardware and is relatively easy to integrate into the system.

I have designed and implemented multiple Open Source and proprietary schedulers during my work on this dissertation. Most notably, I introduced the *Distributed Intensity Online* (DIO) scheduler to target the shared resource contention in the memory hierarchy of Uniform Memory Access (UMA) systems, followed by the *Distributed Intensity NUMA Online* (DINO) scheduler that I designed to improve performance and decrease power consumption on Non Uniform Memory Access (NUMA) servers. As part of my internship in HP Labs, I designed a work conserving scheduler that prioritizes access to the multiple CPU cores on an industry level multicore server, thus managing contention for CPU and improving server power efficiency. Finally, the *Clavis2D* framework extends the contention awareness to the datacenter level and provides a comprehensive cluster scheduling solution that simultaneously takes into account multiple performance- and power-related goals.

My dissertation work utilizes state-of-the-art industry level datacenter infrastructure, does not require any modification or prior knowledge about the workload and provides significant performance and energy benefits on-the-fly.

Keywords: Multicore processors; scheduling; shared resource contention; performance evaluation

Dedicated to the loving memory of my grandmother, Maria Pavlovna Blagodurova.

"Success is not final, failure is not fatal: it is the courage to continue that counts."

— The name of the person who said these words goes here. There are hundreds of websites on the Internet that attribute this quote to Sir Winston Churchill without providing any reference [14]. I liked the quotation, so I wanted to include it at the forefront of my PhD work. Satisfied by my finding, I got curious about exactly when and where Churchill said that.

The direct source that attributes these words to him is a book called "The Prodigal Project: Genesis" by Ken Abraham and Daniel Hart [37]. However, Richard Langworth, a Churchill historian, in the appendix to his book "Churchill By Himself" states that, contrary to the wide spread belief, Churchill actually never said that [8]. According to the Wikipedia [43], the above words may be a misattribution of a similar quote from Don Shula, an American football player and coach: "Success is not forever and failure isn't fatal."

Enjoy the rest of the thesis.

Acknowledgments

Working with many wonderful people was *sine qua non* to my dissertation. First and foremost, I want to mention Sasha, my Doctoral advisor of five years. Plainly speaking, Sasha is the smartest, most graceful person I know! She is a wonderful advisor, caring and understanding. PhD is notable for its stresses, so having an advisor that is attentive to your matters helps enormously. She did a tremendous amount of work teaching me to be a productive, independent scientist. I simply could not thank her enough for spending all those countless days with me brainstorming ideas, helping me to prepare and conduct experiments, present the results, read and write excellent scientific papers. If I had to choose again whom to work with for my PhD, I would choose her for sure! She achieved a lot, but I am confident she has yet more stellar career in front of her. I am very proud that my PhD is the first one that is almost entirely a product of her wonderful supervision.

For several years, I have been working with many brilliant scientists at Hewlett-Packard Laboratories, most notably with Martin Arlitt, Daniel Gmach and Cullen Bash. Martin was my first industrial mentor. He showed me that it is important to think about practicality of your research, the passion that I shared with him ever since. He later introduced me to Daniel and Cullen both of whom were working in the same team. Together, we were involved in the Net-Zero Energy datacenter Big Bet at HP Labs. This is a multi-disciplinary project that involved both engineers and computer scientists. It helped me to realize just how important your research could be if it is part of a project of such scale. For these reasons, I strongly believe that a Computer Science PhD student benefits in a big way from being mentored in industry in addition to their academic endeavours.

I am also grateful to Dr. Jian Pei and Dr. Fabien Hermenier for providing helpful insights on many parts of my Thesis work.

My work would be impossible without collaboration with other students. I would like to thank Daniel Shelepov, Juan Carlos Saez, Ananth Narayan, Mohammad Dashti, Tyler Dwyer, and Jessica Jiang for discussing various research topics with me and helping me evaluate and evolve my ideas.

A special thank you goes to Sergey Zhuravlev, a brilliant collaborator in the early years of my PhD. We did a lot of work together and delivered some fantastic results thanks to his bright mind and keen attitude to research.

Finally, I would like to thank Yaroslav Litus, Nasser Ghazali-Beiklar, Jay Kulkarni and Fabien Gaud for being my good friends throughout my studies. Although we did not have a chance to work together (yet!), their emotional support nourished me a lot, helping me to get through these challenging times.

Cheers to you all, this was a great ride!

Contents

List of Tables

List of Figures

Part I

Prologue

Chapter 1

Introduction

1.1 New Challenges for Datacenter Workload Management

Datacenters have become the platform of choice for a variety of applications, including financial, animation, simulation, scientific, Internet services and many more. The growing popularity of datacenters enables tremendous cost and energy savings for their users. But the paradox is that, in doing so, the energy consumption and carbon dioxide emissions of datacenters themselves is increasing at the rapid rate. The U.S. Environmental Protection Agency has reported that the energy consumption of datacenters located in U.S. was 61 billion Kilowatt-Hours in 2006, which cost $4.5 billion. Today, datacenters world-wide consume more electricity annually than the entire country of Mexico [38].

Another huge problem related to the growing datacenter demand is the greenhouse gas emissions. According to the Greenhouse Gas Equivalencies Calculator available at the US Environmental Protection Agency website [15], a 20 MW 24/7 datacenter that is up for 1 year is equivalent to:

- 23k cars in annual greenhouse gas emissions.
- Carbon dioxide emissions from 280k barrels of oil consumed.
- Carbon dioxide emissions from the electricity use of 15k homes for one year.

In other words, *a single datacenter generates as much greenhouse gas as a small city!*

What can we do to make the datacenters more efficient in terms of their energy consumption and greenhouse gas emissions? To answer this question, we first need to determine where do datacenters spend energy (Figure 1.1)? Various sources provide different data, but, according to most

of them the datacenter servers are the main energy consumers with up to 90% of a total datacenter consumption [13]. As for the energy consumption within each server, the CPU and memory tend to be the dominant consumers [100]. *My thesis addresses the energy efficiency of CPU and memory, the two most energy-hungry datacenter server resources.*

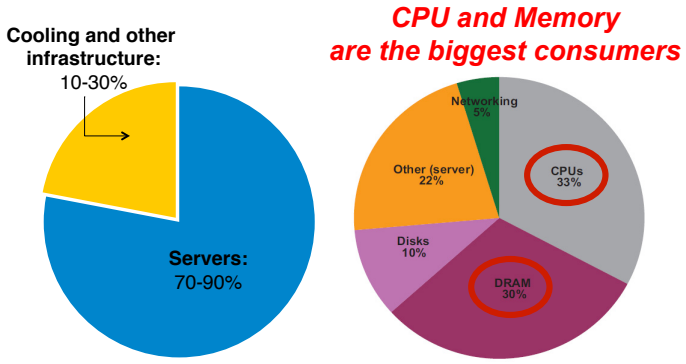


Figure 1.1: Where do datacenters spend energy?

We[1] increase the datacenter energy efficiency by targeting the problem of shared resource contention within the datacenter servers. More precisely, we address the following challenges:

1) Much of the energy is wasted because applications compete for system shared resources and suffer severe performance penalties due to resource contention. Each server is a multicore system, schematically depicted in Figure 1.2, where cores share parts of the memory hierarchy, which we term *memory domains*, compete for resources such as last-level caches (LLC), system request queues and memory controllers. Memory domains, local to the cores, can be accessed in less time than remote ones, and each memory domain has its own memory controller. My thesis work focuses on addressing contention for these shared memory resources. This is a challenging problem for two reasons. First, it is hard to identify the potential bottlenecks on a real, industry-level multicore machine, since modern servers are highly complex. Second, it is unclear how to use the knowledge about the contention to speed up workload execution and reduce power consumption automatically and on-the-fly, which are the essential requirements for today's large scale systems.

2) Despite the significant demand for computational resources, server CPU cores in datacenters tend to be under-utilized. Low utilization means that power is not used as effectively as it could be, as servers use a disproportionate amount of power when idle, relative to when they are busy. For example, energy efficiency (work completed per unit of energy) of commodity server systems

[1] In what follows, unless otherwise stated, I will use the words: I, My, We, Our, interchangeably to refer to my exclusive contributions.

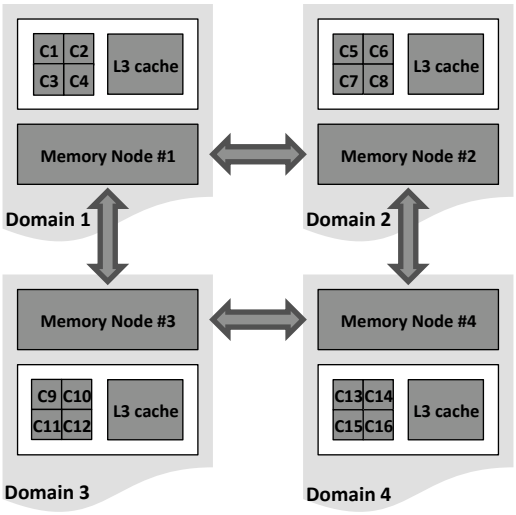


Figure 1.2: A schematic view of a system with four memory domains and four cores per domain. There are 16 cores in total, and a shared L3 cache per domain.

at 30% utilization can be less than half that at 100% [100, 25]. The advent of virtualization enabled the consolidation of several applications onto a server, increasing its utilization. However, even in virtualized resource pools, utilization is typically below 40% [11, 34] because: (a) Most virtualized resource pools are fairly static and don't implement live migration of virtual machines. (b) Most datacenter workloads have very bursty demands thus leading to conservative consolidation decisions. (c) The performance of user-interactive applications drops significantly when they must contend for busy resources. The poor performance of these 'critical' applications at high utilization levels is the main challenge in increasing the utilization of servers in cloud-like datacenters (see the red line on Figure 1.3).

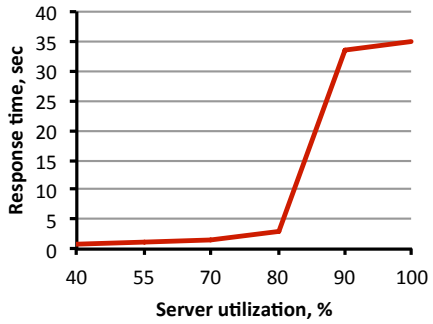


Figure 1.3: Collocated experiments without preferred access.

3) On a datacenter scale, the objectives of performance and energy efficiency are usually tightly coupled, and pursuing one objective requires a trade-off with another. Consider a simple example. The cluster on Figure 1.4 consists of 3 nodes. There are 3 jobs currently running on it: A, B and C. Each job consists of processes. Jobs A and B in our example consist of 2 processes, and job C of only one. We assign processes to nodes. Nodes have several CPU cores (2 in our example). Cluster schedulers typically put each job process on its own CPU core due to their high CPU demand. However, the processes collocated on the same multicore node are sensitive to the resource contention. Thus, one of the scheduling goals in the cluster is to minimize shared resource contention within each cluster multicore node. To mitigate the contention, we can put processes on different nodes, like processes B0 and B1. However, the trade-off is that in doing so, we leave cores on some nodes idle and so we waste power. It is essential to reduce the energy consumption by minimizing the amount of powered up cluster nodes. Another caveat is that processes belonging to the same job communicate, and if we place them on different nodes they will suffer increased communication cost. To preserve network bandwidth and improve latency, we need to maximize intra-node communication in the cluster.

The problem we target is: how to assign processes to nodes in order to simultaneously minimize several performance- and power-related objectives (e.g., contention, power consumption and communication overhead)? Unfortunately, state-of-the-art cluster installations have neither the right algorithms nor means of enforcing an optimal scheduling decision on-the-fly. While previous research in the field advanced state-of-the-art technologies by considering mostly power efficiency, none addressed all three goals simultaneously.

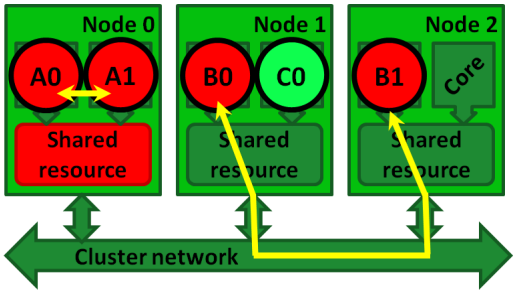


Figure 1.4: A simple job scheduling example.

1.2 Thesis contributions

The single most important contribution of my research is the development of techniques for automatic, on-the-fly detection, measurement and mitigation of shared resource contention in datacenter servers. This in turn reduces workload execution time and consumed datacenter energy. Below are the contributions of this thesis.

Addressing contention for server memory hierarchy in UMA systems. Contention for shared resources on multicore processors remains an unsolved problem in existing systems despite significant research efforts dedicated to this problem in the past. Previous solutions focused primarily on hardware techniques and software page coloring to mitigate this problem. Our goal was to investigate how and to what extent contention for shared resource can be mitigated via thread scheduling. Scheduling is an attractive tool, because it does not require extra hardware and is relatively easy to integrate into the system. Our study is the first to provide a comprehensive analysis of contention-mitigating techniques that use only scheduling. The most difficult part of the problem is to find a classification scheme for threads, which would determine how they affect each other when competing for shared resources. We provide a comprehensive analysis of such classification schemes using a newly proposed methodology that enables us to evaluate these schemes separately from the scheduling algorithm itself and to compare them to the optimal. As a result of this analysis we discovered a classification scheme that addresses not only contention for cache space, but contention for other shared resources, such as the memory controller, memory bus and prefetching hardware. To show the applicability of our analysis we design a new scheduling algorithm called DIO, which we prototype at user level, and experimentally demonstrate that it performs within 2% of the optimal. We also conclude that the highest impact of contention-aware scheduling techniques is not in improving performance of a workload as a whole but in improving quality of service or performance isolation for individual applications and in optimizing system energy consumption.

Addressing contention for server memory hierarchy in NUMA systems. The work on contention-aware scheduling so far assumed that the underlying system is UMA (uniform memory access latencies, single memory controller). Modern multicore systems, however, are NUMA, which means that they feature non-uniform memory access latencies and multiple memory controllers. We discovered that state-of-the-art contention management algorithms fail to be effective on NUMA systems and may even *hurt* performance relative to a default OS scheduler. We further investigate the causes for this behavior and design DINO, the first contention-aware algorithm for NUMA systems.

Addressing contention for server CPU cores. Servers in most cloud datacenters, e.g., in Amazon EC2 or HP CloudSystem, are often under-utilized due to concerns about Service Level Agreement (SLA) violations that may result from resource contention as server utilization increases. This low utilization means that neither the capital investment in the servers nor the power consumed is being used as effectively as it could be. In this work, we present a novel method for managing the collocation of critical (e.g., user interactive) and non-critical (e.g., batch) workloads on virtualized multicore servers. Unlike previous cap-based solutions, our approach improves server utilization while meeting the SLAs of critical workloads by prioritizing resource access using Linux cgroups weights. Extensive experimental results suggest that the proposed work conserving collocation method is able to utilize a server to nearly 100% while keeping the performance loss of critical workloads within the specified limits.

Addressing contention globally on the cluster level. Computing clusters running large distributed jobs can waste lots of energy. State-of-the-art job schedulers overlook the fact that jobs collocated on the same node may compete for the node's shared resources, communicating components of a job may experience significant latency, and the node resources may be underutilized. All these factors can lead to energy waste. Prior work attempted to address pieces of the problem, like minimizing communication distance or reducing idleness, but we are not aware of a solution that addresses the problem comprehensively. We were interested in answering the question: *How do we build a system that simultaneously minimizes contention for shared resources, communication distance and idleness while balancing these objectives according to human preferences?* We designed and implemented *Clavis2D* – a system that addresses this goal. At its heart is a custom multi-objective solver that uses domain-specific knowledge to deliver good scheduling decisions within minutes, as opposed to hours or days. Our experiments on a clusters with up to 16 nodes show that *Clavis2D* delivers energy savings of up to 5 kWh (3kWh on average) and performance improvement of up to 60% (20% on average).

The rest of the dissertation is structured as follows.

Chapter 2 presents our analysis and solution for addressing memory hierarchy contention in UMA systems via scheduling. Chapter 3 describes the design and evaluation of the first NUMA-aware scheduler. Chapter 4 introduces the challenges of collocating workloads of varying CPU usage and presents our answer to this problem. Chapter 5 provides background on cluster architecture and presents our solution to the contention management on the cluster scale. Chapter 6 discusses related work in the area of contention-aware scheduling in datacenters. Chapter 7 summarizes the contributions and lessons of this dissertation and outlines future research.

Part II

Addressing contention locally within each server

Chapter 2

Addressing contention for memory hierarchy in UMA systems

2.1 Introduction

This chapter dives into the topic of shared resource contention in datacenter servers by focusing on the memory hierarchy contention in UMA multicore servers. The multicore processors deployed into such servers have become so prevalent in both desktops and servers that they may be considered the norm for modern computing systems. The limitations of techniques focused on extraction of instruction-level parallelism (ILP) and the constraints on power budgets have greatly stymied the development of large single cores and made multicore systems a very likely future of computing, with hundreds to thousands of cores per chip. In operating system scheduling algorithms used on multicore systems, the primary strategy for placing threads on cores is load balancing. The OS scheduler tries to balance the runnable threads across the available resources to ensure fair distribution of CPU time and minimize the idling of cores. There is a fundamental flaw with this strategy that arises from the fact that a core is not an independent processor but rather a part of a larger on-chip system and hence shares resources with other cores. It has been documented in previous studies [66, 125, 127, 155, 165, 169] that the execution time of a thread can vary greatly depending on which threads run on the other cores of the same chip. This is especially true if several cores share the same last-level cache (LLC).

Figure 2.1 highlights how the decisions made by the scheduler can affect the performance of an application. This figure shows the results of an experiment where four applications were running

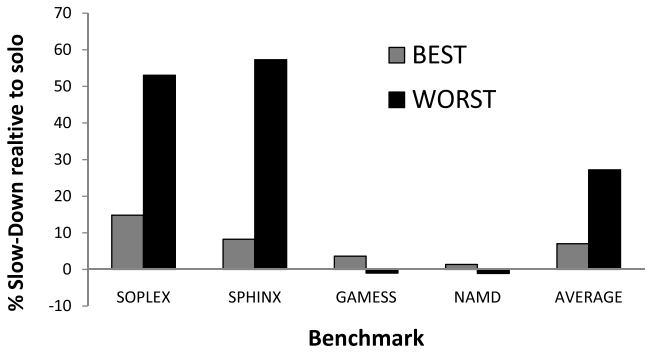


Figure 2.1: The performance degradation relative to running solo for two different schedules of SPEC CPU2006 applications on an Intel Xeon X3565 quad-core processor (two cores share an LLC).

simultaneously on a system with four cores and two shared caches. There are three unique ways to distribute the four applications across the four cores, with respect to the pairs of co-runners sharing the cache; this gives us three unique schedules. We ran the threads in each of these schedules, recorded the average completion time for all applications in the workload, and labeled the schedule with the lowest average completion time as the *best* and the one with the highest average completion time as the *worst*. Figure 2.1 shows the performance degradation that occurs due to sharing an LLC with another application, relative to running solo (contention-free). The best schedule delivers a 20% better average completion time than the worst one. Performance of individual applications improves by as much as 50%.

Previous work on the topic of contention-aware scheduling for multicore systems focused primarily on the problem of *cache contention* since this was assumed to be the main if not the only cause of performance degradation [80, 116, 169, 168, 72]. In this context cache contention refers to the effect when an application is suffering extra cache misses because its co-runners (threads running on cores that share the LLC) bring their own data into the LLC evicting the data of others. Methods such as utility cache partitioning (UCP) [155] and page coloring [66, 169, 191] were devised to mitigate cache contention. The detailed discussion of the related work for this chapter is given in Section 6.2.

Through extensive experimentation on real systems as opposed to simulators we determined that cache contention is not the dominant cause of performance degradation of threads co-scheduled to the same LLC. Along with cache contention other factors like *memory controller contention*,

memory bus contention, and *prefetching hardware contention* all combine in complex ways to create the performance degradation that threads experience when sharing the LLC.

Our goal is to investigate contention-aware scheduling techniques that are able to mitigate as much as possible the factors that cause performance degradation due to contention for shared resources. Such a scheduler would provide speedier as well as more stable execution times from run to run. Any contention aware scheduler must consist of two parts: a classification scheme for identifying which applications should and should not be scheduled together as well as the scheduling policy which assigns threads to cores given their classification. Since the classification scheme is crucial for an effective algorithm, we focused on the analysis of various classification schemes. We studied the following schemes: Stack Distance Competition (SDC) [64], Animal Classes [187], Solo Miss Rate [116], and the Pain Metric. The best classification scheme was used to design a scheduling algorithm, which was prototyped at user level and tested on two very different systems with a variety of workloads.

Our methodology allowed us to identify the *last-level cache miss rate*, which is defined to include *all* requests issued by LLC to main memory including pre-fetching, as one of the most accurate predictors of the degree to which applications will suffer when co-scheduled. We used it to design and implement a new scheduling algorithm called Distributed Intensity (DI). We show experimentally on two different multi-core systems that DI performs better than the default Linux scheduler, delivers much more stable execution times than the default scheduler, and performs within a few percentage points of the optimal. DI needs only the real miss rates of applications, which can be easily obtained online. As such we developed an online version of DI, DI Online (DIO), which dynamically reads miss counters online and schedules applications in real time. Our schedulers are implemented at user-level, and although they could be easily implemented inside the kernel, the user-level implementation was sufficient for evaluation of these algorithms' key properties.

The key contribution of our work is the *analysis demonstrating the effectiveness of various classification schemes in aiding the scheduler to mitigate shared resource contention*. Previous studies focusing on contention-aware scheduling did not investigate this issue comprehensively. They attempted isolated techniques, in some cases on a limited number of workloads, but did not analyze a variety of techniques and did not quantify how close they are to optimal. Therefore, understanding what is the *best* we can do in terms of contention-aware scheduling remains an open question. Our analysis, in contrast, explores a variety of possible classification schemes for determining to what extent the threads will affect each other‘s performance, and we believe we cover most of the schemes

previously proposed in literature as well as introducing our own. In order to perform thorough analysis we devised a new methodology for evaluating classification schemes independently of scheduling algorithms. Using this methodology we compare each classification scheme to the theoretical optimal, and this provides a clear understanding of what is the best we can do in a scheduler. Further, we analyze the extent of performance improvements that can be achieved for different workloads by methodically categorizing the workloads based on the potential speedup they can achieve via contention aware scheduling. This enables us to evaluate the applicability of cache-aware scheduling techniques for a wide variety of workloads. We believe that our work is the first to comprehensively evaluate the potential of scheduling to mitigate contention for shared resources.

The primary application of our analysis is for building new scheduling algorithms that mitigate the effects of shared resource contention. We demonstrate this by designing and evaluating a new algorithm *Distributed Intensity Online*. Our evaluation leads us to a few interesting and often unexpected findings. First of all, we were surprised to learn that if one is trying to improve average workload performance, the default contention unaware scheduler already does a rather good job if we measure performance of a workload over a large number of trials. The reason is that for a given workload there typically exists a number of "good" and "bad" scheduling assignments. In some workloads each of these assignments can be picked with a roughly equal probability if selecting uniformly at random, but in other workloads a good assignment is far more likely to occur than the bad one. A contention unaware default scheduler runs into good and bad assignments according to their respective probabilities, so over time it achieves performance that is not much worse than under a contention-aware algorithm that always picks the good assignment. However, when one is interested in improving performance of individual applications, for example to deliver quality of service guarantees or to accomplish performance isolation, a contention-aware scheduler can offer significant improvements over default, because this scheduler would almost never select a bad scheduling assignment for the prioritized application.

We also found that DIO can be rather easily adjusted to optimize for system energy consumption. On many systems, power consumption can be reduced if the workload is concentrated on a handful of chips, so that remaining chips can be brought into a low-power state. At the same time, clustering the workload on a few chips forces the threads to share memory-hierarchy resources more intensely. This can lead to contention, hurt performance and increase system uptime, leading to *increased* energy consumption. So in order to determine whether threads should be clustered (to save power) or spread across chips (to avoid excessive contention) the scheduler must be able to predict to what extent threads will hurt each other's performance if clustered. We found that DIO was able to make

this decision very effectively, and this lead us to design DIO-POWER – an algorithm that optimizes both performance and power consumption. As a result, DIO-POWER is able to improve energy-delay product over plain DIO, by as much as 80% in some cases.

The rest of the chapter is organized as follows: Section 2.2 describes the classification schemes and policies that we evaluated, the methodology for evaluating the classification schemes separately from the policies and provides the evaluation results for the classification schemes. Section 2.3 attempts to quantify the effects of different factors resulting from contention for shared on-chip resources on performance on multicore CPUs in order to better explain the results of Section 2.2. Section 2.4 describes the contention aware scheduling algorithms that were implemented and tested on real systems. Section 2.5 provides the experimental results for the contention aware scheduling algorithms. Section 2.6 presents and evaluates DIO-POWER. Section 2.7 concludes the chapter.

The contents of Sections 2.2, 2.3 and 2.4 are based on work which was done collaboratively between the author Sergey Blagodurov and his colleague Sergey Zhuravlev.

2.2 Classification Schemes

2.2.1 Methodology

A conventional approach to evaluate new scheduling algorithms is to compare the speedup they deliver relative to a default scheduler. This approach, however, has two potential flaws. First, the schedule chosen by the default scheduler varies greatly based on stochastic events, such as thread spawning order. Second, this approach does not necessarily provide the needed insight into the quality of the algorithms. A scheduling algorithm consists of two components: the information (classification scheme, in our case) used for scheduling decisions and the policy that makes the decisions based on this information. The most challenging part of a cache-aware scheduling algorithm is to select the right classification scheme, because the classification scheme enables the scheduler to predict the performance effects of co-scheduling any group of threads in a shared cache. Our goal was to evaluate the quality of classification schemes separately from any scheduling policies, and only then evaluate the algorithm as a whole. To evaluate classification schemes independently of scheduling policies, we have to use the classification schemes in conjunction with a “perfect” policy. In this way, we are confident that any differences in the performance between the different algorithms are due to classification schemes, and not to the policy.

A "perfect" scheduling policy

As a perfect scheduling policy, we use an algorithm proposed by Jiang et al. [109]. This algorithm is guaranteed to find an optimal scheduling assignment, i.e., the mapping of threads to cores, on a machine with several clusters of cores sharing a cache as long as the *co-run degradations* for applications are known. A co-run degradation is an increase in the execution time of an application when it shares a cache with a co-runner, relative to running solo.

Jiang's methodology uses the co-run degradations to construct a graph theoretic representation of the problem, where threads are represented as nodes connected by edges, and the weights of the edges are given by the sum of the mutual co-run degradations between the two threads. The optimal scheduling assignment can be found by solving a min-weight perfect matching problem. For instance, given the co-run degradations in Table 2.1, Figure 2.2 demonstrates how Jiang's method would be used to find the best and the worst scheduling assignment. In Table 2.1, the value on the intersection of row i and column j indicates the performance degradation that application i experiences when co-scheduled with application j. In Figure 2.2, edge weights show the sum of mutual co-run degradations of the corresponding nodes. For example, the weight of 90.4% on the edge between MCF and MILC is the sum of 65.63% (the degradation of MCF when co-scheduled with MILC) and 24.75% (the degradation of MILC co-scheduled with MCF).

Table 2.1: Co-run degradations of four obtained on a real system. Small negative degradations for some benchmarks occur as a result of sharing of certain libraries. The value on the intersection of row i and column j indicates the performance degradation that application i experiences when co-scheduled with application j.

	mcf	milc	gamess	namd
mcf	48.01%	65.63%	2.0%	2.11%
milc	24.75%	45.39%	1.23%	1.11%
gamess	2.67%	4.48%	-1.01%	-1.21%
namd	1.48%	3.45%	-1.19%	-0.93%

Although Jiang's methodology and the corresponding algorithms would be too expensive to use online (the complexity of the algorithm is polynomial in the number of threads on systems with two cores per shared cache and the problem is NP-complete on systems where the degree of sharing is larger), it is acceptable for offline evaluation of the quality of classification schemes.

Using Jiang's algorithm as the perfect policy implies that the classification schemes we are evaluating must be suitable for estimating co-run degradations. All of our chosen classification schemes answered this requirement: they can be used to estimate co-run degradations in absolute or

in relative terms.

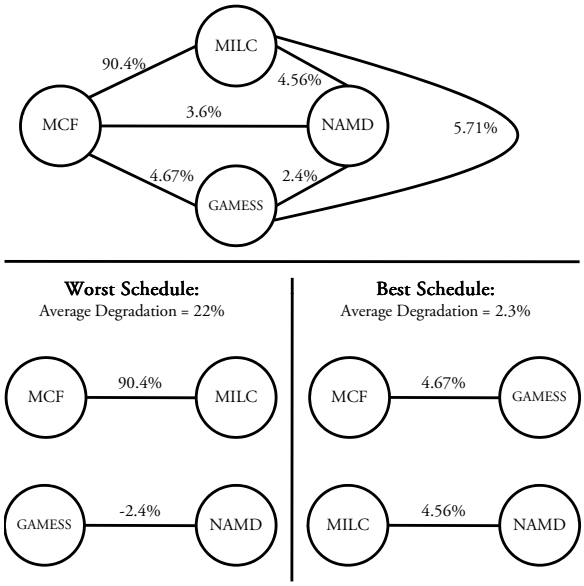


Figure 2.2: An overview of how to use Jiang's method for determining the optimal and the worst thread schedule. Edges connecting nodes are labeled with mutual co-run degradations, i.e., the sum of individual degradations for a given pair. The average degradation for a schedule is computed by summing up all mutual degradations and dividing by the total number of applications (four in our case).

An optimal classification scheme

To determine the quality of various classification schemes we not only need to compare them with each other, but also to evaluate how they measure up to the optimal classification scheme. All of our evaluated classification schemes attempt to approximate relative performance degradation that arbitrary tuples of threads experience when sharing a cache relative to running solo. An optimal classification scheme would therefore have the knowledge of *actual* such degradation, as measured on a real system. To obtain these measured degradations, we selected ten representative benchmarks from the SPEC CPU2006 benchmark suite (the methodology for selection is described later in this section), ran them solo on our experimental system (described in detail in Section 2.5), ran all possible pairs of these applications and recorded their performance degradation relative to solo performance. In order to make the analysis tractable it was performed based on pairwise degradations, assuming that only two threads may share a cache, but the resultant scheduling algorithms are

evaluated on systems with four cores per shared cache as well.

Evaluating classification schemes

To evaluate a classification scheme on a particular set of applications, we follow these steps:

1. Find the *optimal schedule* using Jiang's method and the optimal classification scheme, i.e., relying on measured degradations. Record its average performance degradation (see Figure 2.2).

2. Find the *estimated best schedule* using Jiang's method and the evaluated classification scheme, i.e., relying on estimated degradations. Record its average performance degradation.

3. Compute the difference between the degradation of the optimal schedule and of the estimated best schedule. The smaller the difference, the better the evaluated classification scheme.

To perform a rigorous evaluation, we construct a large number of workloads consisting of four, eight and ten applications. We evaluate all classification schemes using this method, and for each classification scheme report the average degradation above the optimal scheme across all workloads.

Benchmarks and workloads

We selected ten benchmarks from the SPEC2006 benchmark suite to represent a wide range of cache access behaviors. The cache miss rates and access rates for every application in the SPEC2006 benchmark suite were obtained from a third party characterization report [101] and a clustering technique was employed to select the ten representative applications.

From these ten applications we constructed workloads for a four-core, six-core, eight-core, and ten-core processor with two cores per LLC. With the ten benchmarks we selected, there are 210 unique four-application workloads, 210 unique six-application workloads, 45 unique eight-application workloads, and 1 unique ten-application workload to be constructed on each system. There are three unique ways to schedule a four-application workload on a machine with four cores and two shared caches. The number of unique schedules grows to 15, 105, and 945 for the six, eight, and ten-core systems respectively. Using Jiang's methodology, as opposed to running all these 9450 schedules, saves a considerable amount of time and actually makes it feasible to evaluate such a large number of workloads.

2.2.2 The Classification Schemes

For any classification scheme to work it must first obtain some "raw" data about the applications it will classify. This raw data may be obtained online via performance counters, embedded into an application's binary as a signature, or furnished by the compiler. Where this data comes from has a lot to do with what kind of data is required by the classification scheme. The SDC algorithm proposed by Chandra et al. [64] is one of the best known methods for determining how threads will interact with each other when sharing the same cache. The SDC algorithm requires the memory reuse patterns of applications, known as stack distance profiles, as input. Likewise all but one of our classification schemes require stack distance profiles. The one exception is the Miss Rate classification scheme which requires only miss rates as input. The simplicity of the Miss Rate Scheme allowed us to adapt it to gather the miss rates dynamically online making it a far more attractive option than the other classification schemes. However, in order to understand why such a simple classification scheme is so effective as well as to evaluate it against more complex and better established classification schemes we need to explore a wide variety of classification schemes and we need to use stack distance profiles to do so.

A stack distance profile is a compact summary of the application's cache-line reuse patterns. It is obtained by monitoring (or simulating) cache accesses on a system with an LRU cache. A stack distance profile is a histogram with a "bucket" or a position corresponding to each LRU stack position (the total number of positions is equal to the number of cache ways) plus an additional position for recording cache misses. Each position in the stack-distance profile counts the number of hits to the lines in the corresponding LRU stack position. For example, whenever an application reuses a cache line that is at the top of the LRU stack, the number of "hits" in the first position of the stack-distance profile is incremented. If an application experiences a cache miss, the number of items in the miss position is incremented. The shape of the stack-distance profile captures the nature of the application's cache behavior: an application with a large number of hits in top LRU positions has a good locality of reference. An application with a low number of hits in top positions and/or a large number of misses has a poor locality of reference. For our study we obtained the stack-distance profiles using the Pin binary instrumentation tool [132]; an initial profiling run of an application under Pin was required for that. If stack-distance profiles were to be used online in a live scheduler, they could be approximated online using hardware performance counters [169].

We now discuss four classification schemes which are based on the information provided in the stack distance profiles.

SDC

The SDC[1] classification scheme was the first that we evaluated, since this is a well known method for predicting the effects of cache contention among threads [64]. The idea behind the SDC method is to model how two applications compete for the LRU stack positions in the shared cache and estimate the extra misses incurred by each application as a result of this competition. The sum of the extra misses from the co-runners is the proxy for the performance degradation of this co-schedule.

The main idea of the SDC algorithm is in constructing a new stack distance profile that merges individual stack distance profiles of threads that run together. On initialization, each individual profile is assigned a current pointer that is initialized to point to the first stack distance position. Then the algorithm iterates over each position in the profile, determining which of the co-runners will be the "winner" for this stack-distance position. The co-runner with the highest number of hits in the current position is selected as the winner. The winner's counter is copied into the merged profile, and its current pointer is advanced. After the Ath iteration (A is the associativity of the LLC), *the effective cache space for each thread is computed proportionally to the number of its stack distance counters that are included in the merged profile*. Then, the cache miss rate with the new effective cache space is estimated for each co-runner, and the degradation of these miss rates relative to solo miss rates is used as a proxy for the co-run degradation. Note that miss rate degradations do not approximate absolute performance degradations, but they provide an approximation of relative performance in different schedules.

Animal Classes

This classification scheme is based on the animalistic classification of applications introduced by Xie et al. [187]. It allows classifying applications in terms of their influence on each other when co-scheduled in the same shared cache. Each application can belong to one of the four different classes: *turtle* (low use of the shared cache), *sheep* (low miss rate, insensitive to the number of cache ways allocated to it), *rabbit* (low miss rate, sensitive to the number of allocated cache ways) and *devil* (high miss rate, tends to thrash the cache thus hurting co-scheduled applications).

We attempted to use this classification scheme to predict contention among applications of different classes, but found an important shortcoming of the original animalistic model. The authors of

[1] Chandra suggested three algorithms for calculating the extra miss rates. However, only two of them (FOA and SDC) are computationally fast enough to be used in the robust scheduling algorithm. We chose SDC as it is slightly more efficient than FOA.

the animalistic classification proposed that *devils* (applications with a high miss rate but a low rate of reuse of cached data) must be insensitive to contention for shared resources. On the contrary, we found this not to be the case. According to our experiments, *devils* were some of the most *sensitive* applications – i.e., their performance degraded the most when they shared the on-chip resources with other applications. Since devils have a high miss rate they issue a large number of memory and prefetch requests. Therefore, they compete for shared resources other than cache: memory controller, memory bus, and prefetching hardware. As will be shown in Section 2.3, contention for these resources dominates performance, and that is why *devils* turn out to be sensitive.

To use the animalistic classification scheme for finding the optimal schedule as well as to account for our findings about "sensitive" *devils* we use a *symbiosis table* to approximate relative performance degradations for applications that fall within different animal classes. The symbiosis table provides estimates of how well various classes co-exist with each other on the same shared cache. For example, the highest estimated degradation (with the experimentally chosen value of 8) will be for two sensitive devils co-scheduled in the same shared cache, because the high miss rate of one of them will hurt the performance of the other one. Two turtles, on the other hand, will not suffer at all. Hence, their mutual degradation is estimated as 0. All other class combinations have their estimates in the interval between 0 and 8.

The information for classification of applications, as described by Xie et al. [187], is obtained from stack-distance profiles.

Miss Rate

Our findings about "sensitive" devils caused us to consider the miss rate as the heuristic for contention. Although another group of researchers previously proposed a contention-aware scheduling algorithm based on miss rates [116], the hypothesis that the miss rate should explain contention contradicted the models based on stack-distance profiles, which emphasized cache reuse patterns, and thus it needed a thorough validation. We define the miss rate to include all cache line requests issued by the LLC to DRAM. On our Intel Xeon system with L2 as the LLC, this quantity can be measured using the L2_LINES_IN hardware counter.

We hypothesized that identifying applications with high miss rates is beneficial for the scheduler, because these applications exacerbate the performance degradation due to memory controller contention, memory bus contention, and prefetching hardware contention. To attempt an approximation of the "best" schedule using the miss rate heuristic, the scheduler will identify high miss rate applications and separate them into different caches, such that no one cache will have a much higher

total miss rate than any other cache. Since no cache will experience a significantly higher miss rate than any other cache the performance degradation factors will be stressed evenly throughout the system.

In addition to evaluating a metric based on miss rates, we also experimented with other metrics, which can be obtained online using hardware counters, such as cache access rate and IPC. Miss rate, however, turned out to perform the best among them.

Pain

The Pain Classification Scheme is based on two new concepts that we introduce in this work: *cache sensitivity* and *cache intensity*. Sensitivity is a measure of how much an application will suffer when cache space is taken away from it due to contention. Intensity is a measure of how much an application will hurt others by taking away their space in a shared cache. By combining the sensitivity and intensity of two applications, we estimate the "pain" of the given co-schedule. Combining a sensitive application with an intensive co-runner should result in a high level of pain, and combining an insensitive application with any type of co-runner should result in a low level of pain. We obtain sensitivity and intensity from stack distance profiles and we then combine them to measure the resulting pain.

To calculate sensitivity S, we examine the number of cache hits that will most likely turn into misses when the cache is shared. To that end, we assign to the positions in the stack-distance profile *loss probabilities* describing the likelihood that the hits will be lost from each position. Intuitively hits to the Most Recently Used (MRU) position are less likely to become misses than hits to the LRU position when the cache is shared. Entries that are accessed less frequently are more likely to be evicted as the other thread brings its data into the cache; thus we scale the number of hits in each position by the corresponding probability and add them up to obtain the likely extra misses. The resulting measure is the sensitivity value which is shown in equation (Eq. 2.1). Here $h(i)$ is the number of hits to the i-th position in the stack, where $i = 1$ is the MRU and $i = n$ is the LRU for an n-way set associative cache. We use a linear loss probability distribution. As such the probability of a hit in the i-th position becoming a miss is $\frac{i}{n+1}$.

$$S = (\frac{1}{1+n}) \sum_{i=1}^{n} i * h(i) \tag{2.1}$$

Intensity Z is a measure of how aggressively an application uses the cache. As such, it approximates how much space the application will take away from its co-runner(s). Our approach to

measuring intensity is to use the number of last-level cache accesses per one million instructions.

We combine sensitivity S and intensity Z into the Pain metric, which is then used to approximate the co-run degradations required by our evaluation methodology. Suppose we have applications A and B sharing the same cache. Then the Pain of A due to B approximates the relative performance degradation that A is expected to experience due to B and is calculated as the intensity of B multiplied by the sensitivity of A (Eq. 2.2). The degradation of co-scheduling A and B together is the sum of the Pain of A due to B and the Pain of B due to A (Eq. 2.3).

$$Pain(A_B) = S(A) * Z(B) \tag{2.2}$$

$$Pain(A, B) = Pain(A_B) + Pain(B_A) \tag{2.3}$$

Classification Schemes Evaluation

For the purposes of this work we collected stack distances profiles offline using Intel's binary instrumentation tool Pin [101], an add-on module to Pin MICA [163], and our own module extending the functionality of MICA. The stack distance profiles were converted into the four classification schemes described above: SDC, Pain, Miss rates, and Animal. We estimate the extra degradation above the optimal schedule that each classification scheme produces for the four-core, six-core, eight-core and ten-core systems. Additionally, we present the degradations for the worst and random schedules. A random schedule picks each of the possible assignment for a workload with equal probability.

Figure 2.3 shows the results of the evaluation. Lower numbers are better. The Pain, Miss Rate and Animal schemes performed relatively well, but SDC surprisingly did only slightly better than random. Pain performed the best, delivering only 1% worse performance than the optimal classification scheme for all the systems. Interestingly we see that all classification schemes except Pain and Animal do worse as the number of cores in the system grows. In systems with more cores the number of possible schedules grows, and so imperfect classification schemes are less able to make a *lucky* choice.

The Animal scheme did worse than Pain. Animal classes are a rather rough estimation of relative co-run degradations (a lot of programs will fall to the same class), and so the Animal scheme simply cannot achieve the same precision as Pain which takes into account absolute values. The Miss Rate

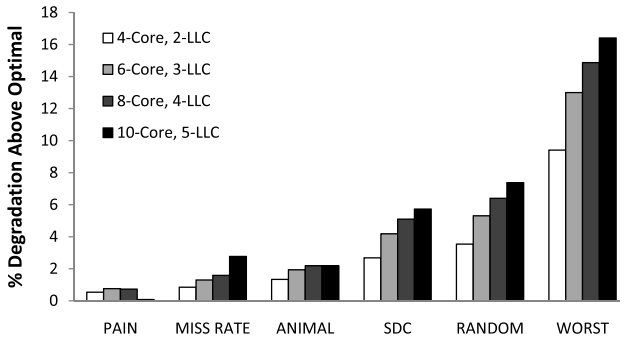


Figure 2.3: Degradation relative to optimal experienced by each classification scheme on systems with different numbers of cores (2 cores per LLC).

scheme performs almost as well as Pain and Animal scheme and yet is by far the easiest to compute either online or offline.

SDC performed worse than Pain and Animal for the following reasons. The first reason is that SDC does not take into account miss rates in its stack distance competition model. So it only works well in those scenarios where the co-running threads have roughly equal miss rates (this observation is made by the authors themselves [64]). When the miss rates of co-running threads are very different, the thread with the higher miss rate will "win" more cache real estate – this fact is not accounted for by the SDC model. Authors of SDC [64] offer a more advanced (but at the same time computationally more expensive) model for predicting extra miss rates which takes into account different miss rates of co-running applications. We did not consider this model in the current work, because we deemed it too computationally expensive to use in an online scheduler.

The second reason has to do with the fact that SDC models the performance effects of cache contention, but as the next section shows, this is not the dominant cause for performance degradation and so other factors must be considered as well. We were initially surprised to find that SDC, a model extensively validated in the past, failed to outperform even such a coarse classification heuristic as the miss rate. In addition to SDC, many other studies of cache contention used stack-distance or reuse-distance profiles for managing contention [64, 155, 165, 169]. The theory behind stack-distance based models seems to suggest that the miss rate should be a *poor* heuristic for predicting contention, since applications with a high cache miss rate may actually have a very poor reuse of their cached data, and so they would be indifferent to contention. Our analysis, however, showed the opposite: miss rate turned out to be an excellent heuristic for contention.

We discovered that the reason for these seemingly unintuitive results had to do with the causes of performance degradation on multicore systems. SDC, and other solutions relying on stack distance profiles such as cache partitioning [165, 155, 187], assumed that the dominant cause of performance degradation is contention for the space in the shared cache, i.e., when co-scheduled threads evict each other data from the shared cache. We found, however, that cache contention is by far not the dominant cause of performance degradation. Other factors, such as contention for memory controllers, memory bus, and resources involved in prefetching, dominate performance degradation for most applications. A high miss rate exacerbates the contention for all of these resources, since a high-miss-rate application will issue a large number of requests to a memory controller and the memory bus, and will also be typically characterized by a large number of prefetch requests.

In the next section we attempt to quantify the causes for performance degradation resulting from multiple factors, showing that contention for cache space is not dominant; these results provide the explanation why a simple heuristic such as the miss rate turns out to be such a good predictor for contention.

2.3 Factors Causing Performance Degradation on multicore systems

Recent work on the topic of performance degradation in multicore systems focused on contention for cache space and the resulting data evictions when applications share the LLC. However, it is well known that cache contention is far from being the only factor that contributes to performance degradation when threads share an LLC. Sharing of other resources, such as the memory bus, memory controllers and prefetching hardware also plays an important role. Through extensive analysis of data collected on real hardware we have determined that contention for space in the shared cache explains only a part of the performance degradation when applications share an LLC. In this section we attempt to quantify how much performance degradation can be attributed to contention for each shared resource assuming a system depicted on Figure 2.4 (eight Intel Xeon X5365 cores running at 3GHz, and 8GB of RAM).

Estimating the contribution that each factor has on the overall performance degradation is difficult, since all the degradation factors work in conjunction with each other in complicated and practically inseparable ways. Nevertheless, we desired a rough estimate of the degree to which each factor affects overall performance degradation to identify if any factor in particular should be the focus of our attention since mitigating it will yield the greatest improvements.

We now describe the process we used to estimate the contributions of each factor to the overall

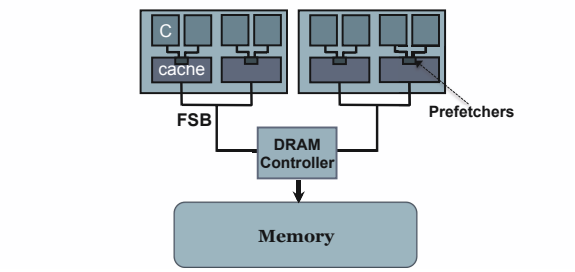


Figure 2.4: A schematic view of a system assumed in this section. Each of 4 chips has 2 cores sharing a LLC cache. Chips are grouped into 2 sockets (physical packages). All cores are equidistant to the main memory.

degradation. Our experimental system is a two-sockets server with two Intel X5365 "Clovertown" quad-core processors. The two sockets share the memory controller hub, which includes the *DRAM controller*. On each socket there are four cores sharing a *front-side bus (FSB)*. There are two L2 caches on each socket, one per pair of cores. Each pair of cores also shares prefetching hardware, as described below. So when two threads run on different sockets, they compete for the DRAM controller. When they run on the same socket, but on different caches, they compete for the FSB, in addition to the DRAM controller. Finally, when they run on cores sharing the same cache, they also compete for the L2 cache and the prefetching hardware, in addition to the FSB and the DRAM controller. To estimate how contention for each of these resources contributes to the total degradation, we measured the execution times of several benchmarks under the following eight conditions:

Solo_PF_ON: Running SOLO and prefetching is ENABLED

Solo_PF_OFF: Running SOLO and prefetching is DISABLED

SameCache_PF_ON: Sharing the LLC with an interfering benchmark and prefetching is ENABLED

SameCache_PF_OFF: Sharing the LLC with an interfering benchmark and prefetching is DISABLED

DiffCache_PF_ON: An interfering benchmark runs on a different LLC but on the same socket and prefetching is ENABLED

DiffCache_PF_OFF: An interfering benchmark runs on a different LLC but on the same socket and prefetching is DISABLED

DiffSocket_PF_ON: An interfering benchmark runs on a different socket and prefetching is ENABLED

DiffSocket_PF_OFF: An interfering benchmark runs on a
different socket and prefetching is DISABLED

As an interfering benchmark for this experiment we used MILC. MILC was chosen for several reasons. First, it has a very high solo miss rate which allows us to estimate one of the worst-case contention scenarios. Second, MILC suffers a negligible increase in its own miss rate due to cache contention (we determined this via experiments and also by tracing MILC's memory reuse patterns, which showed that MILC hardly ever reuses its cached data) and hence will not introduce extra misses of its own when co-run with other applications. We refer to MILC as the *interfering benchmark* and we refer to the test application simply as the *application*. In the experiment where MILC is the test application, SPHINX is used as the interfering benchmark.

