UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

POWER-CONSTRAINED AWARE AND LATENCY-AWARE MICROARCHITECTURAL OPTIMIZATIONS IN MANY-CORE PROCESSORS

by Sudhanshu Shekhar Jha

Improving the Efficiency of Multicore Systems Through Software and Hardware Cooperation

A DISSERTATION PRESENTED
BY
VÍCTOR JAVIER JIMÉNEZ PÉREZ
TO
THE DEPARTMENT OF COMPUTER ARCHITECTURE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

UNIVERSITAT POLITÈCNICA DE CATALUNYA – BARCELONA TECH
BARCELONA
SEPTEMBER 2016

Thesis advisor: Dr. Francisco Cazorla
Thesis advisor: Dr. Mateo Valero Víctor Javier Jiménez Pérez

Improving the Efficiency of Multicore Systems Through Software and Hardware Cooperation

Abstract

Increasing processors' clock frequency has traditionally been one of the largest drivers of performance improvements for computing systems. In the first half of the 2000s, however, it became clear that continuing to increase frequency was not a viable solution anymore. Power consumption and power density became prohibitely costly, and processor manufacturers moved to multicore designs. This new paradigm introduced multiple challenges not present in single-threaded processors. Applications running on multicore systems share different resources such as the cache hierarchy and the memory bus. Resource sharing occurs at much finer degree when cores support multithreading as well. In this case, applications share the processor's pipeline too. Running multiple applications on the same processor allows for better utilization of its resources—which otherwise may just lie idle if an application does not use them. But sharing resources may create interferences between applications running on the system. While the degree of these interferences depends on the nature of the applications, it is typically desirable to reduce them in order to improve efficiency.

Most currently available processors expose a set of sensors and actuators that software can use to monitor and control resource sharing among the applications running on a system. But it is typically up to end users to analyze their workloads of interest and to manually use the actuators provided by the processor. Because of this, in many cases the different mechanisms for controlling resource sharing are simply left unused. In this thesis we present different techniques that rely on software/hardware interaction to monitor and improve application interference—and thus improve system efficiency. First we conduct a quantitative study showing the benefits of hardware/software cooperation on system efficiency. Then we narrow our focus on a given hardware knob: data prefetching. Specifically we develop and evaluate several adaptive solutions for improving the efficiency of hardware data prefetching on multicore systems. The impact of the solutions presented in this thesis, however, goes beyond the particular case of data prefetching. They serve as illustrative examples for developing software/hardware cooperation schemes that enable the efficient sharing of resources in multicore systems.

Resource sharing in a processor is a critical factor that significantly affects system efficiency. But resource sharing also occurs at other levels in a computing system. In large-scale computing facilities applications might also share storage and networking resources for instance. As a case study we consider the design of an energy accounting system relying on hardware/software cooperation for large-scale computing facilities. We explore multiple alternatives for the required sensors and actuators, as well as the inherent trade-offs in the design of such a system.

Contents

Listing of Figures

Dedicated to my wife and parents,
for their constant support.

Acknowledgments

Ever since I was a child, my parents strove to give me a good education. When I became interested in computers, they did not hesitate to buy me one—even if at that time computers were not precisely affordable. Later, they always encouraged me to continue studying, and supported me and my decisions. For all this, I am very grateful to them. Thanks to my brother for bringing joy and fun to my family, and for enduring my (failed) attempts to spark his interest in computers. Amusingly, he is now professionally dealing with computers.

I met my wife soon after starting my PhD. It was supposed to take just a few years, but it ended up taking quite a bit longer. I am thankful for her endless support and for cheering me up during difficult times. My son was born when I was close to finishing this thesis. His needs certainly delayed its end, but the joy he brought to my life simply outshines everything else. I want to thank him for not complaining too much when I needed to devote some time to complete this thesis.

I am indebted to my advisor, Francisco Cazorla, for his direction and encouragement since day one. His constructive criticism had a large influence on my development as a researcher. I am grateful for his trust and the freedom he allowed me to have during my studies. Roberto Gioiosa provided lots of insights and contributed to lively discussions that significantly helped my research. I thank him for his teachings. I want to show my gratitude to Mateo Valero for all his support and for being always there when difficulties appeared on the path.

During the internships I did at the IBM T. J. Watson Research Center, Pradip Bose and Alper Buyuktosunoglu guided my research as well. In a way, they became de facto advisors of my PhD studies. Pradip was always eager to help me no matter what the problem was. Beyond invaluable Alper's mentoring, my friendship with him certainly made my time there more enjoyable. It was great to watch together Barça winning so many titles. I thank Pradip and Alper for all their teachings and all the opportunities that they provided me. I also want to thank Francis P. O'Connell, Canturk Isci, Chen-yong Cher and Eren Kursun for their help while I was at IBM. During my stays at IBM, some friends made me feel closer to home. I am thankful to Ramon, Augusto, Valentin and Alex for it.

The path that led to my PhD studies started when I decided to enrol in the master program at the Computer Architecture Department at UPC. I am grateful to Marisa Gil, Nacho Navarro and Xavi Martorell for their support during that time. During my master studies I was a visitor student at INRIA. I thank Grigori Fursin for the opportunity. I also want to thank Isaac Gelado for all his help with my master thesis.

The C6 building is a special place for me. There, I met many people who made the process towards obtaining my PhD a much better experience. Many of them continue to be my friends to this day. I want to thank Zoraida, Carlos González, Àlex, Jordi, Abhishek, Carlos Boneti, Ramon, Lluís, Javi, Carlos and Isaac. I specially want to thank Carlos Boneti for all his help during the first stages of my

PhD in the CAOS group at Barcelona Supercomputing Center (BSC). I am grateful to all the other members of the CAOS team for their friendship.

Even if they do not really understand very much what my research is about, I am grateful to my friends Cesc and Sergi for being part of my life before, during and (hopefully) after my PhD. Soon, they will not be able to continue to make fun of my eternal student condition. I also want to thank my undergraduate friends: Alex, Juanan, Marc, Bea, Lou, Lucas and many more. The list is simply too long. During all these years I met other people who in one way or another helped me to finish my PhD. I am thankful to all of them.

Last but not the least, I thank all the institutions that supported me during my PhD studies: the Computer Architecture Department at the UPC, the HiPEAC Network of Excellence, the Barcelona Supercomputing Center and IBM Corporation. Specifically, this work has been partially sponsored by Defense Advanced Research Projects Agency (DARPA), Microsystems Technology Office (MTO), under contract no. HR0011-13-C-0022. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This document is: Approved for Public Release, Distribution Unlimited. This work has also been partially supported by: the Spanish Ministry of Economy and Competitiveness (MINECO) under contracts TIN2012-3455 and TIN2015-65316-P; the Spanish Ministry of Science and Technology of Spain under contracts TIN-2007-60625 and JCI-2008-3688, and grants AP-2005-3776 and AP-2005-3318; and the HiPEAC Network of Excellence (IST-004408).

1 Introduction

In the last two decades of the past century computer scientists relied on the scaling laws for CMOS devices to improve processor performance. Increases in clock speed drove processor performance during that time—processor clock frequency doubled approximately every 18 months. In the first half of the past decade, however, the slowdown of Dennard scaling made processor manufacturers to embrace the multicore design (Haensch et al.[45], Horowitz et al.[49]). That move allowed processor performance to continue to increase by scaling up the number of cores instead of the clock rate. But it also introduced multiple challenges not present in single-threaded processors. Suddenly, traditional problems in the parallel computation realm such as thread coscheduling or resource sharing appeared in the context of a single multicore processor.

Multicore processors allow multiple applications to concurrently run on a single processor. This

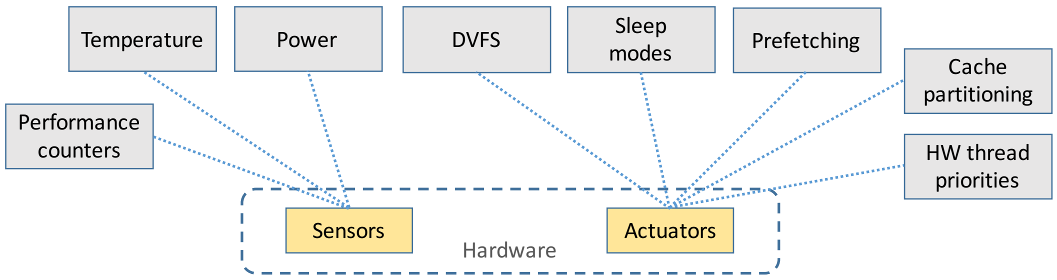


Figure 1.1: Examples of sensors and actuators exposed by hardware to software.

typically increases the utilization of the different units in the processor such as caches and functional units (assuming the processor supports multithreading too). But it also creates interferences between applications running on the system as they compete for the usage of shared resources. In order to alleviate these interferences—and therefore increase performance—several hardware techniques to adaptively manage resource sharing were developed over time (e.g., Tullsen et al.[124] studied several SMT fetch policies and Qureshi & Patt[101] presented a runtime mechanism to partition shared caches). Hardware-based adaptive policies are widely used in modern processors and they have proved successful at increasing performance or improving energy efficiency. Yet, in this thesis we show that there is room for improvement: adaptive policies can be further enhanced by relying on software and hardware cooperation. In this approach, hardware exposes sensors and actuators to software. A software layer uses the sensors to gather all the required information to implement smart adaptive resource-management policies. Such policies then use the actuators to adapt hardware resource sharing to the running workloads based on a certain metric of interest (e.g., performance, power consumption or quality of service).