Estimating Performance Degradation due to DRAM Controller Contention We look at the difference between the solo run and the run when the interfering benchmark is on a different socket. When the interfering benchmark is on a different socket any performance degradation it causes can only be due to DRAM controller contention since no other resources are shared. Equation 2.4 shows how we estimate the performance degradation due to DRAM controller contention.

$$DRAM\_contention = \frac{DiffSocket\_PF\_OFF - Solo\_PF\_OFF}{Solo\_PF\_OFF} \tag{2.4}$$

There are several complications with this approach, which make it a rough estimate as opposed to an accurate measure of DRAM controller contention. First, when the LLC is shared by two applications, extra evictions from cache cause the total number of misses to go up. These extra misses contribute to the DRAM controller contention. In our experimental technique the two applications are in different LLCs and hence there are no extra misses. As a result, we are underestimating the DRAM controller contention. Second, we chose to disable prefetching for this experiment. If we enabled prefetching and put two applications into different LLC then they would each have access to a complete set of prefetching hardware. This would have greatly increased the total number of requests issued to the memory system from the prefetching hardware as compared to the number of requests that can be issued from only one LLC. By disabling the prefetching we are once again underestimating the DRAM controller contention. As such the values that we measure should be considered a lower bound on DRAM controller contention.

Estimating Performance Degradation due to FSB Contention Next, we estimate the degree of performance degradation due to contention for the FSB. To that end, we run the application and the

interfering benchmark on the same socket, but on different LLCs. This is done with prefetching disabled, so as not to increase the bus traffic. Equation 2.5 shows how we estimate the degradation due to FSB contention.

$$FSB\_Contention = \frac{DiffCache\_PF\_OFF - DiffSocket\_PF\_OFF}{Solo\_PF\_OFF} \tag{2.5}$$

Estimating Performance Degradation due to Cache Contention To estimate the performance degradation due to cache contention we take the execution time when an application is run with an interfering co-runner in the same LLC and subtract from it the execution time of the application running with the interfering benchmark in a different LLC of the same socket. This is done with prefetching disabled so as not to increase bus traffic or contend for prefetching hardware. The difference in the execution times between the two runs can be attributed to the extra misses that resulted due to cache contention. Equation 2.6 demonstrates how we estimate performance degradation due to cache contention.

$$Cache\_Contention = \frac{SameCache\_PF\_OFF - DiffCache\_PF\_OFF}{Solo\_PF\_OFF} \tag{2.6}$$

Estimating Performance Degradation due to Contention for Resources Involved in Prefetching Contention for resources involved in prefetching has received less attention in literature than contention for other resources. We were compelled to investigate this type of contention when we observed that some applications experienced a decreased prefetching rate (up to 30%) when sharing an LLC with a memory-intensive co-runner. Broadly speaking, prefetching resources include all the hardware that might contribute to the speed and quality of prefetching. For example, our experimental processor has two types of hardware that prefetches into the L2 cache. The first is the Data Prefetching Logic (DPL) that is activated when an application has two consecutive misses in the LLC and a stride pattern is detected. In this case, the rest of the addresses up to the page boundary are prefetched. The second is the adjacent cache line prefetcher, or the streaming prefetcher. The L2 prefetching hardware is dynamically shared by the two cores using the LLC. The memory controller and the FSB are also involved in prefetching, since they determine how aggressively these requests can be issued to memory. It is difficult to tease apart the latencies attributable to contention for each resource, so our estimation of contention for prefetching resources includes contention for prefetching hardware as well as additional contention for these two other resources. This is an upper bound on the contention for the prefetching hardware itself.

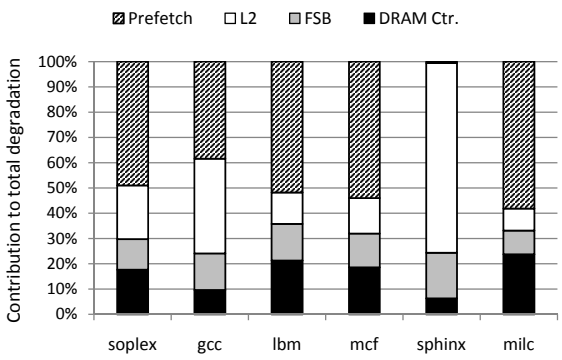


Figure 2.5: Percent contribution that each of the factors have on the total degradation on a UMA multicore system.

We can measure the performance degradation due to prefetching-related resources as the difference between the total degradation and the degradation caused by cache contention, FSB, and DRAM controller contention. Equation 2.7 calculates the total degradation of an application when the LLC is shared by looking at the difference when the interfering benchmark shares the LLC and when the application runs alone. Equation 2.8 shows the calculation of the prefetching degradation.

$$Total\_Degradation = \frac{SameCache\_PF\_ON - Solo\_PF\_ON}{Solo\_PF\_ON} \tag{2.7}$$

$$Prefetching\_Contention = Eq.(2.7) - Eq.(2.6) - Eq.(2.5) - Eq.(2.4) \tag{2.8}$$

Finally, we calculate the degradation contribution of each factor as the ratio of its degradation compared to the total degradation. Figure 2.5 shows the percent contribution of each factor (DRAM controller contention, FSB contention, L2 cache contention, and prefetching resource contention) to the total degradation for six SPEC2006 benchmarks.

The six applications shown in Figure 2.5 are the applications that experience a performance degradation of at least 45% chosen from the ten representative benchmarks. We see from Figure 2.5 that for all applications except SPHINX contention for resources *other than shared cache* is the dominant factor in performance degradation, accounting for more than 50% of the total degradation.

In order to understand whether similar conclusions can be made with respect to other systems, we also performed a similar analysis of degradation-contributing factors on a system that is very different from the Intel system examined so far. We used an AMD Opteron 2350 (Barcelona) system.

The AMD machine has four cores per processor, each core with private L1 instruction and data caches and a private unified L2 cache. There is an 2MB L3 cache shared by all four cores on the chip.

Since there are important differences between the Intel and AMD systems, the degradation-contributing factors are also somewhat different. For instance, while the Intel system is UMA (UMA stands for Uniform Memory Access), the AMD system is NUMA (Non-Uniform Memory Access). This means that each processor on an AMD system has its own DRAM controller and a local memory node. While any core can access both local and remote nodes (attached to other processors), local accesses take less time. Remote accesses take longer and require using the DRAM controller of the remote processor. Because of these differences, there are two additional degradation-contributing factors on this AMD system that were not present on the Intel system: (1) interconnect contention (IC), which would occur if a thread is accessing remote memory and competes for inter-processor interconnects with other threads, and (2) remote latency overhead (RL), which occurs if a thread is accessing remote memory and experiences longer wire delays[2]. Furthermore, prefetching works differently on the AMD system, so we were unable to isolate its effects in the same way that we were able to do on the Intel system. In summary, we identified four performance-degrading factors on the NUMA AMD system that a thread can experience relative to the scenario when it runs on the system alone and accesses the memory in a local node: (1) L3 cache contention, (2) DRAM controller contention, (3) interconnect contention and (4) remote latency. Experiments similar to those that we designed on the Intel system were used to estimate the effect of these factors on performance. One difference is that we used different benchmarks in order to capture the representative trends in degradation breakdown. We provide the detailed explanation of our methodology for estimating the breakdown of degradation factors on the AMD system in the Appendix.

Figure 2.6 shows the contribution of each factor to the performance degradation of the benchmarks as they are co-scheduled with three instances of MILC. Although this system is different from the Intel system examined earlier, the data allows us to reach similar conclusions: shared cache contention (L3) is *not* the dominant cause for performance degradation. The dominant causes are the DRAM controller and interconnect contention.

As in case with the UMA system, this is only an approximation, since contention causing factors on a real system overlap in complex and integrated ways. For example, increasing cache contention

[2]Note that under our definition, RL does not include IC.

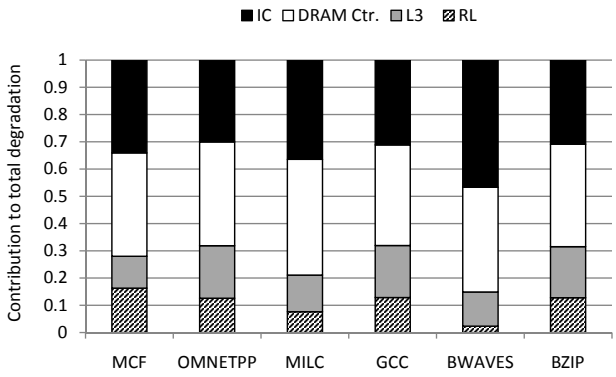


Figure 2.6: Contribution of each factor to the worst-case performance degradation on a NUMA multicore system.

increases other types of contention: if two threads share a cache and have memory access patterns that result in a lot of extra cache misses, that would stress all the memory hierarchy levels placed on the path between the shared cache and main memory of the machine. The results provided are an approximation that is intended to direct attention to the true bottlenecks in the system.

2.3.1 Discussion of Performance-Degrading Factors Breakdown

While cache contention does have an effect on performance degradation, any scheduling strategy that caters to reducing cache contention exclusively cannot and will not have a major impact on performance. The fact that contention for resources other than cache is dominant, explains why the miss rate turns out to be such a good heuristic for predicting contention. The miss rate, which we define to include all DRAM-to-LLC transfers, highly correlates with the amount of DRAM controller, FSB, and prefetch requests as well as the degree of interconnect usage and the sensitivity to remote access latency, and thus is indicative of both the sensitivity of an application as well as its intensity.

2.4 Scheduling Algorithms

A scheduling algorithm is the combination of a classification scheme and a scheduling policy. We considered and evaluated several scheduling algorithms that combined different classification schemes and policies, and in this section we present those that showed the best performance and were also the simplest to implement. In particular, we evaluate two algorithms based on the Miss

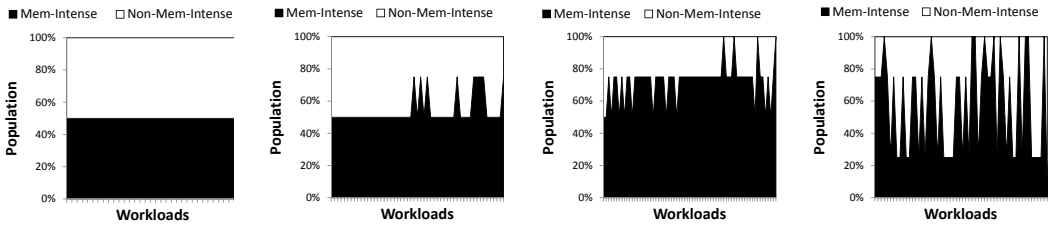
Rate classification schemes, because the miss rate is very easy to obtain online via hardware performance counters. The scheduling policy we used was the *Centralized Sort*. It examines the list of applications, sorted by their miss rates, and distributes them across cores, such that the total miss rate of all threads sharing a cache is equalized across all caches. For the evaluation results of other heuristics and policies, we refer the reader to our technical report [54].

While the Pain classification scheme gave the best performance, we chose not to use it in an online algorithm, instead opting to implement one using the miss rate heuristic. This was done to make the scheduler simpler, thus making more likely that it will be adopted in general-purpose operating systems. Using Pain would require more changes to the operating system than using the miss rate for the following reason. Pain requires stack distance profiles. Obtaining a stack distance profile online requires periodic sampling of data addresses associated with last-level cache accesses using advanced capabilities of hardware performance monitoring counters, as in RapidMRC [169]. Although RapidMRC can generate accurate stack-distance profiles online with low overhead, there is certain complexity associated with its implementation. If a scheduler uses the miss rate heuristic, all it has to do is periodically measure the miss rates of the running threads, which is simpler than collecting stack-distance profiles. Given that the miss rate heuristic had a much lower implementation complexity but almost the same performance as Pain, we thought it would be the preferred choice in future OS schedulers.

In the rest of this section we describe two algorithms based on the Miss Rate classification scheme: *Distributed Intensity* (DI) and *Distributed Intensity Online* (DIO). DIO does not rely on stack-distance profiles, but on the miss rates measured online. DI estimates the miss rate based on the stack-distance profiles; it was evaluated in order to determine if any accuracy is lost when the miss rates are measured online as opposed to estimated from the profiles.

2.4.1 Distributed Intensity (DI)

In the Distributed Intensity (DI) algorithm all threads are assigned a value which is their solo miss rate (misses per one million instructions) as determined from the stack distance profile. The goal is then to spread the threads across the system such that the miss rates are distributed as evenly as possible. The idea is that the performance degradation factors identified in Section 2.3 are all exacerbated by a high miss rate and so we avoid the situation where the concentration of threads with high cumulative miss rate results in increased contention for shared memory resources with the consequent performance bottleneck on the system.



(a) The makeup of workloads which deliver speedups from 25%-15%.

(b) The makeup of workloads which deliver speedups from 15%-10%.

(c) The makeup of workloads which deliver speedups from 10%-5%.

(d) The makeup of workloads which deliver speedups from 5%-0%.

Figure 2.7: The makeup of workloads

The algorithm uses the notion of *memory hierarchy entities* when trying to spread the workload's intensiveness across the system. Memory hierarchy entities are distinct hardware modules (e.g., cores, chips, packages) each of which is located on its own *level of memory hierarchy*. The entities on the upper level are grouped into the larger entities of the lower level which are thus called *containers*. For example, several cores are organized into a chip (chip serves as a container for cores), whereas several chips form a physical package (the package is a container for chips).

The presence of the requested data is first checked against the very fast and small caches that are located close to the core itself. These caches are typically devoted exclusively to its closest core (and are called *private* because of that), hence they can be considered the same entity with the core. If the data is missing in the private caches, it is then looked for in the larger *on-chip cache* shared between all cores on the chip. The latency of servicing the request from the shared on-chip cache is higher than from the private caches. However, the shared cache is still significantly faster than the main memory. If the data is absent in the shared cache, the request goes down the memory hierarchy into the *Front Side Bus* that is shared between several chips on the same physical package (Figure 2.4). There could be many physical packages on the system. Each of the packages in this case will have its own FSB shared between all the chips belonging to this package. The requests from each of the on-package buses then compete for the access of the *DRAM controller*. On the UMA system there is only one such controller per machine. However, the NUMA systems have several DRAM controllers, one per every memory node on the system.

For example, on our testing machines equipped with Intel Xeon and AMD Opteron processors (described in Sections 2.3 and 2.5.1) the entities on each level of the memory hierarchy are defined as shown in Table 2.2. During the initialization stage (at boot), the system determines the number of memory hierarchy levels and the number of distinct entities on each level (as is specified in the *proc*

pseudo filesystem at /sys/devices/system/cpu). DI (described in blocks of pseudo code 1 and 2) then tries to even out the miss rate on all levels of memory hierarchy.

level of memory hierarchy	**type of entity on this level**	**entities per: machine / container**	
		Intel Xeon	**AMD Barcelona**
3	core	8/2	8/4
2	chip	4/2	2/1
1	physical package	2/2	
0	machine	1/-	

Table 2.2: Entities on each level of memory hierarchy.

Algorithm 1 Invocation at the beginning of every scheduling interval

1: *// initialization of the boolean global array Order[l], every member specifies the order of browsing entities on the corresponding level of the memory hierarchy. Each entity has its ID that was given to it by OS. Order[l]=0 means that entities are browsed from the smallest ID to the biggest, Order[l]=1 – vice versa.*
2: **for** $l = 0; l < number\_of\_memory\_hierarchy\_levels >; l++$ **do**
3: $Order[l]$:=0;
4: **end for**
5: sort the threads according to the *miss rate* in descending order
6: let T be the array of sorted threads
7: *// spread the threads across the machine*
8: **while** $T \neq \emptyset$ **do**
9: take the first (the most aggressive) $t \in T$
10: *// invoke DI() to assign the thread*
11: DI(t,$< machine >$, 0)
12: **end while**

To further justify the validity of this method we performed a study on all possible 4-thread workloads that can be constructed from the 10 representative SPEC2006 benchmarks. We computed the percent difference between average co-run degradations for each workload achieved with the optimal solution relative to the worst solution (we refer to this value as the speedup). The highest average speedup relative to the worst schedule was 25%, and the lowest was less than 1%. We broke up the workloads based on the speedup range into which they fit: (25%-15%), (15%-10%), (10%-5%), and (5%-0%), and studied which types of applications are present in each range. For simplicity we define two categories of applications: intensive (above average miss rate), and non-intensive (below average miss rate).

Algorithm 2 DI(thread to schedule t, container e_parent, memory hierarchy level of the container l)

1: Let E_{all} be the array of entities on hierarchy level $l + 1$
2: Let $E_{children}$ be the array of entities on hierarchy level $l + 1$ whose container is e_parent
3: browse the entities in $E_{children}$ in order $Order[l + 1]$ and determine the first entity with the minimum number of allocated threads: $e\_min \in E_{children}$
4: **if** e_min is a $< core >$ **then**
5: *// we have reached the bottom of the memory hierarchy*
6: assign thread t to core e_min
7: **else**
8: increment the number of threads allocated to e_min
9: *// recursively invoke DI() to assign the thread on the lower hierarchy level*
10: DI(t,e_min,$l + 1$)
11: **end if**
12: **if** number of threads allocated to each entity $e \in E_{all}$ is the same **then**
13: *// reverse the order of browsing on this level*
14: $Order[l + 1]$:= NOT $Order[l + 1]$
15: **end if**

We note that according to the ideas of DI, workloads consisting of two intensive and two non-intensive applications should achieve the highest speedups. This is because in the worst case these workloads can be scheduled in such a way as to put both intensive applications in the same cache, creating a cache with a large miss rate, and a performance bottleneck. Alternatively, in the best case these workloads can be scheduled to spread the two intensive applications across caches thus minimizing the miss rate from each cache and avoiding bottlenecks. The difference in performance between these two cases (the one with a bottleneck and the one without) should account for the significant speedup of the workload. We further note that workloads consisting of more than two or fewer than two intensive applications can also benefit from distribution of miss rates but the speedup will be smaller, since the various scheduling solutions do not offer such a stark contrast between creating a major bottleneck and almost entirely eliminating it.

Figure 2.7 shows the makeup of workloads (intensive vs. non-intensive) applications and the range of speedups they offer. The (unlabeled) x-axis identifies all the workloads falling into the given speedup range. We see that the distribution of applications validates the claims of DI. The other claim that we make to justify why DI should work is that miss rates of applications are relatively stable. What we mean by stability is that when an application shares the LLC with a co-runner its miss rate will not increase so dramatically as to make the solution found by DI invalid.

DI assigns threads to caches to even out the miss rate across all the caches. This assignment

is done based on the solo miss rates of applications. The real miss rate of applications will change when they share a cache with a co-runner, but we claim that these changes will be relatively small such that the miss rates are still rather even across caches. Consider Figure 2.8, which shows the solo miss rates of the 10 SPEC2006 benchmarks as well as the largest miss rate observed for each application as it was co-scheduled to share a cache with all other applications in the set.

We see that if the applications were sorted based on their miss rates their order would be nearly identical if we used solo miss rates, maximum miss rates, or anything in between. Only the applications MILC and SOPLEX may exchange positions with each other or GCC and SPHINX may exchange positions depending on the miss rate used. The DI algorithm makes scheduling decisions based on the sorted order of applications. If the order of the sorted applications remains nearly unchanged as the miss rates changes then the solutions found by DI would also be very similar. Hence the solution found by DI with solo miss rates should also be very good if the miss rates change slightly. Through an extensive search of all the SPEC2006 benchmark suite and the PARSEC benchmark suite we have not found any applications whose miss rate change due to LLC contention would violate the claim made above.

The DI scheduler is implemented as a user level scheduler running on top of Linux. It enforces all scheduling decisions via system calls which allow it to bind threads to cores. The scheduler also has access to files containing the solo miss rates. For all the applications it uses solo miss rate estimated using stack distance profiles as the input to the classification scheme.

2.4.2 Distributed Intensity Online (DIO)

DIO is based on the same classification scheme and scheduling policies as DI except that it obtains the miss rates of applications dynamically online via performance counters. This makes DIO more attractive since the stack distance profiles, which require extra work to obtain online, are not required. The miss rate of applications can be obtained dynamically online on almost any machine with minimal effort. Furthermore, the dynamic nature of the obtained miss rates makes DIO more resilient to applications that have a change in the miss rate due to LLC contention. DIO continuously monitors the miss rates of applications and thus accounts for phase changes. To minimize migrations due to phase changes of applications we collect miss rate data not more frequently than once every billion cycles and we use a running average for scheduling decisions. Every billion cycles DIO measures the new miss rate and re-evaluates the thread assignments based on the updated miss rate running average values for the workload.

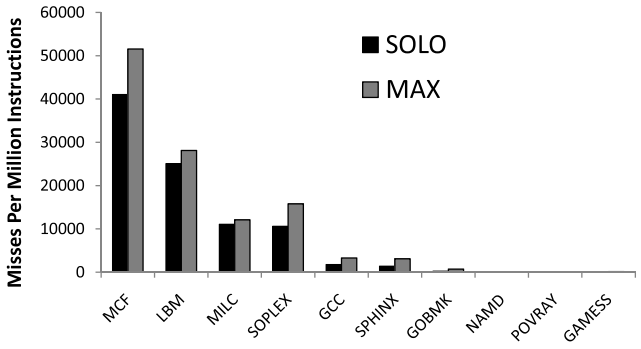


Figure 2.8: The solo and maximum miss rate recorded for each of the 10 SPEC2006 benchmarks.

The DIO scheduler, like DI, manages the assignment of threads to cores using affinity interfaces provided in Linux. As such, it mirrors the actions that would be performed by a kernel scheduler. The key difference is that the kernel scheduler would directly manipulate the runqueues in order to place a thread on a particular core, but a user-level prototype of the scheduler uses affinity-related system calls for that purpose. For example, to swap thread A on core i with thread B on core j we set affinity of A to j and affinity B to i. The kernel performs the actual swapping.

2.5 Evaluation on Real Systems

2.5.1 Evaluation Platform

We performed the experiments on two systems:

Dell-Poweredge-2950 (Intel Xeon X5365) has eight cores placed on four chips. Each chip has a 4MB 16-way L2 cache shared by its two cores. Each core also has private L1 instruction and data caches. In our first series of experiments we used only two chips out of four. This enabled us to verify our analytical results for the 4 thread workloads directly. After that, all eight cores with eight-thread workloads were used.

Dell-Poweredge-R805 (AMD Opteron 2350 Barcelona) has eight cores placed on two chips. Each chip has a 2MB 32-way L3 cache shared by its four cores. Each core also has a private unified L2 cache and private L1 instruction and data caches. All eight cores with eight thread workloads were used. Although this is a NUMA system, we did not study NUMA effects in detail and did not set up our experiments to explicitly stress NUMA effects. Therefore, addressing contention on NUMA systems must be investigated in future work.

The experimental workloads for Section 2.5.2 were comprised of the 14 scientific benchmarks from SPEC CPU 2006 suite chosen using the clustering technique as described in Section 2.2 (see Table 2.3). For the eight-core experiments we created eight-thread workloads by doubling the corresponding four-thread workloads. For example, for the four-thread workload (SOPLEX, SPHINX, GAMESS, NAMD) the corresponding eight-thread workload is (SOPLEX, SPHINX, GAMESS, NAMD, SOPLEX, SPHINX, GAMESS, NAMD). The user-level scheduler starts the applications and binds them to cores as directed by the scheduling algorithm.

We chose to use LAMP, a solution stack of open source software consisting of Linux, Apache, MySQL and PHP, as our testing environment in Section 2.5.3. LAMP is widely used in many areas where efficient storage and retrieving of data is required, e.g., website management and data mining.

Since we are focused on CPU-bound workloads, which are *not* likely to run with more threads than cores [45, 179], we only evaluate the scenarios where the number of threads does not exceed the number of cores. If the opposite were the case, the scheduler would simply re-evaluate the mapping of threads to cores every time the set of running threads changes. The decision of which thread is selected to run would be made in the same way as it is done by the default scheduler. While in this case there are also opportunities to separate competing threads *in time* as opposed to in space, we do not investigate these strategies in this work.

Both systems were running Linux Gentoo 2.6.27 release 8. We compare performance under DI and DIO to the default contention-unaware scheduler in Linux, referring to the latter as DEFAULT. The prefetching hardware is fully enabled during these experiments. To account for the varied execution times of benchmark we restart an application as soon as it terminates (to ensure that the same workload is running at all times). An experiment terminates when the longest application had executed three times.

2.5.2 Results for scientific workloads

Intel Xeon 4 cores We begin with the results for the four-thread workloads on the four-core configuration of the Intel Xeon machine. For every workload we first run the three possible unique schedules and measure the aggregate workload completion time of each. We then determine the schedule with the optimal (minimal) completion time, the worst possible schedule (maximum completion time) and the expected completion time of the random scheduling algorithm (it selects all schedules with equal probability). We then compared the aggregate execution times of DI and DIO with the completion times of OPTIMAL, WORST and RANDOM. We do not present results for

	Workloads			
	2 memory-bound, 2 CPU-bound			
1	**SOPLEX**	**SPHINX**	GAMESS	NAMD
2	**SOPLEX**	**MCF**	GAMESS	GOBMK
3	**MCF**	**LIBQUANTUM**	POVRAY	GAMESS
4	**MCF**	**OMNETPP**	H264	NAMD
5	**MILC**	**LIBQUANTUM**	POVRAY	PERL
	1 memory-bound, 3 CPU-bound			
6	**SPHINX**	GCC	NAMD	GAMESS
	3 memory-bound, 1 CPU-bound			
7	**LBM**	**MILC**	**SPHINX**	GOBMK
8	**LBM**	**MILC**	**MCF**	NAMD

Table 2.3: The workloads used for experiments (devils are highlighted in bold).

the default Linux scheduler because when the scheduler is given a processor affinity mask to use only four cores out of eight, the migrations of threads across cores become more frequent than when no mask is used, leading to results with an atypically high variance. (We do report results under default Linux when we present our experiments using all eight cores.) Figure 2.9 shows the performance degradation above the optimal for every workload with DI, DIO, RANDOM and WORST. The results show that DI and DIO perform better than RANDOM and are within 2% of OPTIMAL.

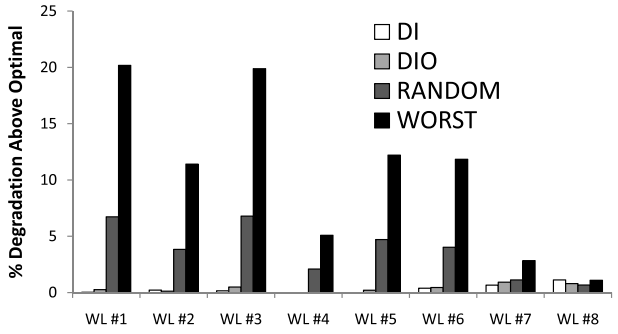


Figure 2.9: Aggregate performance degradation of each workload with DI, DIO, RANDOM and WORST relative to OPTIMAL (low bars are good) for the Intel machine and 4 threads.

Intel Xeon 8 cores Since this setup does not require a processor affinity mask, we evaluated the results of DI and DIO against DEFAULT as in this case DEFAULT does not perform excessive migrations. Figure 2.10 shows the percent aggregate workload speedup (the average of speedups of

all the programs in the workload) over DEFAULT.

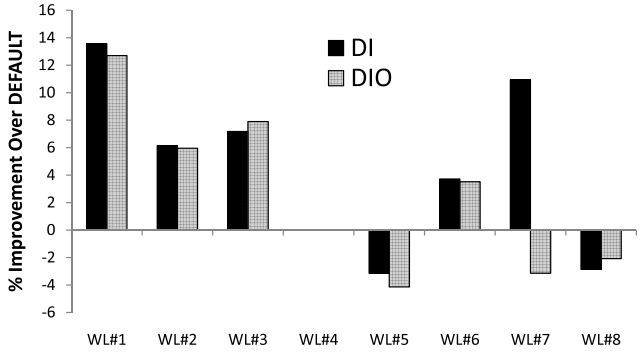


Figure 2.10: Aggregate performance improvement of each workload with DI and DIO relative to DEFAULT (high bars are good) on the Intel system.

We note that although generally DI and DIO improve aggregate performance over DEFAULT, in a few cases they performed slightly worse. However, the biggest advantage of DI and DIO is that they offer much more stable results from run to run and avoid the worst-case thread assignment. This effect is especially significant if we look at performance of individual applications. Figure 2.12(a) shows relative performance improvement for individual applications of the worst-case assignments of DI and DIO over the worst case assignments under DEFAULT. Higher numbers are better. Worst-case performance improvement is obtained by comparing the worst-case performance (across all the runs) under DI and DIO with the worst-case performance under DEFAULT. The results show that DEFAULT consistently stumbles on much worse solutions than DI or DIO and as such there are cases when the performance of individual applications is unpredictably bad under DEFAULT. What this means is that if an application was repeatedly executed on a multicore system under the default scheduler, it could occasionally slowdown by as much as 100%(!) relative to running solo. With DIO, on the other hand, the slowdown would be much smaller and more predictable.

Figure 2.12(b) shows the deviation of the execution time of consecutive runs of the same application in the same workload with DI, DIO and DEFAULT. We note that DEFAULT has a much higher deviation from run to run than DI and DIO. DIO has a slightly higher deviation than DI as it is sensitive to phase changes of applications and as a result tends to migrate applications more frequently.

AMD Opteron 8 cores Finally, we report the results for the same eight-thread workloads on the AMD system. The results for the percent aggregate workload speedup over DEFAULT (Figure 2.11), relative performance improvement of the worst case assignments over DEFAULT (Figure 2.13(a))

and the deviation of the execution times (Figure 2.13(b)) generally repeat the patterns observed on the Intel Xeon machine with eight threads.

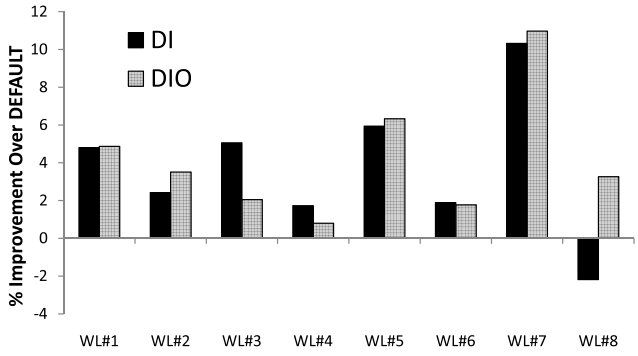
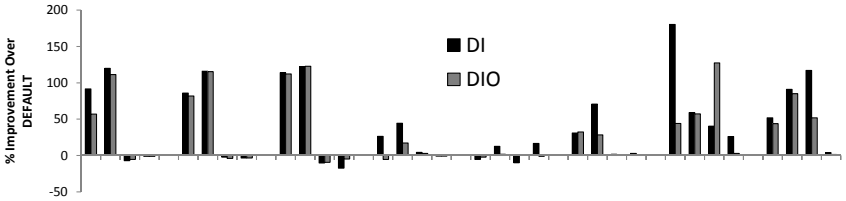
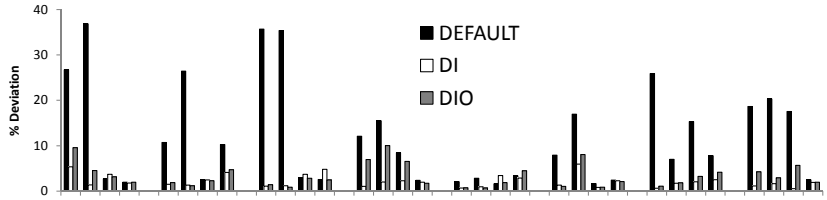


Figure 2.11: Aggregate performance improvement of each workload with DI and DIO relative to DEFAULT (high bars are good) on the AMD system.



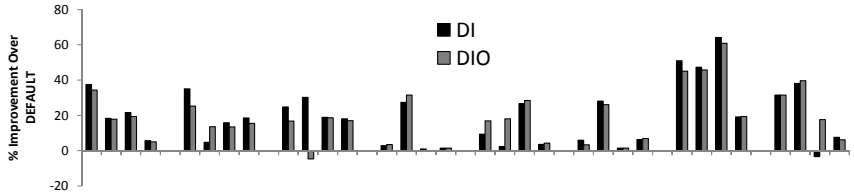
(a) Relative performance improvement of the worst case DI and DIO over the worst case DEFAULT on the Intel system (workload labels are provided in the next Figure).



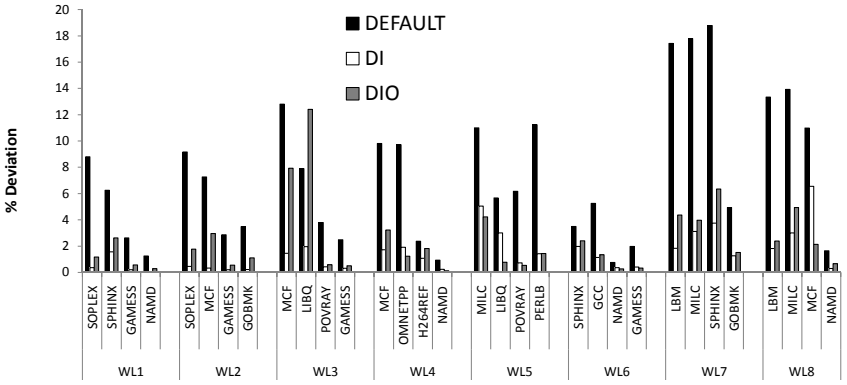
(b) Deviation with DI, DIO and Default (low bars are good) on the Intel system.

Figure 2.12: Relative performance improvement and deviation on the Intel system

We draw several conclusions from our results. First of all, the classification scheme based on miss rates effectively enables to reduce contention for shared resources with scheduling. Furthermore, an algorithm based on this classification scheme can be effectively implemented online as demonstrated by our DIO prototype. Using contention-aware scheduling can help improve overall system efficiency by reducing completion time for the entire workload as well as reduce worst-case



(a) Relative performance improvement of the worst case DI and DIO over the worst case DEFAULT on the AMD system.



(b) Deviation with DI, DIO and Default (low bars are good) on the AMD system.

Figure 2.13: Relative performance improvement and deviation on the AMD system

performance for individual applications. In the former case, DIO improves performance by up to 13% relative to DEFAULT and in the isolated cases where it does worse than DEFAULT, the impact on performance is at most 4%, far smaller than the corresponding benefit. On average, if we examine performance across all the workloads we have tried DEFAULT does rather well in terms of workload-wide performance – in the worst case it does only 13% worse than DIO. But if we consider the variance of completion times and the effect on individual applications, the picture changes significantly. DEFAULT achieves a much higher variance and it is likely to stumble into much worse worst-case performance for individual applications. This means that when the goal is to deliver QoS, achieve performance isolation or simply prioritize individual applications, contention-aware scheduling can achieve much larger performance impacts, speeding up individual applications by as much as a factor of two.

To understand why DEFAULT performs relatively well on average let us discuss several examples. Consider a four core machine where each pair of cores shares a cache. If the workload to be executed on this machine involves two intensive applications and two non-intensive applications

and if the threads are mapped randomly to cores (which is a good approximation for DEFAULT) then there is *only* a $1/3$ probability of stumbling onto the *worst* solution where the intensive applications share the same cache. If there are three intensive applications in the workload and only one non-intensive application then all mappings are relatively equivalent *on average* since two of the intensive applications will experience performance degradation and one will not (the one paired with a non-intensive one). Similarly, workloads with no intensive applications, one intensive applications, and all intensive applications show no real difference between solutions. As such, DEFAULT is able to perform well on average. Therefore, we believe that future research must focus on the performance of individual threads, which can vary greatly under the DEFAULT scheduler, as opposed to trying to improve average performance. This point is further highlighted in our experiments with the LAMP workloads.

2.5.3 Results for the LAMP workloads

In the previous section we showed the effectiveness of Distributed Intensity Online for several scientific workloads consisting of the benchmarks from SPEC CPU 2006 suite. In this section we demonstrate performance improvements that DIO can provide to real multithreaded applications from the LAMP stack.

The main data processing in LAMP is done by Apache HTTP server and MySQL database engine. Both are client-server systems: several clients (website visitors in case of Apache) are concurrently connecting to the server machine to request a webpage or process the data in the database. The server management daemons *apache2* and *mysqld* are then responsible for arranging access to the website scripts and database files and performing the actual work of data storing/retrieval. In our experimental setup we initiated four concurrent web requests to the Apache 2.2.14 web server equipped with PHP version 5.2.12 and four remote client connections to the database server running MySQL 5.0.84 under OS Linux 2.6.29.6. Both *apache2* and *mysqld* are multithreaded applications that spawn one new distinct thread for each new client connection. This client thread within a daemon is then responsible for executing the client‘s request. In our experiments, all eight cores of Intel and AMD machines with eight client threads of *apache2* and *mysqld* were used.

In order to implement a realistic workload, we decided to use the data gathered by the web statistics system for five real commercial websites as our testing database. This data includes the information about website‘s audience activity (what pages on what website were accessed, in what order, etc.) as well as the information about visitors themselves (client OS, user agent information,

browser settings, session id retrieved from the cookies, etc.). The total number of records in the database is more than 3 million.

The memory intensity of the client threads varies greatly depending on what webpage or database operation is being requested by the client. We chose to request webpages that can be found in a typical web statistics system and database operations that are executed in everyday maintenance and analysis of a website‘s activity. Table 2.4 describes the webpages and database operations that were performed in our experiments. A single Apache request corresponds to the retrieval from the server a large dataset corresponding to visitor activity. The time of servicing this lengthy request is about three minutes. A single MySQL request corresponds to performing a batch of operations typical in maintenance of website databases – on our platform this request took about five minutes. Once the request is fulfilled, it is sent again until all requests finish at least three times.

We next report the results for the four workloads consisting of the requests from Table 2.4 on AMD and Intel systems. Aggregate workload speedup over DEFAULT is shown in Figures 2.14 and 2.15. Figures 2.16(a) and 2.17(a) show performance improvement over DEFAULT of the worst-case execution of each request type over a large number of trials. Figures 2.16(b), 2.17(b) show the deviation of the execution times.

To help understand the results, we explain the nature of the workload. The workload executed by Apache is extremely memory-intensive. It issues off-chip request at the rate of 74 misses per 1000 instructions – this is more than three times higher (!) than the most memory-intensive application that we encountered in the SPEC CPU2006 suite. The implication of such unprecedented memory intensity is that on our Intel system, where there is only one memory controller, the memory controller becomes the bottleneck, and our scheduling algorithm, which on the Intel system alleviates contention only for front-side bus controllers, caches, and pre-fetching units (Fig. 2.4), cannot deliver measurable performance improvements. Figure 2.15 illustrates this effect, showing that DIO does not improve performance on that system.

Turning our attention to Figure 2.14, on the other hand, we observe that on the AMD system with multiple memory controllers, DIO *does* improve performance. Apache threads, in particular, significantly benefit from DIO. (MySQL threads are not memory-intensive, so they do not and are not expected to benefit).

At the same time, as we further increase the number of Apache threads in the workload, we observe the same situation as on Intel systems: it is not possible to improve performance for all Apache threads simultaneously due to contention. This situation occurs when the number of Apache threads is greater than the number of memory domains. Apache executes a very memory-intensive

Type of operation (where it is used)	**Client request ID**	**Client request description**
Retrieval of visitor activity (essential for the analysis of the target audience, part of the web-statistics system). *All the web server threads executing retrieval requests are memory intensive (i.e., devils).*	Apache #0	Obtain the *session chart*: the top 100 popular sessions on the website.
	Apache #1	Obtain the *browser chart*: what browsers are used most frequently by the visitors of this website.
	Apache #2	Get the *visitor loyalty*: the rating of the most committed visitors and how many pages every visitor have accessed before.
	Apache #3	Get the *country popularity*: how many pages were accessed from each country.
Maintenance of the website database: inserts (archiving, updating and merging data from several databases), indexing. *All the database server threads executing maintenance requests are CPU intensive (i.e., turtles). Also, since the threads are working with the tables that are stored on the local hard drive, the continuous phases of CPU intensity had sudden drop downs of CPU activity due to accesses to the hard drive*	Mysql #0 - #5	Inserting rows into the database tables with the subsequent creation of the full-text table index.

Table 2.4: Client operations requested.

workload and the only way to completely isolate an Apache thread from contention is to schedule it without any other Apache threads in the memory domain. This is not possible when the number of Apache threads is greater than the number of memory domains. However, what we *can* do is configure DIO to achieve performance isolation for *chosen* threads, as we demonstrate next.

In this "prioritized" mode, DIO will ensure that a chosen memory-intensive thread will run in a memory domain without any other memory-intensive threads. For our LAMP workload, this will result in one Apache thread running in a domain only with "peaceful" (low missrate) MySQL threads. On the Intel system, a thread can be isolated either on a separate shared cache or on a separate physical package (recall Figure 2.4). On the AMD system, a thread isolated in a memory

domain will be isolated both in the physical package and in the shared cache.

Workloads #1-3 in Figures 2.14, 2.16(a), and 2.16(b) are the workloads where one Apache thread is prioritized, executed on an AMD system. Workloads #2c-3p in Figures 2.15, 2.17(a) and 2.17(b) are similar "prioritized" workloads on the Intel system. The prioritized thread is marked with a "(p)" in scenarios where it is isolated in a physical package, and with a "(c)" when it is isolated on the separate cache but not on a separate package (on the Intel system).

We observe that the marked threads – especially those isolated on a separate physical package – achieve a very significant performance improvement over the DEFAULT scheduler on average. The improvements in the worst-case execution times are even more dramatic. DIO delivers these performance improvements, because it is able to identify threads that will suffer from contention if co-scheduled. If one such "sensitive" thread is also assigned a high priority by the user, DIO will be able to isolate it from contention. Note, this is not the same as statically assigning high-priority threads to run on a separate domain – this will unnecessarily waste resources. DIO's ability to identify contention-sensitive threads enables it to deliver isolation only when this is likely to improve performance.

DIO also reduces the deviation in execution times of Apache threads: execution times become much more stable with DIO. Deviation for MySQL threads remains unchanged, because MySQL threads are partially I/O-bound. I/O creates a large variability in execution times, which is not (and cannot be) mitigated by a contention-aware scheduler.

Our overall conclusions drawn from the results with the LAMP workloads are as follows:

- Contention-aware scheduling is especially effective on systems that have multiple contention bottlenecks, such as multiple memory controllers. In that case, a scheduler can spread the load across these several contention points and improve overall performance. On systems where performance is dominated by a single bottleneck (e.g., a single memory controller on the Intel system), effectiveness of contention-aware scheduling diminishes.

- In cases where the workload contains multiple memory-intensive threads it becomes more difficult to improve performance for *every* thread in the workload. This is an inherent limitation of contention-aware scheduling. When contention is high only that much can be done to reduce it by means of shuffling threads across contention points. In that case, the biggest practical benefit of contention-aware scheduling is in providing performance isolation for selected threads. For instance, in a multithreaded workload, these could be threads responsible for handling latency-sensitive queries (e.g., search threads as opposed to indexing threads in

a Google workload).

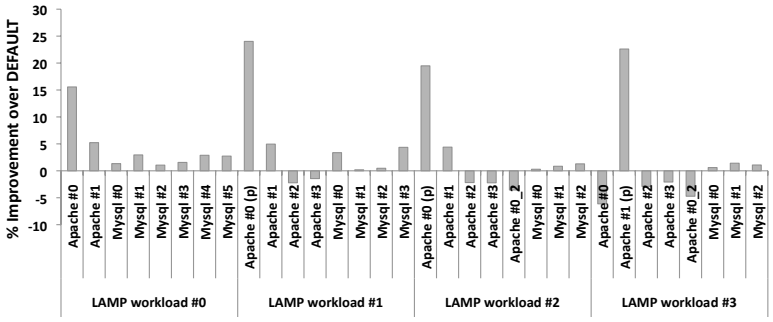


Figure 2.14: Performance improvement per thread of each LAMP workload with DIO relative to DEFAULT (high bars are good) on the AMD system.

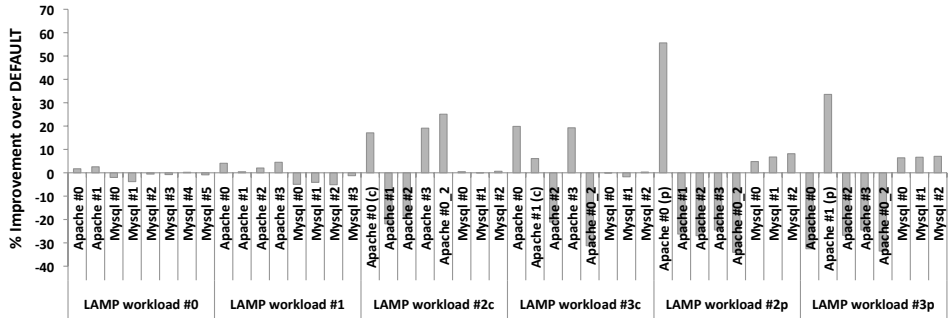
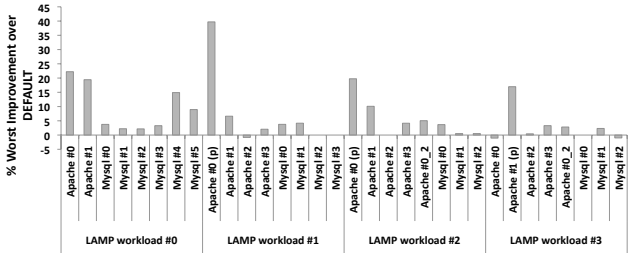


Figure 2.15: Performance improvement per thread of each LAMP workload with DIO relative to DEFAULT (high bars are good) on the Intel system.

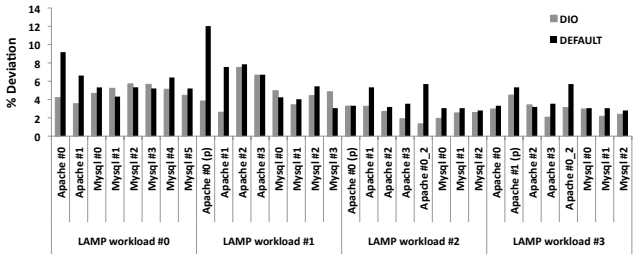
2.6 Minimizing power consumption on multicore systems with resource contention

The Distributed Intensity Online algorithm has an inherent ability to predict when a group of threads co-scheduled on the same memory domain will significantly degrade each other's performance. We found that this ability can be successfully exploited to build a power-aware scheduler (DIO-POWER) that would not only mitigate resource contention, but also reduce system energy consumption.

The key observation behind DIO-POWER is that clustering threads on as few memory domains as possible reduces power consumption. That is because on modern systems memory domains



(a) Relative performance improvement of the worst case DIO over the worst case DEFAULT of each LAMP workload on the AMD system.



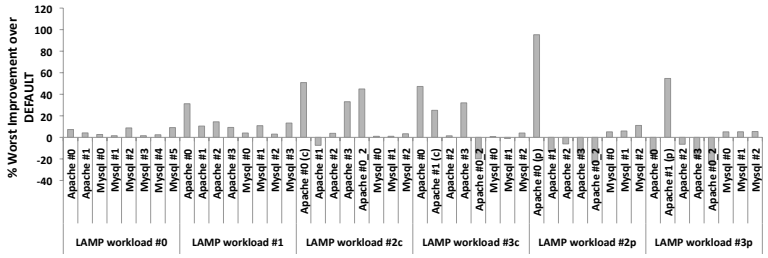
(b) Deviation with DIO and Default (low bars are good) of each LAMP workload on the AMD system.

Figure 2.16: Relative performance improvement of DIO over Default for LAMP workloads on the AMD system

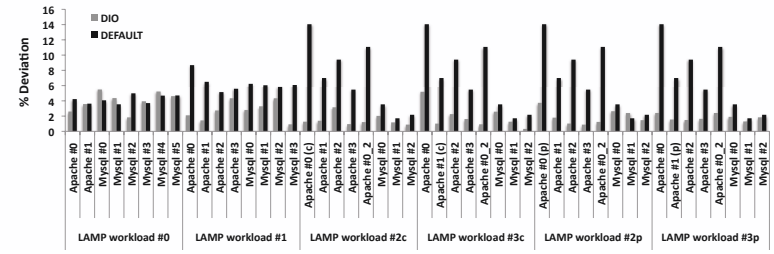
correspond to power domains (i.e., chips) and so leaving an entire chip idle reduces the power that the system consumes. At the same time, clustering threads together may hurt performance. As a result, the workload's execution time may increase, leading to increased system uptime and increased consumption of *energy* (power multiplied by time) despite reduced consumption of power. While a conventional scheduler does not know how to make the right decision (to cluster or not to cluster), DIO-POWER does, because it relies on DI's ability to predict whether a group of threads will hurt each other's performance when clustered.