1.1 Sensors and Actuators

Current processors contain multiple sensors that users can use for different purposes (Bowhill et al.[18], Sinharoy et al.[111]). Some examples of such sensors can be seen in Figure 1.1. Performance counters (PMCs) are a set of special-purpose registers built into the performance monitoring unit (PMU) in a processor. The PMU can be programmed to measure a broad selection of microarchitectural events such as the number of instructions completed, cache misses or cycles that the processor was stalled because of different reasons. Measures obtained from PMCs can help performance monitoring, workload characterization and application tuning (Anderson et al.[5], Zagha et al.[132]). They can also be helpful during the verification of a processor design to validate the exepcted system performance (Srinivas et al.[117]). Typically the OS exposes these special-purpose registers through some interface so that users can use them to measure system performance. Eranian[37] implemented such an interface in the form of a patch for the Linux kernel. Later, native support for PMCs was added to the Linux kernel (Carvalho de Melo[22]).

In addition to performance measurement, modern systems also allow users to obtain power consumption measurements. This information can then be used—potentially together with performance measurements—to build adaptive power management techniques. For instance, AMESTER (IBM Automated Measurements of Systems for Temperature and Energy Reporting software) is an in-house solution to monitor IBM POWER systems (Floyd et al.[38]). This software operates in an out-of-band manner, thus avoiding any performance overhead on the system being measured. Power consumption can be measured for different components (e.g., processor and memory). Other metrics such as temperature, voltage and frequency can also be obtained by reading the sensors in POWER systems.

The IBM POWER7 also includes a power proxy that estimates power consumption for each core based on PMC data (Floyd et al.[38]). Empirical results show that estimations obained from the power proxy are close to the actual power consumption measurements. Some of the advantages of this approach are a finer sampling granularity—the interval can be as small as a few microseconds—as well as the ability to independently estimate power consumption for each core.

Processor designers also expose multiple actuators (or knobs) so that software can program and control the behavior of the processor (see Figure 1.1 for examples of such actuators). Dynamic frequency and voltage scaling (DVFS) (see Chandrakasan et al.[24]), core sleep modes (Floyd et al.[38]), hardware thread priorities (Boneti et al.[16]) or programmable prefetching engines (Sinharoy et al.[111]) are some examples of such knobs. Processors typically expose these actuators to the OS through special-purpose registers or privileged instructions.

1.2 Problem Statement

Support for controlling some of these actuators is built into current operating systems. For instance, Linux contains a mechanism to control processor frequency and voltage based on workload demands (Pallipadi & Starikovskiy[97]). In many cases, however, it is left to the end users to manually program these actuators. Because of that, quite often these actuators end up not being used. The reason is typically the costly workload characterization and optimization process necessary to select the optimal setting for a particular actuator. That process further complicates when users are not just running a small set of workloads, but their systems run a broad mix of workloads with different characteristics.

An adaptive mechanism that tunes hardware settings based on workload characteristics has the potential to increase system efficiency—either from a pure performance perspective or from an energy point of view. A block diagram of such a mechanism is depicted in Figure 1.2: the hardware exposes *sensors* and *actuators* to the software—typically to the OS or some firmware. The *sensor collector* reads the hardware sensors to obtain workload resource usage. Based on that information, a set of adaptive policies take resource allocation decisions. The specific decisions depend on the optimization metric of interest (e.g., performance, power consumption or quality-of-service). The *actuator controller*

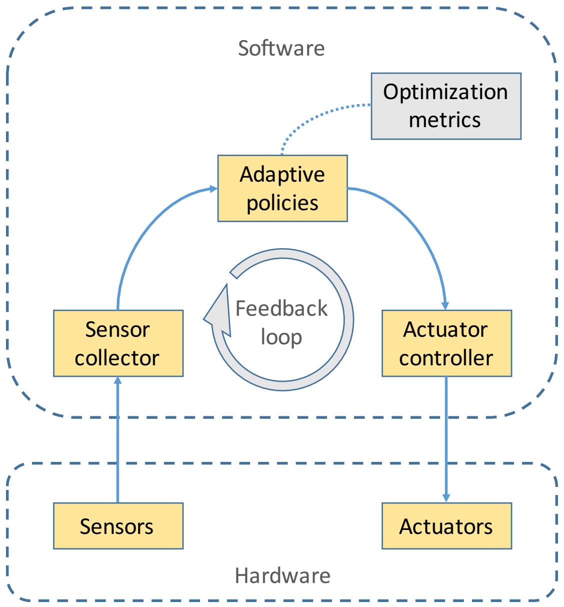


Figure 1.2: Design overview for the proposed adaptive resource management system.

enforces such decisions by configuring the actuators accordingly.