We now present the data (Table 2.5) that shows reduction in power consumption due to clustering memory-intensive and CPU-intensive threads on the Intel machine for three workloads consisting of SPEC CPU 2006 benchmarks. The power consumed in every scenario is given relative to the power consumed by a completely idle machine when no benchmarks are running. Power is measured using power analyzer Extech 380803.

The results for the first workload show the power savings from clustering. Four instances of NAMD were first scheduled together on the same physical package (socket) so that another package



(a) Relative performance improvement of the worst case DIO over the worst case DEFAULT of each LAMP workload on the Intel system.



(b) Deviation with DIO and Default (low bars are good) of each LAMP workload on the Intel system.

Figure 2.17: Relative performance improvement of DIO over Default for LAMP workloads on the Intel system

can be turned off. The increase in power consumption relative to the idle state in this case was then compared with the increase when four NAMDs are spread across the system: one instance per each shared cache. Note that this is exactly how the DIO algorithm will assign threads to cores (it will spread them). As can be seen from the table, the relative power consumption in this case increased by 18W (24%).

Similar results were obtained for the workloads consisting of instances of two different applications: one memory-intensive and one CPU-intensive. Here the corresponding increase in power consumption was as high as 81W (90%) for Power Workload #2 and 50W (48%) for Power Workload #3.

Two more interesting observations can be made from the results in Table 2.5:

- putting devils on the same physical package results in lower power consumption of the system;
- when threads are clustered on the same package, shared cache contention between devils does not contribute to the power consumption increase.

Remember, however, that power savings do not necessarily translate into energy savings: when devils are clustered on the same chip they will extend each other's execution time and, as a result, increase overall energy consumption. Therefore, a power-aware algorithm must be able to determine exactly when clustering is beneficial and when it is not.

We present DIO-POWER, a contention-aware scheduling algorithm that minimizes power consumption while preventing contention. Its structure described in blocks of pseudo code 3 and 4 is similar to that of DIO with the changes highlighted in bold.

Algorithm 3 Invocation at the beginning of every scheduling interval

1: *// initialization of the boolean global array Order[l], every member specifies the order of browsing entities on the corresponding level of the memory hierarchy. Each entity has its ID that was given to it by OS. Order[l]=0 means that entities are browsed from the smallest ID to the biggest, Order[l]=1 – vice versa.*
2: **for** $l = 0; l < number\_of\_memory\_hierarchy\_levels >; l++$ **do**
3: $Order[l]$:=0;
4: **end for**
5: sort the threads according to the *miss rate* in descending order
6: **classify the threads into devils and turtles (we loosely define devils as those with at least 2000 shared cache misses per million instructions)**
7: **determine the number of devils** D
8: **mark the first** D **shared caches along with their containers for scheduling (the rest will remain idle and so can be turned into lower power state)**
9: let T be the array of sorted threads
10: *// spread the threads across the machine* ***while preventing placing devils together in the same shared cache***
11: **while** $T \neq \emptyset$ **do**
12: take the first (the most aggressive) $t \in T$
13: *// invoke DIO-POWER() to assign the thread*
14: DIO-POWER(t,$< machine >$, 0)
15: **end while**

DIO-POWER reduces power consumption on the system by clustering the workload on as few power domains as possible while preventing the clustering of those threads that will seriously hurt each other's performance if clustered. Mildly intensive applications may be clustered together for the sake of energy savings and may thus suffer a mild performance loss. To capture the effect of DIO-POWER on both energy and performance, we used *Energy Delay Product* [90, 172]:

$$EDP = \frac{Energy\_Consumed}{Instructions\_per\_Second} = \frac{Average\_Power * Time * Time}{Instructions\_Retired} \tag{2.9}$$

EDP shows by how much energy savings (relative to the idle energy consumption of the machine) outweigh performance slowdown due to clustering.

In the next experiment we show how DIO-POWER is able to achieve optimal EDP for the dynamic workload mix relative to conventional Linux schedulers. The Linux scheduler can be configured in two modes. The DEFAULT mode, used throughout this chapter, balances load across memory domains (chips) when the number of threads is smaller than the number of cores. DEFAULT-MC is the default scheduler with the *sched_mc_power_savings* flag turned on – in this mode the scheduler attempts to cluster the threads on as few chips as possible, but without introducing any unwarranted runqueue delays.

Algorithm 4 DIO-POWER(thread to schedule t, container e_parent, memory hierarchy level of the container l)

1: Let E_{all} be the array of entities on hierarchy level $l+1$
2: Let $E_{children}$ be the array of entities on hierarchy level $l+1$ whose container is e_parent **and that were marked for scheduling**
3: browse the entities in $E_{children}$ in order $Order[l+1]$ and determine the first entity with the minimum number of allocated threads: $e\_min \in E_{children}$
4: **if** e_min is a $< core >$ **then**
5: *// we have reached the bottom of the memory hierarchy*
6: **if assigning thread** t **to core** e_min **would not result in 2 devils in the same shared cache then**
7: *// we have reached the bottom of the memory hierarchy*
8: assign thread t to core e_min
9: **end if**
10: **else**
11: increment the number of threads allocated to e_min
12: *// recursively invoke DIO-POWER() to assign the thread on the lower hierarchy level*
13: DIO-POWER(t,e_min,$l+1$)
14: **end if**
15: **if** number of threads allocated to each entity $e \in E_{all}$ is the same **then**
16: *// reverse the order of browsing on this level*
17: $Order[l+1]$:= NOT $Order[l+1]$
18: **end if**

Although each version of the Linux scheduler is able to deliver optimal EDP for a specific group of workloads – DEFAULT for predominantly devil workloads where it makes sense to spread applications across domains, and DEFAULT-MC for predominantly turtle workloads where it makes sense to cluster applications, they cannot make the right decision *across all workloads*. So the user cannot configure the scheduler optimally unless the workload is known in advance. In the next

experiment we will show that DIO-POWER is able to match the EDP of the best Linux scheduler for any workload, while each Linux scheduler only does well for a workload that happens to suit its policy.

Figure 2.18 shows how DIO-POWER works for LAMP threads described in the previous section on the Intel system. The workloads consist of four threads (to make the power savings due to clustering possible) with the varying number of Apache threads per workload.

We can see that while the DEFAULT scheduler does well in terms of EDP for the first two workloads with two and four Apache "devil" threads, where spreading these threads across chip makes sense, it does not do well for the third workload where only one Apache thread is present and so it makes sense to cluster threads. Similarly, DEFAULT-MC does well for the third workload, but not for the first two. DIO-POWER, on the other hand, achieves good EDP across all three workloads, matching the performance of the best Linux scheduling mode in each case, but without requiring manual configuration and advance knowledge of the workload.

Finally, Figure 2.19 shows the improvement in terms of EDP for each scientific workload from SPEC CPU 2006 with DIO-POWER relative to DIO on the Intel system. On average, DIO-POWER is able to show a significant improvements in EDP (up to 78%). In the case of workload #8 DIO-POWER was slightly worse than DIO. The reason is that clustering slowdown was able to outweigh power savings in this case.

We are not presenting any results for AMD Opteron systems due to almost absolute insensitivity of AMD's machines to the power savings resulting from leaving chips idle. The power consumption stayed relatively the same regardless of the number of idle chips in the system. The reason for that is probably a lack of a power savings mode for the entire processor package as opposed to any particular core (on Intel machines there are the processor/package C-states, or PC-states, where the entire processor enters lower power mode when all cores on it are idle).

We conclude that the ability of DIO to identify threads that will hurt each other's performance can be used to implement an effective power-savings policy. Existing "power-aware" schedulers in Linux can only be configured statically to cluster threads or spread them apart. DIO-POWER, on the other hand, can decide whether to spread or to cluster dynamically, depending on the properties of the workload. The same heuristic that helps DIO determine whether two threads will hurt each other's performance can help it decide whether energy savings as a result of tighter resource sharing will outweigh the energy loss as a result of contention.

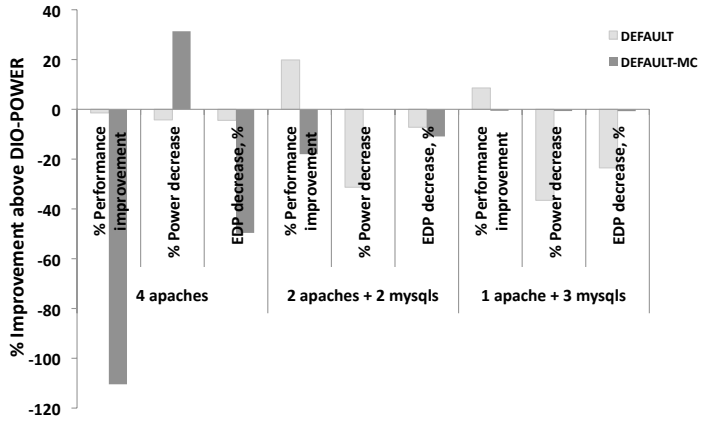


Figure 2.18: The comparison in terms of performance, power consumption and EDP for 3 LAMP workloads with DEFAULT and DEFAULT-MC relative to DIO-POWER (high bars are good) on the Intel system. DIO-POWER is able to provide the best solution in terms of EDP trade-off every time.

2.7 Conclusions

In this work we identified factors other than cache space contention that cause performance degradation in multicore systems when threads share the memory hierarchy. We estimated that other factors like memory controller contention, memory bus contention and prefetching hardware contention contribute more to overall performance degradation than cache space contention. We predicted that in order to alleviate these factors it was necessary to minimize the total number of memory requests issued from each cache. To that end we developed scheduling algorithms DI and DIO that schedule threads such that the miss rate is evenly distributed among the caches.

The Miss Rate heuristic, which underlies the DI and DIO algorithms was evaluated against the best known strategies for alleviating performance degradation due to cache sharing, such as SDC, and it was found to perform near the theoretical optimum. DIO is a user level implementation of the algorithm relying on the Miss Rate heuristic that gathers all the needed information online from performance counters. DIO is simple, can work both at the user and kernel level, and it requires no modifications to hardware or the non-scheduler parts of the operating system. DIO has been shown to perform within 2% of the oracle optimal solution on two different machines. It performs better than the default Linux scheduler both in terms of average performance (for the vast majority of workloads) as well as in terms of execution time stability from run to run for individual applications.

Upon evaluating performance of DIO we concluded that the highest impact from contention-aware scheduling is in its ability to provide performance isolation for “important” applications and

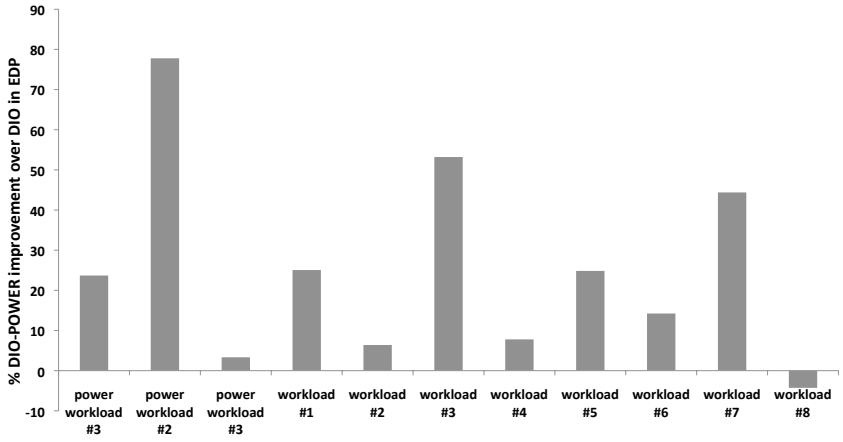


Figure 2.19: Improvement in terms of EDP for each workload with DIO-POWER relative to DIO (high bars are good) on the Intel system.

to optimize system energy consumption. DIO improved stability of execution times and lowered worst-case execution time by up to a factor of two for scientific workloads and by up to 40% for a commercial Apache/MySQL workload. Furthermore, DIO's ability to predict when threads will degrade each other's performance inspired the design of DIO-POWER, an algorithm that catered to system energy-delay product and improved the EDP relative to DIO by as much as 80%.

Workload	Physical package 0				Physical package 1				Power increase (W)	Which cores are busy
	cache 0		cache 1		cache 2		cache 3			
	core 0	core 2	core 4	core 6	core 1	core 3	core 5	core 7		
Power workload #1	NAMD1	NAMD2	NAMD3	NAMD4					76	all cores on one physical package
	NAMD1		NAMD2		NAMD3		NAMD4		94	one core in every shared cache
Power workload #2	**MCF1**	NAMD1	**MCF2**	NAMD2					90	all cores on one physical package (good co-scheduling)
	MCF1	**MCF2**	NAMD1	NAMD2					94	all cores on one physical package (bad co-scheduling)
	MCF1		**MCF2**		NAMD1		NAMD2		126	one core in every shared cache (2 MCFs on one package)
	MCF1		NAMD1		**MCF2**		NAMD2		171	one core in every shared cache (2 MCFs on different packages)
Power workload #3	**MILC1**	POVRAY1	**MILC2**	POVRAY2					104	all cores on one physical package (good co-scheduling)
	MILC1	**MILC2**	POVRAY1	POVRAY2					101	all cores on one physical package (bad co-scheduling)
	MILC1		**MILC2**		POVRAY1		POVRAY2		149	one core in every shared cache (2 MILCs on one package)
	MILC1		POVRAY1		**MILC2**		POVRAY2		154	one core in every shared cache (2 MILCs on different packages)
LAMP Power workload #1	***Apache #0***		***Apache #1***		***Apache #2***		***Apache #3***		118	DIO
	Apache #0	***Apache #1***	***Apache #2***	***Apache #3***					76	DEFAULT with power savings
	Apache #0		***Apache #1***		***Apache #2***		***Apache #3***		113	DIO-POWER
LAMP Power workload #2	***Apache #0***		***Apache #1***		*Mysql #0*		*Mysql #1*		131	DIO
	Apache #0	***Apache #1***	*Mysql #0*	*Mysql #1*					90	DEFAULT with power savings
	Apache #0	*Mysql #0*	***Apache #1***	*Mysql #1*					90	DIO-POWER

Table 2.5: Power consumption on the Intel system for three power workloads from SPEC CPU 2006 suite and 2 LAMP power workloads(devils are highlighted in bold).

Chapter 3

Addressing contention for memory hierarchy in NUMA systems

3.1 Introduction

The contention-aware algorithms that we introduced in Chapter 2 focused primarily on UMA (Uniform Memory Access) systems, where there are multiple shared LLCs, but only a single memory node equipped with the single memory controller, and memory can be accessed with the same latency from any core. However, new multicore systems increasingly use the Non-Uniform Memory Access (NUMA) architecture, due to its decentralized and scalable nature. In modern NUMA systems, there are multiple memory nodes, one per memory domain (see Figure 3.1). Local nodes can be accessed in less time than remote ones, and each node has its own memory controller. When we ran the best known contention-aware schedulers on a NUMA system, we discovered that not only do they not manage contention effectively, but they sometimes even *hurt performance* when compared to a default contention-unaware scheduler (on our experimental setup we observed as much as 30% performance degradation caused by a NUMA-agnostic contention-aware algorithm relative to the default Linux scheduler). The focus of this chapter is to investigate (1) why contention-management schedulers that targeted UMA systems fail to work on NUMA systems and (2) devise an algorithm that would work effectively on NUMA systems.

Why UMA contention-aware algorithms may hurt performance on NUMA systems: The UMA contention-aware algorithms work as follows on NUMA systems. They identify threads that are sharing a memory domain and hurting each other's performance and migrate one of the threads to

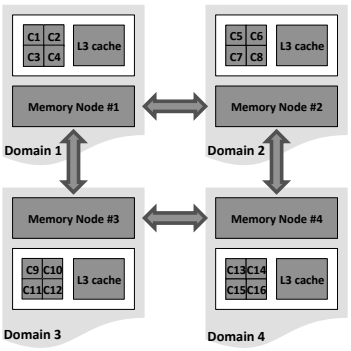


Figure 3.1: A schematic view of a system with four memory domains and four cores per domain. There are 16 cores in total, and a shared L3 cache per domain.

a different domain. This may lead to a situation where a thread's memory is located in a different domain than that in which the thread is running. (E.g., consider a thread being migrated from core C1 to core C5 in Figure 1, with its memory being located in Memory Node #1). We refer to migrations that may place a thread into a domain remote from its memory *NUMA-agnostic migrations*.

NUMA-agnostic migrations create several problems, an obvious one being that the thread now incurs a higher latency when accessing its memory. However, contrary to a commonly held belief that remote access latency – i.e., the higher latency incurred when accessing a remote domain relative to accessing a local one – would be the key concern in this scenario, we discovered that NUMA-agnostic migrations create other problems, which are far more serious than remote access latency. In particular, NUMA-agnostic migrations fail to eliminate contention for some of the key hardware resources on multicore systems and create contention for additional resources. That is why existing contention-aware algorithms that perform NUMA-agnostic migrations not only fail to be effective, but can substantially hurt performance on modern multicore systems.

Research on NUMA-related optimizations to systems is rich and dates back many years. Many research efforts addressed efficient co-location of the computation and related memory on the same node [123, 59, 120, 168, 42, 67]. More ambitious proposals aimed to holistically redesign the operating system to dovetail with NUMA architectures [85, 152, 86, 175, 118]. None of the previous efforts, however, addressed shared resource contention in the context of NUMA systems and the associated challenges. The detailed discussion of the related work for this chapter is given in Section 6.3.

Challenges in designing contention-aware algorithms for NUMA systems: To address this problem, a contention-aware algorithm on a NUMA system must migrate the memory of the thread

to the same domain where it migrates the thread itself. However, the need to move memory along with the thread makes thread migrations costly. So the algorithm must minimize thread migrations, performing them only when they are likely to significantly increase performance, and when migrating memory it must carefully decide which pages are most profitable to migrate. Our work addresses these challenges.

The contributions of our work can be summarized as follows:

- We discover that contention-aware algorithms known to work well on UMA systems may actually *hurt* performance on NUMA systems.
- We identify NUMA-agnostic migration as the cause for this phenomenon and identify the reasons why performance degrades. We also show that remote access latency is not the key reason why NUMA-agnostic migration hurt performance.
- We design and implement *Distributed Intensity NUMA Online* (DINO), a new contention-aware algorithm for NUMA systems. DINO prevents superfluous thread migrations, but when it does perform migrations, it moves the memory of the threads along with the threads themselves. DINO performs up to 20% better than the default Linux scheduler and up to 50% better than Distributed Intensity, which is the best contention-aware scheduler known to us [193].
- We devise a page migration strategy that works online, uses Instruction-Based Sampling, and eliminates on average 75% of remote accesses.

Our algorithms were implemented at user-level, since modern operating systems typically export the interfaces for implementing the desired functionality. If needed, the algorithms can also be moved into the kernel itself.

The rest of this chapter is organized as follows. Section 3.2 demonstrates why existing contention-aware algorithms fail to work on NUMA systems. Section 3.3 presents and evaluates DINO. Section 3.4 analyzes memory migration strategies. Section 3.5 provides the experimental results. Section 3.6 summarizes our findings.

The contents of Section 3.4.1 are based on work which was done collaboratively between the author Sergey Blagodurov and his colleague Mohammad Dashti.

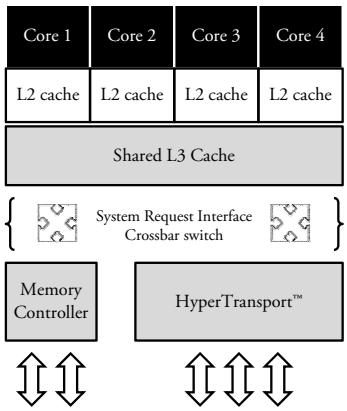


Figure 3.2: A schematic view of a system used in this study. A single domain is shown.

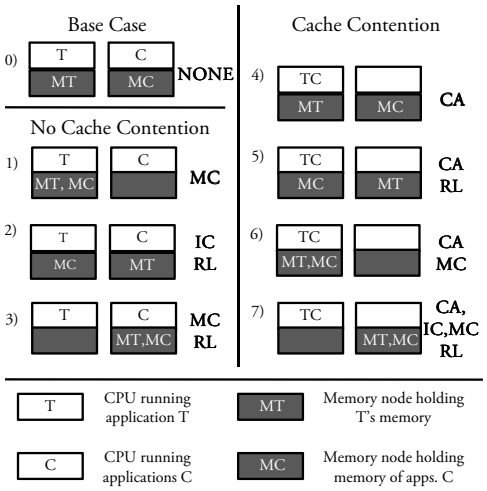


Figure 3.3: Placement of threads and memory in all experimental configurations.

3.2 Why existing algorithms do not work on NUMA systems

As we explained in the introduction, existing contention-aware algorithms perform NUMA-agnostic migration, and so a thread may end up running on a node remote from its memory. This creates additional problems besides introducing remote latency overhead. In particular, NUMA-agnostic migrations fail to eliminate *memory controller contention*, and create additional *interconnect contention*. The focus of this section is to experimentally demonstrate why this is the case.

To this end, in Section 3.2.1, we quantify how contention for various shared resources contributes to performance degradation that an application may experience as it shares the hardware with other applications. We show that memory controller contention and interconnect contention

are the most important causes of performance degradation when an application is running remotely from its memory. Then, in Section 3.2.2 we use this finding to explain why NUMA-agnostic migrations can be detrimental to performance.

3.2.1 Quantifying causes of contention

In this section we quantify the effects of performance degradation on multicore NUMA systems depending on how threads and their memory are placed in memory domains. For this part of the study, we use benchmarks from the SPEC CPU2006 benchmark suite. We perform experiments on a Dell PowerEdge server equipped with four AMD Barcelona processors running at 2.3GHz, and 64GB of RAM, 16GB per domain. The operating system is Linux 2.6.29.6. Figure 3.2 schematically represents the architecture of each processor in this system.

We identify four sources of performance degradation that can occur on modern NUMA systems, such as those shown in Figures 3.1 and 3.2:

- Contention for the shared last-level cache (CA). This also includes contention for the system request queue and the crossbar.
- Contention for the memory controller (MC). This also includes contention for the DRAM prefetching unit.
- Contention for the inter-domain interconnect (IC).
- Remote access latency, occurring when a thread's memory is placed in a remote node (RL).

To quantify the effects of performance degradation caused by these factors we use the methodology depicted in Figure 3.3. We run a target application, denoted as T with a set of three competing applications, denoted as C. The memory of the target application is denoted MT, and the memory of the competing applications is denoted MC. We vary (1) how the target application is placed with respect to its memory, (2) how it is placed with respect to the competing applications, and (3) how the memory of the target is placed with respect to the memory of the competing applications. Exploring performance in these various scenarios allows us to quantify the effects of NUMA-agnostic thread placement.

Figure 3.3 summarizes the relative placement of memory and applications that we used in our experiments. Next to each scenario we show factors affecting the performance of the target application: CA, IC, MC or RL. For example, in Scenario 0, an application runs contention-free with its

memory on a local node, so no performance-degrading factors are present. We term this the *base* case and compare to it the performance in other cases. The scenarios where there is cache contention are shown on the right and the scenarios where there is no cache contention are shown on the left.

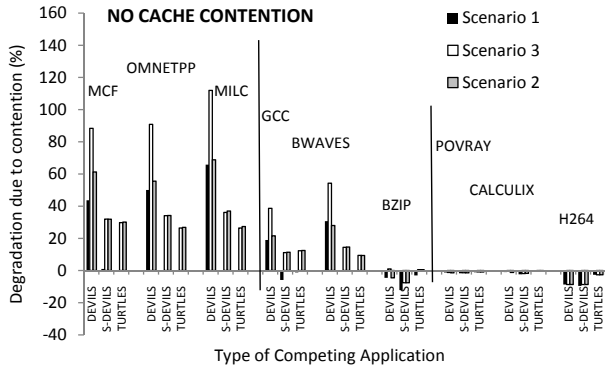
We used two types of target and competing applications, classified according to their memory intensity: *devil* and *turtle*. The terminology is borrowed from an earlier study on application classification [187]. Devils are memory intensive: they generate a large number of memory requests. We classify an application as a devil if it generates more than two misses per 1000 instructions (MPI). Otherwise, an application is deemed a turtle. We further divide devils into two subcategories: *regular devils* and *soft-devils*. Regular devils have a miss rate that exceeds 15 misses per 1000 instructions. Soft-devils have an MPI between two and 15. Solo miss rates, obtained when an application runs on a machine alone, are used for classification.

We experimented with nine different target applications: three devils (*mcf*, *omnetpp* and *milc*), three soft-devils, (*gcc*, *bwaves* and *bzip*) and three turtles (*povray*, *calculix* and *h264*).

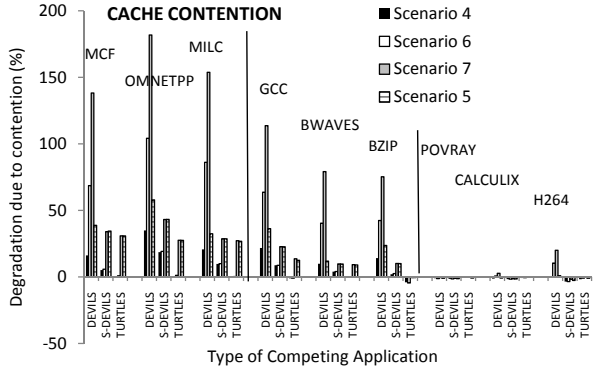
Figure 3.4 shows how an application's performance degrades in Scenarios 1-7 from Figure 3.3 relative to Scenario 0. Performance degradation, shown on the y-axis, is measured as the increase in completion time relative to Scenario 0. The x-axis shows the type of competing applications that were running concurrently to generate contention: devil, soft-devil, or turtle.

These results demonstrate a very important point exhibited in Scenario 3: when a thread runs alone on a memory node (i.e., there is no contention for cache), but its memory is remote and is in the same domain as the memory of another memory-intensive thread, performance degradation can be very severe, reaching 110% (see MILC, Scenario 3). One of the reasons is that the threads are still competing for the *memory controller* of the node that holds their memory. But this is exactly the scenario that can be created by a NUMA-agnostic migration, which migrates a thread to a different node without migrating its memory. This is the first piece of evidence showing why NUMA-agnostic migrations will cause problems.

We now present further evidence. Using the data in these experiments, we are able to estimate how much each of the four factors (CA, MC, IC, and RL) contributes to the overall performance degradation in Scenario 7 – the one where performance degradation is the worst. For that, we compare experiments that differ from each other precisely by one degradation factor involved. This allows us to single out the influence of this differentiating factor on the application performance. Figure 3.5 shows the breakdown for the devil and soft-devil applications. Turtles are not shown, because their performance degradation is negligible. The overall degradation for each application relative to the base case is shown at the top of the corresponding bar. The y-axis shows the fraction



(a) No cache contention



(b) Cache contention

Figure 3.4: Performance degradation due to contention, cases 1-7 from Figure 3.3 relative to running contention free (case 0).

of the total performance degradation that each factor causes. Since contention causing factors on a real system overlap in complex and integrated ways, it is not possible to obtain a precise separation. These results are an approximation that is intended to direct attention to the true bottlenecks in the system.

The results show that of all performance-degrading factors contention for cache constitutes only a very small part, contributing at most 20% to the overall degradation. And yet, NUMA-agnostic migrations eliminate only contention for the shared cache (CA), leaving the more important factors (MC, IC, RL) unaddressed! Since the memory is not migrated with the thread, several memory-intensive threads could still have their memory placed in the same memory node and so they would compete for the memory controller when accessing their memory. Furthermore, a migrated thread

could be subject to the remote access latency, and because a thread would use the inter-node interconnect to access its memory, it would be subject to the interconnect contention. In summary, NUMA-agnostic migrations fail to eliminate or even exacerbate the most crucial performance-degrading factors: MC, IC, RL.

3.2.2 Why existing contention management algorithms hurt performance

Now that we are familiar with causes of performance degradation on NUMA systems, we are ready to explain why existing contention management algorithms fail to work on NUMA systems. Consider the following example. Suppose that two competing threads A and B run on cores C1 and C2 on a system shown in Figure 1. A contention-aware scheduler would detect that A and B compete and migrate one of the threads, for example thread B, to a core in a different memory domain, for example core C5. Now A and B are not competing for the last-level (L3) cache, and on UMA systems this would be sufficient to eliminate contention. But on a NUMA system shown in Figure 1, A and B are still competing for the memory controller at Memory Node #1 (MC in Figure 3.5), assuming that their memory is physically located in Node #1. So by simply migrating thread B to another memory domain, the scheduler does not eliminate one of the most significant sources of contention – contention for the memory controller.

Furthermore, the migration of thread B to a different memory domain creates two additional problems, which degrade thread B's performance. Assuming that thread B's memory is physically located in Memory Node #1 (all operating systems of which we are aware would allocate B's memory on Node #1 if B is running on a core attached to Node #1 and then leave the memory on Node #1 even after thread migration), B is now suffering from two additional sources of overhead: interconnect contention and remote latency (labeled IC and RL respectively in Figure 3.5). Although remote latency is not a crucially important factor, interconnect contention could hurt performance quite significantly.

To summarize, NUMA-agnostic migrations in the existing contention management algorithms cause the following problems, listed in the order of severity according to their effect on performance: (1) They fail to eliminate memory-controller contention; (2) They may create additional interconnect contention; (3) They introduce remote latency overhead.

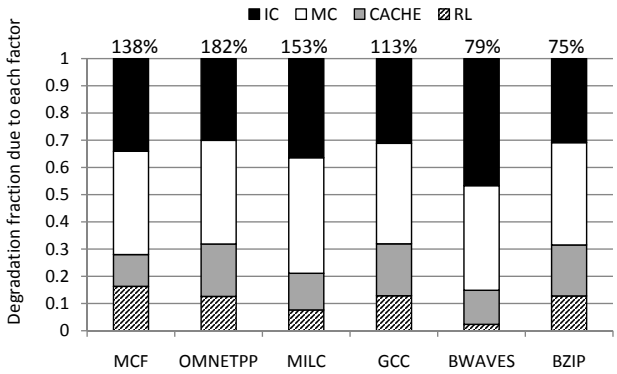


Figure 3.5: Contribution of each factor to the worst-case performance degradation.

3.3 A Contention-Aware Scheduling Algorithm for NUMA Systems

We design a new contention-aware scheduling algorithm for NUMA systems. We borrow the contention-modeling heuristic from the *Distributed Intensity* (DI) algorithm, because it was shown to perform within 3% percent of optimal on non-NUMA systems[1] [193]. Other contention aware algorithms use similar principles as DI [116, 140].

We begin by explaining how the original DI algorithm works (Section 3.3.1). For clarity we will refer to it from now on as *DI-Plain*. We proceed to show that simply extending DI-Plain to migrate memory – this version of the algorithm is called *DI-Migrate* – is not sufficient to achieve good performance on NUMA systems. We conclude with the description of our new *DI-NUMA Online*, or DINO, that in addition to migrating thread memory along with the thread eliminates superfluous migrations and unlike other algorithms improves performance on NUMA systems.

3.3.1 DI-Plain

DI-Plain works by predicting which threads will interfere if co-scheduled on the same memory domain and placing those threads on separate domains. Prediction is performed online, based on performance characteristics of threads measured via hardware counters. To predict interference, DI uses the *miss-rate heuristic* – a measure of last-level cache misses per thousand instructions, which includes the misses resulting from hardware pre-fetch requests. As we and other researchers showed in earlier work the miss-rate heuristic is a good approximation of contention: if two threads have a

[1] Although some experiments with DI reported in [193] were performed on a NUMA machine, the experimental environment was configured so as to eliminate any effects of NUMA.

high LLC miss rate they are likely to compete for shared CPU resources and degrade each other's performance [193, 53, 116, 140].

Even though the miss rate does not capture the full complexity of thread interactions on modern multicore systems, it is an excellent predictor of contention for memory controllers and interconnects – key resource bottlenecks on these systems – because it reflects how intensely threads use these resources. Detailed study showing why the miss rate heuristic works well and how it compares to other modeling heuristics is reported in [193, 53].

DI-Plain continuously monitors the miss rates of running threads. Once in a while (every second in the original implementation), it sorts the threads according to their miss rates, and assigns them to memory domains so as to co-schedule low-miss-rate threads with high-miss-rate threads. It does so by first iterating over the sorted threads starting from the most memory-intensive (the one with the highest miss rate) and placing each thread in a separate domain, iterating over domains consecutively. This way it separates memory-intensive threads. Then it iterates over the array from the other end, starting from the least memory-intensive thread, placing each on an unused core in consecutive domains. Then it iterates from the other end of the array again, and continues alternating iterations until all threads have been placed. This strategy results in balancing the memory intensity across domains. DI-Plain performs no memory migration when it migrates the threads.

Existing operating systems (Linux, Solaris) would not move the thread's memory to another node when a thread is moved to a new domain. Linux performs new memory allocations in the new domain, but will leave the memory allocated before migration in the old one. Solaris will act similarly[2]. So on either of these systems, if the thread after migration keeps accessing the memory that was allocated on another domain, it will cause negative performance effects described in Section 3.2.

3.3.2 DI-Migrate

Our first (and obvious) attempt to make DI-Plain NUMA-aware was to make it migrate the thread's memory along with the thread. We refer to this "intermediate" algorithm in our design exploration as *DI-Migrate*. The description of the memory migration algorithm is deferred until Section 3.4, but the general idea is that it detects which pages are actively accessed and migrates them to the new node along with a chunk of surrounding pages. For now we present a few experiments comparing

[2]Solaris will perform new allocations in the new domain if a thread's home *lgroup* – a representation of a thread's home memory domain – is reassigned upon migration, but will not move the memory allocated prior to home lgroup reassignment. If the lgroup is unchanged, even new memory allocations will be performed in the old domain.

DI-Plain with DI-Migrate. Our experiments will reveal that memory migration is insufficient to make DI-Plain work well on NUMA systems, and this will motivate the design of DINO.

Our experiments were performed on the same system as described in Section 3.2.1.

The benchmarks shown in this section are scientific applications from SPEC CPU2006 and SPEC MPI2007 suites with reference sets in both cases. (In a later section we also show results for the multithreaded Apache/MySQL workload.) We evaluated scientific applications for two reasons. First, they are CPU-intensive and often suffer from contention. Second, they were of interest for our partner Western Canadian Research Grid (WestGrid) – a network of compute clusters used by scientists at Canadian universities and in particular by physicists involved in ATLAS, an international particle physics experiment at the Large Hadron Collider at CERN. The WestGrid site at our university is interested in deploying contention management algorithms on their clusters. Prospect of adoption of contention management algorithms in a real setting also motivated their user-level implementation – not requiring a custom kernel makes the adoption less risky. Our algorithms are implemented on Linux as user-level daemons that measure threads' miss rates using `perfmon`, migrate threads using scheduling affinity system calls, and move memory using the `numa_migrate_pages` system call.

For SPEC CPU we show one workload for brevity; complete results are presented in Section 3.5. All benchmarks in the workload are launched simultaneously and if one benchmark terminates it is restarted until each benchmark completes three times. We use the result of the second execution for each benchmark, and perform the experiment ten times, reporting the average of these runs.

For SPEC MPI we show results for eleven different MPI jobs. In each experiment we run a single job, each comprised of 16 processes. We perform ten runs of each job and present the average completion times.

We compare performance under DI-Plain and DI-Migrate relative to the default Linux Completely Fair Scheduler, to which we refer as Default. Standard deviation across the runs is under 6% for the DI algorithms. Deviation under Default is necessarily high, because being unaware of resource contention it may force a low-contention thread placement in one run and a high-contention mapping in another. Detailed comparison of deviations under different schedulers is also presented in Section 3.5.

Figures 3.6 and 3.7 show the average completion time improvement for the SPEC CPU and SPEC MPI workloads respectively (higher numbers are better) under DI algorithms relative to Default. We draw two important conclusions. First of all, DI-Plain often *hurts* performance on NUMA systems, sometimes by as much as 36%. Second, while DI-Migrate eliminates performance loss

and even improves it for SPEC CPU workloads, it fails to excel with SPEC MPI workloads, hurting performance by as much as 25% for GAPgeofem.

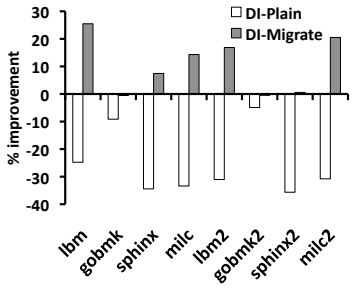


Figure 3.6: Improvement of completion time under DI-Plain and DI-Migrate relative to the Default for a SPEC CPU 2006 workload.

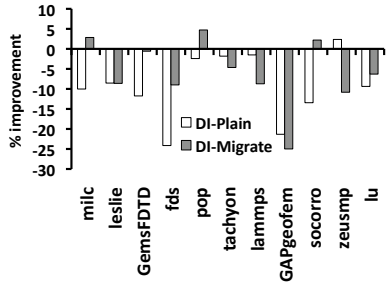


Figure 3.7: Improvement of completion time under DI-Plain and DI-Migrate relative to Default for eleven SPEC MPI 2007 jobs.

Our investigation revealed DI-Migrate migrated processes a lot more frequently in the SPEC MPI workload than in the SPEC CPU workload. While fewer than 50 migrations per process per hour were performed for SPEC CPU workloads, but as many as 400 (per process) were performed for SPEC MPI! DI-Migrate will migrate a thread to a different core any time its miss rate (and its position in the array sorted by miss rates) changes. For the dynamic SPEC MPI workload this happened rather frequently and led to frequent migrations.

Unlike on UMA systems, thread migrations are not cheap on NUMA systems, because you also have to move the memory of the thread. No matter how efficient memory migrations are, they will never be completely free, so it is always worth reducing the number of migrations to the minimum, performing them only when they are likely to result in improved performance. Our analysis of DI-Migrate behaviour for the SPEC MPI workload revealed that oftentimes migrations resulted in a thread placement that was not better in terms of contention than the placement prior to migration.

This invited opportunities for improvement, which we used in design of DINO.

3.3.3 DINO

Motivation

DINO's key novelty is in eliminating superfluous thread migrations – those that are not likely to reduce contention. Recall that DI-Plain (Section 3.3.1) triggers migrations when threads change their miss rates and their relative positions in the sorted array. Miss rates may change rather often, but we found that it is not necessary to respond to every change in order to reduce contention.

This insight comes from the observation that while the miss rate is an excellent heuristic for predicting relative contention at *coarse* granularity (and that is why it was shown to perform within 3% of the optimal oracular scheduler in DI) it does not perfectly predict how contention is affected by small changes in the miss rate.

Figure 3.8 illustrates this point. It shows on the x-axis SPEC CPU 2006 applications sorted in the decreasing order by their performance degradation when co-scheduled on the same domain with three instances of itself, relative to running solo. The bars show the miss rates and the line shows the degradations[3]. In general, with the exception of one outlier *mcf*, if one application has a much higher miss rate than another, it will have a much higher degradation. But if the difference in the miss rates is small, it is difficult to predict the relative difference in degradations.

What this means is that it is not necessary for the scheduler to migrate threads upon small changes in the miss rate, only upon the large ones.

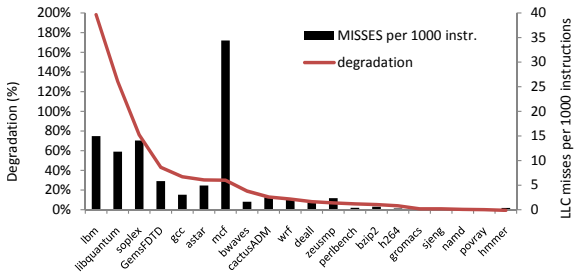


Figure 3.8: Performance degradation due to contention and miss rates for SPEC CPU2006 applications.

[3]We omit several benchmarks whose counters failed to record during the experiment.

Thread classification in DINO and multithreaded support

To build upon this insight, we design DINO to organize threads into broad *classes* according to their miss rates, and to perform migrations only when threads change their class, while trying to preserve thread-core affinities whenever possible. Classes are defined as follows (again, we borrow the animalistic classification from previous work):

Class 1: ***turtles*** – fewer than two LLC misses per 1000 instructions.
Class 2: ***devils*** – 2-100 LLC misses per 1000 instructions.
Class 3: ***super-devils*** – more than 100 LLC misses per 1000 instructions.

Threshold values for classes were chosen for our target architecture. Values for other architectures should be chosen by examining the relationship between the miss rates and degradations on that architecture.

Before we describe DINO in detail, we explain the special new features in DINO to deal with multithreaded applications.

First of all, DINO tries to co-schedule threads of the same application on the same memory domain, provided that this does not conflict with DINO's contention-aware assignment (described below). This assumes that performance improvement from co-operative data sharing when threads are co-scheduled on the same domain are much smaller than the negative effects of contention. This is true for many applications [190]. However, when this assumption does not hold, DINO can be extended to predict when co-scheduling threads on the same domain is more beneficial than separating them, using techniques described in [110] or [168].

When it is not possible to co-schedule all threads in an application on the same domain, and if threads actively share data, they will put pressure on memory controller and interconnects. While there is not much the scheduler can do in this situation (re-designing the application is the best alternative), it must at least avoid migrating the memory back and forth, so as not to make the performance worse. Therefore, DINO detects when the memory is being "ping-ponged" between nodes and discontinues memory migration in that case.

DINO algorithm description

We now explain how DINO works using an example.

In every rebalancing interval, set to one second in our implementation, DINO reads the miss rate of each thread from hardware counters. It then determines each thread's class based on its miss rate.

To reduce the influence of sudden spikes, the thread only changes the class if it spent at least 7 out of the last 10 intervals with the missrate from the new class. Otherwise, the thread's class remains the same. We save this data as an array of tuples <new_class, new_processID, new_threadID>, sorted by memory-intensity of the class (e.g., super-devils, followed by devils and followed by turtles). Suppose we have a workload of eight threads containing two super-devils (D), three devils (d) and three turtles (t). Threads numbered <0, 3, 4, 5> are part of process 0. The remaining threads, numbered 1, 2, 6 and 7 each belong to a separate process, numbered 1, 2, 3 and 4 respectively [4]. Then the sorted tuple array will look like this:

```
new_class:     D D  d d d  t t t
new_processID: 0 4  0 2 3  0 0 1
new_threadID:  0 7  4 2 6  3 5 1
```

DINO then proceeds with the computation of *the placement layout* for the next interval. The placement layout defines how threads are placed on cores. It is computed by taking the most aggressive class instance (a super devil in our example) and placing it on a core in the first memory domain `dom0`, then the second aggressive (also a super devil) – on a core in the second domain and so on until we reach the last domain. Then we iterate from the opposite end of the array (starting with the least memory-intensive instance) and spread them across domains starting with `dom3`. We continue alternating between two ends of the array until all class instances have been placed on cores. In our example, for the NUMA machine with four memory domains and two cores per domain, the layout will be computed as follows:

```
domain:      dom0  dom1  dom2  dom3
new_core:    0 1   2 3   4 5   6 7
layout:      D t   D t   d t   d d
```

Although this example assumes that the number of threads equals the number of cores, the algorithm generalizes for scenarios when the number of threads is smaller or greater than the number of cores. In the latter case, each core will have T "slots" that can be filled with threads, where $T = num\_threads/num\_cores$, and instead of taking one class-instance from the array at a time, DINO will take T.

Now that we determined the layout for class-instances, we are yet to decide which thread will fill each core-class slot – any thread of the given class can potentially fill the slot corresponding

[4] DINO assigns a unique thread ID to each thread in the workload.

to the class. In making this decision, we would like to match threads to class instances so as to *minimize the number of migrations*. And to achieve that, we refer to the matching solution for the old rebalancing interval, saved in the form of a tuple array: <old_domain, old_core, old_class, old_processID, old_threadID> for each thread.

Migrations are deemed superfluous if they change thread-core assignment, while not changing the placement of class-instances on cores. For example, if a thread that happens to be a devil (d) runs on a core that has been assigned the (d)-slot in the new assignment, it is not necessary to migrate this thread to another core with a (d)-slot. DI-Plain did not take this into consideration and thus performed a lot of superfluous migrations. To avoid them in DINO we first decide the thread assignment for any tuple that preserves core-class placement according to the new layout. So, if for a given thread `old_core = new_core` and `old_class = new_class`, then the corresponding tuple in the new solution for that thread will be <new_core, new_class, old_processID, old_threadID>.

For example, if the old solution were:

```
domain:           dom0  dom1  dom2  dom3
old_core:         0 1   2 3   4 5   6 7
old_class:        D t   d t   d t   d t
old_processID:    0 1   2 0   0 0   3 4
old_threadID:     0 1   2 3   4 5   6 7
```

then the initial shape of the new solution would be:

```
domain:           dom0  dom1  dom2  dom3
new_core:         0 1   2 3   4 5   6 7
new_class:        D t   D t   d t   d d
new_processID:    0 1     0   0 0   3
new_threadID:     0 1     3   4 5   6
```

Then, the threads whose placement was not determined in the previous step – i.e., those whose old class is not the same as their current core's new class, as determined by the new placement, will fill the unused cores according to their new class:

```
domain:           dom0  dom1  dom2  dom3
new_core:         0 1   2 3   4 5   6 7
new_class:        D t   D t   d t   d d
new_processID:    0 1   4 0   0 0   3 2
new_threadID:     0 1   7 3   4 5   6 2
```

Now that the thread placement is determined, DINO makes the final pass over the thread tuples to take care of multithreaded applications. For each thread A it checks if there is another thread B of the same multithreaded application (`new_processID(A) = new_processID(B)`) among the thread tuples not yet iterated so that B is not placed in the same memory domain with A. If there is one, we check the threads that are placed in the same memory domain with A. If there is a thread C in the same domain with A, such that `new_processID(A) != new_processID(C)` and `new_class(B) = new_class(C)` then we switch tuples B and C in the new solution. In our example this would result in the following assignment:

```
domain:           dom0  dom1  dom2  dom3
new_core:         0 1   2 3   4 5   6 7
new_class:        D t   D t   d t   d d
new_processID:    0 0   4 1   0 0   3 2
new_threadID:     0 3   7 1   4 5   6 2
```

DINO has complexity of $O(N)$ in the number of threads. Since the algorithm runs at most once a second, this has little overhead even for a large number of threads. We found that more frequent thread rebalancing did not yield better performance. Relatively infrequent changes of thread affinities mean that the algorithm is best suited for long-lived applications, such as the scientific applications we target in our study, data analytics (e.g., MapReduce), or servers. When there's more threads than cores coarse-grained rebalancing is performed by DINO, but fine-grained time sharing of cores between threads is performed by the kernel scheduler. If threads are I/O- or synchronization-intensive and have unequal sleep-awake periods, any resulting load imbalance must be corrected, e.g., as in [140].

DINO's Effect on Migration Frequency

Table 3.1: Average number of memory migrations per hour of execution under DI-Migrate and DINO for applications evaluated in Section 3.3.2.

	SPEC CPU2006					SPEC MPI2007				
	soplex	*milc*	*lbm*	*gamess*	*namd*	*leslie*	*lamps*	*GAP*	*socorro*	*lu*
DI-Migrate	36	22	11	47	41	381	135	237	340	256
DINO	8	6	5	7	6	2	1	3	2	1

We conclude this section by demonstrating how DINO is able to reduce migration frequency relative to DI-Migrate. Table 3.1 shows the average number of memory migrations per hour of execution under DI-Migrate and DINO for different applications from the workloads evaluated in Section 3.3.2. The results for MPI jobs are given for one of its processes and not for the whole job. Due to space limitations, we show the numbers for selected applications that are representative of the overall trend. The numbers show that DINO significantly reduces the number of migrations. As will be shown in Section 3.5, this results in up to 30% performance improvements for jobs in the MPI workload.

3.4 Memory migration

The straightforward solution to implement memory migration is to migrate the entire resident set of the thread when the thread is moved to another domain. This does not work for the following reasons. First of all, for multithreaded applications, even those where data sharing is rare, it is difficult to determine how the resident set is partitioned among the threads. Second, even if the application is single-threaded, if its resident set is large it will not fit into a single memory domain, so it is not possible to migrate it in its entirety. Finally, we experimentally found that even in cases where it is possible to migrate the entire resident set of a process, this can hurt performance of applications with large memory footprints. So in this section we describe how we designed and implemented a memory migration strategy that determines which of the thread's pages are most profitable to migrate when the thread is moved to a new core.

3.4.1 Designing the migration strategy

In order to rapidly evaluate various memory migration strategies, we designed a simulator based on a widely used binary instrumentation tool for x86 binaries called Pin [132]. Using Pin, we collected memory access traces of all SPEC CPU2006 benchmarks and then used a cache simulator on top of Pin to determine which of those accesses would be LLC misses, and so require an access to memory.

To evaluate memory migration strategies we used a metric called *Saved Remote Accesses* (SRA). SRA is the percent of the remote memory accesses that were eliminated using a particular memory migration strategy (after the thread was migrated) relative to not migrating the memory at all. For example, if we detect every remote access and migrate the corresponding page to the thread's new memory node, we are eliminating all remote accesses, so the SRA would be 100%.

Each strategy that we evaluated detects when a thread is about to perform an access to a remote domain, and migrates one or more memory pages from the thread's virtual address space associated with the requested address. We tried the following strategies: ***sequential-forward*** where K pages including and following the one corresponding to the requested address are migrated; ***sequential-forward-backward*** where $K/2$ pages sequentially preceding and $K/2$ pages sequentially following the requested address are migrated; ***random*** where randomly chosen K pages are migrated; ***pattern-based*** where we detect a thread's memory-access pattern by monitoring its previous accesses, similarly to how hardware pre-fetchers do this, and migrate K pages that match the pattern. We found that sequential-forward-backward was the most effective migration policy in terms of SRA.

Another challenge in designing a memory migration strategy is minimizing the overhead of detecting which of the remote memory addresses are actually being accessed. Ideally, we want to be able to detect every remote access and migrate the associated pages. However, on modern hardware this would require unmapping address translations on a remote domain and handling a page fault every time a remote access occurs. This results in frequent interrupts and is therefore expensive.

After analyzing our options we decided to use hardware counter sampling available on modern x86 systems: PEBS (Precise Event-Based Sampling) on Intel processors and IBS (Instruction-Based Sampling) on AMD processors. These mechanisms tag a sample of instruction with various pieces of information; load and store instructions are annotated with the memory address.

While hardware-based event sampling has low overhead, it also provides relatively low sampling accuracy – on our system it samples less than one percent of instructions. So we also analysed how SRA is affected depending on the sampling accuracy as well as the number of pages that are being migrated. The lower the accuracy, the higher the value of K (pages to be migrated) needs to be to achieve a high SRA. For the hardware sampling accuracy that was acceptable in terms of CPU overhead (less than 1% per core), we found that migrating 4096 pages enables us to achieve the SRA as high as 74.9%. We also confirmed experimentally that this was a good value for K (results shown later).

3.4.2 Implementation of the memory migration algorithm

Our memory migration algorithm is implemented for AMD systems, and so we use IBS, which we access via Linux performance-monitoring tool `perfmon` [78].

Migration in DINO is performed in a user-level daemon running separately from the scheduling daemon. The daemon wakes up every ten milliseconds, sets up IBS to perform sampling, reads the

next sample and migrates the page containing the memory address in the sample (if the sampled instruction was a load or a store) along with K pages in the application address space that sequentially precede and follow the accessed page. Page migration is effected using the *numa_move_pages* system call.

3.5 Evaluation

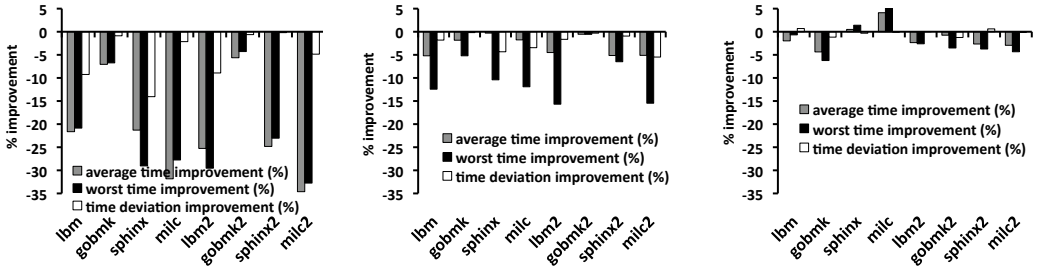
3.5.1 Workloads

In this section we evaluate DINO implemented using the migration strategy described in the previous section. We evaluate three workload types: SPEC CPU2006 applications, SPEC MPI2007 applications, and LAMP – Linux/Apache/MySQL/PHP.

We used two experimental systems for evaluation. One was described in Section 3.2.1. Another one is a Dell PowerEdge server equipped with two AMD Barcelona processors running at 2GHz, and 8GB of RAM, 4GB per domain. The operating system is Linux 2.6.29.6. The experimental design for SPEC CPU and MPI workloads was described in Section 3.3.2. The LAMP workload is described below.

The LAMP acronym is used to describe the application environment consisting of Linux, Apache, MySQL and PHP. The main data processing in LAMP is done by the Apache HTTP server and the MySQL database engine. The server management daemons *apache2* and *mysqld* are responsible for arranging access to the website scripts and database files and performing the actual work of data storage and retrieval. We use Apache 2.2.14 with PHP version 5.2.12 and MySQL 5.0.84. Both *apache2* and *mysqld* are multithreaded applications that spawn one new distinct thread for each new client connection. This client thread within a daemon is then responsible for executing the client‘s request.

In our experiment, clients continuously retrieve from the Apache server various statistics about website activity. Our database is populated with the data gathered by the web statistics system for five real commercial websites. This data includes the information about website‘s audience activity (what pages on what website were accessed, in what order, etc.) as well as the information about visitors themselves (client OS, user agent information, browser settings, session id retrieved from the cookies, etc.). The total number of records in the database is more than 3 million. We have four Apache daemons, each responsible for handling a different type of request. There are also four MySQL daemons that perform maintenance of the website database.



(a) 1 page(4KB) (b) 256 pages (1MB) (c) 4096 pages (16MB)

Figure 3.9: Performance improvement with DINO as K is varied relative to whole-resident-set migration for SPEC CPU.

We further demonstrate the effect that the choice of K (the number of pages that are moved on every migration) has on performance of DINO. Then we compare DINO to DI-Plain, DI-Migrate and Default.

3.5.2 Effect of K

Two of our workloads, SPEC CPU and LAMP demonstrate the key insights, and so we focus on those workloads. We show how performance changes as we vary the value of K. We compare to the scenario where DINO migrates the thread's entire resident set upon migrating the thread itself. The per-process resident sets of the two chosen workloads could actually fit in a single memory node on our system (it had 4GB per node), so whole-resident-set migration was possible. For SPEC CPU applications, resident sets vary from under a megabyte to 1.6GB for *mcf*. In general, they are in hundreds of megabytes for memory-intensive applications and much smaller for others. In LAMP, MySQL's resident set was about 400MB and Apache's was 120MB.

We show average completion time improvement (for Apache/MySQL this is average completion time per request), worst-case execution time improvement, and deviation improvement. Completion time improvement is the average over ten runs. To compute the worst-case execution time we run each workload ten times and record the longest completion time. Improvement in deviation is the percent reduction in standard deviation of the average completion time.

Figure 3.9 shows the results for the SPEC CPU workloads. Performance is hurt when we migrate a small number of pages, but becomes comparable to whole-resident-set migration when K reaches 4096. Whole-resident set migration actually works quite well for this workload, because migrations are performed infrequently and the resident set is small.

However upon experimenting with the LAMP workload we found that whole-resident set migration was detrimental to performance, most likely because the resident sets were much larger and also because this is a multithreaded workload where threads share data. Figure 3.10 shows performance and deviation improvement when $K = 4096$ relative to whole-resident-set migration. Performance is substantially improved when $K = 4096$. We experimented with smaller values of K, but found no substantial differences on performance.

We conclude that migrating very large chunks of memory is acceptable for processes with small resident sets, but not advisable for multithreaded applications and/or applications with large resident sets. DINO migrates threads infrequently, so a relatively large value of K results in good performance.

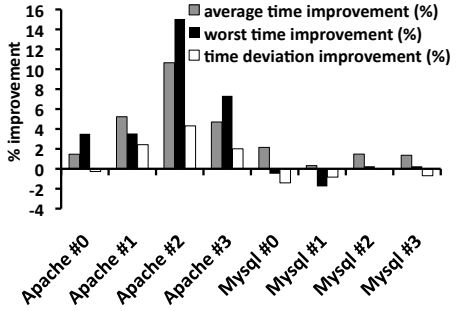
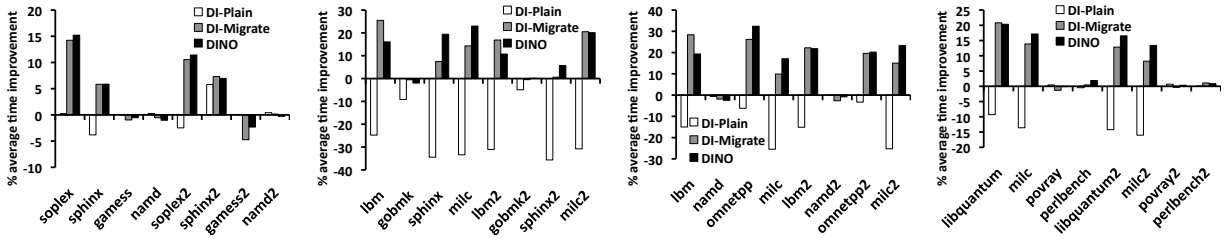


Figure 3.10: Performance improvement with DINO for $K = 4096$ relative to whole-resident-set migration for LAMP.

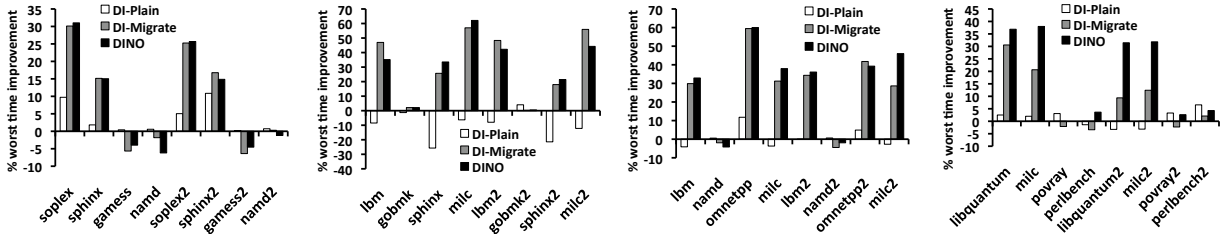
3.5.3 DINO vs. other algorithms

We compare performance under DINO, DI-Plain and DI-Migrate relative to Default, and similarly to the previous section, report completion time improvement, worst-case execution time improvement and deviation improvement.

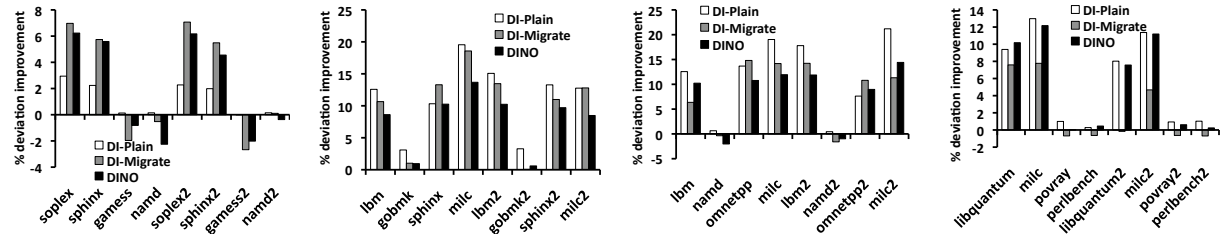
Figures 3.11-3.13 show the results for the three workload types, SPEC CPU, SPEC MPI and LAMP respectively. For SPEC CPU, DI-Plain hurts completion time for many applications, but both DI-Migrate and DINO improve, with DINO performing slightly better than DI-Migrate for most applications. Worst-case improvement numbers show a similar trend, although DI-Plain does not perform as poorly here. Improvements in the worst-case execution time indicate that a scheduler is able to avoid pathological thread assignments that create especially high contention, and produce more stable performance. Deviation of running times is improved by all three schedulers relative to Default.



(a) Average execution time improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.



(b) Worst-case execution time improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.



(c) Deviation improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.

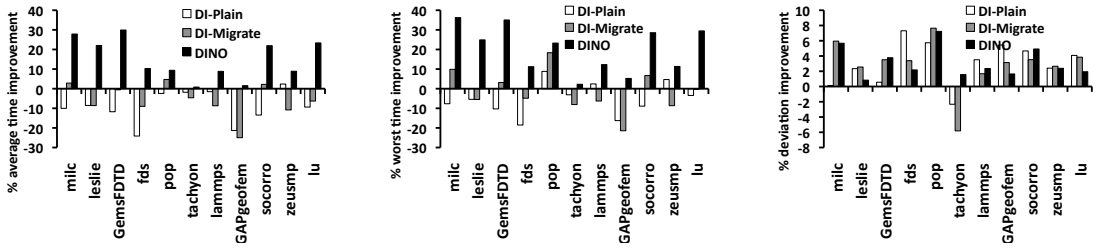
Figure 3.11: DINO, DI-Migrate and DI-Plain relative to Default for SPEC CPU 2006 workloads.

As to SPEC MPI workloads (Figure 3.12) only DINO is able to improve completion times across the board, by as much as 30% for some jobs. DI-Plain and DI-Migrate, on the other hand, can hurt performance by as much as 20%. Worst-case execution time also consistently improves under DINO, while sometimes degrading under DI-Plain and DI-Migrate.

LAMP is a tough workload for DINO or any scheduler that optimizes memory placement, because the workload is multithreaded and no matter how you place threads they still share data, putting pressure on interconnects. Nevertheless, DINO still manages to improve completion time and worst-case execution time in some cases, to a larger extent than the other two algorithms.

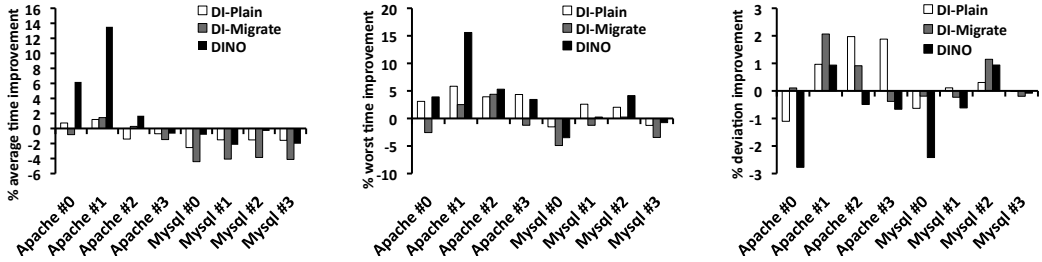
3.5.4 Discussion

Our evaluation demonstrates that DINO is significantly better at managing contention on NUMA systems than the DI algorithm designed without NUMA awareness or DI that was simply extended



(a) Completion time improvement (b) Worst-case time improvement (c) Deviation improvement

Figure 3.12: DINO, DI-Migrate and DI-Plain relative to Default for SPEC MPI 2007.



(a) Completion time improvement (b) Worst-case time improvement (c) Deviation improvement

Figure 3.13: DINO, DI-Migrate and DI-Plain relative to Default for LAMP.

with memory migration. Multiprocess workloads representative of scientific Grid clusters show excellent performance under DINO. Improvements for the challenging multithreaded workloads are less significant as expected, and wherever degradation occurs for some threads it is outweighed by performance improvements for other threads.

3.6 Conclusions

We discovered that contention-aware algorithms designed for UMA systems may hurt performance on systems that are NUMA. We found that contention for memory controllers and interconnects occurring when thread runs remotely from its memory are the key causes. To address this problem we presented DINO: a new contention management algorithm for NUMA systems. While designing DINO we found that simply migrating a thread's memory when the thread is moved to a new node is not a sufficient solution; it is also important to eliminate superfluous migrations: those that add to migration cost without providing the benefit. The goals for our future work are (1) devising metric for predicting a trade-off between performance degradation and benefits from thread sharing and (2) investigate the impact of using small versus large memory pages during migration.

Chapter 4

Addressing contention for server CPU cores

4.1 Introduction

So far we considered only a certain class of workloads in this dissertation: those that make intensive use of the memory hierarchy on the modern datacenter servers. These are CPU intensive batch computations that need to be finished by a certain deadline. There are, however, many transactional workloads in datacenters that, while not consuming much of the server resources, require immediate access to the resources they do need. In this chapter we describe our work on managing contention for CPU resources of the machine, where we take advantage of this diversity in performance requirements and resource usage of the two popular workload classes.

Unlike in Chapters 2 and 3, where the workload was 100% CPU intensive (and so did not require core sharing), in this chapter we assume that the applications are sharing same CPU cores. We react to this change in the setup by not only scheduling workload across the cores, but also *prioritizing* the access to resources from different programs based on their class. This allows to consolidate more workloads on the same servers and thus save energy. The savings come from the fact that servers in datacenters tend to be under-utilized. Low utilization means that power is not used as effectively as it could be, as servers use a disproportionate amount of power when idle, relative to when they are busy. For example, energy efficiency (work completed per unit of energy) of commodity server systems at 30% utilization can be less than half that at 100% [100, 25]. Low utilization also contributes to over-provisioning of IT equipment and increased capital expenditures (CapEx).

The advent of virtualization enabled consolidation of several applications onto a server, increasing its utilization. However, even in virtualized resource pools, utilization is typically below 40% [11, 34, 135] due to: (1) Most virtualized resource pools are fairly static and don't implement live migration of virtual machines (VMs). (2) Most datacenter workloads have very bursty demands thus leading to conservative consolidation decisions. (3) The performance of user-interactive applications drops significantly when they must contend for busy resources. The poor performance of these 'critical' applications at high utilization levels is the main challenge in increasing the utilization of servers in datacenters.

In this chapter, we address the challenge of increasing server utilization while maintaining Service Level Agreements (SLAs) by pairing critical workloads such as transactional or interactive workloads with non-critical workloads such as batch processing jobs or High Performance Computing (HPC) applications. Typically, a critical workload has strict performance requirements specified in an SLA, whereas non-critical workloads are more delay tolerant. Most workloads can be categorized into one of these two groups [100, 62].

In this work, we assume a virtualized datacenter that uses a *work conserving approach* (i.e., physical resources are shared among hosted VMs). Much previous work in virtualized datacenter management uses a *non-work conserving approach* commonly referred to as resource capping where each workload has restricted access to the resources it can use per interval [62, 147, 192, 87, 183, 111, 106, 182, 141, 131, 89, 134, 150]. To guarantee workload isolation, the sum of the caps per each resource must not exceed the available resource capacity. Unfortunately, enforcing caps typically leads to low average utilizations. The *work conserving approach* however, typically leads to higher utilization but no resource access guarantees can be made. In practice, an expert chooses the consolidation factor, i.e., the utilization level up to which workloads will be consolidated.

Our solution is to consolidate both batch and interactive workloads onto each server, enabling a very high utilization level (80% and above) [51]. At higher utilizations, the performance of the hosted workloads is likely to degrade due to contention for shared server resources. We prevent this by providing prioritized access to physical resources using Linux Control Groups (cgroups) *cpu.shares* [5]. Since the term shares is often used in cap-based work to denote a portion of CPU allocated to a particular job, we refer to Linux CPU shares as CPU weights in this chapter, to avoid confusion. We ensure that critical workloads have preferred access to physical resources such that they exhibit similar performance when consolidated with non-critical workloads as compared to when they are not. We demonstrate that the performance for low priority, non-critical workloads, which are typically less response-time sensitive, remains acceptable as long as server resources are

not excessively over-provisioned.

Other work improves server utilization by using weights to control resource assignment in work conserving mode [73, 185, 161, 83, 166, 126, 129]. In this work, we introduce a novel approach that fully loads the server with critical and non-critical loads. None of the previous work investigated whether a work-conserving approach can be used to significantly increase resource utilization up to full server capacity while maintaining acceptable performance. We have developed a working prototype and we show that it is feasible to achieve high server utilization with a detailed experimental study using real workloads. The detailed discussion of the related work for this chapter is given in Section 6.4.

We evaluate our techniques on the CPU, network and IO-intensive datacenter workloads. In this work, we focus on increasing the CPU utilization while considering other physical resource constraints. The motivation behind it is the fact that the energy consumption of the server increases mainly with its CPU consumption [100]. We note that our approach can also be applied in the presence of network and I/O utilization bottlenecks. Our work provides several unique contributions:

- We demonstrate that static cgroups weights can be used to increase overall server utilization while maintaining the performance of critical workloads. We quantify the benefits and show the impact to critical and non-critical workloads.
- We evaluate the correlation between the CPU access weights and the workload CPU consumption. We further demonstrate that intelligent assignment of critical workload virtual CPUs to physical CPUs can completely mitigate the impact of collocation even at very high utilization levels.
- We show that dynamic weight management can preserve SLAs when multiple bursty critical workloads share resources. We present a model that dynamically assigns weights based on performance and past resource consumption. We show that this approach can be used to provide fairness or performance isolation between workloads.

The remainder of this chapter is organized as follows. An overview of the system under study is provided in Section 4.2. Section 4.3 describes our provisioning method for maximizing server utilization using static resource access weights. Section 4.4 explores dynamic CPU access management of critical workloads, followed by a summary of our work in Section 4.5.

4.2 System overview

4.2.1 Workload description

Table 4.1 describes the workloads that we use for our experiments. We use RUBiS and Wiki as representatives for critical workloads and combinations of financial, animation, simulation and scientific applications as non-critical workloads [50, 49, 68, 28]. Both critical applications are Web applications. Wiki represents a fairly large and computational intensive application that triggers multiple queries to the database or Memcached server to create its dynamic Web pages. Wiki response times in our setup are typically on the order of 500 milliseconds (ms) and above. In contrast, RUBiS is a much faster application with response times on the order of tens of milliseconds.

We divide our non-critical workloads into two groups. In Group A we use a set of Swaptions, Facesim, and FDS. These are popular applications representing three typical types of the datacenter batch loads: finance, animation and simulation. Swaptions and Facesim are multi-threaded, mostly CPU intensive, with Facesim and FDS being partially I/O bound. They have no network communication. Group B is comprised of LU, BT and CG, which are CPU bound, network intensive MPI jobs. They represent a typical HPC workload. MPI applications are becoming increasingly important in the datacenter as more and more HPC workloads are being moved ”into the cloud” (i.e., hosted in a remote, shared, virtualized datacenter).

Next, we describe the performance metrics that we use to measure the performance level of the workloads. For transactional workloads, we measure the average, 95th and 99th percentile response time and request rate every 5 seconds. We obtain the response times for each request by using information from the client (i.e., the workload generator); however, in a real environment, response times from the Web server log files can be used. We created a monitor that parses the workload generator log files and calculates the performance metrics at runtime.

The number of CPU cycles consumed is used to estimate the progress of batch workloads. Empirical studies reveal that these workloads consume a certain amount of CPU time (with small variation) to complete. Hence, we use CPU consumption in each measurement interval as a performance metric for the batch workloads under study. We compare these values with the ”solo performance” of the batch workload (i.e., performance obtained when the batch workload has exclusive use of the server) to determine the degree of slowdown resulting from collocation. For example, if the CPU utilization of a batch workload is decreased to half the utilization of the solo run then the slowdown would be 50%.

Class	*Program*	*Description*
Critical	RUBiS	RUBiS is modeled after eBay.com and implements the core functionality of an auction site: selling, browsing and bidding. It is widely used as a benchmark in research [177, 184, 95]. We used a 3-tier RUBiS setup in our experiments (Apache – JBoss – MySQL).
	Wiki	Wiki is a realistic Web hosting benchmark based on WikiBench [176] and MediaWiki, which is the application used to host wikipedia.org. It uses real Wikipedia database dumps and generates traffic using publicly available traces. Our Wiki setup consists of a workload generator and three tiers: Apache – MySQL – Memcached.
Non-critical	Swaptions	Swaptions is a financial analysis benchmark suite from Intel that mimics an RMS (recognition, mining and synthesis) workload that uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of Swaptions [49]. Swaptions is a multi-threaded application.
	Facesim	Facesim is animation software that takes a model of a human face and a time sequence of muscle activations and computes a visually realistic animation of the modeled face [49]. Facesim is a multi-threaded application.
	Fire Dynamics Simulator (FDS)	Fire Dynamics Simulator (FDS) is a fire simulator that computes a computational fluid dynamics model of fire-driven fluid flow [68]. FDS is a single-threaded application.
	LU, BT, CG	LU (Lower-Upper Gauss-Seidel solver), BT (Block Tri-diagonal solver) and CG (Conjugate Gradient, irregular memory access and communication) are popular scientific programs written in MPI (Message Passing Interface) programming model [28].

Table 4.1: Critical and non-critical workloads used in this Study.

4.2.2 Experimental Testbed

Our testbed consists of five HP ProLiant BL465c G7 servers. Each is equipped with two 12-core AMD Opteron 6176 (2.3 GHz) CPUs. Each server has 5 MB shared cache and 64 GB of main memory distributed across 4 NUMA memory nodes. All servers are connected via a 10 Gb/s Ethernet network and have access to 4 TB of NFS storage.

We consider an environment where applications are hosted within a common pool of virtualized servers. Each application or application tier runs in a virtual machine (VM). The resources of a physical server, including CPU, memory, disk and network I/O bandwidth, are shared by the hosted

VMs. In our prototype implementation, we use KVM as the virtualization platform [19] running on Ubuntu 11.10. We further installed the latest stable Linux kernel version 3.4.

We prioritize the access to CPU cycles for the collocated workloads by adjusting the parameter *cpu.shares*, which is a configuration parameter available through Linux Control Groups (*cgroups*). It specifies how VMs are given processor time by the scheduler. In our setup, we are able to vary CPU shares values from 2 up to 262,144 for each process. These values are relative to each other. A value twice as high for a process compared to another denotes that it has access to twice as many CPU cycles as the other process. In the literature, this is often referred to as work conserving mode [84].

Linux Control Groups also allow prioritizing access to resources other than CPU. Currently, priority-based management of I/O and network access is supported via *blkio* and *net_prio* modules, respectively. Although we did not observe significant contention for disk and network in our experiments, we note that the method introduced in this chapter can be directly extended to non-CPU resources by using these Control Groups capabilities.

Table 4.2 summarize the system parameters that are considered and the values used in our experiments.

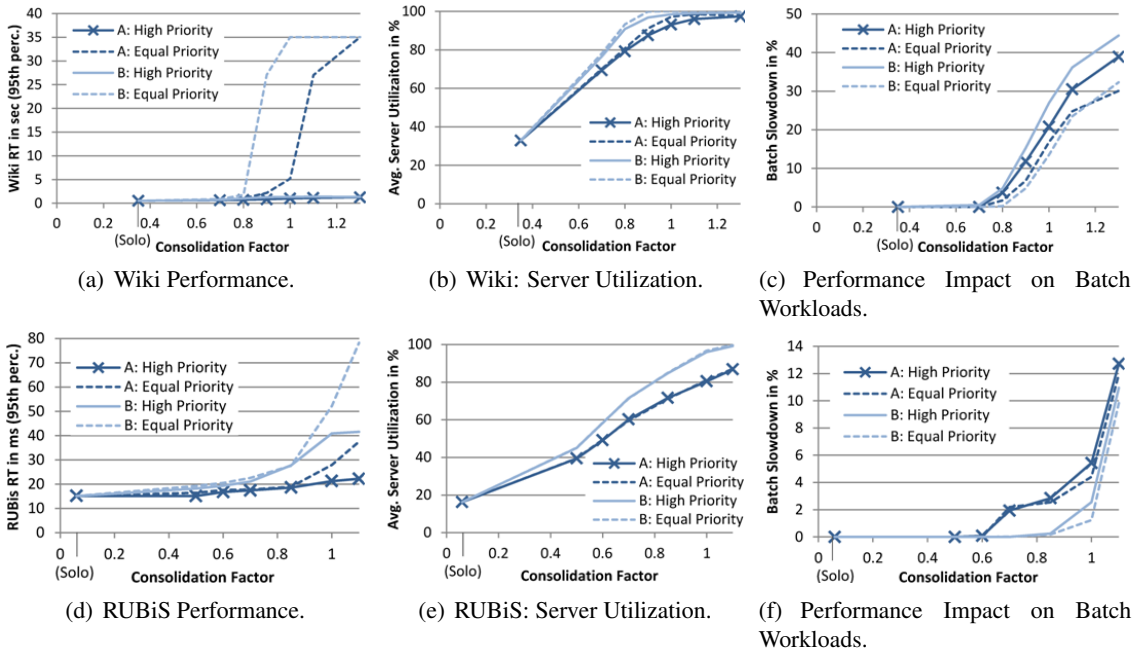
Parameter	*Range of Values in our Experiments*
Server utilization	Ranges from 1% in case of the solo RUBiS run with 45 requests per second up to 100% in case of the collocated runs of one or two critical applications with three batch jobs.
Measurement granularity	We measure performance metrics of critical and non-critical applications every 5 seconds.
Cgroups cpu.shares (CPU weights)	We vary the weight values from 2 (the minimum possible value) up to 262,144 (the maximum possible value) for each process.
Linux kernel versions	3.4, 3.0

Table 4.2: System Parameters.

4.3 Driving server utilization up

This section explores how well we can prioritize the access to CPU between collocated workloads using static CPU weights. We evaluate in detail the performance impact on critical and non-critical workloads.

4.3.1 Workload Collocation using Static Weights



(a) Wiki Performance.

(b) Wiki: Server Utilization.

(c) Performance Impact on Batch Workloads.

(d) RUBiS Performance.

(e) RUBiS: Server Utilization.

(f) Performance Impact on Batch Workloads.

Figure 4.1: Workload Collocation using Static Prioritization for Scenario A (Swaptions, Facesim, FDS) and Scenario B (LU, BT, CG).

For the experiments shown in Figure 4.1, we vary the load on the server by changing the number and size of batch workloads consolidated with the critical workload on the server. To change the size of batch workloads, we vary the number of virtual CPUs assigned to each batch VM. A consolidation factor of one indicates that the sum of the demands of all hosted workloads equals the machines capacity, i.e., the workloads hosted on the machine are expected to consume all available CPU resources. A consolidation factor above one indicates that demand for CPU resources is higher than the server capacity and thus some workloads will not get their CPU demand fully satisfied. As a baseline we run the critical workload in isolation. We then evaluate its performance when collocating with non-critical workloads.

Further, we consolidate the critical workload with two different sets of non-critical workloads. In Scenario A, we use Swaptions, Facesim, and FDS, whereas in Scenario B we use LU, BT, and CG as the non-critical workloads. Figure 4.1 compares two static weight approaches. The first approach Equal Priority follows the default settings in cgroups and assigns a CPU weight of 1,024 to each process. The High Priority approach provides preferred CPU access to critical workloads. Since the

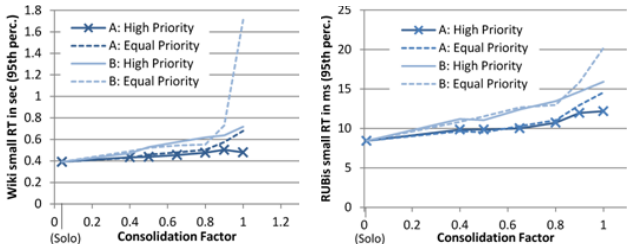
critical applications have several tiers, we set the CPU weight value for each tier individually. We use a CPU weight of 10,000 for critical workloads and a weight of 2 for non-critical workloads, i.e., a critical workload is allowed to use 5,000 times more CPU cycles per interval than a non-critical workload. This mostly ensures that non-critical VMs will only be given access to CPU cycles that are not requested by critical VMs.

Figures 4.1(a), 4.1(b) and 4.1(c) present the results of collocating Wiki with non-critical workloads. We performed each experiment multiple times and observed little variation in the results. Wiki is an open loop benchmark and the Wiki client (WikiBench) is configured to submit 40 requests per second. Figure 4.1(a) shows the 95th percentile response time of Wiki in seconds with respect to the consolidation factor. We use a standard MediaWiki setup that has a 95th percentile response time of 0.5s when running solo on the server, resulting in an overall system utilization of about 35% as shown in Figure 4.1(b). In accordance with Human Computer Interaction (HCI) research, we consider response times below 2s as acceptable application responsiveness to humans [174] (lower values are desirable, while values over 2s are deemed unacceptable). When consolidating Wiki with non-critical workloads, the results show that its performance remains acceptable until a consolidation factor of 0.8. For example, in Scenario A the 95th percentile response time increases to 1.2s when using equal weights compared to 0.75s when prioritizing Wiki. However, as soon as the server starts getting overloaded, i.e., the server utilization gets above 90% in Scenario A or 80% in Scenario B, the performance of Wiki suffers greatly with equal weights. We note that the response time does not exceed 35s because requests time out in the WikiBench client. Giving Wiki preferred access to CPU improves its performance vastly. Even in the highly overloaded case when consolidating workloads of 1.3 times the server's capacity, the 95th percentile response time is still below 1.4s in both scenarios. While we do not recommend operating servers at this load level, we note that unexpected load increases might push a server into such overload situations. Providing preferred access to critical workloads helps mitigate their performance degradation until remedial actions can be taken. Figure 4.1(c) shows the performance loss for the non-critical applications. As soon as the server utilization approaches 100%, the non-critical workloads suffer from access to fewer resources. The figure shows that the slowdown of non-critical workloads increases by up to 12% when providing preferred CPU access to critical workloads. For example, in the High Priority case with a consolidation factor of 1.3 in Scenario B the batch workloads experience a slowdown of 44% compared to 32% with equal weights. We believe that the additional performance degradation of the batch jobs through prioritizing critical workloads is tolerable as various data suggests that the batch deadline estimates are usually padded by 20 to 40% across a wide spectrum of systems [12, 89].

Figures 4.1(d), 4.1(e), and 4.1(f) shows the corresponding results using RUBiS as a critical workload. The RUBiS client is configured to trigger an average of 450 requests per second (req/s). RUBiS has a much smaller footprint than Wiki, requiring only 6% of the server's CPU resources. In the solo run, RUBiS achieves a 95th percentile response time of 15 ms. When prioritizing RUBiS, its response time at a consolidation factor of 1.1 is still below 34 ms for Scenario A and 50 ms for Scenario B. The difference in impact between the two scenarios is due to workload A being partially I/O bound. As a result, it generates less contention than B for computational resources of the server. We note that the RUBiS response times are still short enough that they fall into the "crisp" response time category defined by the HCI community [174]. This means that a human user of the RUBiS application would be unlikely to notice any difference between the response times in the solo case and the prioritized collocation case, even at a collocation factor of 1.1.

4.3.2 Impact of Critical Workload Size on Performance

The likelihood that workloads with equal CPU demand per request get all their demands satisfied depends on their size. For example, if Workload 1 services twice as many requests as Workload 2 we would need to give Workload 1 twice as large a weight as Workload 2, such that both have an equal probability to satisfy their demands. Figure 4.2 shows the impact of the critical workload size on its performance. For both Wiki and RUBiS we reduce the request rate to 10% of its original rate. The figures show that the benefits of prioritization depend on the size of the critical workloads. Although both applications improve their performance when having preferred access to CPU, the differences to the default equal priority settings is lower. This indicates that adjusting CPU weights is more important for larger critical workloads.



(a) Wiki Performance. (b) RUBiS Performance.

Figure 4.2: Impact of Critical Workload Size.

4.3.3 Providing Different Weights to Critical Workloads

This section evaluates the impact of different CPU weights for critical workloads. We consolidate two Wiki applications. Wiki 1 has a request rate of 4 req/s resulting in 4% server utilization, and Wiki 2 has 40 req/s resulting in 35% utilization. Further, we consolidate them with non-critical workloads from Scenario B and vary the consolidation factor from 0.4 to 1.1 by changing the size of the non-critical workloads. Figure 4.3 compares four different configurations:

1) *1k-1k-1k*: Each workload has a default weight setting of 1,024 (1k). This represents the *Equal Priority* approach from the previous sections.

2) *10k-10k-2*: Both critical applications Wiki 1 and Wiki 2 have a CPU weight of 10k and the non-critical workloads have a weight of 2. These are the same weight settings as in the *High Priority* approach.

3) *10k-256k-2*: The smaller Wiki 1 has a CPU weight of 10k and the larger Wiki 2 has a weight of 256k. Non-critical workloads have a CPU weight of 2.

4) *256k-10k-2*: The smaller Wiki 1 has a CPU weight of 256k and the larger Wiki 2 has a weight of 10k. Non-critical workloads have a CPU weight of 2.

The first Configuration *1k-1k-1k* provides the worst performance of critical workloads. As indicated in Section 4.3.2, the critical workload size has an effect on its performance. Given equal weights of 10k in Configuration 2) the smaller Wiki 1 consistently achieves better response times. The differences are even larger in Configuration 4) where the smaller application Wiki 1 has the highest weights. Interestingly, its performance improvement is marginal whereas the Wiki 2 suffers greatly. Providing preferred access to the larger critical application Wiki 2 allows balancing performance between the critical workloads. In Configuration 3) both Wiki applications exhibit comparable performance.

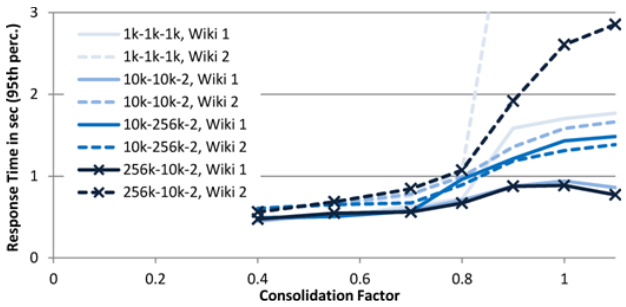
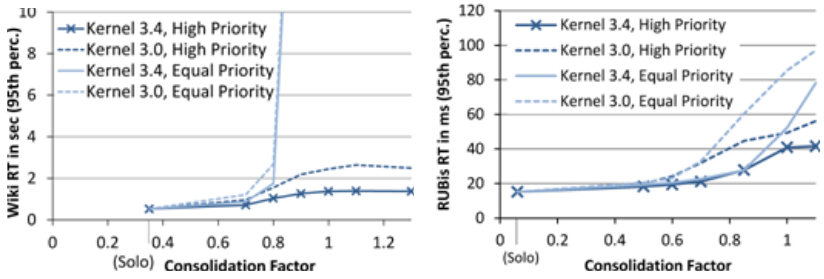


Figure 4.3: Collocating two Critical WLs with Non-critical WLs. (Scenario B).

4.3.4 Impact of Kernel Version on Performance

The feature for fine-grained CPU resource control via Linux control groups was added to the kernel in version 2.6.24 and since then many performance improvements to the kernel scheduler have been made. To illustrate the progress we compare the results of kernel 3.4 from Figure 4.1(a) and 4.1(d) in Scenario B with corresponding results with the default Ubuntu 11.10 kernel version 3.0. The results are shown in Figure 4.4. We note that the performance of critical workloads improved in all scenarios, especially when providing preferred access to CPU. For example, in Figure 4.4(a) the response time of Wiki with high priority at very high system load is twice as long with the 3.0 kernel compared to the 3.4 kernel. The performance differences for RUBiS between the two kernel versions are smaller than for Wiki but still significant, as shown in Figure 4.4(b). While the critical application benefitted from a newer kernel version, the performance of the batch workloads remained unchanged. This indicates that switching costs between Linux processes and the allocation of virtual machine CPUs to physical CPU cores has improved significantly with newer kernel versions [20, 31]. Indeed, we have measured a 39% reduction in cross-core VCPU migrations in the kernel 3.4 relative to 3.0. The average VCPU waiting time in a physical core run queue has also decreased by 17%.

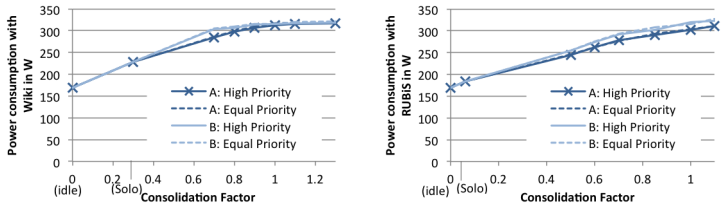


(a) Wiki (Scenario B). (b) RUBiS (Scenario B).

Figure 4.4: Impact of Kernel Version.

4.3.5 The Issue of Idle Power Consumption

Figure 4.5 presents the power measurement data for our experimental setup with different collocation factors. To measure power, we use the *Integrated Lights-Out (iLO 3)* feature of the Proliant servers. iLO is an embedded server management technology that makes it possible to monitor and perform activities on an HP server from a remote location. The iLO Ethernet card has a separate network connection (and its own IP address) to which we periodically connect via HTTPS and request



(a) Power consumption of consolidation scenarios with Wiki.

(b) Power consumption of consolidation scenarios with RUBiS.

Figure 4.5: Server power consumption.

the server's current consumed power.

In Section 4.3.1 we have shown that the proposed collocation method allows to utilize the server to its full capacity while keeping the performance degradation for the critical workloads within their SLA. We have also shown that the slowdown that batch workloads experience due to collocation is tolerable even for the high collocation factors of 1 and above. Figure 4.5 demonstrates why fully loading the server is necessary. While the power consumption (Figures 4.5(a) and 4.5(b)) proportionately increases with the server load (Figures 4.1(b) and 4.1(e)), the *idle* power consumption is almost 50% of the peak power consumption. Hence, it is desirable to offset the idle power consumption by collocating workloads on fewer servers, granted that we can keep performance impact from collocation within limits.

4.3.6 Pinning VCPUs of Critical Workloads

In this section, we investigate the reasons for the performance decrease of critical workloads with high weights when increasing workload collocation. In particular, we noticed that the virtual CPUs (VCPUs) belonging to the VMs of critical workloads often were allocated on a subset of the available physical cores, which resulted in the critical VCPUs competing with each other for CPU cycles. To determine whether this affected performance, we pinned each critical VCPU to a single physical CPU core. In our setup, the Wiki application has fewer VCPUs assigned to it than the number of physical CPUs on the server. The VCPUs of non-critical workloads remained unbound such that the kernel scheduler is able to map them freely. Figure 4.6(a) shows that with preferred access to CPU and VCPUs pinned, Wiki's performance is almost independent of the physical server utilization; compared to the solo run (0.52 seconds) its 95th percentile response time increases to 0.68 seconds at a consolidation factor of 0.7. Even at a very high consolidation factor of 1.3 its response time remains at 0.68 seconds.

Figure 4.6(b) indicates that the performance impact on non-critical workloads is small when pinning critical workload VCPUs. While batch workloads are slowed down more at consolidation factors below 0.9, they actually improve their performance in the *High Priority* scenario for higher consolidation factors when pinning critical VCPUs.

Figures 4.6(a) and 4.6(b) show that we can make resource sharing more efficient by improving the default allocation of VCPUs to physical CPUs when prioritizing workloads. The default allocation for KVM is performed by the standart Linux kernel load balancer. Below we briefly describe how it works and identify its limitations.

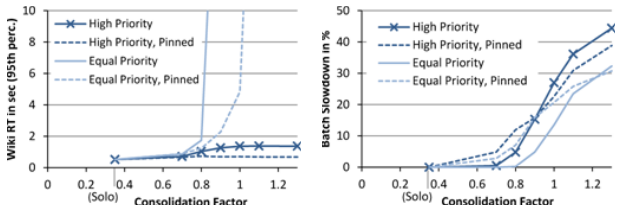
All the physical cores in Linux are organized into *scheduling domains* which reflect the topology of the system: the cores themselves are the scheduling domains of the lowest level, followed by domains that group cores adjusted to the same cache, socket and NUMA memory node. The load on each core is defined as the number of tasks in the per-core run queue. Linux attempts to balance the load (queue lengths) system wide by periodically invoking the load balancing algorithm on every core to pull tasks from longer to shorter queues.

Balancing is done progressing up the scheduling domain hierarchy and at each level the load balancer determines how many tasks need to be moved between two domains of the same level to balance the sum of the loads in those domains. The frequency with which the load balancer is invoked is dependent on both the scheduling domain level and the instantaneous load. The frequency of balancing and the number of migrations decreases as the level in the scheduling domain hierarchy increases, because the migrations between higher level domains are deemed more costly.

The load balancer may fail to balance run queues because of constraints on migrating tasks. In particular, the balancer will never migrate the currently running task, and it will resist migrating cache hot tasks, where a task is designated as cache-hot if it has executed recently (approx. 5ms) on the core (except for migration between CPU hardware contexts). This is a simple locality heuristic that ignores actual memory usage. Only if repeated attempts to balance tasks across domains fail, the load balancer will migrate cache-hot tasks. If even migrating cache-hot tasks fails to balance groups, the balancer will wake up a special kernel migration thread, which walks the domains from the busiest core up to the highest domain level to which this core belongs, searching for an underloaded core to push tasks to.

Although the default load balancer has some priority awareness (e.g., the balancing will fail if all tasks on the remote queue have a higher priority value), the kernel migration thread performs the balancing more aggressively. It does not perform the priority comparison, so the balancing is likely to succeed [137]. The issue is that this approach may be counter-productive for collocation

management in which high priority tasks often have transactional nature and could consume small amount of resources. In this case, Linux will place them into the same runqueue which will result in them competing with each other rather than with low priority CPU intensive batch jobs (the ideal case). To prevent this, we pin the VCPU to physical CPUs so that each critical process is running in its dedicated runqueue. Section 4.4.4 further investigates the case when the total number of critical VCPUs exceeds the available number of physical CPU cores.



(a) Wiki Performance. (b) Non-Critical WL Performance.

Figure 4.6: Pinning VCPUs of Critical Workloads (Scenario B).

4.3.7 Summary of Collocation with Static Weights

The previous sections showed that collocating critical and non-critical workloads achieved server utilizations of 80% and above without significant performance decreases for the critical workloads. Although the 95th percentile response times of critical workloads increased between 41% and 98% when comparing a solo run and a collocated run with up to 80% server utilization, it is important to note that based on HCI research [174] we anticipate minimal consequences for end users. In particular, in the solo experiments the physical resources are highly over provisioned. This results in much lower server utilization. While our approach increases the response times to improve server utilization, we maintain the response times in similar HCI ”categories” [174] (e.g., ”crisp” for the RUBiS application), such that typical users will not notice the difference in performance.

The experimental results show that when collocating workloads, providing critical workloads preferred access to physical resources improves their performance significantly, while having only a minor effect on non-critical workloads. Further, providing prioritized access to resources helps mitigate performance losses during sudden overload situations. Finally, providing prioritized access to CPU resources using static CPU weights in cgroups is available in most Linux distributions and does not incur additional overhead.

4.4 Dynamic Prioritization of Critical SLAs

Section 4.3 showed the benefits of providing prioritized CPU access to workloads using static CPU weights. In this section, we evaluate the impact on performance when weights are controlled dynamically. We consider two scenarios: in the *Fair* scenario, we adjust CPU weights based on application performance, e.g., the response time. Thus, two applications with similar SLAs should have an equal chance to achieve their SLA goal. In the *Isolation* scenario, we consider performance SLAs with a specified maximum load for each application, e.g., a maximum request rate. We then assign CPU weights based on the *Fair* scenario as long as request rates stay below the agreed maximum level. If the request rate of an application exceeds the maximum level then this application receives lower CPU weights in order to reduce the impact on other applications.

4.4.1 Model for Dynamical CPU Weights

In this section, we assume that each critical workload has an SLA assigned that specifies a desired response time RT^{SLA}. Our approach first determines how much CPU should be allocated to each workload in the next control interval based on its current performance and CPU consumption, similar to a non-work conserving approach. We then translate these CPU allocation values to CPU weights. To demonstrate our approach, we use a simple feedback controller. We note that more sophisticated feed-back or feed-forward controllers as presented in [192, 87, 183, 46] could be easily integrated.

First, we determine a performance factor PF for each critical application i in Equation (1). $RT_i(t)$ is the 95th percentile response time of the application in the current interval PF and RT_i^{SLA} is its response time requirement. $PF_i(t)$ ranges from -1 to ∞. A value below 0 indicates that the application meets its response time requirement, whereas a value larger than 0 denotes by how much it misses the requirement.

$$PF_i(t) = \frac{RT_i(t) - RT_i^{SLA}}{RT_i^{SLA}} \quad (1)$$

Next, we determine how much CPU should be allocated to the workload in the next interval based on its current performance factor and CPU consumption. We note that applications are typically comprised of several tiers. For instance, the Wiki application contains a Web server, a Database and a Memcached tier, each running in its own VM. $C_{i,j}(t)$ is the current CPU consumption of tier j of application i as a fraction of the total server CPU capacity and $C_{i,j}(t+1)$ is the desired CPU allocation for the next interval $t+1$:

$$C_{i,j}(t+1) = C_{i,j}(t) \times (1 + PF_i(t)) \quad (2)$$

Finally, we convert the desired CPU allocation values into the CPU weights $W_{i,j}(t+1)$. In this work, we use a simple linear model for the translation. The intuition behind the transformation is that in the case when the physical server is fully utilized or even overloaded the CPU weights determine the amount of CPU each workload tier is able to access. Setting CPU weights in relation to the desired CPU allocation values gives each workload tier an identical probability to satisfy all of its demands.