The design of such an adaptive mechanism follows the principle of hardware-software codesign (Shalf et al.[109]), which proposes deeper collaboration between the hardware design and the application teams. At the core of this approach lies an iterative optimization loop where application design influences hardware design decisions, and vice versa. It also encourages workload autotuning to optimize applications for the specific platforms they run on. An adaptive resource management mechanism has the potential to optimize a system well beyond the time while the system is being designed—further optimizing the system once it has been delivered to its users.

In this thesis we explore the potential of leveraging hardware actuators to improve system efficiency. First we conduct a quantitative study that shows the benefits of a hardware/software cooperation approach. Then we narrow our focus to a given knob: hardware data prefetching. This choice is based on its potential performance impact—caches continue to be critical to system performance—and the lack of adaptive solutions that tailor this knob. We present and evaluate different adaptive techniques that rely on hardware/software cooperation in order to intelligently control prefetching on a multicore system. The results show that our policies significantly increase performance and might reduce memory bandwidth and power consumption too—effectively making the system more efficient. Resource sharing occurs as well at other levels in a computing system. For instance, in large-scale computing facilities applications—and their users—might share a wide variety of resources, ranging from processor and memory to storage and networking. In this context we describe the design of an energy accounting system for large-scale computing facilities, and we analyze different forms of hardware/software cooperation required to build such a system.

1.3 Using Hardware Data Prefetching as an Actuator

Hardware data prefetching is a well-known technique to help alleviate the so-called *memory wall* problem (Wulf & McKee[129]) and hide memory latency. The technique relies on the fact that many applications exhibit spatial locality (i.e., once a given memory address is accessed, it is very likely that surrounding addresses will be accessed in the near future). Upon a data cache miss for a given address, the prefetcher may speculatively bring consecutive blocks corresponding to the addresses that the application is likely to access in the future. More complex prefetch implementations may detect access patterns to non-consecutive data (e.g., pointer-based list traversal).

Many general purpose server-class microprocessors in the field today rely on data prefetch engines to improve performance for memory-intensive workloads. Some prefetch engines allow users to change some of their parameters. In current commercial systems, however, the hardware prefetcher is typically enabled in a default configuration during system bring-up, and dynamic reconfiguration of the prefetch engine is not an autonomic feature. Nonetheless, commonly used prefetch algorithms—when applied in a fixed, non-adaptive mode—will not help performance in many cases. In fact, they may actually degrade it due to useless bus bandwidth consumption and cache pollution. These problems exacerbate with current chip multiprocessors (CMP) and simultaneous multithreading (SMT) processors, which contain a significant number of cores (e.g., IBM POWER7 has 8 cores). And given the current trends future processors will have larger core counts. Executing many threads concurrently can potentially stress the bandwidth between processor and memory (Rogers et al.[104]). In this scenario, bandwidth can easily be saturated even without the presence of data prefetching. Enabling data prefetch may degrade system performance, since prefetches will fight with demand loads for the scarce available bandwidth.

In this thesis, we present multiple adaptive solutions that dynamically adapt the prefetching configuration to the running workloads. We first present a scheme that optimizes the prefetching setting for every thread running on a system. Then, we describe a system-wide solution that tackles the problem of managing global memory bandwidth by intelligently shifting prefetching bandwidth resources among applications. Our mechanisms successfully achieve significant performance improvements. In some cases they are also capable of reducing power consumption.

We use the IBM POWER7 (Sinharoy et al.[111]) as the vehicle for this study, since: (i) it represents a state-of-the-art high-end processor, with a mature data prefetch engine that has evolved significantly since the POWER3 time-frame; and (ii) it provides facilities for accurate measurement of performance and power metrics. POWER7 contains a programmable prefetch engine that is able to prefetch consecutive data blocks as well as those separated by a non-unit, constant stride. The processor system is provided to customers with a default prefetch setting that targets to improve performance for most applications. But users can manually override the default setting via the operating system, if needed. Users can specify some parameters such as the prefetch depth and whether strided prefetch and prefetch for store operations should be enabled or not. Changing the prefetch configuration affects workloads in different ways depending on the workload nature. This is the case even within the class of scientific-engineering applications, which are generally amenable to data prefetch. While the optimal prefetch setting—if known—can lead to a significant performance improvement, the corollary to this, as we show in this thesis, is that blindly setting a configuration may reduce performance and waste power consumption.