For the linear transformation model shown in Equation (3), we set $W^{max} = 256k$, which is the available maximum CPU weight value on our systems. Further, we set $W^{min} = 1k$, which is the default CPU weight on our systems. The initialization of W^{max} and W^{min} in a different setup must be done based on the available control means and datacenter requirements. A lesser value of W^{max} would imply less room to differentiate between several critical applications, while higher value of W^{min} would additionally mean more isolation for critical jobs from the non-critical influence. $C^{max}(t+1)$ is the maximum desired CPU allocation across all critical application tiers. The CPU weight $W_{i,j}(t+1)$ of application tieri,j for the next interval is defined as:

$$W_{i,j}(t+1) = (W^{max} - W^{min}) \times \frac{C_{i,j}(t+1)}{C^{max}(t+1)} + W^{min} \quad (3)$$

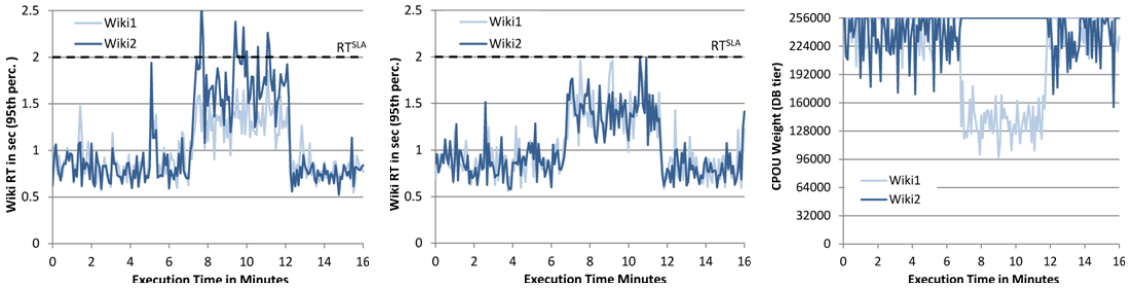
4.4.2 Providing Fairness to Critical Workloads

This section demonstrates how dynamic CPU weights can provide fairness to the critical workloads, i.e., they are able to achieve a similar level of performance regardless of their load (user requests). For Wiki applications, we specify a 95th percentile response time requirement of 2 seconds. We chose this SLA, because HCI research suggests that response times below 2s are acceptable application responsiveness to humans [174]. We note that even in our solo setup Wiki exhibits fairly long response times. We believe response times could be improved by tweaking the application.

We then consolidate two Wiki applications, which have a default request rate of 20 req/s, together with the batch workloads from Scenario B. Each Wiki utilizes about 20% of the CPU resources of the physical server. The total server utilization is around 80%. After 7 minutes, Wiki 2 experiences a 100% increase in its load for 5 minutes. Figure 4.7(a) shows the Wiki performances assigning static CPU weights as in the *10k-10k-2* Configuration from Section 4.3.3. During the spike the performance impact on Wiki 2 is higher than on Wiki 1 resulting in a SLA violation for Wiki 2 18% of the time.

Figure 4.7(b) shows results for the *Fair* Scenario where CPU weights are assigned dynamically every 5 seconds according to the model of Section 4.4.1. The figure shows that in this scenario

the two critical applications, Wiki 1 and Wiki 2, exhibit similar performance over the complete run. During the spike, the CPU weights for Wiki 1 decrease such that Wiki 2 is able to achieve a comparable performance. The CPU weights for Wiki 1 and 2 are shown in Figure 4.7(c). We note that both applications perform within their response time requirement of 2.0 seconds.



(a) Performance with Static Priorities. (b) Performance with Dynamic Priorities (fair). (c) CPU Weights of DB Tiers (fair).

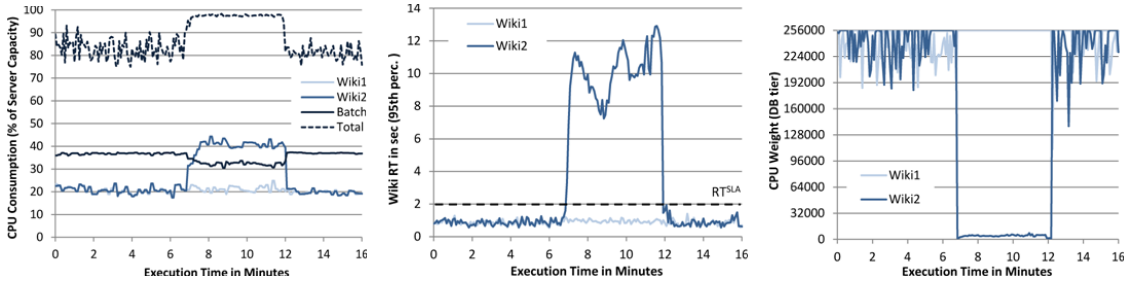
Figure 4.7: Workload Collocation during Spikes Providing Fairness.

4.4.3 Achieving Workload Isolation with Dynamic Prioritization

The big drawback of the work conserving mode is the reduced degree of isolation. For example, workloads that drastically increase their demands can steal CPU resources from other workloads with negative impact. In this section, we consider an SLA where the performance requirement (e.g., response time) is restricted by a certain maximum load (e.g., maximum number of req/s). For the experiment, we set the maximum allowed load to 30 req/s. If a critical application i exceeds the maximum load then we reduce its response time requirement RT_i^{SLA} in Equation (1) to a large value. This results in lower CPU weights and mitigates its impact on other workloads. We note that a migration controller as presented in [192] can be used to migrate off workloads if server load exceeds a maximum threshold for too long. As long as load stays within the maximum load our approach assigns dynamic CPU weights as in Section 4.4.2.

Figure 4.8 shows the results of the *Isolation* Scenario. When the load of Wiki 2 exceeds 30 req/s, its response time requirement is set to a million seconds, practically infinity. The controller then reduces the CPU weights for Wiki 2 as shown in Figure 4.8(c). By adjusting the weights, we are able to mitigate the performance impact on Wiki 1 during the spike as shown in Figure 4.8(b). As expected, the performance of Wiki 2 drops significantly for the time it exceeds the maximum allowed load. Figure 4.8(a) shows the CPU consumption for each application during the experiment. Despite

the lower weights for Wiki 2 during the spike, its CPU consumption increases due to the increased demand.



(a) CPU Consumption (isolation scenario).

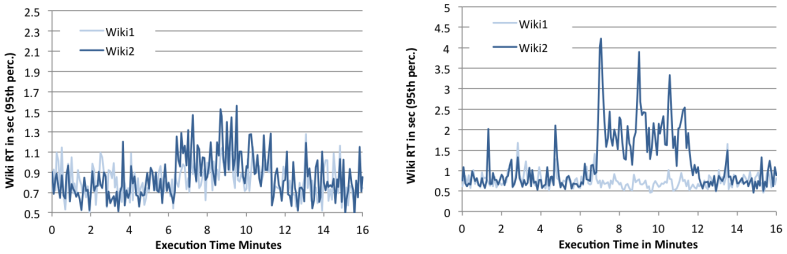
(b) Performance with Dynamic Priorities (isolation).

(c) CPU Weights of DB Tiers (isolation).

Figure 4.8: Workload Collocation during Spikes Providing Isolation.

4.4.4 Improving Critical Performance with Dynamic Pinning

Section 4.3.6 demonstrated that we can make resource sharing more efficient by improving the allocation of VCPUs to physical CPUs when prioritizing workloads. This section utilizes this collocation method by dynamically pinning critical VCPUs for the case where the total number of critical VCPUs exceeds the available number of physical CPU cores. We do that by dynamically monitoring CPU intensiveness of each critical VCPU. We then confine the top N most computationally intensive critical VCPUs to their own dedicated physical cores, where N is the number of cores on the server, and leave the non-critical CPU unbound. The experimental results of combing the dynamic pinning approach with dynamic weighting are shown on Figure 4.9. As can be seen, the critical performance has further improved for both Fair and Isolation scenarios.



(a) Performance with Dynamic Priorities and Pinning (fair).

(b) Performance with Dynamic Priorities and Pinning (isolation).

Figure 4.9: Workload Collocation with Dynamic Priorities and Pinning.

4.5 Conclusion

A long-standing problem associated with datacenters is server under-utilization, which increases both CapEx and OpEx costs. In this chapter, we present a novel method for improving server CPU utilization through weight-based collocation management of critical (e.g., user interactive) and non-critical (e.g., batch) workloads on virtualized multi-core servers. Our experimental results reveal that the proposed collocation method is able to utilize a server to near full capacity with small critical workload performance loss. In future work, we will consider hard deadlines for batch jobs and investigate scenarios where other resources than CPU are the bottleneck.

Part III

Addressing contention globally on the cluster level

Chapter 5

Addressing cluster contention via multi-objective scheduling

5.1 Introduction

While the objectives of performance and energy efficiency could be viewed as orthogonal, we found that they are usually tightly coupled, and pursuing one objective requires a trade-off with another, especially when dealing with many servers at once. We also realized that, to get the most out of contention management, we would have to make rebalancing decisions on a datacenter scale rather than within each standalone server (which was the topic of Chapters 2, 3 and 4). This chapter is devoted to our solution for the contention management that spans many datacenter servers altogether.

The definition of a datacenter in this context is broad: from tightly coupled CPU nodes in supercomputers, to loosely coupled clusters of (possibly heterogeneous) nodes managed by High Performance Computing (HPC) job schedulers, Hadoop, Dryad or similar middleware. What unites these systems is that: (1) at large scale they consume massive amounts of energy, and (2) their resource management algorithms are not designed to optimally manage energy consumption.

Energy consumption: Datacenters worldwide consume nearly two percent of the world electricity (more than the entire country of Mexico), and the consumption has been steadily growing over the years despite gains in efficiency [38]. The financial costs of energy are coupled with environmental costs, most notably carbon dioxide emissions [75]. According to the US Environmental Protection Agency, a supercomputer may soon produce as much greenhouse gas as a small city [16].

Inadequate energy management: A serious shortcoming of job schedulers used on computing clusters is the lack of *energy-conscious space management*. Space management is the assignment

of jobs to compute nodes at a given snapshot in time. An energy-conscious space management algorithm would make sure that: **(1)** jobs or their components sharing a node are not thrashing the resources within the node, because this can waste energy [52]; **(2)** job components communicating across the network are placed as physically close as possible, since longer network delays significantly contribute to energy consumption; **(3)** jobs are packed to compute nodes as tightly as possible, as long as performance is not hurt, because under-utilized nodes waste energy [100, 25].

Although existing solutions have dabbed into energy-conscious space management – for instance, Quincy collocates Dryad jobs with their data [105], Entropy [97] attempts to minimize the number of active nodes – none of the cluster management algorithms comprehensively addresses all three objectives. The challenge in designing a system that does is that these objectives may be in mutual conflict, creating tension between energy consumption and performance. For instance, collocating two components of a job on the same node may reduce energy consumption and communication overhead, but at the same time hurt performance if there is contention for the node's shared resources. Aiming to compact the jobs on as few nodes as possible may exacerbate intra-node resource contention and hurt performance, and in some cases even increase energy consumption.

We designed and built *Clavis2D*, a system that addresses these goals. *Clavis2D* assigns jobs or job components to nodes so as to minimize contention for intra-node shared resources, such as CPU caches and memory bandwidth (existing job schedulers already manage contention for CPU cycles or physical RAM space, so *Clavis2D* does not contribute in that space). *Clavis2D* collocates communicating jobs (or components) on the same node or within a short-range communication distance of a cluster so as to minimize communication overhead. Further, it compacts the jobs onto the nodes as tightly as possible, but without compromising performance. To balance the often conflicting objectives of minimizing intra-node resource contention, communication distance, and the power required by the cluster nodes, *Clavis2D* models the problem as a multi-objective optimization.

Contributions: The main contribution of our work is a new cluster scheduling algorithm that balances multiple performance- and energy-related objectives. In this chapter we chose to minimize intra-node contention, communication distance and the number of active nodes, but our framework is general enough that we can easily add other objectives, such as minimizing CPU overload, SLA violation, etc.

Multi-objective scheduling is a difficult problem. Finding the optimal solution is NP-hard, and even computing an approximation can take a long time. This was a serious challenge in our work, because *Clavis2D* needs to compute a solution online. To that end, we improved a generic constraint solver to use domain-specific knowledge of the problem, so it is able to approximate a solution much

faster.

Systems that balance multiple objectives typically need to be guided by human input, because there needs to be an authority that decides whether one goal is more important than another. Unfortunately, cluster management systems with which we are familiar typically leave the methods for choosing the exact configuration parameters to guesswork. As we felt this was a substantial limitation, we augmented a GUI tool called Vismon [58] to display to the user a Pareto front of good solutions, which essentially show the trade-offs between multiple goals. With this information, the user can identify which solutions match his or her preferences. User preferences are then automatically converted to system configuration parameters.

As a case in point, we implement *Clavis2D* for HPC clusters – distributed systems most commonly used for scientific computing. Whether it is processing massive amounts of data from the Large Hadron Collider [17], simulating a new airplane wing design [4], or providing patients with an instant diagnosis [18], these problems are typically solved on HPC clusters. HPC clusters run compute-intensive jobs consisting of communicating components, so contention for CPU caches and memory bandwidth as well as communication overhead are very serious issues. Furthermore, significant portions of these clusters remain under-utilized and waste energy, so a solution like *Clavis2D* can have a large impact. Nevertheless, the design principles used in *Clavis2D* are applicable to other cluster management systems. We evaluated *Clavis2D* on clusters with up to 16 nodes and achieved energy savings of up to 5 kWh (3kWh on average) and performance improvement of up to 60% (20% on average).

While there exists a plethora of work addressing time management of HPC clusters, asking how to manage job queues to achieve various performance and energy objectives [89, 113, 112, 167, 146, 124], none of it touched upon the subject of space management, which is crucial from energy-saving perspective and is focus of this chapter. The detailed discussion of the related work for this chapter is given in Sections 6.5 through 6.8.

The rest of this chapter is organized as follows: Section 5.2 provides background on the technical aspects of HPC cluster scheduling and discusses its limitations. Section 5.3 describes the choice of some key components of the system under test. Section 5.4 introduces the mechanisms behind *Clavis2D*, while Section 5.5 presents the algorithm. Section 5.6 shows the experimental results, and Section 5.7 concludes.

5.2 Background and Motivation

An *HPC cluster* is a group of linked computers, working together closely *thus in many respects forming a single computer* for the purpose of solving advanced computation problems. There are multiple, mostly identical, servers that are running multiple applications. Each server is a NUMA (Non-Uniform Memory Access) multicore system. The servers (nodes) in the HPC cluster are connected through a cluster network and are treated by a cluster resource management system as a whole. HPC cluster is a batch processing system: it executes jobs at a time chosen by the cluster scheduler according to the requirements set upon job submission, defined scheduling policy and the availability of resources. That differs from an interactive system (for example, a typical Linux desktop with shell) where commands are executed when entered via the terminal or a transactional system (i.e., a database or a web server), where the jobs are executed as soon as they are initiated by a transactional request from outside the cluster.

A job submitted to the HPC cluster is typically a shell script which contains a program invocation and a set of attributes allowing cluster user to manage the job after submission and to request the resources necessary for the job execution. The attributes specify the duration of the job (*walltime*)[1], offer control over when a job is eligible to be run, what happens to the output when it is completed and how the user is notified when the job completes. One important attribute is *the resource list*. The list specifies the amount and type of resources needed by the job in order to execute. The cluster job can request a number of processors, the amount of physical memory, the swap or the disk space.

Jobs running on HPC clusters are typically implemented as collections of processes communicating over the network, e.g., using MPI – the **M**essage **P**assing **I**nterface. So the processes of the same job could be either co-located on the same physical node or spread across multiple nodes.

Figure 5.1 describes an HPC cluster assumed within this study and a typical job management cycle in it. If there are job submission requests posted in the last scheduling interval, the framework goes through steps 1–7. Otherwise, only steps 2–7 are being executed. The cluster resource management system usually comprises a resource manager (RM) and a job scheduler (JS) [93, 104]. The main task of the resource manager is to set up a queuing system for users to submit their jobs (step 1). Other tasks include maintaining a list of available compute resources and sharing this information with the cluster scheduler (step 2), so the latter is able to form the order in which jobs will be executed on the cluster (step 3) [154]. Most resource managers have an internal, built-in job

[1] In HPC job management systems, a user must estimate the time it will take to complete the job and provide this information to the job scheduler when submitting a job.

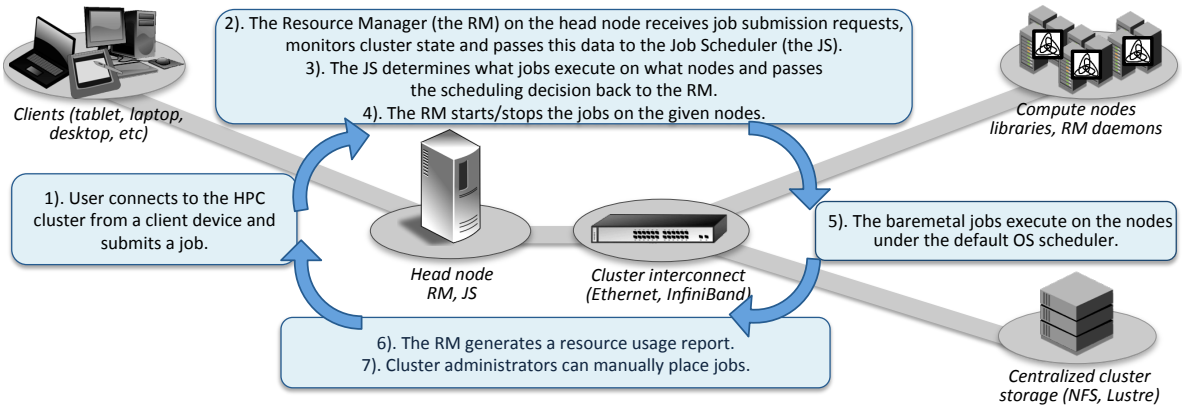


Figure 5.1: A typical HPC cluster setting.

scheduler, but system administrators usually substitute an external scheduler for the internal one to enhance functionality (the features provided by external cluster schedulers usually include advanced reservation, backfilling, preemption and many more [154]).

When the time comes for a job to start running, the JS tells the RM to launch the job processes on the specified cluster nodes (step 4). How exactly the processes comprising the job are assigned to nodes is determined by the cluster configuration. There are two process-to-node assignment policies: ***core round robin*** (CoreRR) and ***node round robin*** (NodeRR). CoreRR keeps assigning processes to the same node until all cores are occupied; only then the next idle node is chosen. NodeRR assigns processes to a node only until it fills a certain fraction of cores on that node; then it moves to the next available node [93, 104]. Once the RM has launched the processes, the job is left to run on the assigned nodes for the walltime requested in the submission script. During this period, the job processes are usually scheduled by the OS kernel running locally on the node (step 5).

The rest of the job management cycle has to do with the cluster monitoring and control. The RM often has the ability to report the status of the previously submitted jobs and their resource usage to the user upon request (step 6). Because the cluster jobs can run for the extended periods of time (a few hours to a few days or even weeks), there is a chance that some of the cluster nodes assigned to the job may falter causing the entire job to crash. That is why it is essential to have some form of periodic checkpointing supported by the job so that its state can be restored after the crash (step 7). The checkpointing mechanism is often application-specific, although special libraries can sometimes be used, BLCR being one example [3, 181]. The difficulty with the first method is that it is program dependent: only the application developers know what data to save on the disk for the later restore. While BLCR offers a more universal approach, the restore process may not go

as planned: some programs do not work with BLCR and those that do work may experience PID conflicts when being restored on different nodes. In both cases, the open connections of the moving jobs have to be terminated.

There are a few key limitations to this system.

1. It does not take into account contention for CPU caches and memory bandwidth that may occur if several processes are assigned to the same node with the CoreRR assignment. The resulting performance degradation, and the corresponding energy waste, can be as large as $3\times$ relative to a process running contention-free [52].
2. It does not take into account that when heavily communicating processes are placed on separate nodes (e.g., with NodeRR assignment), network communication overhead can waste energy and hurt performance.
3. When using the NodeRR assignment, it can leave many cores unused. As a result, the cluster as a whole ends up using more nodes than it could if the jobs were more tightly packed. Idle nodes could then be powered down to save energy. Deciding whether to tightly pack the jobs (CoreRR) or to spread them apart (NodeRR) is challenging, because we have to take into account the characteristics of individual jobs (contention-sensitive or communication sensitive) and the corresponding contention and communication overheads.
4. The existing infrastructure does not allow moving a process from one node to another after it begins execution. Processes execute on bare metal, as opposed to on a VM, and generic checkpoint-restore mechanisms are not robust[2].

To illustrate the tangible impact of these shortcomings, consider performance data in Figure 5.2. The figure shows performance difference of the NodeRR relative to CoreRR assignment (higher numbers mean performance improvement). Performance is measured as job completion time. We use the 8-node cluster described in Section 5.6 (the *Parapluie* cluster). Each node has 24 cores, and we run a single job with 24 processes. CoreRR places all of the job's processes on a single node; NodeRR places half of the job's processes on one node, and half on another.

Figure 5.2 demonstrates that the impact of the assignment policy is mixed and has to be tailored to job type. Some jobs perform better under NodeRR assignment – these are the jobs whose processes compete with each other for shared node resources when assigned using the CoreRR policy.

[2]We tried using the state-of-the-art checkpoint-restore mechanism BLCR [3, 181], but were often unable to restore processes on a new node because of PID conflicts. Application-specific checkpoint mechanisms are not widely supported.

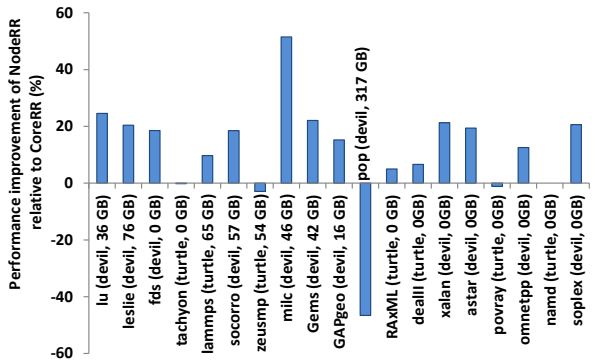


Figure 5.2: Comparison of CoreRR and NodeRR process-to-node assignments for SPEC MPI2007 and SPEC HEP runs.

Other jobs are indifferent to the assignment policy. Yet other jobs (e.g., *pop*) suffer under NodeRR, because they experience communication overhead when their processes are scheduled on different nodes. Furthermore, the magnitude of performance effects is rather variable. Some jobs do better under NodeRR, but the improvement is small, so we still may want to pack them using CoreRR so as to reduce the number of idle cores.

What we want is a cluster management algorithm that will find a placement of processes that optimally (or nearly optimally) meets these three objectives (1) minimizing contention for shared resources, (2) minimizing communication overhead and (3) minimizing the power required by the cluster nodes.

This requires weighting the importance of each of the three objectives, detecting contention and communication overhead, finding an assignment that will minimize the two while ensuring that the total power we use is minimal, and migrating the processes to new nodes.

The following two sections describe the two key pieces of our solution. Section 5.4 describes how we detect contention and communication overhead and enable migration of job processes. Section 5.5 describes the new job placement algorithm.

5.3 The Choice of the key framework components

In this section, I will provide the discussion of why a particular cluster scheduler (Maui) and Resource Manager (Torque) were chosen to be used as a baseline within the project. A description of how the chosen scheduler works is also provided.

5.3.1 The Choice of the Cluster Scheduler and Resource Manager

The three most popular cluster schedulers were analyzed during this step: Condor, Maui and Moab (Table 5.1). The choice was made in favor of Maui. The reasons are the following. Maui offers several novel opportunities for scheduling that the sligtly outdated Condor is not capable to provide. It is also more robust than Condor which was reflected in the fact that several university cluster installations has switched recently from Condor to Maui ([40] for instance). Moreover, Maui is being used by the Compute Canada clusters (in particluar, by the BC based WestGrid) and by the local HPC cluster in the Faculty of Applied Science at SFU (The Colony HPC cluster). This is important when considering future work for the project. While Moab, arguably the most popular cluster scheduler in the commercial setups, has more capabilities than Maui (especially in the amount of monitoring data that is provided by the scheduler), it is, at the same time, proprietary and quite expensive to install. Maui, on the other hand, is open-source and free. Maui and Moab was developed by the same company, with Moab being essentially an extension for Maui. That means that, when the framework developed within this project will be mature enough, it can be integrated into Moab as well. The choice of Maui as the job scheduler largely determined choosing Torque as the project's RM since is is closely integrated with (and as popular as) Maui itself (Table 5.2).

5.3.2 Maui Cluster Scheduler

Maui has a two-phase scheduling algorithm. During the first phase, the high-priority jobs are scheduled using *advance reservation*. In the second phase, a *backfill* algorithm is used to schedule low-priority jobs between previously scheduled jobs. Advance reservation uses execution time predictions (walltime) provided by the users to reserve resources (such as CPUs and memory) and to generate a schedule. Hence, it serves as the mechanism by which the scheduler guarantees the availability of a set of resources at a particular time. Given a schedule with advance-reserved, high-priority jobs and a list of low-priority jobs, a backfill algorithm tries to fit the small jobs into scheduling gaps. This allocation does not alter the sequence of jobs previously scheduled, but improves system utilization by running low priority jobs in between high-priority jobs. To use backfill, the scheduler also requires a runtime estimate, which is supplied by the user when jobs are submitted. The following is the sequence of work flow of Maui [93, 154, 104]:

0. Maui starts a new iteration when the following event(s) occur:

a) Job or resource change event occurs (i.e. job termination, node failure)

b) Reservation boundary event occurs

Scheduler	*Description*
RM internal schedulers	Open Source. Usually are very simple and are hence not used in a realistic cluster environment.
Maui	Open Source. Maui extends the capabilities of base RM schedulers by adding the following features: - Job priority policies and configurations - Job advance reservation support - QOS support including resource access control - Extensive fairness policies - Non-intrusive 'Test' modes - and many more [23].
Moab	Like Maui, is capable of supporting multiple scheduling policies, dynamic priorities, reservations, and fairshare capabilities. In addition, has integrated billing mechanisms and an advanced graphical cluster administration with integrated documentation and wizards. Unfortunately, Moab is proprietary software which makes it difficult to analyze within the academia.
Condor	Supports many features implemented in Maui (backfill, reservations, etc.). Maui, however, became predominant scheduler for academic HPC environments (for example, our local WestGrid cluster facilities use Maui for scheduling) so we opted for it.

Table 5.1: Comparison of different job schedulers.

c) A configurable timer expires

d) Via an external command

1. On every iteration, Maui obtains updated data from RM regarding node, job state, configuration.

2. Historical statistics and usage information for running and completed jobs are updated.

3. Refresh Reservations. This step adjusts existing reservation incorporating updated node availability. Changes in node availability may also cause reservations to slide forward or backward in time. Reservations may be created or removed. If the job that possess a reservation is idle (waiting in the queue) it is provided an immediate access to the reserved resources.

4. A list is generated that contains all jobs which can be feasibly scheduled. Availability of resources, job credentials, etc. are taken into account in generating this list.

5. Prioritize feasible jobs according to the historical usage information (to enforce resource fair

Resource manager	*Description*
OpenPBS	An initial Open Source RM capable of a basic job maintenance and control functionality. It was a starting point for many advanced RMs, including Torque.
Torque	This is an advanced Open Source RM based on the original OpenPBS project. Torque incorporates many advances in the areas of scalability, reliability, and functionality and is currently in use at tens of thousands of leading government, academic, and commercial sites throughout the world.
SLURM	Simple Linux Utility for Resource Management is an open-source RM designed for Linux clusters. It has many good features including portability and highly tolerance of system failures (including failure of the node executing its control functions). Torque, however, is developed by the same company as Maui and is meant to be used with it by default. It is also believed that SLURM is hard to configure properly to work with Maui ([35] for instance).

Table 5.2: Comparison of different resource managers.

sharing), various job attributes, scheduling policies and resources required by each jobs.

6. Schedule jobs in priority order, starting the jobs it can and creating advanced reservations for those it cannot until it has made reservations for the top N jobs where N is a cluster configurable parameter.

7. Backfill. Maui determines current available backfill windows in between reservations and attempts to fill them with the remaining jobs which are eligible to run during these holes.

Maui supports job preemption. High-priority jobs can preempt lower-priority or backfill jobs if resources to run the high-priority jobs are not available. In some cases, resources reserved for high-priority jobs can be used to run low-priority jobs when no high-priority jobs are in the queue. However, when high-priority jobs are submitted, these low-priority jobs can be preempted to reclaim resources for high-priority jobs.

5.4 Clavis2D: the mechanisms

This section describes the mechanisms enabling *Clavis2D*, namely: (a) online detection of contention for shared resources, (b) online detection of communication overhead and (c) a virtualization solution to enable migration of processes between nodes. Figure 5.3 highlights in bold the changes that we make to the "vanilla" framework.

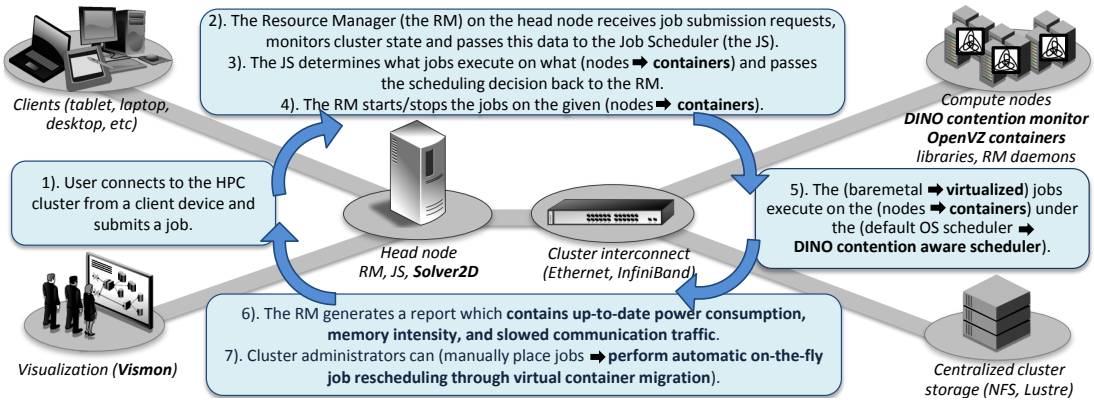


Figure 5.3: HPC cluster setting with *Clavis2D* modifications highlighted in bold.

Quantifying shared resource contention: *Clavis2D* needs to measure the degree of contention for shared resources and determine when the contention is high enough to warrant spreading the processes loosely across nodes, leaving some cores idle. *Clavis2D* addresses contention for chip-wide shared resources, such as last-level caches, memory controllers, memory bandwidth and prefetch hardware. Prior work showed that when processes running on the same chip compete for these resources, their performance can degrade by as much as 3× relative to running contention-free [52, 193].

Directly measuring the amount of contention is very difficult on modern hardware. The best known approach is to approximate it using *memory intensity*, which is simply the rate of off-chip memory requests generated by a process or thread [193]. Off-chip memory requests comprise last-level cache misses and hardware prefetches.

We classify processes based on their memory intensity into "animalistic" categories adopted from an earlier study [187]: *devils* and *turtles*. Devils suffer performance degradation in excess of 10% when more than one are co-scheduled on the cores of the same chip. Their corresponding rate of memory requests exceeds 4.5 per 1000 instructions. Turtles have a performance degradation of less than 10% and their memory request rate is below 4.5. Figure 5.2, in parentheses near application names, shows the classes for workloads from SPEC MPI 2007 and SPEC High Energy Physics (HEP).

The memory request rates and performance degradations needed to delineate the categories are hardware specific, but we found them to be rather stable across several different makes and models of multicore processors we tested. In any case, the categories can be quickly established using simple benchmarks.

The memory intensity metric enables us to quickly characterize the amount of contention of each process online, simply by measuring its memory request rate using hardware performance counters. *Clavis2D* includes a daemon on each node that performs these measurements and communicates the memory intensity of each process to the RM (step 6 in Figure 5.3). The RM in turn passes this information to *Solver2D*, a component of *Clavis2D* running on the head node (see Figure 5.3), which then uses it in the job assignment algorithm.

In addition to using memory intensity in cluster-wide scheduling decisions, *Clavis2D* also uses it for node-local scheduling, in order to avoid placing competing devils on the same chip whenever we have more than one running on the same node. For that purpose we borrow the DINO algorithm described in previous work [52].

Quantifying communication overhead: We identify the jobs whose performance is lagging because of accessing *a slow link*: a part of the cluster interconnect that provides an inferior communication performance to the job. We consider two forms of the communication overhead: (a) When the processes of the job are spread across multiple nodes, as opposed to be collocated within a node, they communicate via a cross-node interconnect, which is usually much slower than the internal node interconnect (e.g., AMD HyperTransport or Intel QuickPath). This form of communication overhead only matters for jobs that can fit within a node, as bigger jobs will access cross-node interconnect regardless of their placement. (b) When the job processes communicate via a slower channel that links several node racks together. Due to the geographical distribution of the racks and the cost of the equipment, the delay and capacity of the cross-rack channels is often much worse than that of the links within each rack [91], so if a process communicates off-rack it can experience a slowdown.

We measured the sensitivity of performance to the amount of data exchanged via the cluster interconnect. We used the same 24-process jobs as in Figure 5.2, but ran two jobs of the same program type on a node, 12 processes from each job. The remaining processes of each job where running on another node. This way we had the same level of contention as with the CoreRR assignment in Figure 5.2, but the network delays were more significant, because each job was split between two nodes. By comparing the performance in this setup with the CoreRR setup of Figure 5.2, we were able to gauge the slowdown due to network communication. We found that most of the programs do not take a big performance hit: 17 out of 19 jobs saw no measurable degradation, one suffered the degradation of 10%. Only one job (pop) suffered a 40% performance degradation due to communication overhead, and it was characterized by very intense network traffic. Hence, we use the amount

of traffic exchanged via a slow link as the metric for communication intensity. *Clavis2D* uses traffic accounting with iptables [39] to measure the traffic exchanged by every process on the system and detects those that use a slow link to communicate.

Virtualization: Using virtualization was crucial for *Clavis2D*, because we needed the ability to migrate running processes across the nodes. We evaluated a number of existing options, including VMWare, KVM, Xen, and narrowed down our choice to OpenVZ. OpenVZ uses a single patched Linux kernel that runs both on the host and in the guests. It uses a common file system so each VM is just a directory of files that is isolated using chroot and special Linux kernel modules. However, because it does not have the overhead of a full VM, it is very fast and efficient. Different studies have shown an average performance overhead of 1-3% for virtualizing with OpenVZ as compared to using a standalone server [30, 158], and we observed similar overheads.

While the topic of virtualizing cluster workload is not novel, the topic of virtualizing HPC workload is. Our framework contains several non-trivial engineering details of implementing an online migration of the HPC jobs across the nodes in a cluster. Such migration will not work with out-of-the-box virtualization solutions, because HPC jobs are tightly coupled and will often crash when attempting to migrate some of its processes. Due to space limitations, we omit these implementation details, but we will release them online along with the rest of the *Clavis2D* distribution.

In order to run cluster jobs inside virtual containers as opposed to bare-metal, we install the cluster monitoring tools (pbs_moms) and software libraries (OpenMPI, etc) inside the OpenVZ virtual containers instead of on the nodes themselves (Figure 5.3). The RM and the JS thus treat containers as compute nodes when performing cluster-wide queue management. That allows for a seamless integration with the JS: *Clavis2D* moves the containers across the nodes when the scheduling assignment changes, while the JS manages the job queue of the workload running inside the containers.

We create two containers per node on our experimental clusters. It allows to mitigate the contention effects while not consuming much of a disk space or network traffic during container migrations.

5.5 Clavis2D: the algorithm

The problem of energy-conscious space management is to find a balance between several conflicting goals: decreasing energy consumption, decreasing shared resource contention and improving network locality. In *Clavis2D*, we model this problem as a multi-objective optimization (MO).

There are many ways to solve MO problems [71, 26]. In *a priori* methods, preference information is first asked from the decision maker and then a solution best satisfying these preferences is found. *Scalarizing* is a popular approach to implementing a priori methods, in which a multiobjective optimization problem is formulated as combination of single-objective optimization problems (e.g., minimizing a weighted sum of goals) [27, 26]. The advantage of a priori methods is that they *automatically* find a small number of desired solutions by allowing to prioritize each goal through numerical weights. As such, these methods can be used by an automatic scheduling framework. The disadvantage is that it is often hard to determine appropriate values for the weights.

In contrast, *a posteriori* methods aim at producing a Pareto front of optimal solutions for the task: each one of these solutions is better than the rest with respect to at least one objective. These methods generate a large, diverse set of solutions that the administrator can choose from. The catch is that they require human involvement to pick the right solution every time.

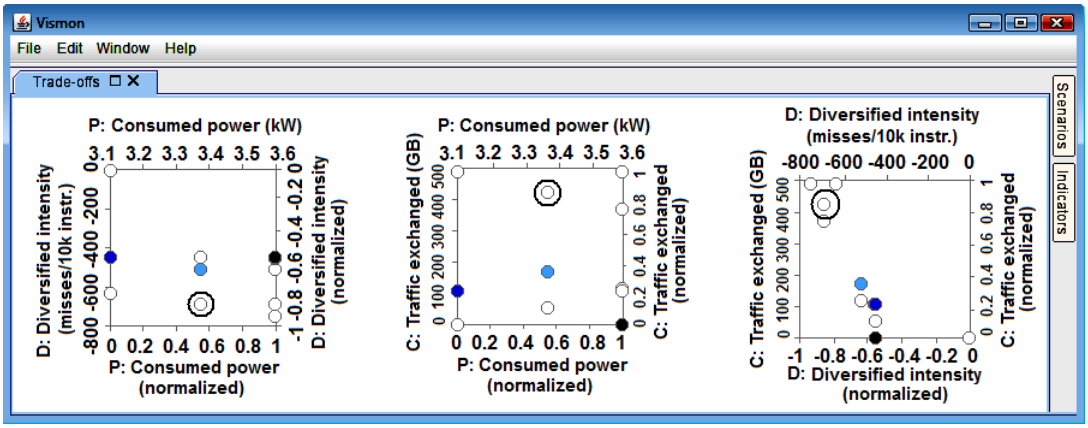
In *Clavis2D* we combine the two approaches: first, we use a popular a posteriori genetic algorithm (MOGA) to generate a Pareto front of possible solutions for a representative workload. These solutions are then visualized, allowing the cluster administrator to select schedules that are deemed acceptable for a particular cluster. This step needs to be performed only once. *In choosing particular schedules of a representation workload, the system administrator expresses his or her preferences with respect to balancing the optimization objectives.* The chosen schedules are then used to calibrate the weights for the automatic solver that finds solution for an arbitrary cluster workload. This is done through a well-studied linear scalarization. We next describe both optimization stages.

5.5.1 Stage 1: Calibrating Weights

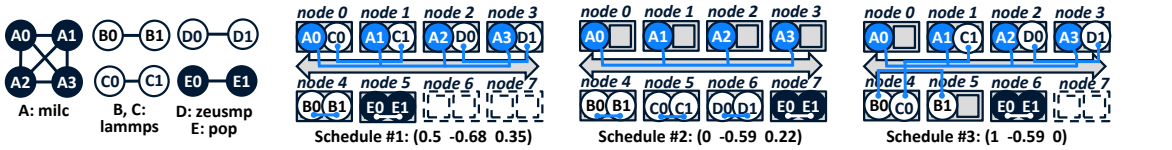
To find a Pareto front we use *gamultiobj* solver from Matlab. *Gamultiobj* finds minima of multiple functions using the Non-dominated Sorting Genetic Algorithm (NSGA-II) [71, 124]. In its search the algorithm favors solutions that are ”genetically better” (have minimum objective values as of yet). It then tries to crossover those solutions to find the better ones. Besides solutions with better ”genes”, the algorithm also favors solutions that can help increase the diversity of the population (the spread in goal values) even if they have a lower fitness value, as it is important to maintain the diversity of population for convergence to an optimal solution. We used the custom population type for the solver and adopted a partially matched crossover and mutation operators to our problem.

We visualize the Pareto front using a tool called *Vismon*. Vismon was created to facilitate analysis of trade-offs in decisions related to fisheries [58], and we adopted its source code to our domain. The Pareto front that resulted from running *gamultiobj* on our 8-node *Parapluie* cluster is shown

in Figure 5.4. Every schedule is characterized by its power consumption, communication overhead and contention. Hence, to visualize these three properties in 2D we need three $({}_3C_2)$ screens, each one showing how each schedule faired for each pair of objectives. The axes show the objectives and the dots represent the schedules. More formally, the objectives are defined as follows:



(a) Vismon trade-offs screen (the chosen solutions are colored).



(b) A workload and three of its schedules chosen by the system administrator (the collocated devils are in dark).

Figure 5.4: Choosing solutions from Pareto front with Vismon.

P: The total power consumption of the schedule. The lower this value for the schedule, the more energy efficient the schedule is.

D: *Diversified intensity* – a cumulative difference between container memory intensities on each node. Higher values are better as bigger differences in memory intensities among collocated containers contributes to low shared resource contention on a node (i.e. devils tend to be collocated with turtles). For a better schedule, this goal needs to be maximized, hence we use its negation.

C: Total traffic exchanged via a slow link. Lower values identify robust schedules with low communication overhead.

The last two objectives are subject to a constraint: in order to get the performance boost from contention or slow communication decrease, we need to provide it for all the containers within a job. For example, we need to isolate all the devils within a job from other devils to provide a performance boost. If there is at least one devil container within a job that is collocated with another

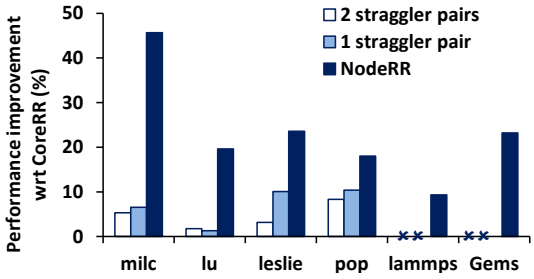


Figure 5.5: Relative comparison of CoreRR and assignments with varying degree of contention.

devil, there will likely be no performance benefit for the job. This is known as the *straggler's problem*: the lagging container slows down the entire job [128]. It is illustrated in Figure 5.5, where the respective devil jobs were running on 8 containers. We compared the following cases relative to CoreRR (4 nodes, 2 devils per node): *2 straggler pairs* (4 nodes, 1 devil per node and 2 nodes, 2 devils per node), *1 straggler pair* (6 nodes, 1 devil per node and 1 node, 2 devils per node) and NodeRR (8 nodes, 1 devil per node). When there is at least one lagging pair, the performance benefit from contention mitigation is very limited, sometimes jobs even hang due to divergence of running times in their components.

Armed with the information presented by Vismon, the system administrator must pick the schedules (by clicking the points on the screens) that best express his or her preferences with respect to balancing among the objectives. Here the administrator is guided by the knowledge of performance of representative workloads on his or her cluster. In our case, we assume that the administrator has seen the data similar to what we presented in Section 5.4, and so he or she knows that (a) contention can be a serious problem, (b) communication overhead is typically not large with the exception of jobs exchanging massive quantities of traffic. Also, for the sake of the example, we assume that in this case the administrator is a more concerned with performance than energy.

With these goals and information in mind, the administrator is to pick at least three acceptable schedules on each screen in Figure 5.4(a). The colored points are those selected by the administrator. Figure 5.4(b) shows the actual job placement represented by those points. Notice how the administrator chose solutions that: (a) cover the entire spectra for the power objective; (b) provide consistently low values for the contention factor and (c) have the network traffic that varies, but is not exceedingly high. The administrator avoids some points, like the one circled on Figure 5.4(a), that while resulting in low contention also gives us high communication overhead.

Once the acceptable schedules are chosen, *Clavis2D* proceeds with calibrating the weights for

their later use in the weighted sum minimization. The task is to find numerical values for α, β and γ from a system of three linear equations of the form:

$$\alpha P_n - \beta D_n + \gamma C_n = F \tag{5.1}$$

where F is the objective function that is to be minimized, P_n, D_n and C_n are normalized goal values from schedules #1-3. The result of solving this system for the values shown on Figure 5.4(b) with *mldivide* Matlab operator is as follows: $0.15 \times F$, $1.94 \times F$ and $0.68 \times F$. For the normalized goals only the relative value of the weights matter [41], so we use 0.15, 1.94 and 0.68 as our weights representing relative importance of each goal on *Parapluie* cluster.

5.5.2 Stage 2: Approximating the Optimal Schedule

Solver2D is the component of *Clavis2D* that computes the approximation to the optimal schedule using a popular open-source solver called Choco [171]. Finding a scheduling assignment is similar to the bin-packing problem. Here we also need to pack items (containers) into bins (nodes) so that the size of the items put into bins does not exceed the bin's capacity (core count). Our problem complicates bin-packing in that the items that we are packing have two additional attributes (devil or not, communicates via a slow link or not), and we need to account for these attributes while packing the bins.

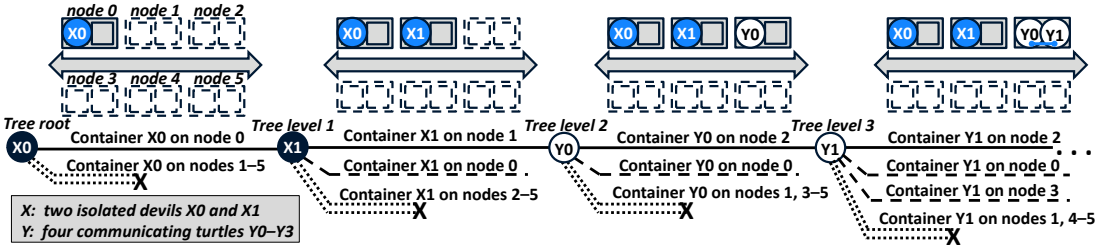


Figure 5.6: Example of a Branch-and-Bound enumeration search tree created by *Solver2D*.

Bin-packing is NP-hard. Choco looks for an optimal solution by creating and then traversing an *enumeration search tree* (Figure 5.6). On each tree level we evaluate the placement of a particular container. The branches are possible nodes that this container can be placed on. Traversing this tree can take a very long time, many hours and even days, unless the number of nodes and containers is very small. We compute scheduling decisions online, so we only have a few *minutes* to complete the traversal. If we terminate the traversal early, which is a common approach, we are unlikely to receive a good approximation for the optimal solution given these severe time constraints. To address this

problem, we use *domain-specific knowledge to eliminate the parts of the tree that are unlikely to contain good solutions*.

There are two ways in which we can eliminate parts of the traversal: *custom branching* and *custom bounding*. A generic solver will create as many branches in the tree as necessary to explore *all* assignments of containers to nodes. However, from our point of view, not all of these assignments are interesting to consider. To eliminate useless assignments, we create a ***custom branching strategy***, described below. Another problem with a generic solver is that it will traverse *every* branch it creates. However, by looking at the initial placement of containers that was done prior to reaching a particular branching decision, we can sometimes already predict that following that branch any further will not result in a better scheduling assignment than the incumbent (the best assignment found so far). In that case, it is better to skip that branch altogether, pruning a large part of the tree. In order to predict whether a particular branch is worth skipping we design several ***custom bounding functions*** that we also describe below.

1) *Custom branching.* For our scheduling task, it is redundant to evaluate all possible assignments of containers to nodes. In constraint programming this is often referred to as symmetry breaking. For example, suppose that we need to decide possible server values for a container Y1 from Figure 5.6 which is classified as a communicating turtle. There are already 3 containers instantiated before that one in the tree: one is a turtle Y0 communicating with Y1 and the rest two are non-communicating devils X0 and X1. All three previously assigned containers are alone on their respective servers. Additionally, three servers are currently idle – no container is assigned to them. So, when deciding how to branch Y1, we only need to take 3 branches: one idle server, one server with a communicating turtle and one server with a devil. We do not explore other branches: the schedules they offer will not result in a different weighted sum outcome and hence can be pruned. Such are the dotted branches in Figure 5.6. We call branches that need to be evaluated *fundamentally different*. They are pictured as dashed and solid lines in Figure 5.6 (the solid lines mark the currently evaluated solution).

In a custom branching strategy we also decide what branches to evaluate first. This is called *branch prioritization* and it is important when the weights for each objective component in Equation 5.1 are different. For instance, exploring branches which result in a separated devils before branches that lead to collocated devils could give a better solution faster if the contention is more important than power. Figure 5.6 illustrates branch prioritization for $\gamma > \beta > \alpha$.

2) *Naive bounding function.* The idea behind the *Naive* bounding function, is that we estimate each addend from Equation 5.1 independently of other addends. For example, to calculate the

total power consumption, we assume that all the containers are packed on as few nodes as possible regardless of their contention class or communication patterns. Similarly, for the diversified intensity we assume that each "devilish" container is put on a node by itself, unless no idle nodes are present. To estimate the total amount of traffic exchanged over slow links we assume that the small jobs can fit within a node and that off-rack traffic is only experienced by the jobs that cannot fit within a rack.

3) *Elaborate bounding function.* The naive bounding function is not very precise in its estimate: in the real world, the three goals are interdependent. For example, decreasing the power consumption usually increases the number of collocated devils. The *Elaborate* bounding function is based on a more strict simplification according to which the addends from Equation 5.1 are estimated from a potential workload placement. Let us return to the example from Figure 5.6. We see that in the path currently taken by the solver (the solid lines), four containers (X0, X1, Y0, Y1) are already instantiated. To estimate the best possible F that can be obtained in the subtree stemming from Y1, we first calculate the possible range for the number of collocated turtle pairs tt and the number of collocated devil-turtle pairs dt that can be formed with yet unassigned containers Y2 and Y3. The number of collocated devil pairs dd for this problem is determined by tt and dt, so each pair (tt, dt) describes how the unassigned containers of a particular class are placed with respect to other containers. In our case, $tt \in \{0, 1\}$ (both Y2 and Y3 are turtles and the rest of the turtles Y0 and Y1 are already together); and $dt \in \{0, 1, 2\}$ (because the two already assigned devils X0 and X1 can share a node with a turtle).

We then iterate through the potential workload placements represented by pairs (tt, dt) and estimate the three goals for each of the pairs. The estimations are not independent anymore, but are based on a particular class distribution within the cluster, which makes them more precise. The best estimated F across all iterations is the value returned by the elaborate function. To make the calculation fast, we process each iteration in its own thread, which takes advantage of the multiple CPUs and works very well in practice.

4) *Elaborate fathoming.* The bounding function in each bud node estimates the best future values for addends from (1). Fathoming is a branch-and-bound technique that piggybacks on these calculations and attempts to greedily find a good schedule that is in accordance with the bounding predictions. This is done as follows. We know that the elaborate bounding function predicts the number of collocated turtles tt in the best schedule. Knowing this, the fathoming generates a greedy schedule by taking two random unassigned turtles, placing them on an idle node and then removing those turtles from further consideration. After this action, the value of tt is decremented. Fathoming proceeds until $tt = 0$ or no more turtles are available, after which devil-turtle pairs are taken and so

on. Once the fathoming schedule is formed, the solver compares it to the incumbent: if the fathom is better than the incumbent, the fathom replaces incumbent.

5.5.3 Solver efficiency

Row	Features enabled:							Performance (up to 128 containers per job):			
	Best Random	Custom branching?	Naïve bounding function?	Elaborate bounding function?	Elaborate fathoming?	Prune top % branches?	Time to solve, sec	Weighted sum relative to the best one	Average tree nodes evaluated (solutions tried for random)	Average fraction of tree nodes pruned	Average fraction of tree levels explored
0	yes						60	3.14x	505078		
1		no	no	no	no	no	60	5.18x	338789	0.00	0.00
2		yes	no	no	no	no	60	1.29x	663692	0.00	0.01
3		yes	yes			no	60	1.29x	3600690	0.41	0.04
4		yes		yes	no	no	60	1.23x	2267173	0.81	0.67
5		yes		yes	yes	no	60	1x	376533	0.80	0.93
6		yes		yes	yes	yes	60	1.03x	58055	0.88	0.96
7		yes		yes	yes	yes	600	0.99x	6934349	0.89	0.97
8		yes		yes	yes	yes	6000	0.99x	71722876	0.90	0.97

Table 5.3: Solver evaluation (custom branching strategy).

Table 5.3 provides the evaluation data for our customized solver. The average results are shown for more than 5,000 runs of the solver. In each run, we terminated the search after 60 seconds and obtained the best weighted sum found so far. On several occasions we experimented with longer running times (600 and 6000 seconds) to see if this will give us better solutions. We varied the custom features enabled in the solver (from Section 5.5.2) to evaluate their impact. We also varied the weights assigned to each objective from 1 to 10 with the increment of 1, i.e. we minimized the weighted sum with the weights set to $\{1\ 1\ 1\}$, $\{1\ 1\ 2\}$, $\{1\ 2\ 1\}$, $\{1\ 2\ 2\}$ and so on. We used memory intensity and communication patterns of SPEC MPI and SPEC HEP applications. We configured the workload to have 8 jobs with up to 128 containers per job. We configured the number of nodes to be the same as the maximum number of containers, hence 1024 nodes at the maximum. The number of nodes and containers was sufficiently large that it was not feasible to find the truly optimal weighted sum within a feasible period of time. So to evaluate the quality of the solution we simply compare the weighted sum found by each solver to the best one found across all solver runs.

Row 0 shows the results for a simple solver that randomly searches for a solution and upon completion gives the best one it could find. We then evaluate the solutions with progressively more optimization features (rows 1–5). The metrics to pay attention to are: *weighted sum relative to the best* (smaller is better) and *average fraction of tree levels explored* (higher is better).

The results suggest that the proposed optimization features are largely beneficial in terms of either approaching the best weighted sum or in the percentage of the problem space explored. The most efficient solver is the one that uses all the optimization features and the most sophisticated

bounding function (row 5). This solver reduces the weighted sum by 3× on average relative to the random solver and explores 93% of the search tree.

In row 6 we show the solver that prunes more than the bounding function suggests: now a branch is pruned if the estimated bounding function value is greater than the incumbent multiplied by a coefficient 0.99. This further increases the number of tree levels explored by 3%, but does not correspond to a better weighted sum. This means that, if a solution with a better weighted sum exists within these 3% of the nodes, it differs from the best weighted sum by no more than 1%. We also note that, due to our branch prioritization, it is unlikely that the best solution is located at the end of the tree traversal. Rows 7 and 8 provide results for longer solver executions. As can be seen, longer execution times only marginally improve the solution, so we stick with 60-second executions, which enables us to re-evaluate scheduling assignments every 60 seconds.

5.6 Evaluation

5.6.1 Experimental results on OpenVZ overhead

In this Section we demonstrate what performance benefits *Clavis2D* provides on Systems cluster. **Systems** is a proof-of-concept cluster of our university lab. The servers that comprise the cluster have the following hardware configuration:

Two Dell-Poweredge-R805 (AMD Opteron 2435 Istanbul) servers have twelve cores placed on two chips. Each chip has a 6MB cache shared by its six cores. It is a NUMA system: each CPU has an associated 16 GB memory block (32 GB in total). Each server had a single 70 GB SCSI hard drive. We denote these machines as nodes X and Y.

One Dell-Poweredge-R905 (AMD Opteron 8435 Istanbul) server has 24 cores placed on four chips. Each chip has a 5 MB L3 cache shared by its six cores. Each core also has a private unified L2 cache and private L1 instruction and data caches. It is a NUMA system: each CPU has an associated 16 GB memory block (64 GB in total). The server was configured with a single 76 GB SCSI hard drive. We treat this machine as two 12 core nodes with 2 NUMA domains each. 1GbE network is used as an interconnect between all the nodes.

First, we performed the experiments to see if DINO contention-aware algorithm provides any benefits to the MPI cluster workloads when scheduling on the node level. We run all the MPI jobs with 24 ranks each. Series A on Figure 5.7 shows the average performance improvement for devils and turtles across different runs. As can be seen, DINO outperforms Linux Default for most of the applications tested.

Next, we wanted to demonstrate the benefits from the contention-aware cross-node scheduling. Series B shows that, for all the devils, there is a benefit in scheduling a job with 6 ranks per node (and taking twice as many nodes) rather than scheduling it with 12 ranks per node. The challenge here is to detect which ranks are devils and then spread only those apart. Series C–F shows the performance improvement of a solution where the job class is detected online and devils are spread across the nodes with OpenVZ container migration. We make the following conclusions from these results:

1) OpenVZ containers incur little overhead on its own (Series C). There can be performance degradation after migration though, if contention-unaware algorithm (like Linux Default) is used on the cluster nodes (Series E). Our contention-aware DINO algorithm (Series F), on the other hand, is able to provide solution, which is close to the best separated one of Series B (when ranks are started on different nodes from the start and no migration occurred).

2) Separating devil processes across nodes is beneficial for the overall performance of an MPI job (Series F). The influence of such separation is twofold: it relieves pressure on shared resources within a node, but it also increases pressure on the network. Series F results state that, even for a slow 1GbE connection and migration overhead included, performance benefit of reducing shared resource contention outweighs the interconnect slowdown for all devils. Those jobs that cannot benefit from separation (turtles) or those for whom migration penalty outweighs performance benefits (semi-devils) can be dynamically detected by the framework and used for load balancing (to plug the idle slots on partially loaded nodes).

3) It is generally very hard to precisely measure what is the impact of each of the two conflicting factors on the overall performance of a job, since they interact in complex ways. Series D provides an estimate of the sensitivity of our workloads to exchanging data via the 1GbE interconnect. We obtained it by increasing the number of jobs running on the node: instead of 12 ranks of a single job, the ranks of two jobs were present on the node (6 ranks of each job). Both jobs were different instances of the same program. Such experimental configuration keeps the shared resource contention on the same level (12 ranks per node), but it increases the pressure on the network. Series D shows that most of the programs we considered do not take big hit when using the network and so the performance benefit of Series F is roughly due to the relief in shared resource contention.

5.6.2 Cluster experimental configuration

Table 5.4 shows the two clusters used for our experiments. Both clusters used 1GbE and InfiniBand networks for connecting the nodes. We use Maui as our cluster scheduler, Torque as a resource

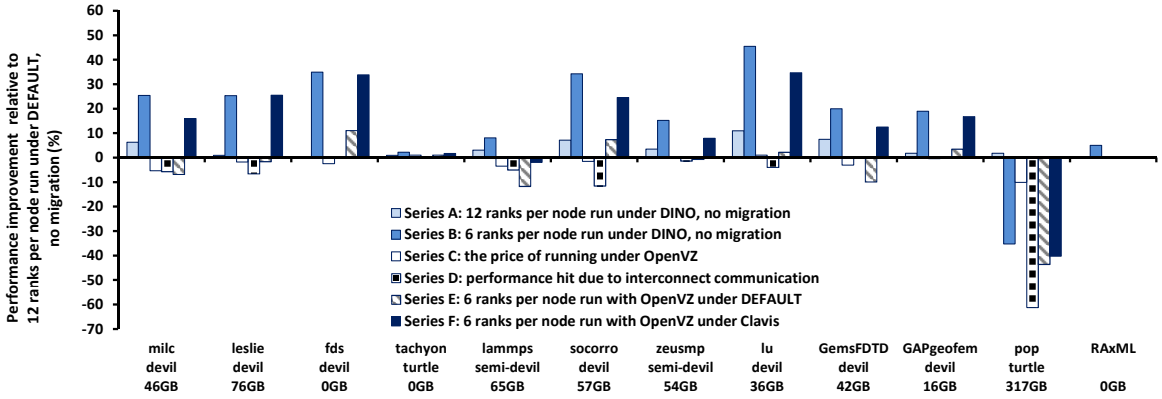


Figure 5.7: Results of the contention-aware experiments on Systems cluster.

manager, SPEC MPI2007 and SPEC HEP as our workload. All nodes were running Linux 2.6.32.

Cluster	Nodes	CPU #, make, model	Cores/CPU	RAM
Parapluie (Grid 5000) [57]	8	4 AMD 6164 HE (Magny-Cours)	6	48GB
India (FutureGrid) [82]	16	2 Intel X5570 (Nehalem)	4	24GB

Table 5.4: Clusters used for experiments.

1. **Parapluie** cluster hosted by Grid5000. The nodes have the following hardware configuration: HP Proliant DL165 G7 (AMD Opteron 6164 HE Magny-Cours) servers have 24 cores placed on four chips. Each chip has a 5 MB L3 cache shared by its six cores. Each core also has a private unified L2 cache and private L1 instruction and data caches. It is a NUMA system: each chip has an associated 12 GB memory block, for a total of 48 GB main memory. The servers were configured with a single SCSI hard drive. The nodes are connected through 1GbE and InfiniBand networks. We measure power with *ipmi* package. We conduct experiments on 192 cores (8 node) cluster that we create within the Parapluie infrastructure.[3]

2. **India** cluster hosted by FutureGrid. The nodes have the following hardware configuration: IBM iDataPlex (Intel Xeon X5570 Nehalem) servers have 8 cores placed on two chips. Each chip has an 8 MB L3 cache shared by its four cores. Each core also has a private unified L2 cache and

[3]Setting up the experimental testbed for cluster research is challenging, because it is hard to get root access to many industry-standard servers for an academic researcher as the facilities that do have the necessary resources are usually reluctant to give such access to their fine-tuned infrastructure. We will share the instructions and scripts on how to deploy a research-friendly cluster testbed using FutureGrid and Grid5000 resources.

private L1 instruction and data caches. It is a NUMA system: each CPU has an associated 12 GB memory block, for a total of 24 GB main memory. The servers were configured with a single SCSI hard drive. The nodes are connected through 1GbE and InfiniBand networks. We measure power with *ipmi* package. We conduct experiments on 128 cores (16 node) cluster that we create within the India infrastructure.

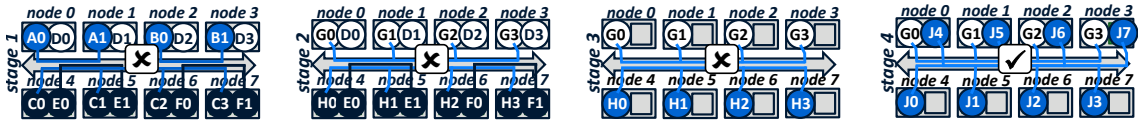
We compare performance and energy consumption under three job schedulers: *Baseline*, *Expert* and *Clavis2D*. *Baseline* is the state-of-the-art scheduler used in HPC clusters. With *Baseline* the cluster is configured once (or very rarely) to use either CoreRR or NodeRR policy. The policy is chosen by the system administrator based on some knowledge about performance of representative workloads on the particular cluster. For instance, for the Parapluie cluster, the administrator would use performance data similar to that shown in Figure 5.2 to decide that NodeRR would be a more favorable policy. Given the limitation of state-of-the-art cluster schedulers, we expect that *Baseline* would often create schedules that lead to poor performance and energy waste. Therefore, *Baseline* should be considered as the lower bound.

The *Expert* scheduler is at the opposite end of the spectrum. Here we assume that job placement is performed by a knowledgeable system administrator who can peek one step ahead into the job queue and who is willing to manually place jobs to nodes based on the knowledge of their performance characteristics (contention-intensive or not, communication-intensive or not). Since typical HPC clusters do not use virtualization, we assume that the administrator is unable to migrate jobs after they began execution. Nevertheless, clever placement of jobs prior to beginning of execution gives an important advantage, so we expect to see very good performance and energy with the *Expert* scheduler. At the same time, it is not feasible to expect job scheduling to be performed manually by a human, so *Expert* is an idealized, unrealistic scenario.

To mimic a typical HPC environment, we create a job queue inside the cluster. Jobs are scheduled to run in the order of their arrival. Some jobs have dependencies: they cannot run before the job they depend on completes. In the figures that follow we will show the placement of jobs in the cluster across several stages. A stage begins when a new set of jobs is assigned to run and ends when those jobs complete.

5.6.3 Experimental results on Parapluie

Figure 5.8 shows the placement of jobs under *Baseline*.[4] There are four stages, each lasting 6 hours, giving us a day-long cluster-wide experiment. Job containers are shown in different colors to indicate their memory intensity. Turtle containers are white. Devil containers collocated with other devil containers (suffering from contention!) are shown in black. Devil containers collocated with turtle containers or running alone (no contention) are shown in blue. Containers that exchange lots of network traffic are connected by a solid line.



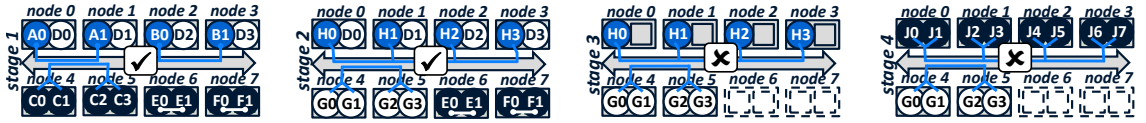
(a) At the start, the cluster is fully loaded with jobs A–F.

(b) Jobs G and H can only be started after A–C.

(c) Jobs D–F have finished.

(d) Job J can only be started after job H.

Figure 5.8: Baseline NodeRR experiments on Parapluie cluster.



(a) Expert manually consolidates E and F, since they are hurt when accessing cluster network.

(b) D is put together with H which makes H run faster.

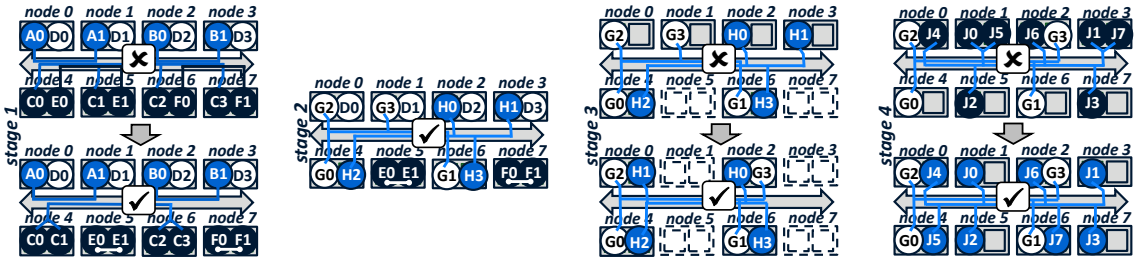
(c) There is no way for an expert to collocate H and G after D has finished.

(d) Expert does not have enough space to fully spread J apart, ends up with a straggler's problem.

Figure 5.9: Expert on Parapluie cluster, requires offline profiling and regular human involvement.

Baseline can stumble into good schedules, e.g., when it collocates the devil containers of A and B with the turtle containers of D (stage 1, Figure 5.8(a)) or when it spreads the devil containers of J apart and fills in the gaps with the turtle containers of H (stage 4, Figure 5.8(d)). At the same time, there are many situations, when it enforces a bad schedule. For example, separating the containers of communication-intensive E and F hurts their performance (stages 1 and 2). Collocating A and B with D results in a later collocation of two turtles (D and G), which is wasteful considering that those turtles could be collocated with the devil H instead (stage 2), reducing resource contention for H. Finally, lack of virtualization and online workload profiling makes it impossible to detect that the jobs G and H in stage 3 are complimentary and could be collocated to save power.

[4]The videos with more scenarios are available online at [10] thanks to Jessica Jiang.



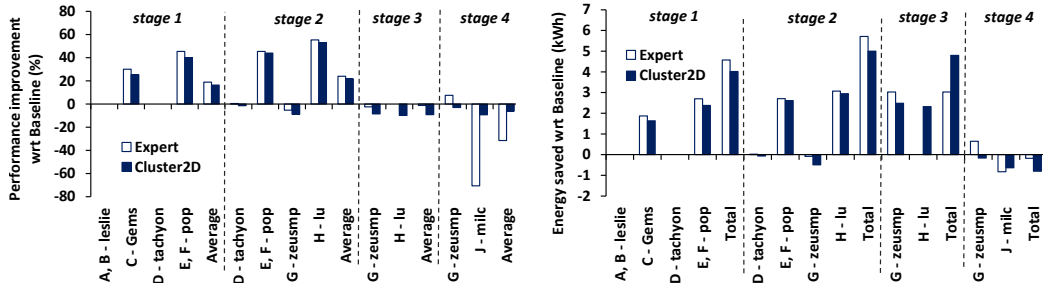
(a) *Clavis2D* dynamically consolidates E and F.

(b) *Clavis2D* collocates H with D and G to make H run faster.

(c) *Clavis2D* dynamically consolidates G and H on fewer nodes to save power.

(d) *Clavis2D* rearranges J to make it run faster, G containers plug the holes.

Figure 5.10: *Clavis2D* automatic experiments on Parapluie cluster.



(a) The performance results of Parapluie runs.

(b) The energy measurements from Parapluie runs.

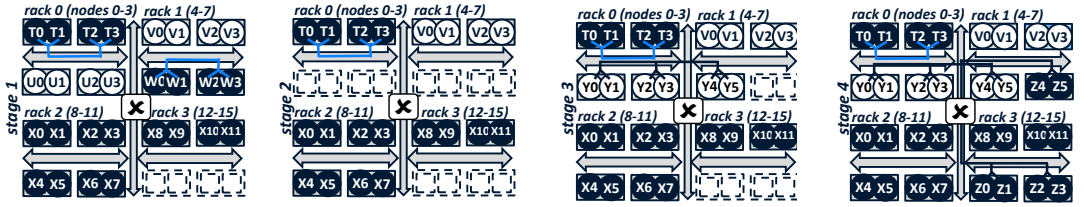
Figure 5.11: Experimental results on Parapluie cluster.

The *Expert* scheduler (Figure 5.9) is able to detect that E and F need to be consolidated (stage 1, Figure 5.9(a)). It also makes sure that devil H is placed with turtle D upon its launch and that the containers of G stay together and do not mix with E and F even though this could reduce contention (stage 2, Figure 5.9(b)). This results in a significant (up to 60%) speed up for the job execution on stages 1 and 2 (Figure 5.11(a)). Total energy savings associated with the jobs completed ahead of time reach 6 kWh (Figure 5.11(b)). However, good decisions made by *Expert* initially, affect the later schedules. In stage 3, the well-placed jobs D, E and F have terminated, leaving a poor schedule in which G and H consume more power than they have to. In stage 4, the previously placed G prevents us from spread J to use more nodes, thus causing the straggler's problem and resulting in performance loss of more than 70% (Figure 5.11(a)).

Clavis2D is able to find reasonable solutions at every stage automatically (Figure 5.10). If the initial placement is poor *Clavis2D* migrates the container to achieve a better mapping (hence we show the *before* and *after* migration placement in Figure 5.10). Migration is performed without

affecting the Maui job queue. Besides stages 1 and 2, *Clavis2D* is also able to reach the most energy-efficient solution in stage 3 (Figure 5.10(c)), almost 2 kWh better than that of *Expert* in total. In stage 4, it is able to prevent significant performance loss by dynamically reshuffling containers of F and H across the cluster (Figure 5.10(d)). Interestingly, the energy consumption of the schedule found by *Clavis2D* in stage 4 does not differ from that of the *Expert*, even though *Expert* uses fewer nodes. That is because *Expert* severely slows down J (by more than 60%) making it run longer and consume more energy.

5.6.4 Experimental results on India cluster



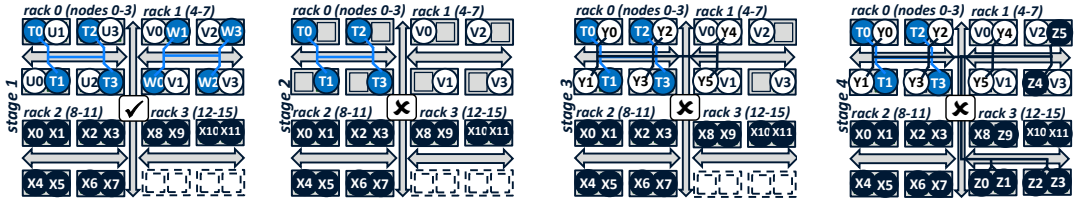
(a) At the start, the cluster is loaded with jobs T–X.

(b) Jobs U and W have finished.

(c) Job Y's advanced reservation has begun.

(d) Job Z's advanced reservation has begun.

Figure 5.12: Baseline CoreRR experiments on India cluster.



(a) Expert manually shuffles containers of T, U and V, W to boost T and W.

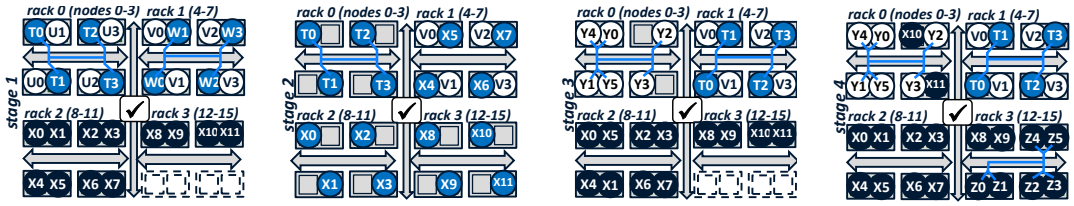
(b) Expert is unable to spread containers of X apart to increase performance.

(c) The job Y now has to suffer remote communication.

(d) Expert is unable to reshuffle containers of X and Z to prevent remote communication for Z.

Figure 5.13: Expert on India cluster, requires offline profiling and regular human involvement.

In the India cluster, nodes are divided among 4 racks that are connected via a slow link. We use the *netem* kernel module to induce latency between the racks as described in [29], thus emulating a geographically distributed system. In such a cluster, the administrator is primarily concerned with avoiding the communication overhead of accessing the remote racks. We also assume that the power is as important as performance, and so CoreRR is chosen for the *Baseline* configuration.



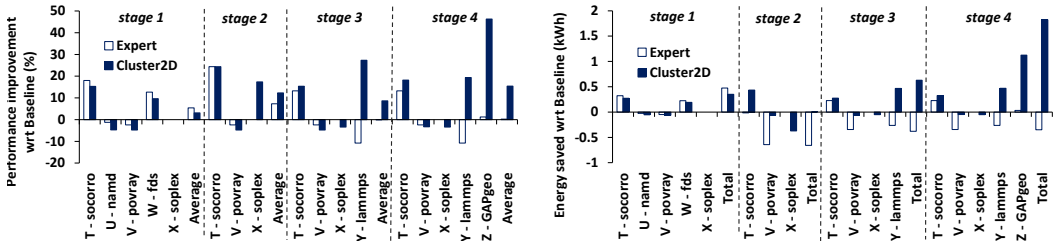
(a) *Clavis2D* dynamically collocates T, U and V, W.

(b) *Clavis2D* increases X's performance before reservations of Y and Z.

(c) *Clavis2D* puts Y within a rack to prevent slower remote communication.

(d) *Clavis2D* reshuffles containers of X and Z to avoid remote traffic for Z.

Figure 5.14: *Clavis2D* automatic experiments on India (initial placement is omitted due to space limitations).



(a) The performance results of India cluster runs.

(b) The energy measurements from India cluster runs.

Figure 5.15: Experimental results on India cluster.

It is well known that servers consume a significant amount of power when they are idle, often more than 50% of peak power [25]. This is indeed the case on both Parapluie and India. To show how *Clavis2D* adapts its decisions to changes in hardware, we here assume that the administrator enabled dynamic power management algorithms on the nodes [167] in order to reduce their idle power. We assume that as a result the idle power consumption on India has been reduced from 50% of the peak power, to only 25%.

Job placement under *Baseline* is shown in Figure 5.12. The job queue in these runs is comprised of 7 jobs, T through Z, three of which (U, V and X) are SPEC HEP jobs. SPEC HEP is the official CPU performance benchmark used on clusters processing LHC data [36]. It is very common for LHC workloads to utilize huge parts of the cluster, so the biggest job X (soplex) in our testbed requires 37% of the computing facilities (12 containers). The sheer size of this job makes it difficult to improve its performance in stage 1. The devil jobs T and W, on the other hand, can receive more than 10% boost if we pair them with turtles U and V. Both *Expert* and *Clavis2D* do just that (Figures 5.13(a) and 5.14(a)).

Despite our cluster being nearly fully utilized most of the time, in stage 2 it happens to have a

pocket of six idle nodes (Figure 5.12(b)). This is because jobs Y and Z that should run next have advanced reservations and are waiting for their time slot. This opportunity is seized by *Clavis2D* to spread the containers of X on different nodes (Figure 5.14(b)), giving X a 20% boost in performance (Figure 5.15). By letting X use more nodes, *Clavis2D* sacrifices energy, losing 0.4 kWh relative to *Baseline*. The loss in energy is rather small due to using dynamic power management, and *Clavis2D* is aware of that, because its weights were calibrated using the power data from the "energy-savvy" India cluster. If a similar situation had occurred on the Parapluie cluster, where idle power consumption is much larger, *Clavis2D* would have consolidated X, because power loss outweighs performance benefits in that case.

The jobs Y and Z on stages 3 and 4 arrive at the time when every rack on the cluster is partially loaded. As a result, neither *Baseline* nor *Expert* are able to place them entirely within a rack, triggering slow cross-rack communication. Only *Clavis2D* is able to avoid this by reshuffling containers of already running jobs, thus freeing some rack space for the newcomers. Because of that, *Clavis2D* outperforms *Baseline* and *Expert* in both performance and energy savings (Figure 5.15).

5.7 Conclusion

We have shown that *Clavis2D* saves energy and improves performance on computing clusters, because in scheduling jobs it balances three important goals: minimizing contention for shared resources, reducing communication distance and avoiding resource idleness. Underlying *Clavis2D*is a custom multi-objective solver that can be enhanced to address additional performance and energy-related goals. Although we evaluated *Clavis2D* on typical scientific computing clusters, we believe it can also deliver benefits in other computing infrastructures, such as Hadoop clusters.

Part IV

Epilogue

Chapter 6

Related work

6.1 Introduction

This chapter is dedicated to the discussion of the work related to my thesis. Section 6.2 starts by discussing the topic of contention-aware scheduling for multicore systems. The previous work primarily focused on the problem of *cache contention* since this was assumed to be the main if not the only cause of performance degradation [80, 116, 169, 168, 72]. In this context cache contention refers to the effect when an application is suffering extra cache misses because its co-runners (threads running on cores that share the LLC) bring their own data into the LLC evicting the data of others. It has been documented in previous studies [66, 125, 127, 155, 165, 169] that the execution time of a thread can vary greatly depending on which threads run on the other cores of the same LLC cache. Methods such as utility cache partitioning (UCP) [155] and page coloring [66, 169, 191] were devised to mitigate cache contention.

Section 6.3 is devoted to the research on NUMA-related system optimizations, which is rich and dates back many years. Many research efforts addressed efficient co-location of the computation and related memory on the same node [123, 59, 120, 168, 42, 67]. More ambitious proposals aimed to holistically redesign the operating system to dovetail with NUMA architectures [85, 152, 86, 175, 118]. None of the previous efforts, however, addressed shared resource contention in the context of NUMA systems and the associated challenges.

Our solution to the contention for CPU resources of the machine presented in Chapter 4 is not the only work conserving technique that exists for multicore systems. Other work improves server utilization by using weights to control resource assignment in work conserving mode [73, 185, 161, 83, 166, 126, 129]. In this work, we introduce a novel approach that fully loads the server with critical and non-critical loads. None of the previous work presented in Section 6.4 investigated

whether a work-conserving approach can be used to significantly increase resource utilization up to full server capacity while maintaining acceptable performance.

While there exists a plethora of work addressing time management of HPC clusters, asking how to manage job queues to achieve various performance and energy objectives [89, 113, 112, 167, 146, 124], none of it touched upon the subject of space management, which is crucial from energy-saving perspective and is focus of this chapter. The detailed discussion of work related to datacenter wide contention management issues is given in Sections 6.5 through 6.8.

6.2 Related work on memory hierarchy contention within a datacenter server

In the work on Utility Cache Partitioning [155], a custom hardware solution estimates each application's number of hits and misses for all possible number of ways allocated to the application in the cache (the technique is based on stack-distance profiles). The cache is then partitioned so as to minimize the number of cache misses for the co-running applications. UCP minimizes cache contention given a particular set of co-runners. Our solution, on the other hand, decides which co-runners to co-schedule so as to minimize contention. As such, our solution can be complementary to UCP and other solutions relying on cache partitioning [165].

Tam, Azimi, Soares and Stumm [169] similarly to several other researchers [66, 125, 127, 191] address cache contention via software-based cache partitioning. The cache is partitioned among applications using page coloring. Each application is reserved a portion of the cache, and the physical memory is allocated such that the application's cache lines map only into that reserved portion. The size of the allocated cache portion is determined based on the marginal utility of allocating additional cache lines for that application. Marginal utility is estimated via an application's reuse distance profile, which is very similar to a stack-distance profile and is approximated online using hardware counters [169]. Software cache partitioning, like hardware cache partitioning, is used to isolate workloads that hurt each other. While this solution delivers promising results, it has two important limitations: first of all, it requires non-trivial changes to the virtual memory system, a very complicated component of the OS. Second, it may require copying of physical memory if the application's cache portion must be reduced or reallocated. Given these limitations, it is desirable to explore options like scheduling, which are not subject to these drawbacks.

Moscibroda and Mutlu demonstrated the problem with shared resource contention in memory controllers [142]. They showed that shared memory controller contention on real dual-core and

quad-core systems can be a significant problem, causing largely unfair slowdowns, reduced system performance and long periods of service denial to some applications. They later addressed the problem by designing fair memory controllers that reduce and manage interference in the main memory subsystem [144, 145, 115]. Several researchers addressed shared resource contention due to pre-fetchers as well as in on-chip networks [69, 76, 92, 122]. The solutions proposed offer enhancements to certain parts of the memory hierarchy rather than target the shared resource contention problem in the whole system via scheduling. Hence, we see this work as complementary to ours.

Herdrich et al. proposed rate-based QoS techniques, which involved throttling the speed of the core in order to limit the effects of memory-intensive applications on its co-runners [96]. Rate-based QoS can be used in conjunction with scheduling to provide isolation from contention for high-priority applications.

Several prior studies investigated the design of cache-aware scheduling algorithms. Symbiotic Jobscheduling [164] is a method for co-scheduling threads on simultaneous multithreading processors (SMT) in a way that minimizes resource contention. This method could be adapted to co-schedule threads on single-threaded cores sharing caches. This method works by trying (or sampling) a large number of thread assignments and picking the ones with the best observed rate of instructions per cycle.

The drawback of this solution is that it requires a sampling phase during which the workload is not scheduled optimally. When the number of threads and cores is large, the number of samples that must be taken will be large as well.

Fedorova et. al designed a cache-aware scheduler that compensates threads that were hurt by cache contention by giving them extra CPU time [80, 79]. This algorithm accomplishes the effect of fair cache sharing, but it was not designed to improve overall performance.

Mars et. al [134] suggested to throttle down less important SPEC CPU applications that could hurt performance of the more important ones. In this work, we instead focus on scheduling techniques as means of mitigating the shared resource contention, i.e., we do not penalize the aggressive apps, but rather migrate them away from the sensitive programs in the memory hierarchy of the multicore server.

Tang et. [170] has shown that, depending on the co-runner, sharing LLC and FSB can have a significant impact. Based on an application's characteristics in terms of their resource usage when running alone, she suggested both a one-time and an adaptive learning strategies to mitigate the contention. In the latter case, the solution puts various thread-to-core mappings against each other to learn which mapping performs best. Our solution instead relies on an online profiling of the

applications and a dynamic, reactive rescheduling of the workload without exhaustive permutation of the possible mappings prior to making the decision.

The idea of distributing benchmarks with a high rate of off-chip requests across different shared caches was also proposed in earlier studies [116, 72]. The authors propose to reduce cache interference by spreading the intensive applications apart and co-scheduling them with non-intensive applications. Cache misses per cycle were used as the metric for intensity. Our idea of DI is similar, however we provide a more rigorous analysis of this idea and the reasons for its effectiveness, and also demonstrate that DI operates within a narrow margin of the optimal. The other papers did not provide the same analysis, so it was difficult to judge the quality of the algorithm. Further, previous work did not resolve the controversy between two approaches to model contention: Chandra demonstrated that reuse-distance models provide some of the most accurate measures of contention [64, 165], but this contradicted the hypothesis that the rate of off-chip requests can be a good heuristic for predicting contention. Our work reconciled these two approaches, explaining that Chandra's model is suitable for systems where cache contention is the dominant cause of performance degradation, but on real systems, where other contention factors are equally important, LLC miss rates turn out to be a very good heuristic for contention.

6.3 Related work on NUMA-awareness within a datacenter server

Research on NUMA-related optimizations to systems is rich and dates back many years. Many research efforts addressed efficient co-location of the computation and related memory on the same node [123, 59, 120, 168, 42, 67]. More ambitious proposals aimed to holistically redesign the operating system to dovetail with NUMA architectures [85, 152, 86, 175, 118]. None of the previous efforts, however, addressed shared resource contention in the context of NUMA systems and the associated challenges.

Li et al. in [123] introduced AMPS, an operating system scheduler for asymmetric multicore systems that supports NUMA architectures. AMPS implemented a NUMA-aware migration policy that can allow or deny thread migration requested by the scheduler. The authors used the *resident set size* of a thread in deciding whether or not the OS schedule is allowed to migrate thread to a different domain. If the migration overhead were expected to be high the migration would be disallowed. Our scheduler, instead of prohibiting migrations, detects which pages are being actively accessed and moves them as well as surrounding pages to the new domain.

LaRowe et al. [120] presented a dynamic multiple-copy policy placement and migration policy

for NUMA systems. The policy periodically reevaluates its memory placement decisions and allows multiple physical copies of a single virtual page. It supports both migration and replication with the choice between the two operations based on reference history. A directory-based invalidation scheme is used to ensure the coherence of replicated pages. The policy applies a freeze/defrost strategy: to determine when to defrost a frozen page and trigger reevaluation of its placement is based on both time and reference history of the page. The authors evaluate various fine-grained page migration and/or replication strategies, however, since their test machine only has one processor per NUMA node, they do not address contention. The strategies developed in this work could have been very useful for our contention aware scheduler if the inexpensive mechanisms that the authors used for detecting page accesses were available to us. Detailed page reference history is difficult to obtain without hardware support; obtaining it in software may cause overhead for some workloads.

Goglin et al. [88] developed an effective implementation of the *move_pages* system call in Linux, which allows the dynamic migration of large memory areas to be significantly faster than in previous versions of the OS. This work is integrated in Linux kernel 2.6.29 [88], which we use for our experiments. The *Next-touch* policy, also introduced in the paper to facilitate thread-data affinity, works as follows: the application marks pages that it will likely access in the future as *Migrate-on-next-touch* using a new parameter to the `madvise()` system call. The Linux kernel then ensures that the next access to these pages causes a special page fault resulting in the pages being migrated to their threads. The work provides developers with an opportunity to improve memory proximity for their programs. Our work, on the other hand, improves memory proximity by using hardware counters data available on every modern machine. No involvement from the developer is needed.

Linux kernel since 2.6.12 supports the *cpusets* mechanism and its ability to migrate the memory of the applications confined to the cpuset along with their threads to the new nodes if the parameters of a cpuset change. Schermerhorn et al. further extended the cpuset functionality by adding an automatic page migration mechanism to it [162]: if enabled, it migrates the memory of a thread within the cpuset nodes whenever the thread migrates to a core adjacent to a different node. Two options for the memory migration are possible. The first is a lazy migration, when the kernel attempts to unmap any anonymous pages in the process's page table. When the process subsequently touches any of these unmapped pages, the swap fault handler will use the "migrate-on-fault" mechanism to migrate the misplaced pages to the correct node. Lazy migration may be disabled, in which case, automigration will use direct, synchronous migration to pull all anonymous pages mapped by the process to new node. The efficiency of lazy automigration is comparable to our memory migration solution

based on IBS (we performed experiments to verify). However, automigration requires kernel modification (it is implemented as a collection of kernel patches), while our solution is implemented on user level. Cpuset mechanism needs explicit configuration from the system administrator and it does not perform contention management.

In [168] the authors group threads of the same application that are likely to share data onto neighbouring cores to minimize the costs of data sharing between them. They rely on several features of Performance Monitoring Unit unique to IBM Open-Power 720 PCs: the ability to monitor CPU stall breakdown charged to different microprocessor components and using the data sampling to track the sharing pattern between threads. The DINO algorithm introduced in our work complements [168] as it is designed to mitigate contention between applications. DINO provides sharing support by attempting to group threads of the same application and their memory on the same NUMA node, but as long as co-scheduling multiple threads of the same application does not contradict with a contention-aware schedule. In order to develop a more precise metric that assesses the effects of performance degradation versus the benefits from co-scheduling, we would need stronger hardware support, such as that available on IBM Open-Power 720 PCs or on the newest Nehalem systems (as demonstrated by the member of our team [110]).

The VMware ESX hypervisor supports NUMA load balancing and automatic page migration for its virtual machines (VMs) in commercial systems [42]. ESX Server 2 assigns each virtual machine a home node on whose processors a VM is allowed to run and its newly-allocated memory comes from the home node as well. Periodically, a special rebalancer module selects a VM and changes its home node to the least-loaded node. In our work we do not consider load balancing. Instead, we make thread migration decisions based on shared resource contention. To eliminate possible remote access penalties associated with accessing the memory on the old node, ESX Server 2 performs page migration from the virtual machine's original node to its new home node. ESX selects migration candidates based on finding hot remotely-accessed memory from page faults. The DINO scheduler, on the other hand, identifies hot pages using Instruction-Based Sampling. No modification to the OS is required.

The SGI Origin 2000 system [67] implemented the following hardware-supported [121] mechanism for co-location of computation and memory. When the difference between remote and local accesses for a given memory page is greater than a tunable threshold, an interrupt is generated to inform the operating system that the physical memory page is suffering an excessive number of remote references and hence has to be migrated. Our solution to page migration is different in that it detects "hot" remotely accessed pages via Instruction-Based Sampling, and performs migration in

the context of a contention-aware scheduler.

In a series of papers [85] [152] [86] [175] the authors describe a novel operating system Tornado specifically designed for NUMA machines. The goal of this new OS is to provide data locality and application independence for OS objects thus minimizing penalties due to remote memory access in a NUMA system. The K42 [118] project, which is based on Tornado, is an open-source research operating system kernel that incorporates such innovative design principles like structuring the system using modular, object-oriented code (originally demonstrated in Tornado), designing the system to scale to very large shared-memory multiprocessors, avoiding centralized code paths and global locks and data structures and many more. K42 keeps physical memory close to where it is accessed. It uses large pages to reduce hardware and software costs of virtual memory. K42 project has resulted in many important contributions to Linux, on which our work relies. As a result, we were able to avoid deleterious effects of remote memory accesses without requiring changes to the applications or the operating system. We believe that our NUMA contention-aware scheduling approach that was demonstrated to work effectively in Linux can also be easily implemented in K42 with its inherent user-level implementation of kernel functionality and native performance monitoring infrastructure.

6.4 Related work on managing contention for CPU within a datacenter server

Most prior work in workload management uses some form of capping to distribute resources among collocated workloads in non-work conserving mode [62, 147, 192, 87, 183, 111, 106, 182, 141, 131, 89, 134, 150]. Examples are limiting CPU resource utilization in Xen with CPU caps or limiting network bandwidth through Linux Control Groups. With such approaches, non-critical workloads only have access to limited amounts of server resources. If the controller does not perform runtime adjustments of the caps, the utilization of the non-critical workloads will stay low even when resources are available, resulting in lower overall server utilization.

Other work improves server utilization by using weights to control resource assignment in work conserving mode. The work by Diao et al. prioritizes and dispatches incoming requests to service agents based on the SLA attainment target in a simulation testbed [73]. Others use existing techniques such as CPU shares, Xen CPU weights, or Linux real-time priorities to collocate interactive jobs [185, 161, 83] and mix them with batch jobs [166, 126, 129]. By controlling resource access priorities of collocated workloads, a controller is able to adjust to workload demand changes allowing applications to use resources when available. In this work, we introduce a novel approach that

fully loads the server with critical and non-critical loads. None of the previous work investigated whether a work-conserving approach can be used to significantly increase resource utilization up to full server capacity while maintaining acceptable performance. We have developed a working prototype and we show that it is feasible to achieve high server utilization with a detailed experimental study using real workloads. We leave the evaluation of financial and power saving benefits of full server utilization for future work.

Our solution works very well with traditional sizing tools [46, 130, 133, 159, 186, 47], which determine the optimal amount of critical workload and ensure that not too many critical jobs are placed on a system. Further, sizing tools can ensure that batch jobs get enough resources to avoid starvation. We then use reactive, weight-based workload management to maintain performance of the critical applications. Our weight-based management solution is complimentary to more sophisticated predictive controllers that consolidate workloads based on their resource profiles.

Cap-based techniques that enable existing middleware to manage mixed batch and transactional workloads have been previously proposed [63, 62]. A workload profiler monitors the nodes' resource utilization and estimates an average CPU requirement for each application request. The system then predicts the performance of the transactional application for a given allocation of CPU resources. The authors use simulation to demonstrate the benefits of their approach.

Bubble-up [133] tries to collocate workloads onto the same server that, according to prediction, will not conflict. A profiling run for each application is required for making the prediction. During the run, the application is collocated with a special 'bubble' that generates a profiling report. While the proposed approach works well for the examined Google workload, application profiling may be challenging in general, because data center workloads may be unknown in advance.

In order to address QoS requirements in Q-Clouds [147], VMs are first profiled in a solo run to determine the amount of resources needed to attain the desired QoS. The Q-Clouds scheduler then leaves a prescribed amount of unused resources on each server. The Q-Clouds implementation also dynamically adjusts VCPU caps of the VMs through the Hyper-V hypervisor.

Several hybrid provisioning approaches [192, 87, 183, 46] combine prediction and reaction while performing cap-based SLA management in a data center. They use historical trace analysis to predict future workload demands and to determine the amount of workload given to any particular server. In addition, they implement a reactive controller that handles unforeseen events.

The Active CoordinaTion (ACT) approach described in [111] relies on the notion of Class-of-Service, a multi-dimensional vector that expresses the resource expectations of Xen VMs. Based on these vectors it caps VMs in their resource usage. ACT uses historic VM behavior to infer future

trends.

Sandpiper [185] relocates VMs from overloaded to under-utilized nodes using a black-box approach that is OS and application agnostic and a gray-box approach that exploits OS and application-level statistics. Explicit SLA violations must be detected on a per-virtual server basis in the gray-box approach – a hotspot is flagged if the memory utilization, response time or request rate exceed an SLA threshold. Before resolving the hotspot via migrations, Sandpiper estimates the additional resource needs for overloaded VMs using a high percentile of the CPU and network bandwidth distribution as a proxy for the peak demand.

The creators of GreenSlot [89] present a parallel batch job scheduler for data centers powered by PV energy. They predict the amount of available energy in the near future and schedule workloads to maximize green energy consumption while meeting the jobs' deadlines. If grid energy must be used, the scheduler selects times when it is inexpensive.

By colocating with other tenants of an Infrastructure as a Service (IaaS) offering, IaaS users could reap significant cost savings by judiciously sharing their use of the fixed-size instances offered by IaaS providers [106]. Each server instance is characterized by a number of resources (CPU, network, memory, and disk space) which constitute dimensions of the instance capacity vector. Workloads associated with an instance are similarly characterized by a multi-dimensional utilization vector. User needs are spelled out as SLAs defined over the various resources of the instance (e.g., CPU, memory, local storage, network bandwidth). Based on this input, a desirable VM configuration is calculated, reflecting what the service deems best for each customer in the system. The framework then carries out the necessary migrations of VMs.

The Entropy resource manager [97] efficiently maintains the cluster in a configuration (a mapping of VMs to nodes), that uses a minimum number of nodes. As authors themselves point out, the configurations considered in the paper are fairly simple, because in the clusters available in their Grid'5000 experimental testbed, every node has only a single processor. Our study, on the other hand, assumes cluster nodes to be multicore servers. It is handling the contention within a multicore server that our framework is targeted at. Entropy does not control resource utilization of the collocated workloads based on the SLA level, which is the focus of our work.

To preserve high performance-to-power ratios in a clustered architecture, some authors [195, 143, 153] suggested tuning a cluster's performance and power characteristics by scaling the frequency of the nodes. Instead of taking the DVFS approach to save power, we utilize the collocated servers to their full capacity as this provides the best power efficiency on modern multicore servers. We note that the proposed solution can be used along with the collocation management methods that

we covered to handle the jobs inside each server.

Hashemian et al. [95] previously discovered that the response times reported by the RUBiS client generator could be grossly inaccurate, and that the generated workloads were less realistic than expected, causing server scalability to be incorrectly estimated. Using experimentation, they demonstrate how the Java virtual machine and the Java network library are the root causes of these issues. In this work, we use authors' fixes to correctly generate the load for RUBiS.

In the paper [60] authors explored theoretically how to systematically improve the effective utilization of computer systems, while maintaining a desired responsiveness for delay-sensitive workloads. They demonstrated that by carefully managing resources, workloads, and scheduling delay-tolerant workloads around the higher priority workload(s), achieving this goal is feasible.

While data centers host multiple services, the network does little to prevent a traffic flood in one service from affecting other services around it. To overcome these limitations the authors of VL2 [91] provide a solution in which each service has the illusion that all the servers assigned to it, and only those servers, are connected by a single non-interfering Ethernet switch. To achieve that, they adopt Valiant Load Balancing (VLB) where each server independently picks a path at random through the network for each of the flows it sends to other servers in the data center. Common concerns with VLB, such as the extra latency and the consumption of extra network capacity caused by path stretch, are overcome by the environment's setup (propagation delay is very small inside a data center) and the topology, which includes an extra layer of switches that packets bounce off of). VL2 provides network isolation to the cloud appliances, much like our work provides CPU isolation for the collocated virtualized workloads. Hence, the two solutions are complementary.

Our collocation methods rely on the ability of cloud environment to monitor the performance metrics of the running workloads and compare them with their target SLAs on-the-fly. The works presented in [189, 151] provide a generic, distributed monitoring and evaluation frameworks that can be used in conjunction with our solution in the cloud.

Many scientists utilize high-throughput computing (HTC)-enabled systems, where jobs are designed to opportunistically 'scavenge' unused resource cycles from other workloads and be interrupted when higher priority jobs do need resources. The work presented in [135] proposes a cloud infrastructure that deploys backfill VMs, each of which utilizes an entire idle node. A backfill VM is terminated when the resource is needed to satisfy an on-demand request. The work assumes that, whenever a backfill VM is terminated to free space for an on-demand user VM, the remaining resources on the node remain idle. This is different from our paper, because we consider the case of *simultaneously* collocating multiple on-demand and opportunistic jobs onto the same node.

In Linux, the generic scheduler supports a way of prioritizing processes through the notion of *scheduling classes* [137]. Scheduling classes are used to decide which thread should be run next on a processor core. There are two scheduling classes implemented in 2.6.32 and later kernels:

1) Completely Fair Scheduling class: schedules tasks following the Completely Fair (CF) algorithm. Threads whose policy is set to SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE are scheduled by this scheduling class.

2) Real Time scheduling class: schedules tasks following real-time mechanisms defined in the POSIX standard. Threads whose policy set to SCHED_FIFO, SCHED_RR are scheduled using this scheduling class.

Linux supports five possible scheduling policies:

- SCHED_NORMAL is used for normal processes.
- SCHED_BATCH is for CPU-intensive batch processes that are not interactive. Processes of this type are, in a way, the least privileged: they will never pre-empt another process handled by the CF scheduler and will therefore not disturb interactive tasks. The class is well suited for non-interactive tasks.
- SCHED_IDLE tasks are also of low importance in scheduling decisions, but this time because their priority is set to a low value. Note that SCHED_IDLE is, despite its name, not responsible for scheduling the idle loop. The kernel provides a separate mechanism for this purpose.
- SCHED_RR and SCHED_FIFO are used to implement soft real-time processes. SCHED_RR implements a round robin method, while SCHED_FIFO uses a first-in-first-out mechanism. Although there exist several runqueues, each of which corresponds to the real-time processes with the same priority level, the processes from lower priority runqueues cannot preempt processes from higher priority runqueues, which makes collocation management difficult.

When the scheduler is invoked, it queries the scheduling classes to determine which task is supposed to run next. Scheduling classes are related in a flat hierarchy: real-time processes are most important, so they are handled before completely fair processes, which are, in turn, given preference to the idle tasks that are active on a CPU when there is nothing better to do. Every task belongs to exactly one of the scheduling classes.

This scheme suggests a natural way to setup a priority-based collocation management: the critical processes could be assigned the policies of Real Time scheduling class (SCHED_FIFO, SCHED_RR) or retain the default SCHED_NORMAL policy, while batch processes would be assigned SCHED_BATCH. Control groups offer similar functionality, plus they allow for a fine grained

prioritization between several critical tasks by assigning any priority from 2 up to 262,144, something that the scheduling classes lack.

6.5 Related work on shared resource contention within an HPC cluster

Consolidation is an approach that migrates tasks within a cluster as their computational requirements change. The goal of Entropy resource manager [97] is to efficiently maintain the cluster in a configuration (a mapping of VMs to nodes), that is (1) viable (gives every VM access to sufficient memory and every active VM access to own processing unit), and (2) optimal (uses minimum number of nodes). For this, the Entropy iteratively 1) waits to be informed by the special sensors that a VM has changed state, from active to inactive or vice versa, 2) tries to compute a reconfiguration plan starting from the current configuration that requires the fewest possible migrations and leaves the cluster in a viable, optimal configuration, and 3) if successful, initiates migration of the VMs, if the new configuration uses fewer nodes than the current one, or if the current configuration is not viable.

In the experimental setup assumed in the study each task is encapsulated in a VM managed by Xen, for which efficient migration techniques are available. As authors themselves point out, the configurations considered in the paper are fairly simple, because in the clusters available in their Grid'5000 experimental testbed, every node has only a single processor and all nodes have the same amount of memory. Our study, on the other hand, assumes cluster nodes to be multicore NUMA servers, possibly heterogeneous. Entropy assumes that each node in a cluster provides a certain amount of memory and number of processing units, and each VM requires a certain amount of memory, and, if active, a processing unit. These constraints, while useful, must be refined with more detailed information about the requirements of each VM and its sensitivity to the contention from different cluster jobs. These fine-grained metrics, as well as the contention-aware optimization algorithm based on them will be developed within our study.

By colocating with other tenants of an Infrastructure as a Service (IaaS) offering, IaaS users could reap significant cost savings by judiciously sharing their use of the fixed-size instances offered by IaaS providers. [106] presents the Colocation as a Service (CaaS) framework that consists from two sets of services. CaaS strategic services that are executed in background identify coalitions of self interested users that would benefit from colocation on shared instances. The output of these services is the desirable VM configuration, reflecting what the service deems best for each customer

in the system. CaaS operational services provide the information necessary for the reconfigurations mandated by strategic services by tracking CPU, network and memory utilization of each VM. They then carry out the necessary migrations of VMs. Operational services are online since they involve the performance profiling and migration of VMs.

The authors describe XCS a prototype implementation of CaaS on top of the Xen hypervisor. The reason why Xen was used is because it supports physical resource reservations, as well as live migration using iterative bandwidth adapting (pre-copy or post-copy) memory page transfer algorithms. XCS implements strategic services based on a game-theoretic formulation of colocation; it features concurrent migration heuristics; and offers monitoring and accounting services at both the hypervisor and VM layers. Each server instance in XCS is referred to as Physical Machines (PM). PM is characterized by a number of resources (CPU, network, memory, and disk space) which constitute dimensions of the instance capacity vector. Workloads associated with an instance are similarly characterized by a multi-dimensional utilization vector. The authors used a cluster of 3.2 GHz quad-core hosts (the PMs), each of which with 3 GB of memory. The servers are connected to each other using a one gigabit Ethernet, and to a network file server containing the VMs disk images. PlanetLab traces were used to create an experimental workload.

The efficiency of CaaS framework could be improved by using finer grained metrics to predict the resource utilization of each VM in a shared cluster environment. In our work, we intend to explore such metrics and provide the means of using them to reduce shared contention in a pursuit of several optimization goals as is described in the Outline. The contention metrics can also be beneficial for efficient collocation of self-interested users. Hence, we see our work as complementary to [106].

In [81] authors present a model to predict contention effects in clustered environments. The model provides a basis for predicting execution times for applications on clusters of workstations which can be used to perform performance-efficient scheduling. CPU share of each application on cluster nodes was used to predict the aggregate slowdown of a job in the cluster. Each cluster node is assumed to have only one CPU. While CPU usage metric is still useful in case of multicore NUMA nodes to predict the slowdown for the I/O and communication-bound workloads, it might not detect the contention between several memory intensive applications as easily, since they still consumed 100% of the CPU (just like during solo run) if the core would be allocated to them fully.

Koukis and Koziris [117] present the design and implementation of a gang-like scheduling algorithm (meaning that processors across all nodes of the cluster are to be allocated to all related application threads simultaneously) aimed at improving the throughput of multiprogrammed workloads

on clusters of SMPs. The algorithm selects the processes to be co-scheduled so as not to saturate nor underutilize the memory bus or network link bandwidth. Its input data are acquired dynamically using hardware monitoring counters and a modified NIC firmware. Because of the use of hardware counter data to make a scheduling decision, this work is the closest to ours in terms of implementation. Our work, however, focuses on the resource contention optimization algorithm that optimizes the performance of the virtualized cluster jobs, deviation of the results, provides performance boost and increase in power efficiency of the cluster as a whole. The improvement in application throughput which is the goal of [117] serves only as a secondary objective in our work (throughput will be improved as long as it does not contradict with the main objective(s) described above). The experimental setup in [117] assumed that all processes were spawned directly under the control of the Linux scheduler, using the mpirun command. The authors then compared its performance with the default Linux scheduler (O(1) at the time [117] was written). While using a default Linux scheduler in a cluster setup can be justified for the small number of nodes, the industry-size clusters require state-of-the-art cluster schedulers, like Maui and Moab, to make scheduling decisions, since these schedulers support features like scalability, fulfilling user specified constraints, dynamic priorities, reservations, and fairshare capabilities, necessary for a big cluster operation. Because of that, within our work, we mainly target at comparing the performance of our techniques with the state-of-the-art cluster schedulers on industry-scale clusters. An important limitation of [117] is the lack of virtualization support. This feature is commonly used to migrate jobs within the cluster and we will incorporate it into our setup.

The Quincy framework presented in [105] addresses the problem of scheduling concurrent jobs on clusters where application data is stored on the computing nodes with large local hard drives, a common case in systems such as MapReduce, Hadoop, and Dryad. Maintaining high bandwidth between arbitrary pairs of computers in such environment becomes increasingly expensive as the size of a cluster grows, particularly since hierarchical networks are the norm for current distributed computing clusters. If computations are not placed close to their input data, the network can therefore become a bottleneck. The challenge for a cluster scheduling algorithm in this setting is that the requirements of fairness and locality often conflict. This is because a strategy that achieves optimal data locality will typically delay a job until its ideal resources are available, while fairness benefits from allocating the best available resources to a job as soon as possible after they are requested, even if they are not the resources closest to the computations data. Quincy's fairness goal is that a job which runs for t seconds given exclusive access to a cluster should take no more than Jt seconds when there are J jobs concurrently executing on that cluster. The main contribution of the paper

is a new, graph-based framework for cluster scheduling under a fine grain cluster resource-sharing model with locality constraints.

The Quincy testbed described in the paper restrict each computer to run only one task at a time since jobs do not supply any predictions about their resource consumption. We, on the other hand, assume situation where several cluster jobs can share the NUMA multicore nodes of the cluster. By taking into account the information supplied at every job's submission and online monitoring of job behaviour, our scheduler intent to make optimization decisions what jobs are allowed to share resources. Quincy assumes that tasks are independent of each other so killing one task will not impact another. This independence between tasks is in contrast to programming models like MPI assumed in our study in which tasks execute concurrently and communicate during their execution. MPI jobs are made up of sets of stateful processes communicating across the network, and killing or moving a single process typically requires the restart of all of the other processes in the set.

To preserve high performance to power ratios in a clustered architecture, authors of [143] claim that the power consumption of cluster nodes should be in proportion to the performance improvements they yield. They propose the implementation of heterogeneous cluster nodes that have varying delay and power characteristics. A clusters performance and power characteristic is tuned by scaling the frequency of its nodes. The paper proposes a dynamic adaptation policy that takes advantage of program metrics that estimate the performance potential of allocating node resources to jobs and selects a cluster configuration that either meets a fixed power budget or minimize a metric such as Energy*Delay^2 (ED^2). The DVFS decisions are based on the following two observations: (1) for the programs with high data cache miss rates there is usually not enough work to keep the processor busy for the entire duration of a cache miss, which implies that the execution units can execute at low frequencies and still complete their work before the cache miss returns. (2) For programs with high branch misprediction rates, executing instructions at a low frequency can severely degrade performance because it increases the time taken to detect a branch misprediction. At the end of every interval of one million committed instructions, hardware counters that track L1 and L2 cache miss rates and the number of branch mispredictions are examined, the applications are classified accordingly and the DVFS decision are being made for the next interval. The DVFS approach to saving power, while being efficient on the level of separate cluster nodes, has its limitations on a cluster-wide scope. In our contention optimization algorithm, the scheduling decisions to consolidate the workload on as few chips as possible so as not to decrease performance till a certain extent (specified by a system administrator or a user) will be enforced through job migration and turning off the entire nodes of the cluster and/or the entire chips on the NUMA multicore node (if the hardware supports

this feature). The latter approach reaps greater benefits when it comes to power savings in a cluster environment.

6.6 Related work on virtualization and virtual machine migration

The consolidation of multiple virtualized customer applications onto multicore servers introduces performance interference between collocated workloads, significantly impacting application QoS. To address this challenge, authors of Q-Clouds [147] advocate that the cloud should transparently provision additional resources as necessary to achieve the performance that customers would have realized if they were running in isolation. Q-Clouds is a QoS-aware control framework that uses online feedback to build a multi-input multi-output (MIMO) model that captures performance interference relationships between applications, and uses it to perform the resource management necessary to fulfil QoS requirements. This functionality is also utilized to allow applications to specify multiple levels of QoS as application Q-states. The lowest Q-state is the minimal performance that an application requires, but higher Q-states may be defined by the customer when they are willing to pay for higher levels of QoS.

In order to determine resource requirements in Q-Clouds, VMs are first profiled on a staging server to determine the amount of resources needed to attain a desired level of QoS in an interference-free environment. During VM placement, the Q-Clouds scheduler then leaves a prescribed amount of unused resources on each server for their later allocation in subsequent online control if required by QoS. The implementation described in the paper includes the ability to cap the VCPU utilizations of guest VMs, thereby limiting accessibility to CPU resources. Such implementation limits the evaluation to interference effects that can be directly affected by varying CPU resources. Q-Clouds implementation dynamically adjusts CPU resource allocations provided to guest VMs by setting the associated VCPU cap through the hypervisor. Based upon these inputs (VCPU caps) and outputs (guest QoS feedback in terms of MIPS, millions of instructions executed per second), authors implement the system controller in Matlab. The Matlab component of the controller runs on a separate machine, and communicates with the hardware monitoring agent over the network for updated QoS feedback. The authors evaluate Q-Clouds on Intel Nehalem based multicore machine with the Hyper-V virtualization platform.

The differences between Q-Clouds and our work are in the following: (1) While Q-Clouds is QoS centric, the optimization algorithm that we propose targets several goals, including deviation reduction, performance boost and smart packing of jobs to reduce power consumption. It will be

able to dynamically alter between those goals depending on the current load and the cluster user preferences. (2) In our work we intend to use hardware counters to get the metrics, informative about shared resource contention within the cluster. These metrics will allow us to estimate the performance slowdown incurred by contention online, without any knowledge about the workload obtained beforehand. Using the Q-Clouds' MIPS metric to detect a slowdown due to resource contention requires a profiling run. (3) The VCPU capping approach used in Q-Clouds has its limitations when optimizing scheduling goals, other than fairness. Hence, in our work we intend to optimize contention by scheduling applications on the cluster nodes.

Sandpiper [185] is a reconfiguration engine to relocate VMs from overloaded to under-utilized nodes. When a migration between two nodes is not directly feasible, the system identifies a set of VMs to swap in order to free a sufficient amount of resources on the destination node. Then the sequence of migrations is executed. This approach requires some space for temporarily hosting VMs on either the source or the destination node. Sandpiper implements a black-box approach that is fully OS- and application-agnostic and a gray-box approach that exploits OS- and application-level statistics (processor, network interface and memory swap statistics for each virtual server). Authors implement their techniques in Xen on a prototype datacenter consisting of twenty 2.4Ghz Pentium 4 servers with at least 1GB of RAM connected over gigabit ethernet. VMs in the experiments run Apache, PHP, and MySQL dynamic workloads. The algorithm and metrics used in this study, while being an important first step towards the sufficient amount of information for the contention optimization algorithm, need additional data to support advanced scheduling decisions of workload consolidation in an environment where various levels of memory hierarchy within the node (shared caches, front-side buses, memory controllers and chip interconnects on NUMA machine, etc.) and hierarchical interconnect between the nodes (the active hardware and channels of communication) are shared between various virtualized jobs.

The Active CoordinaTion (ACT) approach described in [111] relies on the notion of Class-of-Service, a multi-dimensional vector that expresses the resource expectations of a clouds guest VMs (CPU and network utilization in the implementation proposed), based on which it maps VMs onto Platform Units (set of platform resources of different types provided by the firmware at machine boot time and dynamically via performance counters). Using these abstractions, ACT can perform active management of a cluster workload that relies on continuous monitoring of the guest VMs runtime behavior and on an adaptive resource allocation algorithm, also presented within this work. The implementation suggested used CPU utilization and IO bandwidth as components for Platform Units to comply with the respective Class-of-Service components.

ACT management uses historic (observed) information regarding VM behavior to guess its future trends. The algorithm makes adjustments in each of the resources represented in the platform unit. Due to the limitations of the experimental testbed, the possible VM migration to another node (in case the current node does not have the resources to host it anymore) was emulated by pausing the VM in question. When more resources become available in the physical machine, the paused VMs are started again, thereby roughly emulating a new VM being migrated into the current physical machine from elsewhere in the cluster. The experimental analysis of the ACT prototype, built for Xen-based platforms, uses RUBiS, Hadoop, and SPEC fpr evaluation. Although this work lacks the experiments with MPI applications, typical for HPC clusters and does not take the contention for shared resources into account, its management algorithm is interesting for its use of dynamic vector-like entities to describe both VMs to schedule as well as the machines to schedule it onto. Unfortunately, the small number of components considered (CPU and NIC utilization) limits the applicability of this approach when, for instance, computationally intensive applications are competing for the access to memory controller on a NUMA multicore node.

[138] presents the exhaustive set of experimental evaluations of the importance of resource sharing in a virtualized cluster environment conducted on both serial and parallel versions of NAS benchmarks running in KVM virtual machines. The benchmarks were first profiled running solo and then with interference from other benchmarks on the same node. UMA machines with Intel E5440 Quad-Core processors were used as cluster nodes. From these placement experiments, several conclusions were made , namely that (1) by doing real time low level hardware monitoring of the virtual machines we can make scheduling decisions minimizing the resource contention and maximizing the throughput and (2) for the virtualization technology authors chose, there is a very minimal overhead of using it. Experimental evaluations demonstrate that inefficient placement might lead to resource contention causing degradation of upto 30%. In almost all kind of experiments, by making better placement decisions taking into account the statistics generated by hardware counters, the performance can be improved by more than 10%. The placement experiments presented in the paper confirm the preliminary conclusions provided within this work. Our project, however, takes one step further and focuses on devising an online scheduling algorithm that would find the optimal scheduling decisions automatically.

High performance computers are typified by cluster machines constructed from multicore nodes and using high performance interconnects like Infiniband (IB). Virtualizing such platforms implies the shared use of not only the nodes and node cores, but also of the cluster interconnect. In [157]

authors present a detailed study of the implications of sharing these resources by the set of microbenchmarks, using standard x86-based quadcore nodes, the Xen hypervisor to virtualize platform nodes and exploiting Infinibands native hardware support for its simultaneous use by multiple virtual machines. Measurements are conducted with multiple VMs deployed per node. Results indicate that multiple applications can share the clusters multicore nodes without undue effects on the performance of Infiniband until their total required bandwidth exceeds available IB resources. The results also state that, depending on the types of interactions between application processes and the amounts of IO performed, it can be beneficial to structure individual components as separate VMs rather than placing them into a single VM. Such placements can avoid undesirable interactions between guest OS-level and hypervisor level schedulers. We will take these conclusions into account when devising the cluster-wide scheduling algorithm that takes into account contention for cluster interconnects.

The work presented in [173] starts from the premise suggested in previous research that a specific class of applications is better suited to a particular type of virtualization scheme or implementation. In particular, Xen has been shown to perform with little overhead for compute-bound applications, while I/O bound applications incur more performance penalty. Authors of [173] argue that such a generalization should not go beyond the applications that were actually studied due to versatility in the HPC workload. The authors analyzed HPL and SP HPC benchmarks to determine the impact of Xen on these applications and compare their penalty profiles so that the following question can be answered: are the overhead profiles for a particular type of benchmarks (such as compute-bound) similar or are there grounds to conclude otherwise?

The experimental cluster comprised of 16 Pentium 4 nodes connected by a 100Mb Ethernet switch. Oprofile was used to study four hardware events: clock-unhalted, ITLB miss, DTLB miss, and L2Cache miss. For each event, authors gather the breakdown of the samples of an application into various parts such as application code, library code, kernel modules, kernel code, and hypervisor code. They found that, while the overall performance penalty does not differ much between HPL and SP, their breakdown profiles are not similar. Xen impacts the various parts of these applications in different ways. It is therefore possible that different applications in the same class may be impacted more differently than HPL or SP. These results are very interesting to us, since we want to schedule the applications based on their personal needs as much as possible. If, for instance, a cluster job suffered the main virtualization overhead while spending time on hypervisor level, it might be because of the contention from virtual drivers of several VMs for accessing the real hardware through hypervisor interface. Using performance counter data, we can dynamically detect this and

then reschedule jobs as necessary to reduce the contention for the shared hypervisor resources.

6.7 Related work on satisfying resource constraints

In [81] authors present the model of calculating a slowdown imposed on cluster jobs when jobs are assigned to nodes based on a set of constraints, such as memory availability and data locality. The aggregate slowdown due to constraint fulfilment is calculated from the extra amount of work relative to the uniform case where work is assigned evenly among the nodes (the constraints are absent in this case). The model calculates the slowdown using coarse-grained factors (the fraction of a job assigned to a particular node, number of nodes in the cluster, etc.) and does not use any information about contention sensitivity of jobs to shared resources. The approach to calculate the slowdown as the difference between constrained and unconstrained cases, however, deserves a detailed analysis within our study. If it proves to work with contention-aware metrics, the contention-aware model of calculating slowdown can be used in our optimization algorithm.

In [113] authors focus on performance penalty that a virtualization layer added to a scientific Grid incurs on the jobs with tight deadlines. The penalty could lead to unpredictable deadline estimation for the running jobs in the system. Since most of the scientific clouds cater the needs of different HPC communities, and have strict policies that it would kill the user jobs when they over run their allocated time limits, this can result in significant reduction in utilization efficiency. In the paper, authors attempted to tackle this problem by developing a scheduling technique for virtual machines which monitors the workload types (whether they are memory intensive, CPU intensive or network I/O bound) and the rate of missed and respected deadlines, and then try to predict whether the job would be meeting its deadline based on its past experience within the current cluster schedule. The simulation results show the increase in overall job throughput in the system due to timely termination of the executing jobs, which would have never completed otherwise. This work explores the notion of classifying workload jobs based on its type which can be useful in our classification as well. We, however, focus our work on how sensitive the particular cluster job to the various contention effects from other applications within the cluster. Having this goal in mind, the classification scheme can include, for instance, the memory intensiveness of an MPI process, or the average time it takes to deliver an MPI message through the network with the given monitored network load. We believe that the contention-oriented approach will allow us to make the informative scheduling decisions on-the-fly, without waiting for the respawn of a cluster job.

In [97] authors propose the Entropy resource manager for homogeneous clusters, which performs consolidation based on constraint programming and takes migration overhead into account. The use of constraint programming allows Entropy to find mappings of tasks to nodes that are better than those found by heuristics based on local optimizations, and that are frequently globally optimal in the number of nodes. It does so by using a depth first search. A Constraint Satisfaction Problem (CSP) is defined as a set of variables, a set of domains that represent the set of possible values that each variable can take on and a set of constraints that represent required relations between the values of the variables. A solution for a CSP is a variable assignment (a value for each variable) that simultaneously satisfies the constraints. Entropy solves two phases of constraint problems. The first phase, based on constraints describing the set of VMs and their CPU and memory requirements, computes a placement using the minimum number of nodes and a reconfiguration plan to achieve that placement. The second phase, based on a refined set of constraints that take feasible migrations into account, tries to improve the plan, to reduce the number of migrations required.

To solve CSPs, Entropy uses the Choco library. Choco [6, 7] incrementally checks the viability and minimality of a configuration as it is being constructed and discards a partial configuration as soon as it is found to be non-viable or to use more than the minimum number of nodes found so far. This strategy reduces the number of configurations that must be considered. The constraint of viability has to be taken into account both in the final configuration and also during migration. A migration is feasible if the destination node has a sufficient amount of free memory and, when the migrated VM is active, if the destination node has a free processing unit. The use of the specialized constraint programming library to solve the scheduling tasks definitely deserves close investigation within our project. One might wonder, however, whether the task of finding an acceptable solution that way on-the-fly is feasible as several more contention-related metrics come into play.

6.8 Related work on addressing contention globally

*Clavis2D*s unique property is that it addresses the issues of space management (shared memory-hierarchy contention and communication overhead) that may lead to energy waste. Among existing cluster management systems that came the closest to addressing a similar goal was Q-Clouds [147]. Q-Clouds tracks contention among co-located VMs, but instead of separating VMs to different physical nodes, it allocates extra CPU headroom: makes sure there is enough idle CPU cycles on a node so that a VM suffering from interference compensates for any performance loss by using more CPU

cycles (as proposed in [80]). Unlike *Clavis2D*, Q-Clouds uses offline profiling, and a feedback controller to compensate for performance interference; it is not designed to trade-off between multiple objectives.

There are a number of systems that, like *Clavis2D*, target space management, but focus on its different aspects, mainly CPU cycles and physical RAM capacity, and use different management strategies. Sandpiper [185], in particular, detects hotspots on physical servers - a condition when a specific resource, such as the CPU, is overloaded - and relieves the hotspot by migrating one of more VMs to a different server. The approach taken by Sandpiper is fundamentally different from that of *Clavis2D*; it can be characterized as local and reactive. The system reacts to locally-observed crises. *Clavis2D*, on the contrary, aims to achieve a global objective and pro-actively re-arranges VM placement to meet that goal. The generality of the multi-objective optimization framework used in *Clavis2D*allows adding other objectives, such as minimizing CPU and memory overload as in Sandpiper.

We are aware of two systems that, like *Clavis2D*, address the cluster scheduling policy from a global perspective and balance multiple objectives: Quincy [105] and Entropy [97]. Quincy, similarly to *Clavis2D*, targets space management (it aims to locate a task close to a file server holding its data), and furthermore balances multiple, and sometimes conflicting, objectives, such as fairness, latency and throughput. Quincy represents the scheduling problem as a graph and reduces it to the min-flow problem. *Clavis2D* uses a classical multi-objective branch-and-bound framework. Both are valid solutions, but the nice property of the problem representation used in *Clavis2D* is that its generality allows us to exploit it in more than one way: for instance, using widely-available tools we are able to compute a Pareto front of optimal solutions and provide this data to the system administrator to guide him in selecting the weight parameters for the system. We are not aware of any other system that is able to do that.

Entropy [97] shares a few similarities with *Clavis2D*: it aims to minimize the number of utilized nodes, while meeting CPU and memory constraints of the virtual machines. *Clavis2D* addresses a different space management problem than Entropy, but its key difference is that it adapts Choco to use the domain specific knowledge in order to significantly reduce the time needed to traverse the space of available solutions. One shortcoming of *Clavis2D* relative to Entropy is that it does not currently take into account the cost of VM migration. However, the generality of the problem representation allows us to easily add migration cost as one of the objectives in future work.

There is a plethora of work addressing time management of HPC clusters, asking how to manage job queues to achieve various performance and energy objectives [89, 113, 112, 167, 146, 124].

Space limitations do not permit discussing this work in detail, but the bottom line is that none of it touched upon the subject of space management, which is crucial from energy-saving perspective.

In [167] authors consider a trade-off between energy consumption and acceptance ratio of jobs in a virtualized cluster. The approach is based on dynamic configuration of host and its VMs to achieve the minimum energy necessary to execute a certain number of jobs accepted for execution from the queue. To achieve this, authors devise an energy model of multicore processor architecture based on the number of active cores, the average running frequency, and memory. The host resources that are not needed to execute the workload are powered off with DVFS. We note that in our work we consider HPC workloads that are 100% CPU intensive and use all the cores allocated to them. However, in case of partially IO bound load, the DVFS techniques to save power are helpful.

In [146] authors present a new auction mechanism for advanced job reservations in HPC clusters that seeks to increase fairness and efficiency of scheduling in a batch queued environment by eliciting honest information from users at the time they make a reservation. The approach is to employ a pricing mechanism that is incentive compatible: it uses game-theory to ensure that users self-interests are maximized when they honestly reveal their preferences to the allocation mechanism.

HPC clusters is not the only platform that hosts HPC workloads these days. The inherent scalability and flexibility of the cloud offerings, coupled with the absence of capital investment costs has made cloud an attractive platform for the HPC users who cannot afford significant upfront investments or whose workload demand is sporadic or bursty [61, 160, 156, 136]. The downsides of cloud are generally higher operational costs and virtualizatrion overheads due to cloud infrastructure being optimized for Web- and transaction-oriented applications, rather than HPC jobs [108, 77, 103].

Some projects suggested to use the cloud computing resources to host HPC clusters for educational purposes, e.g., StarHPC [107] and ViteraaS [74]. These projects do a great job in paving the way for a virtualized HPC clusters. They do not, however, address a multi-objective dynamic scheduling of job VMs, which is the focus of our work. Several research projects are working on reducing the virtualization overhead for HPC jobs in cloud [99, 98]. The authors of [94, 51] are devising methods for efficient collocation of transactional and HPC workloads in cloud. This problem is not typical for HPC clusters that are usually dedicated to running HPC jobs in batch mode.

We conclude with the related work that addresses various aspects in emerging field of **virtualized HPC clusters**. Previous work on the subject was largely concerned with high overhead of virtualization for HPC [173, 139, 149]. Our framework is using OS-level OpenVZ as a confirmed [30, 158] low overhead virtualization solution for HPC. The work is underway on making

other virtualization solutions (including VMWare, KVM and Xen) achieve near-baremetal performance in HPC clusters [44, 157, 119, 188, 102, 178].

The works of Walters et al. [180] and Naughton et al. [148] address the challenge of implementing fault tolerance in HPC domain through the use of a low overhead virtualization.

Although virtualizing HPC clusters is a recurring theme in research, the space sharing optimization techniques in HPC so far did not go beyond manual job placement. Our work is the first one that integrates all the necessary components to do that efficiently and actually demonstrates how the automated space sharing scheduling in the HPC clusters based on several optimization goals can be done. We also show how the suggested framework works in a typical HPC environment with advanced reservations and job dependencies.

Chapter 7

Summary

7.1 Lessons, Contributions, and the Big Picture

We conclude this dissertation by summarizing its lessons and contributions and discussing how the pieces of our work relate to each other. We then provide insight into the future datacenter management research. We end with a closing note.

This dissertation described 4 schedulers that address contention for shared memory and CPU resources in datacenter servers. The solutions targeted increase in energy efficiency of the datacenter infrastructure. This goal was achieved through smart energy-aware scheduling, increase in workload performance (total execution time), improving server utilization or balancing across several energy- and performance-related goals.

Sometimes a significant improvement in the execution time for a workload as a whole is difficult to achieve, as we learned from our experience on addressing contention for server memory hierarchy in UMA systems. By performing within 2% of the optimal, our new contention-aware scheduling algorithm DIO made the highest impact in improving quality of service and performance isolation for individual applications and in optimizing system energy consumption by consolidating workload at times *it does not hurt* performance. We also discovered that to properly address the shared resource contention in the UMA memory hierarchy, we must target not only contention for cache space, but also contention for other shared resources, such as the memory controller, memory bus and prefetching hardware.

We then attempted to adapt our contention-aware techniques for systems with NUMA memory architecture, in which non-uniform memory access latencies and multiple memory controllers exist. In doing so, we quickly realized that the state-of-the-art contention management algorithms fail to be effective on NUMA systems and may even *hurt* performance relative to a default OS scheduler.

While investigating the causes of this behavior, we discovered contention for memory controllers and cross-chip interconnect to be the major factors hurting performance on NUMA systems and that *the scheduling of memory is as important as the scheduling of threads* on these systems. As a result, we designed DINO, the first contention-aware algorithm for NUMA systems.

So far we considered only a certain class of workloads in our research: those that make intensive use of the memory hierarchy on the modern datacenter servers. These are CPU intensive batch computations that need to be finished by a certain deadline. There are, however, many transactional workloads in datacenters that, while not consuming much of the server resources, require immediate access to the resources they do need. In our work on managing contention for CPU resources of the machine, we wanted to take advantage of this diversity in performance requirements and resource usage of the two popular workload classes by collocating them together and prioritizing access from transactional applications at the expense of batch. We learned that, contrary to the popular belief, *work conserving collocation methods are feasible in state-of-the-art installations*: they allow almost 100% server utilization coupled with acceptable transactional performance. As one example, we implemented such collocation with Linux cgroups weights. We also learned that default OS load balancing mechanisms are not priority aware and need to be augmented with core affinity for high-priority tasks to fulfil their SLAs.

While the objectives of performance and energy efficiency could be viewed as orthogonal, we found that they are usually tightly coupled, and pursuing one objective requires a trade-off with another. We also realized that, to get the most out of contention management, we would have to make rebalancing decisions on a datacenter scale rather than within each standalone server. We were interested in answering the question: How do we build a cluster management system that simultaneously minimizes contention for shared resources, communication distance and idleness while balancing these objectives according to human preferences? One lesson we learned while answering this question is that *combining different approaches of targeting the same problem often yields the best solution in the end.* Such was the idea of using a genetic multi-objective optimization solver together with branch-and-bound solver that minimized a weighted sum of multiple goals. They complemented each other, which led us to design and implementation of *Clavis2D* – a system with a custom multi-objective solver at its heart that uses domain-specific knowledge to deliver good scheduling decisions within minutes, as opposed to hours or days. Our experiments on a clusters with up to 16 nodes show that *Clavis2D* delivers energy savings of up to 5 kWh (3kWh on average) and performance improvement of up to 60% (20% on average). For a reasonably large datacenter with a $30M electricity bill that 20% less energy due to faster execution would correspond

to $6M/year savings. Faster execution saves money!

7.2 A Glance into the Future

> Every two days now we create as much information as we did from the dawn of civilization up until 2003.
>
> Eric Schmidt, former CEO of Google

The data explosion is being experienced these days by many fields, which makes Big Data a rich topic with many possible avenues of investigation. Big data usually includes data sets with sizes beyond the ability of commonly-used software tools to capture, store and process the data within a tolerable time. I believe that the future research in datacenter management should revolve around the development of fast, energy efficient solutions for information retrieval and analysis. I anticipate that the applications of this research will be numerous and diverse, for the simple reason that Big Data computation has immediate relevance to anything that deals with a large scale system. The remainder of this section briefly describes some specific potential research projects.

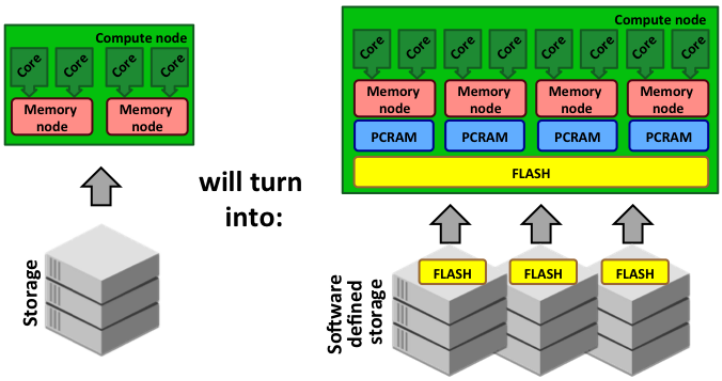


Figure 7.1: Memory hierarchy in Exascale era.

a) *Memory hierarchy in Exascale era.* The next-generation memory system architectures for datacenters are currently being developed. The idea is that the memory will be split into two or more types: the regular fast DRAM and a slower, bigger FLASH or PCRAM phase-change memory (Figure 7.1). The challenge is how and when to move the data between different memory blocks so as to ensure the robust execution of the Exascale workloads. The new types of memory will be managed by using constructs that we are already familiar with, such as reservations, limits and

shares. The FLASH solid state drives are also increasingly used in the software-defined storage networks for caching hot data from the commodity hard drives.

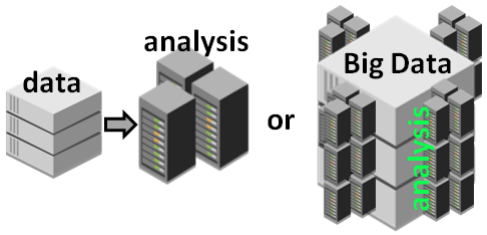


Figure 7.2: Memory hierarchy in Exascale era.

b) *Data placement.* The common approach taken nowadays in most datacenters is to move data from storage close to processing units for analysis. Unfortunately, this method tends to put considerable pressure on the datacenter interconnect or the wide area networks that transfer the data from the place it is stored to the datacenter that processes it. The recent emergence of large file transfer services like Globus Online and long haul InfiniBand and Ethernet switches only exacerbates the problem. We believe that a different paradigm could be investigated that places processing around Big Data storage (Figure 7.2). In the HPC domain, this emphasizes the role of local HPC facilities as data is often generated on site at university or industry campuses. With the introduction of portable, compact datacenters like HP EcoPOD, it became possible to create a cloud organization in which compute power migrates close to the data and only when analysis is needed. The difficulties in consolidating data from many small sensors into a single data warehouse suggest off-loading some of the computations to the sensors themselves.

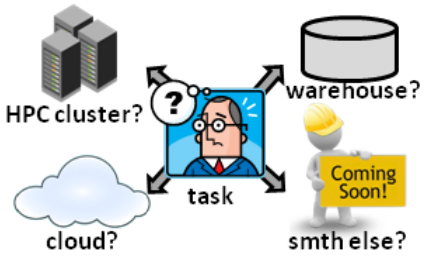


Figure 7.3: How to choose a datacenter for a given large scale analytic task?

c) *How to choose a datacenter for a given large scale analytic task?* (Figure 7.3) As Big Data markets grow, so will the choice of resources that allow the processing of many diverse analytic tasks. A data owner will soon face the question of what resources to utilize for data analysis. HPC

supercomputers are known to be very efficient at tackling large scale distributed tasks. The clusters, however, are quite costly to purchase and maintain and are highly contended for. Although clouds do not have the integrity of supercomputers, they nevertheless are very attractive platforms to perform large scale analytic computations, due to their high availability, scalability and absence of capital investment costs from users. Yet very little data is available on side-by-side performance comparison, price and energy efficiency of clouds when dealing with large scale analytics.

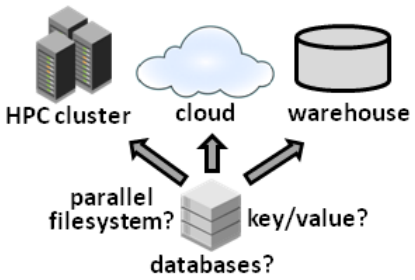


Figure 7.4: What storage organization is most suitable for each datacenter type?

d) *What storage organization is most suitable for each datacenter type?* (Figure 7.4) How to seamlessly integrate different types of storage (parallel filesystems, key-value pairs, database files) with different forms of computation? The type of storage used in a large scale datacenter installation often influences the software that is running on top of it. For example, the SkyServer project was designed to handle millions of information retrieval requests as a parallel database cluster. This caused the development of many user-defined database functions that extract and process astronomy data. Opting instead for a parallel filesystem (e.g., Lustre) as the main storage organization would prompt for writing MPI handlers instead of database functions.

I am convinced that, without fast and energy-efficient processing, information is very difficult to utilize. Companies are at risk of accumulating unprocessed data without the ability to extract valuable information on-the-fly while it still matters. I believe my past experience in dealing with different types of datacenters will help me answer these questions.

7.3 A Closing Note

Contention for shared resources in today's datacenters is a very serious issue that dramatically impairs the workload execution. This dissertation investigated how and to what extent contention for shared resources can be mitigated via workload scheduling. Scheduling is an attractive tool, because

it does not require extra hardware and is relatively easy to integrate into the system. My work addresses the contention for both shared memory and CPU resources within each multicore server, as well as across many servers on a datacenter scale.

The proposed solutions were shown to improve performance and energy efficiency of intensive computations in High Performance Computing (HPC) clusters. HPC is gaining huge momentum these days due to its ability to process large amounts of data. I have also designed and implemented several effective solutions for the collocation of transactional- and batch-oriented analytic workloads in cloud computing environments. Although clouds do not have the integrity of supercomputers, they nevertheless are very attractive platforms for performing massive analytic computations due to their high availability, scalability and absence of capital investment costs. I am convinced that the future of the shared resource contention research in datacenters lies closely with the Big Data challenge: a desperate need of the industry to process huge amounts of data on-the-fly. Companies are at risk of accumulating unprocessed data without the ability to extract valuable information while it still matters.

This dissertation has furthered the knowledge on development of techniques for automatic, on-the-fly detection, measurement and mitigation of shared resource contention in datacenter servers, as well as paved avenues for future research.

Bibliography

[1] AMD64 Architecture Programmer's Manual Volume 2: System Programming. *[Online] Available:* `http://support.amd.com/us/Processor\_TechDocs/24593.pdf`.

[2] BIOS and Kernel Developer's Guide (BKDG) for AMD Family 10h Processors. *[Online] Available:* `http://mirror.leaseweb.com/NetBSD/misc/cegger/hw\_manuals/amd/bkdg\_f10\_pub\_31116.pdf`.

[3] BLCR library. *[Online] Available: https://upc-bugs.lbl.gov/blcr-dist/.*

[4] Boeing catches a lift with High Performance Computing. *[Online] http://hpc4energy.org/hpc-road-map/success-stories/boeing/.*

[5] cgroups. *[Online] Available: http://www.kernel.org/doc/Documentation/cgroups/.*

[6] Choco installation tip 1. *[Online] Available:* `http://sourceforge.net/p/choco/discussion/335512/thread/edb47611`.

[7] Choco installation tip 2. *[Online] Available:* `http://sourceforge.net/p/choco/discussion/335512/thread/467c8307`.

[8] Clarification for the quotation from Richard Langworth. *[Online] Available: http://richardlangworth.com/falsequotes*.

[9] Clavis: a user level scheduler for Linux. *[Online] Available:* `http://clavis.sourceforge.net/`.

[10] Clavis2D videos. *[Online] http://clusterscheduling.blogspot.ca/.*

[11] Considerations to Consolidate and Virtualize. *[Online] Available: http://www.dell.com/downloads/global/solutions/public/white_papers/dell_sql_cons_virt_wp1.pdf.*

[12] Core Algorithms of the Maui Scheduler. *[Online] Available: http://www.eecs.harvard.edu/ chaki/bib/papers/jackson01maui.pdf.*

[13] Google Data Centers Efficiency: How we do it. *[Online] Available:* `http://www.google.ca/about/datacenters/efficiency/internal/`.

[14] Google search results for the Churchill quotation. *[Online] Available: https://www.google.ca/search?q=Success+is+not+final%2C +failure+is+not+fatal%3A+it+is+the+courage+to+continue+that+counts.*

[15] Greenhouse Gas Equivalencies Calculator. *[Online] Available:* `http://www.epa.gov/cleanenergy/energy-resources/calculator.html`.

[16] Greenhouse Gas Equivalencies Calculator. *[Online] http://www.epa.gov/cleanenergy/energy-resources/calculator.html.*

[17] Historic CERN discovery made possible in part by Compute Canada resources. *[Online] https://computecanada.ca/index.php/en/about-us/news/general-news/12-news-english/188-historic-cern-discovery-made-possible-in-part-by-compute-canada-resources.*

[18] IBM's Watson supercomputer to give instant medical diagnoses. *[Online] http://latimesblogs.latimes.com/technology/2011/09/ibm-watson-wellpoint.html.*

[19] Kernel Based Virtual Machine. *[Online] Available: http://www.linux-kvm.org/page/Main_Page.*

[20] Kernel news. *[Online] Available: http://www.linux-magazine.com/content/download/65332/508584/file/094-095_kernelnews.pdf.*

[21] Linux load balancing mechanisms. *[Online] Available:* `http://nthur.lib.nthu.edu.tw/bitstream/987654321/6898/13/432012.pdf`.

[22] Linux scheduling domains. *[Online] Available:* `http://lwn.net/Articles/80911/`.

[23] Maui Cluster Scheduler features. *[Online] Available: http://www.clusterbuilder.org/encyclopedia/alphabetized/m/maui-cluster-scheduler.php.*

[24] Maui Scheduler Administrator's Guide. *[Online] Available: http://www.clusterresources.com/products/maui/docs/mauiadmin.shtml.*

[25] Microsoft's Top 10 Business Practices for Environmentally Sustainable Data Centers. *[Online] Available: http://cdn.globalfoundationservices.com/documents/ MSFT-Top10BusinessPracticesforESDataCentersAug12.pdf.*

[26] Multi-objective optimization. *[Online] Available:* `http://en.wikipedia.org/wiki/Multi-objective_optimization`.

[27] Multi-objective optimization by Kevin Duh. *[Online] Available:* `http://www.kecl.ntt.co.jp/icl/lirg/members/kevinduh/papers/duh11multiobj-handout.pdf`.

[28] NAS. *[Online] Available: http://www.nas.nasa.gov/publications/npb.html.*

[29] Network emulation with netem kernel module. *[Online] Available: http://www.linuxfoundation.org/ collaborate/workgroups/networking/netem*.

[30] Open VirtualiZation. *[Online] Available: http://en.wikipedia.org/wiki/OpenVZ*.

[31] Oracle Kernel Release Notes. *[Online] Available: https://oss.oracle.com/ol6/docs/RELEASE-NOTES-UEK2-en.html*.

[32] PCI/PCI Express Configuration Space Access. *[Online] Available:* `http://developer.amd.com/Assets/pci\%20-\%20pci\%20express\%20configuration\%20space\%20access.pdf`.

[33] Pfmon user guide. *[Online] Available:* `http://perfmon2.sourceforge.net/pfmon\_usersguide.html`.

[34] Server virtualization has stalled, despite the hype. *[Online] Available: http://www.infoworld.com/print/146901*.

[35] SLURM/Maui configuration notes. *[Online] Available: http://www.supercluster.org/pipermail/mauiusers/2005-January/001442.html*.

[36] SPEC High Energy Physics. *[Online] http://w3.hepix.org/benchmarks/doku.php/*.

[37] The direct source that attributes the quotation to Churchill is the book "The Prodigal Project: Genesis" by Ken Abraham and Daniel Hart. *[Online] Available: http://www.nsrider.com/quotes/success.html*.

[38] TR from Jonathan Koomey. *[Online] Available: http://www.koomey.com/post/8323374335*.

[39] Traffic accounting with iptables. *[Online] http://openvz.org/Traffic_accounting_with_iptables*.

[40] Transition to Torque/Maui. *[Online] Available: http://www.urc.uncc.edu/urc/old-announcements/transition-to-torquemaui/*.

[41] Using Weighted Criteria to Make Decisions. *[Online] Available:* `http://mathforum.org/library/drmath/view/72033.html`.

[42] VMware ESX Server 2 NUMA Support. White paper. *[Online] Available: http://www.vmware.com/pdf/esx2_NUMA.pdf*.

[43] Wikipedia suggests the source for the quotation. *[Online] Available: http://en.wikiquote.org/wiki/Winston_Churchill*.

[44] Qasim Ali, Vladimir Kiriansky, Josh Simons, and Puneet Zaroo. Performance evaluation of HPC benchmarks on VMware's ESXi server. In *Proceedings of the 2011 international conference on Parallel Processing*, Euro-Par'11, pages 213–222, 2012.

[45] Dieter an Mey, Samuel Sarholz, and Christian Terboven et al. The RWTH Aachen SMP-Cluster User's Guide, Version 6.2. 2007.

[46] Martin Arlitt, Cullen Bash, Sergey Blagodurov, Yuan Chen, Tom Christian, Daniel Gmach, Chris Hyser, Niru Kumari, Zhenhua Liu, Manish Marwah, et al. Towards the design and operation of net-zero energy data centers. In *Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2012 13th IEEE Intersociety Conference on*, pages 552–561. IEEE, 2012.

[47] A. Beloglazov and R. Buyya. Managing Overloaded Hosts for Dynamic Consolidation of Virtual Machines in Cloud Data Centers under Quality of Service Constraints. *Parallel and Distributed Systems, IEEE Transactions on*, 24(7):1366–1379, 2013.

[48] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via Scheduling: Managing Shared State in Video Games. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 7–7, 2010.

[49] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Princeton University, 2008.

[50] Sergey Blagodurov and Martin Arlitt. Improving the efficiency of information collection and analysis in widely-used IT applications. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, ICPE '11, pages 359–370, 2011.

[51] Sergey Blagodurov, Daniel Gmach, Martin Arlitt, Yuan Chen, Chris Hyser, and Alexandra Fedorova. Maximizing Server Utilization while Meeting Critical SLAs through Weight-Based Collocation Management. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, 2013.

[52] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 1–1, 2011.

[53] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-Aware Scheduling on Multicore Systems. *ACM Trans. Comput. Syst.*, 28:8:1–8:45, December 2010.

[54] Sergey Blagodurov, Sergey Zhuravlev, Serge Lansiquot, and Alexandra Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *Simon Fraser University, Technical Report 2009-16*, 2009.

[55] Martin J. Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on NUMA Systems. *[Online] Available:* `http://www.linuxinsight.com/files/ols2004/bligh-reprint.pdf`, 2004.

[56] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Journal of parallel and distributed computing*, pages 207–216, 1995.

[57] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *Int. J. High Perform. Comput. Appl.*, 20(4):481–494, November 2006.

[58] Maryam Booshehrian, Torsten Muller, Randall M. Peterman, and Tamara Munzner. Vismon: Facilitating Analysis of Trade-Offs, Uncertainty, and Sensitivity In Fisheries Management Decision Making. *Comp. Graph. Forum*, 31(3pt3):1235–1244, June 2012.

[59] Timothy Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*, pages 1–1, Berkeley, CA, USA, 1993. USENIX Association.

[60] Niklas Carlsson and Martin Arlitt. Towards more effective utilization of computer systems. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, ICPE '11, pages 235–246, 2011.

[61] Adam G. Carlyle, Stephen L. Harrell, and Preston M. Smith. Cost-Effective HPC: The community or the cloud? In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 169–176, 2010.

[62] David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguadé. Enabling resource sharing between transactional and batch workloads using dynamic application placement. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 203–222, 2008.

[63] David Carrera, Malgorzata Steinder, Ian Whalley, Jordi Torres, and Eduard Ayguade. Autonomic Placement of Mixed Batch and Transactional Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):219–231, February 2012.

[64] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, 2005.

[65] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[66] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[67] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Evaluation of the Memory Page Migration Influence in the System Performance: the Case of the SGI O2000. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 121–129, 2003.

[68] Roger Curry, Cameron Kiddle, Nayden Markatchev, Rob Simmonds, Tingxi Tan, Martin Arlitt, and Bruce Walker. Facebook meets the virtualized enterprise. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 286–292, 2008.

[69] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. Application-aware prioritization mechanisms for on-chip networks. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–291, 2009.

[70] Arnaldo Carvalho de Melo. Performance counters on Linux, the new tools. *[Online] Available:* `http://linuxplumbersconf.org/2009/slides/Arnaldo-Carvalho-de-Melo-perf.pdf`.

[71] Kalyanmoy Deb and Deb Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., 2001.

[72] Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. vGreen: a System for Energy Efficient Computing in Virtualized Environments. volume 16, pages 6:1–6:27, November 2010.

[73] Yixin Diao and Aliza Heching. Closed loop performance management for service delivery systems. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 61–69, 2012.

[74] Frank Doelitzscher, Markus Held, Christoph Reich, and Anthony Sulistio. ViteraaS: Virtual Cluster as a Service. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 652–657, 2011.

[75] Manuel F. Dolz, Juan C. Fernández, Sergio Iserte, Rafael Mayo, and Enrique S. Quintana-Ortí. A simulator to assess energy saving strategies and policies in HPC workloads. *SIGOPS Oper. Syst. Rev.*, 46(2):2–9, July 2012.

[76] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326, 2009.

[77] Yaakoub El-Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. Exploring the Performance Fluctuations of HPC Workloads on Clouds. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 383–387, 2010.

[78] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *MSPC '08: Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness*, pages 26–30. ACM, 2008.

[79] Alexandra Fedorova. *Operating system scheduling for chip multithreaded processors*. PhD thesis, Cambridge, MA, USA, 2006. AAI3245133.

[80] Alexandra Fedorova, Margo I. Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 25–38, 2007.

[81] Silvia M. Figueira and Francine Berman. Modeling the Slowdown of Data-Parallel Applications in Homogeneous and Heterogeneous Clusters of Workstations. In *Proceedings of the Seventh Heterogeneous Computing Workshop*, HCW '98, pages 90–, 1998.

[82] Geoffrey C. Fox, Gregor von Laszewski, Javier Diaz, Kate Keahey, Jose Fortes, Renato Figueiredo, Shava Smallen, Warren Smith, and Andrew Grimshaw. FutureGrid - a reconfigurable testbed for Cloud, HPC and Grid Computing. 07/2012 2012.

[83] Marc E. Frincu and Ciprian Craciun. Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 267–274, 2011.

[84] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-Conserving Optimal Real-Time Scheduling on Multiprocessors. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 13–22, 2008.

[85] Ben Gamsa, Orran Krieger, and Michael Stumm. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 87–100, 1999.

[86] Benjamin Gamsa, Orran Krieger, and Michael Stumm. Optimizing IPC Performance for Shared-Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, ICPP '94, pages 208–211, 1994.

[87] Anshul Gandhi, Yuan Chen, Daniel Gmach, Martin Arlitt, and Manish Marwah. Minimizing Data Center SLA Violations and Power Consumption via Hybrid Resource Provisioning. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, pages 1–8, 2011.

[88] Brice Goglin and Nathalie Furmento. Enabling High-Performance Memory Migration for Multithreaded Applications on Linux. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–9, 2009.

[89] Íñigo Goiri, Ryan Beauchea, Kien Le, Thu D. Nguyen, Md. E. Haque, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. GreenSlot: scheduling energy consumption in green datacenters. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 20:1–20:11, 2011.

[90] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal of Solid State Circuits*, 31(9):1277–1284, 1996.

[91] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 51–62, 2009.

[92] Boris Grot, Stephen W. Keckler, and Onur Mutlu. Preemptive virtual clock: a flexible, efficient, and cost-effective QOS scheme for networks-on-chip. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–279, 2009.

[93] Abhishek Gupta and Gautam Barua. Cluster Schedulers. *[Online] Available: http://tinyurl.com/crdkq6f*.

[94] Abhishek Gupta, Laxmikant V. Kalé, Dejan Milojicic, Paolo Faraboschi, and Susanne M. Balle. HPC-Aware VM Placement in Infrastructure Clouds. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13, pages 11–20, 2013.

[95] Raoufehsadat Hashemian, Diwakar Krishnamurthy, and Martin Arlitt. Web Workload Generation Challenges - An Empirical Investigation. volume 42 of *Wiley Software Practice and Experience*, pages 629–647, 2012.

[96] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep 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, 2009.

[97] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 41–50, 2009.

[98] Marius Hillenbrand. Towards Virtual InfiniBand Clusters with Network and Performance Isolation, 2011.

[99] Marius Hillenbrand, Viktor Mauch, Jan Stoess, Konrad Miller, and Frank Bellosa. Virtual InfiniBand clusters for HPC clouds. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pages 9:1–9:6, 2012.

[100] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[101] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-Independent Workload Characterization. *IEEE Micro*, 27(3):63–72, 2007.

[102] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 125–134, 2006.

[103] K.Z. Ibrahim, S. Hofmeyr, and C. Iancu. Characterizing the Performance of Parallel Applications on Multi-socket Virtual Machines. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 1–12, 2011.

[104] SAEED IQBAL, RINKU GUPTA, and YUNG-CHIN FANG. Job Scheduling in HPC Clusters. *[Online] Available: http://www.dell.com/downloads/global/power/ps1q05-20040135-fang.pdf*.

[105] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[106] Vatche Ishakian, Raymond Sweha, Jorge Londono, and Azer Bestavros. Colocation as a Service: Strategic and Operational Services for Cloud Colocation. *Network Computing and Applications, IEEE International Symposium on*, pages 76–83, 2010.

[107] C. Ivica, J.T. Riley, and C. Shubert. StarHPC – Teaching parallel programming within elastic compute cloud.

[108] Keith R. Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J. Wasserman, and Nicholas J. Wright. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 159–168, 2010.

[109] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pages 220–229, 2008.

[110] Ali Kamali. Sharing Aware Scheduling on Multicore Systems. Master's thesis, Simon Fraser University Technical Report, Vancouver, Canada, 2010.

[111] M. Kesavan, A. Ranadive, A. Gavrilovska, and K. Schwan. Active CoordinaTion (ACT) - toward effectively managing virtualized multicore clouds. In *In Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 23–32, 2008.

[112] Yacine Kessaci, Nouredine Melab, and El-Ghazali Talbi. A pareto-based GA for scheduling HPC applications on distributed cloud infrastructures. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 456–462, 2011.

[113] Omer Khalid, Ivo Maljevic, Richard Anthony, Miltos Petridis, Kevin Parrott, and Markus Schulz. Dynamic scheduling of virtual machines running HPC workloads in scientific grids. In *Proceedings of the 3rd international conference on New technologies, mobility and security*, NTMS'09, pages 319–323, 2009.

[114] Wooyoung Kim and Michael Voss. Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Softw.*, 28:23–31, January 2011.

[115] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[116] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, 2008.

[117] Evangelos Koukis and Nectarios Koziris. Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems - Volume 1*, ICPADS '06, pages 345–354, 2006.

[118] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, 2006.

[119] John R. Lange, Kevin Pedretti, Peter Dinda, Patrick G. Bridges, Chang Bae, Philip Soltero, and Alexander Merritt. Minimal-overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 169–180, 2011.

[120] Richard P. LaRowe, Jr., Carla Schlatter Ellis, and Mark A. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Transactions on Parallel and Distributed Systems*, 3:686–701, 1991.

[121] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 241–251, 1997.

[122] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-Aware DRAM Controllers. In *HPCA 16: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture*, pages 200–209, 2008.

[123] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-core Architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 53:1–53:11, 2007.

[124] Xiaohui Li, Lionel Amodeo, Farouk Yalaoui, and Hicham Chehade. A multiobjective optimization approach to solve a parallel machines scheduling problem. *Adv. in Artif. Intell.*, 2010:2:1–2:10, January 2010.

[125] Jochen Liedtke, Hermann Haertig, and Michael Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 213, 1997.

[126] Bin Lin and Peter A. Dinda. VSched: Mixing Batch And Interactive Virtual Machines Using Periodic Real-time Scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 8–, 2005.

[127] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA 2008)*, pages 367–378, 2008.

[128] Jimmy Lin and Chris Dyer. Data-intensive text processing with MapReduce. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials '09, pages 1–2, 2009.

[129] Xiaocheng Liu, Chen Wang, Bing Bing Zhou, Junliang Chen, Ting Yang, and Albert Y. Zomaya. Priority-Based Consolidation of Parallel Workloads in the Cloud. *Parallel and Distributed Systems, IEEE Transactions on*, 24(9):1874–1883, 2013.

[130] Zhenhua Liu, Yuan Chen, Cullen Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. Renewable and cooling aware workload management for sustainable data centers. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 175–186, 2012.

[131] Emerson Loureiro, Paddy Nixon, and Simon Dobson. A Fine-Grained Model for Adaptive On-Demand Provisioning of CPU Shares in Data Centers. In *Proceedings of the 3rd International Workshop on Self-Organizing Systems*, IWSOS '08, pages 97–108, 2008.

[132] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[133] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 248–259, 2011.

[134] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 257–265, 2010.

[135] Paul Marshall, Kate Keahey, and Tim Freeman. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 205–214, 2011.

[136] Toni Mastelic, Drazen Lucanin, Andreas Ipp, and Ivona Brandic. Methodology for trade-off analysis when moving scientific applications to cloud. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 281–286, 2012.

[137] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., 2008.

[138] Uday Kiran Medisetty, Vicenc Beltran, David Carrera, Marc Gonzalez, Jordi Torres, and Eduard Ayguade. Efficient HPC application placement in Virtualized Clusters using low level hardware monitoring. *[Online] Available: http://www.udaykiranm.com/hpcvirt.pdf*.

[139] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev.*, 2006.

[140] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166, 2010.

[141] D. Minarolli and B. Freisleben. Utility-based resource allocation for virtual machines in Cloud computing. In *Proceedings of the 2011 IEEE Symposium on Computers and Communications*, ISCC '11, pages 410–417, 2011.

[142] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–18, 2007.

[143] Naveen Muralimanohar, Karthik Ramani, and Rajeev Balasubramonian. Power Efficient Resource Scaling in Partitioned Architectures through Dynamic Heterogeneity. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, volume 0, pages 100–111, 2006.

[144] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160, 2007.

[145] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 63–74, 2008.

[146] Andrew Mutz and Rich Wolski. Efficient auction-based grid reservations using dynamic programming. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–8, 2008.

[147] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 237–250, 2010.

[148] Thomas Naughton, Geoffroy Vallée, Christian Engelmann, and Stephen L. Scott. A case for virtual machine based fault injection in a high-performance computing environment. In *Proceedings of the 2011 international conference on Parallel Processing*, Euro-Par'11, pages 234–243, 2012.

[149] Lucas Nussbaum, Fabienne Anhalt, Olivier Mornard, and Jean-Patrick Gelas. Linux-based virtualization for HPC clusters. In *Proceedings of the Linux Symposium*, 2009.

[150] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 289–302, 2007.

[151] Simon E. Parkin and Graham Morgan. Toward reusable SLA monitoring capabilities. *Wiley Software Practice and Experience*, 42(3):261–280, March 2012.

[152] Eric Parsons, Ben Gamsa, Orran Krieger, and Michael Stumm. (De-)Clustering Objects for Multiprocessor System Software. In *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, IWOOOS '95, pages 72–, 1995.

[153] Vinicius Petrucci, Enrique V. Carrera, Orlando Loques, Julius C. B. Leite, and Daniel Mosse. Optimized Management of Power and Performance for Virtualized Heterogeneous Server Clusters. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 23–32, 2011.

[154] David Jackson Quinn, David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the Maui scheduler. pages 87–102, 2001.

[155] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

[156] Dinesh Rajan, Anthony Canino, Jesus A. Izaguirre, and Douglas Thain. Converting a High Performance Application to an Elastic Cloud Application. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 383–390, 2011.

[157] Adit Ranadive, Mukil Kesavan, Ada Gavrilovska, and Karsten Schwan. Performance implications of virtualizing multicore cluster machines. HPCVirt '08.

[158] Nathan Regola and Jean-Christophe Ducom. Recommendations for Virtualization Technologies in High Performance Computing. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 409–416, 2010.

[159] Jerry Rolia, Ludmila Cherkasova, Martin Arlitt, and Artur Andrzejak. A capacity management service for resource pools. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 229–237, 2005.

[160] Eduardo Roloff, Matthias Diener, Alexandre Carissimi, and Philippe O. A. Navaux. High Performance Computing in the cloud: Deployment, performance and cost efficiency. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE Second International Conference on*, pages 371–378, 2012.

[161] Akkarit Sangpetch, Andrew Turner, and Hyong Kim. How to tame your VMs: an automated control system for virtualized services. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, pages 1–16, 2010.

[162] Lee T. Schermerhorn. Automatic Page Migration for Linux. 2007.

[163] Daniel Shelepov and Alexandra Fedorova. Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. In *Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, 2008.

[164] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News*, 28(5):234–244, 2000.

[165] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 117, 2002.

[166] Orathai Sukwong, Akkarit Sangpetch, and Hyong S. Kim. SageShift: Managing SLAs for highly consolidated cloud. In *INFOCOM'12*, pages 208–216, 2012.

[167] Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel. Energy efficient scheduling of HPC-jobs on virtualize clusters using host and VM dynamic configuration. *SIGOPS Oper. Syst. Rev.*, 46(2):19–27, July 2012.

[168] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58, 2007.

[169] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 121–132, 2009.

[170] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 283–294, 2011.

[171] The Choco Team. Choco: An open source Java constraint programming library. *3rd constraint solver competition*.

[172] Mark Horowitz Thomas, Thomas Indermaur, and Ricardo Gonzalez. Low-Power Digital Design. In *In IEEE Symposium on Low Power Electronics*, pages 8–11, 1994.

[173] Anand Tikotekar, Geoffroy Vallée, Thomas Naughton, Hong Ong, Christian Engelmann, and Stephen L. Scott. Euro-par 2008 workshops - parallel processing. chapter An Analysis of HPC Benchmarks in Virtual Machine Environments, pages 63–71. 2009.

[174] Niraj Tolia, David G. Andersen, and M. Satyanarayanan. Quantifying Interactive User Experience on Thin Clients. *Computer*, 39:46–52, March 2006.

[175] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical Clustering: a Structure for Scalable Multiprocessor Operating System Design. *J. Supercomput.*, 9(1-2):105–134, 1995.

[176] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia Workload Analysis. Technical Report IR-CS-041, Vrije Universiteit, Amsterdam, The Netherlands, September 2007 (revised: June 2008).

[177] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 291–302, 2005.

[178] Geoffroy Vallee. Management of virtual large-scale high-performance computing systems. In *Linux Symposium*, 2011.

[179] Ruud van der Pas. The OMPlab on Sun Systems. In *Proceedings of the First International Workshop on OpenMP*, 2005.

[180] John Paul Walters and Vipin Chaudhary. A fault-tolerant strategy for virtualized HPC clusters. *J. Supercomput.*, 50(3):209–239, December 2009.

[181] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 43:1–43:12, 2008.

[182] Rui Wang, Dara Marie Kusic, and Nagarajan Kandasamy. A distributed control framework for performance management of virtualized computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, ICAC '10, pages 89–98, 2010.

[183] Zhikui Wang, Yuan Chen, D. Gmach, S. Singhal, B. J. Watson, W. Rivera, Xiaoyun Zhu, and C. D. Hyser. AppRAISE: application-level performance management in virtualized server environments. *IEEE Transactions on Network and Service Management*, 6, 2009.

[184] Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 366–387, 2008.

[185] Timothy Wood, Prashant Shenoy, and Arun. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, pages 229–242, 2007.

[186] Zhen Xiao, Weijia Song, and Qi Chen. Dynamic Resource Allocation Using Virtual Machines for Cloud Computing Environment. *Parallel and Distributed Systems, IEEE Transactions on*, 24(6):1107–1117, 2013.

[187] Yuejian Xie and Gabriel Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *In Proceedings of CMP-MSI, held in conjunction with ISCA-35*, 2008.

[188] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for HPC systems. In *Proceedings of the 2006 international conference on Frontiers of High Performance Computing and Networking*, ISPA'06, pages 474–486, 2006.

[189] C.-H. Philip Yuen and S.-H. Gary Chan. Scalable Real-Time Monitoring for Distributed Applications. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2330–2337, 2012.

[190] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 203–212, 2010.

[191] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*, pages 89–102, 2009.

[192] Xiaoyun Zhu, Don Young, Brian J. Watson, Zhikui Wang, Jerry Rolia, Sharad Singhal, Bret McKee, Chris Hyser, Daniel Gmach, Rob Gardner, Tom Christian, and Lucy Cherkasova. 1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center. In *Proceedings of the 2008 International Conference on Autonomic Computing*, ICAC '08, pages 172–181, 2008.

[193] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 129–142, 2010.

[194] Sergey Zhuravlev, Alexandra Fedorova, and Sergey Blagodurov. AKULA: A Fast Simulator for Thread Placement Algorithms on Large Multicore Systems. *Simon Fraser University Technical Report*.

[195] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of Energy-Cognizant Scheduling Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1447–1464, 2013.

Appendix A

Developing a contention-aware scheduler under Linux

A.1 Introduction

In the era of increasingly multicore systems, memory hierarchy is adopting non-uniform distributed architectures. NUMA systems, which have better scalability potential than their UMA counterparts, have several memory nodes distributed across the system. Every node is physically adjacent to a subset of cores, but physical address space of all nodes is globally visible, so cores can access memory in a local as well as remote nodes. Therefore, the time it takes to access data is not uniform and varies depending on the physical location of the data. If a core sources data from a remote node, performance may suffer because of remote latency overhead and delays resulting from interconnect contention, which occurs if lots of cores access large amounts of data remotely [52]. These overheads can be mitigated if the system takes care to co-locate the thread with its data as often as possible [52, 123, 59, 120, 168, 42, 67, 118]. This can be accomplished via NUMA-aware scheduling algorithms.

Recent introduction of multicore NUMA machines into High-performance computing (HPC) clusters also raised the question whether the necessary scheduling decisions can be made at user-level, as cluster schedulers are typically implemented at user level [154, 24]. User-level control of thread and memory placement is also useful for parallel programming runtime libraries [48, 114, 56, 65], which are subject to renewed attention because of proliferation of multicore processors.

The Clavis user level scheduler that we present in this chapter is a result of research reflected in

several conference and journal publications [52, 53, 193, 194]. It is released as an open source [9]. Clavis can support various scheduling algorithms under Linux operating system running on multicore and NUMA machines. It is written in C so as to ease the integration with the default OS scheduling facilities, if desired.

The rest of this chapter is organized as follows: Section A.2 provides an overview of NUMA-related Linux scheduling techniques for both threads and memory. Section A.3 describes the essential features that have to be provided by an OS for a user level scheduler to be functional, along with the ways to obtain them in Linux. Section A.4 demonstrates how hardware performance counters and instruction-based sampling can be used to dynamically monitor the system workload at user level. Section A.5 introduces Clavis, which is built on top of these scheduling and monitoring facilities.

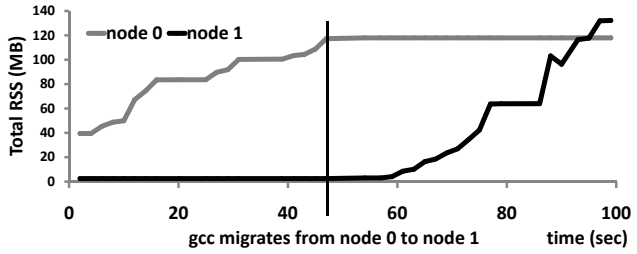
A.2 Default Linux scheduling on NUMA systems

Linux uses the principle *local node first* when allocating memory for the running thread.[1] When a thread is migrated to a new node, that node will receive newly allocated memory of the thread (even if earlier allocations resulted on a different node). Figure A.1 illustrates Linux memory allocation strategy for two applications from SPEC CPU 2006 suite: *gcc* and *milc*. Both applications were initially spawned at one of the cores local to memory node 0 of a two-node NUMA system (AMD Opteron 2350 Barcelona), and then in the middle of the execution were migrated to the core local to the remote memory node 1.

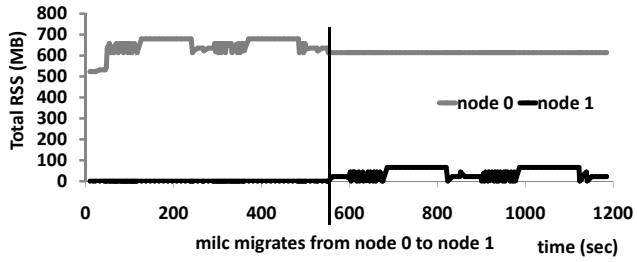
It is interesting to note in Figure A.1 that the size of thread's memory on the old node remains constant after migration. This illustrates that Linux does not migrate the memory along with the thread. Remote memory access latency, in this case, results in performance degradation: 19% for *milc* and 12% for *gcc*. While *gcc* allocates and uses memory on the new node after migration (as evident from the figure), *milc* relies exclusively on the memory allocated before migration (and left on the remote node). That is why, *milc* suffers more from being placed away from its memory.

Linux Completely Fair Scheduler (CFS) tries to compensate for the lack of memory migration by reducing the number of thread migrations across nodes. This is implemented via the abstraction of *scheduling domains*: a distinct scheduling domain is associated with every memory node on the system. The frequency of thread migration across domains is controlled by masking certain events

[1] From now on we assume 2.6.29 kernel, unless it is explicitly stated otherwise.



(a) gcc



(b) milc

Figure A.1: Memory allocation on Linux when the thread is migrated in the middle of its execution and then stays on the core it has migrated on. New memory is always allocated on the new node, old memory stays where it was allocated.

that typically cause migrations, such as context switches [55, 22, 21]. With scheduling domains in place, the system reduces the number of inter-domain migrations, favouring migrations within a domain.

Thread affinity to its local scheduling domain does improve memory locality, but could result in poor load balance and performance overhead. Furthermore, memory-intensive applications (those that issue many requests to DRAM) could end up on the same node, which results in contention for that node's memory controller and the associated last-level caches. Ideally, we need to: (a) identify memory intensive threads, (b) spread them across memory domains, and (c) migrate memory along with the threads. Performance benefits of this scheduling policy were shown in previous work [52, 53, 193].

Section A.3 describes how to obtain the necessary information to enforce these scheduling rules on user level. Section A.4 shows how to identify memory intensive threads using hardware performance counters. Section A.5 puts it all together and presents the user-level scheduling application.

A.3 User-level scheduling and migration techniques under Linux

Linux OS provides rich opportunities for scheduling at user level. Information about the state of the system and the workload, necessary to make a scheduling decision, can be accessed via *sysfs* and *procfs*. Overall, scheduling features available at user level can be separated into two categories: those that provide information for making the right decision – we call them *monitoring features*, and those that let us enforce this decision – *action features*. Monitoring features provide relevant information about the hardware, such as the number of cores, NUMA nodes, etc. They also help identify threads that show high activity levels (e.g., CPU utilization, I/O traffic) and for which user level scheduling actually matters. Table A.1 summarizes monitoring features and presents ways to implement them at user level. Action features provide mechanisms for binding threads to cores and migrating memory. They are summarized in Table A.2.

As can be seen from the tables, many features are implemented via system calls and command-line tools (for example, binding threads can be performed via `sched_setaffinity` call or `taskset` tool). Using system calls in a user level scheduler is a preferred option: unlike command-line tools they do not require spawning a separate process and thus incur less overhead and do not trigger recycling of PIDs. Some command line tools, however, have a special *batch mode*, where a single instantiation remains alive until it is explicitly terminated and its output is periodically redirected to a file or to `stdout`. In Clavis, we only use system calls and command-line tools in batch mode.

A.4 Monitoring hardware performance counters

Performance counters are special hardware registers available on most modern CPUs as part of Performance Monitoring Unit (PMU). These registers obtain the information about certain types of hardware events, such as retired instructions, cache misses, bus transactions, etc. PMU models from Intel and AMD offer hundreds of possible events to monitor covering many aspects of microarchitecture's behaviour. Hardware performance counters can track these events without slowing down the kernel or applications. They also do not require the profiled software to be modified or recompiled [70, 78].

On modern AMD and Intel processors, the PMU offers two modes in which profiling can be performed. In the first mode, the PMU is configured to profile only a small set of particular events,

but for many retired instructions.[2] This mode of profiling is useful for obtaining *a high level* profiling data about the program execution. For example, a PMU configured in this mode is able to track the last level cache (LLC) miss rate and trigger an interrupt when a threshold number of events have occurred (Section A.4.1). This mode, however, does not allow us to find out which particular instruction caused a cache miss.

In the second mode, the PMU works in the opposite way: it obtains detailed information about retired instructions, but the number of profiled instructions is very small. The instruction sampling rate is determined by a sampling period, which is expressed in cycles and can be controlled by end-users. On AMD processors with Instruction-Based Sampling (IBS), execution of one instruction is monitored as it progresses through the processor pipeline. As a result, various information about it becomes available, such as instruction type, logical and physical addresses of the data access, whether it missed in the cache, the latency of servicing the miss, etc. [78, 2] In Section A.4.2 we provide an example of using IBS to obtain logical addresses of the tagged load or store operations. These addresses can then be used by the scheduler to migrate recently accessed memory pages after migration of a thread [52]. On Intel CPUs similar capabilities are available via Precise Event-Based Sampling (PEBS).

A.4.1 Monitoring the LLC miss rate online

As an example of using hardware performance counters to monitor particular hardware events online, we will show how to track LLC miss rate per core on an AMD Opteron systems. Previous research showed that LLC miss rate is a good metric to identify memory intensive threads [116, 52, 53, 193]. Threads with a high LLC miss rate will perform frequent memory requests (hence the term memory-intensive) and so their performance will strongly depend on the memory subsystem. LLC misses have a latency of hundreds of cycles, but can take even longer if the necessary data is located on the remote memory node. Accessing remote memory node requires traversing the cross-chip interconnect, and so LLC-miss latency would increase even further if the interconnect has high traffic. As a result, an application with higher LLC miss rate could suffer higher performance overhead on NUMA systems than an application which does not access memory often.

Many tools to gather hardware performance counter data are available for Linux, including

[2] The exact number of events that can be tracked in parallel depends on available counter registers inside the PMU and usually varies between one and four. A special monitoring software like `perf` or `pfmon`, however, can monitor more events than there are actual physical registers via event multiplexing.

`oprofile`, `likwid`, `PAPI`, etc. In this chapter we focus on two tools that we use in our research: `perf` and `pfmon`. The choice of a tool depends on the Linux kernel version. For Linux kernels prior to 2.6.30, `pfmon` [78] is probably the best choice as it supports all the features essential for user level scheduling, including detailed description of a processor's PMU capabilities (what events are available for tracking, the masks to use with each event, etc), counter multiplexing and periodic output of intermediate counter events (necessary for online monitoring). `Pfmon` requires patching the kernel in order for the user level tool to work. The support for `pfmon` was discontinued since 2.6.30 in favour of the vanilla kernel profiling interface PERF_EVENTS and a user-level tool called `perf` [70]. `Perf` generally supports the same functionality as `pfmon` (apart from a periodic output of intermediate counter data, which we added). PERF_EVENTS must be turned on during kernel compilation for this tool to work.

The server we used has two AMD Opteron 2435 Istanbul CPUs, running at 2.6 GHz, each with six cores (12 total CPU cores). It is a NUMA system: each CPU has an associated 8 GB memory block, for a total of 16 GB main memory. Each CPU has 6 MB 48-way L3 cache shared by six cores. Each core also has a private unified 512 KB 16-way L2 cache and a private 64 KB 2-way L1 instruction and data caches. The client machine was configured with a single 76 GB SCSI hard drive. To track the LLC miss rate (number of LLC misses per instruction), the user-level scheduler must perform the following steps:

1) Get the layout of core IDs spread among the nodes of the server. On a two socket machine with 6 core AMD Opteron 2435 processors, the core-related output of `numactl` would look like:

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10
node 1 cpus: 1 3 5 7 9 11
```

2) Get the information about L3_CACHE_MISSES and RETIRED_INSTRUCTIONS events provided by the given PMU model (the event names can be obtained via `pfmon -L`):

```
# pfmon -i L3_CACHE_MISSES
Code     : 0x4e1
Counters : [ 0 1 2 3 ]
Desc     : L3 Cache Misses
Umask-00 : 0x01 : [READ_BLOCK_EXCLUSIVE] :
```

```
Read Block Exclusive (Data cache read)
Umask-01 : 0x02 : [READ_BLOCK_SHARED] :
Read Block Shared (Instruction cache read)
Umask-02 : 0x04 : [READ_BLOCK_MODIFY] :
Read Block Modify
Umask-03 : 0x00 : [CORE_0_SELECT] :
Core 0 Select
Umask-04 : 0x10 : [CORE_1_SELECT] :
Core 1 Select
<...>
Umask-08 : 0x50 : [CORE_5_SELECT] :
Core 5 Select
Umask-09 : 0xf0 : [ANY_CORE] :
Any core
Umask-10 : 0xf7 : [ALL] :
All sub-events selected

# pfmon -i RETIRED_INSTRUCTIONS
Code     : 0xc0
Counters : [ 0 1 2 3 ]
Desc     : Retired Instructions
```

As seen from the output, L3_CACHE_MISSES has a user mask to configure. The high 4 bits of the mask byte specify the monitored core, while the lower ones tell `pfmon` what events to profile. We would like to collect all types of misses for the given core. Hence, all three meaningful low bits should be set. We will configure the ”core bits” as necessary, so that, for example, core 1 user mask will be 0x17.

While RETIRED_INSTRUCTIONS is a core-level event and can be tracked from every core on the system, L3_CACHE_MISSES is a Northbridge (NB), node-level event [2]. Northbridge resources, including memory controller, crossbar, HyperTransport and LLC events are shared across all cores on the node. To monitor them from user level on AMD Opteron CPUs, the profiling application must start *only one session per node from a single core on the node* (any core on the node can be chosen for that purpose). Starting more than one profiling instance per node for NB events will

result in a monitoring conflict and the profiling instance will be terminated.

3) To get periodic updates on LLC misses and retired instructions for every core on the machine, the scheduler needs to start two profiling sessions on each memory node. One session will access a single core on the node (let it be core 0 for the first node and core 1 for the second) and periodically output misses for all cores on the chip and instructions for this core by accessing NB miss event and this core's instruction event. Another instance will access the rest of the cores from the node and collect retired instruction counts from the other cores. The two sessions for node 0 would then look like so:

```
pfmon --system-wide --print-interval=1000 \
--cpu-list=0 --kernel-level --user-level \
--switch-timeout=1 \
-e L3_CACHE_MISSES:0x07,L3_CACHE_MISSES:0x17,\
L3_CACHE_MISSES:0x27,L3_CACHE_MISSES:0x37 \
-e L3_CACHE_MISSES:0x47,L3_CACHE_MISSES:0x57,\
RETIRED_INSTRUCTIONS

pfmon --system-wide --print-interval=1000 \
--cpu-list=2,4,6,8,10 --kernel-level \
--user-level \
--events=RETIRED_INSTRUCTIONS
```

In the first session, there are two event sets to monitor, each beginning with `--events` keyword. The maximum number of events in each session is equal to the number of available counters inside PMU (four, according to `pfmon -i` output above). `Pfmon` will use event multiplexing to switch between the measured event sets with the frequency `--switch-timeout` milliseconds. Monitoring is performed per core as is designated by `--system-wide` option[3] in kernel and user level for all events. Periodic updates will be given at 1000 ms intervals.

[3] `Pfmon` and `perf` can monitor the counters in two modes: system-wide and per-thread. In per-thread mode, the user specifies a command for which the counters are monitored. When the process running that command is moved to another core, the profiling tool will switch the monitored core accordingly. In the system-wide mode, the tool does not monitor a specific program, but instead tracks all the processes that execute on a specific set of CPUs. A system-wide session cannot co-exist with a per-thread session, but a system wide session can run concurrently with other system wide sessions as long as they do not monitor the same set of CPUs [33]. NB events can only be profiled in system-wide mode.

The scheduler launches similar profiling sessions on the rest of the system nodes, but replaces the core IDs as is seen in `numactl --hardware` output.

4) At this point, scheduler has the updated information about LLC misses and instructions for every core on the system, thus it can calculate the miss rate for every core. The data collected with `perf` and `pfmon` on each node contains "LLC missrate – core" pairs that characterize the amount of memory intensiveness within each node online. In order to make a scheduling decision, we need to find out the id of the thread that is running on a given core so the pair will turn into "LLC missrate – thread ID". This can be done via *procfs* (see Table A.1). While it is possible to tell which threads are executing on the same core, there is currently no way to attribute individual miss rate to every thread due to limitation of measuring NB events on user level[4]. Fortunately, we did not find this to be a show stopper in implementing the user-level scheduler: the LLC miss rate as a metric of memory intensiveness is only significant for compute bound threads, and those threads are usually responsible for most activity on the core (launching a workload with more than one compute-bound thread per core is not typical).

The above steps also apply when using `perf` instead of `pfmon` under the latest kernel versions (we used 2.6.36 kernel with `perf`). The only challenge is that `perf` only includes several basic counters (cycles, instructions retired and so on) into its symbolic interface by default. The rest of the counters, including NB events and their respective user masks have to be accessed by directly addressing a special Performance Event-Select Register (PerfEvtSeln) [1]. Below are the invocations of `perf` with raw hardware event descriptors for the two sessions on node 0:

```
perf stat -a -C 0 -d 1000 \
-e r4008307e1 -e r4008317e1 -e r4008327e1 \
-e r4008337e1 -e r4008347e1 -e r4008357e1 \
-e rc0

perf stat -a -C 2,4,6,8,10 -d 1000 -e rc0
```

As can be seen, the names or raw hardware events in `perf` begin with an "r". Bits 0-7, 32-35 of the register are dedicated to the event code. Bits 8-15 are for the user mask. Bits 16-31 are reserved with the value 0x0083. If the event code is only 1 byte long (0xC0 for RETIRED_INSTRUCTIONS), there is no need to specify the rest of the code bits and, hence, mention all the reserved bytes in between.

[4]We are currently working on kernel changes that will allow measuring per-thread LLC at user level.

A.4.2 Obtaining logical address of a memory access with IBS

IBS is AMD's profiling mechanism that enables the processor to select a random instruction fetch or micro-op after a programmed time interval has expired and record specific performance information about the operation. The IBS mechanism is split into two modules: instruction fetch performance and instruction execution performance. Instruction fetch sampling provides information about instruction TLB and instruction cache behavior for fetched instructions. Instruction execution sampling provides information about micro-op execution behavior [2]. For the purpose of obtaining the address of the load or store operation that missed in the cache, the instruction execution module has to be used as follows:

1) First of all, the register MSRC001_1033 (IbsOpCtl, Execution Control Register) needs to be configured to turn IBS on (bit 17) and set the sampling rate (bits 15:0). According to the register mnemonic, IbsOpCtl is in MSR (Model Specific Registers) space with the 0xC0011033 offset. MSR registers can be accessed from user level in several ways: (a) through x86-defined RDMSR and WRMSR instructions, (b) through command-line tools `rdmsr` and `wrmsr` available from `msr-tools` package, (c) by reading or writing into /dev/cpu/<CID>/msr file (MSR support option must be turned on in the kernel for that)[5].

2) After IBS is configured, execution sampling engine starts the counter and increments it on every cycle. When the counter overflows, IBS tags a micro-op that will be issued in the next cycle for profiling. When the micro-op is retired, IBS sets the 18th bit of IbsOpCtl to notify the software that new instruction execution data is available to read from several MSR registers, including MSRC001_1037 (IbsOpData3, Operation Data 3 Register).

3) At that point, the user level scheduler determines if the tagged operation was a load or store that missed in the cache. For that, it checks the 7th bit of IbsOpData3. If the bit was set by IBS, the data cache address in MSRC001_1038 (IbsDcLinAd, IBS Data Cache Linear Address Register) is valid and ready to be read from.

4) After the scheduler gets the linear address, it needs to clear the 18th bit of IbsOpCtl that was set during step 2, so IBS could start counting again towards the next tagged micro-operation.

[5] Although accessing MSR registers from user level is straight-forward, they are not the only CPU registers that can be configured that way. For example, turning a memory controller prefetcher on/off can only be done via F2x11C register from PCI-defined configuration space. For that, command line tools `lspci` and `setpci` from `pciutils` package can be used under Linux [32].

A.5 Clavis: an online user level scheduler for Linux

Clavis is a user-level application that is designed to test efficiency of scheduling algorithms on real multicore systems[6]. It is able to monitor the workload execution online, gather all the necessary information for making a scheduling decision, pass it to the scheduling algorithm and enforce the algorithm's decision. Clavis is released as an Open Source project [9]. It has three main phases of execution:

- *Preparation.* During this phase, Clavis starts the necessary monitoring programs in batch mode (`top`, `iotop`, `nethogs`, `perf` or `pfmon`, etc.) along with the threads that periodically read and parse the output of those programs. In case the workload is predetermined, which is useful for testing, Clavis also analyzes a launch file with the workload description and places the information about the workload into its internal structures (see below).
- *Main loop*. In each scheduler iteration, Clavis monitors the workload, passes the collected information to the scheduling algorithm and enforces algorithm's decision on migrating threads across cores and migrating the memory. It also maintains various log files that can be used later to analyze each scheduling experiment. The main cycle of execution ends if any of the following events occur: the timeout for the scheduler run has been reached; all applications specified in the launch file have been executed at least NR times, where NR is a configuration parameter specified during invocation.
- *Wrap-up*. In this stage, the report about the scheduler's work and the workload is prepared and saved in the log files. The report includes average execution time of each monitored application, the total number of pages that were migrated, the configuration parameters used in this run and so on.

Clavis can either detect the applications to monitor online or the workload can be described by the user in a special launch file. In the first case, any thread on the machine with high CPU utilization (30% as seen in the `top` output), high disk read/write traffic (50 KB/sec) or high network activity (1MB/sec on any interface) will be detected and its respective process will be incorporated into scheduler's internal structures for future monitoring. All the thresholds are configurable. Alternatively, the user can create a launch file in which case the scheduler will start the applications

[6]The word *clávis* means "a key" in Latin. In the past, Clavis greatly helped us to "unlock" the pros and cons of several scheduling algorithms that we designed in the systems lab at SFU.

specified in it and monitor them throughout its execution. Launch file can contain any number of records with the following syntax:

```
<label> <launch time> <invocation string>
***rundir <rundir>
***thread 0 [<CPU ID>] -or-
***numa thread 0 [<CPU ID>, <NODE ID>]
<...>
***thread N [<CPU ID>] -or-
***numa thread N [<CPU ID>, <NODE ID>]
```

Each record describes a single application, possibly multithreaded. In the record, the user can specify a label that will be assigned to the application, which will then represent the application in the final report. If no label is specified, or if the application was detected at runtime, the binary name of the executable is used as a label. The launch time of the application since the start of the scheduler is entered next. This field is ignored when Clavis was started with ”random” parameter, in which case the scheduler randomizes workload start time. The invocation string and run directory for each program are mandatory fields. In case of multithreaded applications, user can specify additional parameters that will be associated with the program threads. Usually, they are core and node IDs the given thread and its memory should be pinned to. The user, however, can utilize these fields to pass any data to the devised scheduling algorithm (e.g., offline signatures for each program thread).

Clavis is a multithreaded application written in C. It has the following file structure:

- *signal-handling.c* - implementation of the scheduler's framework: monitoring, enforcing scheduling decisions and gathering info for the logs.
- *scheduler-algorithms.c* - the user defined implementation of the scheduling algorithms is located here. This file contains several examples of scheduling algorithm implementations with different complexity to start with.
- *scheduler-tools.c* - a collection of small helpful functions that are used throughout the scheduler work.
- *scheduler.h* - a single header file.

Possible modes of Clavis execution will depend on the number of implemented scheduling algorithms. Clavis supports two additional modes on top of that: (1) a simple binding of the workload

to the cores and/or nodes specified in the launch file with the subsequent logging and monitoring of its execution; (2) monitoring the workload execution under the default OS scheduler. Table A.3 lists the log files that Clavis maintains throughout its execution. The source code of the scheduler, samples of the log files, algorithm implementation examples and the user level tools modified to work with Clavis are available for download from [9].

A.6 Conclusion

In this chapter we discussed facilities for implementing user-level schedulers for NUMA multicore systems available in Linux. Various information about the multicore machine layout and the workload is exported to user space and updated in a timely manner by the kernel. Programs are also allowed to change workload thread schedule and its memory placement as necessary. Hardware performance counters, available on all major processor models, are capable of providing additional profiling information without slowing down the workload under consideration. The Clavis scheduler introduced in this chapter is an Open Source application written in C that leverages opportunities for user level scheduling provided by Linux to test the efficiency of scheduling algorithms on NUMA multicore systems.

Monitoring feature	*Description, how to get on user level*
Information about core layout and memory hierarchy of the machine	Data for each core is located at /sys/devices/system/cpu/, including: ./cpu<ID>/cpufreq/cpuinfo_cur_freq - current frequency of the given core. ./cpu<ID>/cache/index<CID>/shared_cpu_list - cores that share a <CID>-th level cache with the given core. ./cpu<ID>/cache/index<CID>/size - cache size.
Which cores share a NUMA memory node	Can be obtained via sysfs by parsing the contents of /sys/devices/system/node/node<NID>/cpulist The same information is also available with the `numactl` package by issuing `numactl --hardware` from the command line.
Which core the given thread is running on	The latest data is stored in the 39-th column of the /proc/<PID>/task/<TID>/stat file, available via proc pseudo fs.
Detection of multithreaded applications	In Linux threads are mostly treated as separate processes. To determine, which of them belong to the same application, the scheduler can read /proc/<PID>/task/<TID>/status file, which contains TGID field common to all the threads of the same application. The thread for which TID = PID = TGID is the main thread of the application. If it terminates, all the rest of the threads are usually terminated with it.
Memory stored on each NUMA node for the given app	The file /proc/<PID>/numa_maps contains the node breakdown information for each memory range assigned to the application in number of memory pages (4K per page). In case of multithreaded programs, the same information can also be obtained from /proc/<PID>/task/<TID>/numa_maps.
Compute bound threads	The threads can be detected by measuring the number of *jiffies* (a jiffy is the duration of one tick of the system timer interrupt) during which the given thread was scheduled in user or kernel mode. This information can be obtained via /proc/<PID>/task/<TID>/stat file (columns 13th and 14th).
I/O bound threads	The `iotop` command-line tool provides the information about read and write traffic from hard drive per specified interval of time for every such thread.
Network bound threads	The `nethogs` command-line tool is able to monitor the traffic on the given network interface and break it down per process.
Detection of memory intensive threads	Refer to Section A.4.

Table A.1: Scheduling features for monitoring as seen from user level.

Action feature	*Description, how to get on user level*
Thread binding	To periodically rebind the workload threads, user level scheduler can use `sched_setaffinity` system call that takes cpu mask and rebinds the given thread to the cores from the mask. The thread will then run only on those cores as is determined by the default kernel scheduler (CFS). The same action can be performed by the `taskset` command line tool.
Specifying memory policy per thread	Detailed description is provided in the Linux Symposium paper by Bligh et al. also devoted to running Linux on NUMA systems [55].
Memory migration	Memory of the application can be migrated between the nodes in several ways: A *coarse-grained migration* is available via `numa_migrate_pages` system call or `migratepages` command line tool. When used, they migrate *all* pages of the application with the given PID from old-nodes to new-nodes (these 2 parameters are specified during invocation). *Fine-grained migration* can be performed with `numa_move_pages` system call. This call allows to specify logical addresses of the pages that have to be moved. The feature is useful if the scheduler is able to detect what pages among those located on the given node are ”hot” (will be used by the thread after its migration to the remote node). *Automatic page migration*. Linux kernel since 2.6.12 supports the *cpusets* mechanism and its ability to migrate the memory of the applications confined to the cpuset along with their threads to the new nodes if the parameters of a cpuset change. Schermerhorn et al. further extended the cpuset functionality by adding an automatic page migration mechanism to it: if enabled, it migrates the memory of a thread within the cpuset nodes whenever the thread migrates to a core adjacent to a different node. The automatic memory migration can be either coarse-grained or fine-grained, depending on configuration [162]. Automigration feature requires kernel modification (it is implemented as a collection of kernel patches).

Table A.2: Scheduling features for taking action as seen from user level.

Log filename	*Content description*
scheduler.log	The main log of the scheduler, contains information messages about the changes in the workload (start/termination of the eligible programs and threads), migration of the memory to/from nodes, the information about a scheduling decision made by the algorithm along with the metrics the algorithm based its decision on, etc. The final report is also stored here upon program or scheduler termination.
mould.log	Information about what core each workload thread has spent the run on and for how long it was there. The log format: time mark since the start of the scheduler (in scheduler iterations): program label sAppRun[program ID in scheduler].aiTIDs[thread ID in scheduler] (#run number) was at coreID-th core for N intervals
numa.log	Contains the updated information about node location of the program's memory footprint in the format: time mark: label sAppRun[progID].aiTIDs[threadID] (#run number) number of pages on the 0-th node <...> number of pages on the last node for N intervals
vector.log	This log contains the updated data about the resources consumed by each program that is detected online or launched by the scheduler. The log format is: time mark: PID label CPU core utilization in % MISS RATE Memory utilization in % TRAFFIC SNT RCVD IO WRITE READ
systemwide.log	The updated information about the monitored hardware counters is dumped here in every scheduling iteration.
time.log	Number of seconds every program was running along with the time it has spent on user and kernel level.

Table A.3: Log files maintained by Clavis during its run.