1.4 Contributions

This dissertation makes the following contributions:

1. We characterize the impact of multiple actuators on a real CMP/SMT platform. We also assess the potential of adaptive resource-management techniques that leverage hardware actuators.

2. We conduct an extensive characterization of the hardware data prefetching unit included in the IBM POWER7 processor. We analyze the impact of multiple configurations on both performance and power consumption. To that end, we use a combination of well-known benchmarks as well as microbenchmarks.

3. We present and evaluate a runtime-based adaptive prefetching mechanism capable of improving performance via dynamically setting the optimal prefetching configuration, without the need for a priori profile information. Our adaptive scheme increases performance up to 2.7X and 1.3X compared to the default prefetching configuration for single-threaded and multiprogrammed workloads, respectively. Our mechanism is able to reduce memory power consumption in some cases. We also study the implementation of such an adaptive prefetching scheme within the OS kernel. After implementing our adaptive mechanism into the Linux kernel, we have observed similar performance improvements to those obtained by the userspace implementation.

4. We provide a motivation for prefetch-based bandwidth shifting and a characterization of the performance-bandwidth trade-off for multiple benchmarks. We introduce a metric that esti-

mates prefetch usefulness for a given thread based solely on performance counters commonly available in current processors. A novel bandwidth shifting mechanism is capable of significantly improving system performance by taking bandwidth away from benchmarks that do not use prefetching in an efficient way, and giving it to prefetch-efficient benchmarks. The mechanism does not require any hardware support, and it is able to obtain up to 18.5% speedup (10-11% on average). We also study the impact of bandwidth shifting in extreme cases where one benchmark is highly prefetch-efficient and the other uses prefetching inefficiently. Our results show that bandwidth shifting achieves much larger speedups (>1.6X). We also evaluate the impact of the bandwidth shifting mechanism on power consumption.

5. In the last chapter we look beyond the single-system case. We explore how sensors and actuators can be used to accurately track per-task energy consumption in large-scale computing facilities. We study the different trade-offs inherent in the design of an energy accounting solution. We also show how such approach can be beneficial for both users and owners of the facility.

During this thesis we ported AMESTER—an internal power measurement tool at IBM (see Floyd et al.[38], Lefurgy et al.[70])—to C++. By doing so, we could extend the framework where our adaptive policies are implemented with support for power measurement. In addition to that, members of the System and Technology Group (STG) at IBM collaborated in the research conducted during this thesis. We believe the results from our research have provided valuable feedback to IBM's product group.

1.5 List of Publications

The different parts in this thesis have been published in international conferences or technical journals. The following list contains the details for the publications that spawned from this thesis.

- Jiménez, V., Cazorla, F. J., Gioiosa, R., Valero, M., Boneti, C., Kursun, E., Cher, C., Isci, C., Buyuktosunoglu, A., & Bose, P. (2010). Power and thermal characterization of POWER6 system. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT (pp. 7–18)

- Jiménez, V., Cazorla, F. J., Gioiosa, R., Valero, M., Boneti, C., Kursun, E., Cher, C., Isci, C., Buyuktosunoglu, A., & Bose, P. (2011b). Characterizing power and temperature behavior of power6-based system. *IEEE Journal Emerging and Selected Topics in Circuits and Systems, JETCAS*, 1(3), 228–241

- Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., & O'Connell, F. P. (2012). Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT (pp. 137–146).: ACM. *Best Paper Award Nominee.*

- Jiménez, V., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., & Mealey, B. G. (2014). Adaptive prefetching on POWER7: improving performance and power consumption. *ACM Transactions on Parallel Computing, TOPC*, 1(1), 4

- Jiménez, V., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., Cazorla, F. J., & Valero, M. (2015). Increasing multicore system efficiency through intelligent bandwidth shifting. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, HPCA (pp. 39–50).: IEEE

- Jiménez, V., Cazorla, F. J., Gioiosa, R., Kursun, E., Isci, C., Buyuktosunoglu, A., Bose, P., & Valero, M. (2011a). Energy-Aware Accounting and Billing in Large-Scale Computing Facilities. *IEEE Micro*, 31(3), 60–71

Chapter 7 in this thesis orginated a new line of research. The following two publications expand some of contents in this thesis.

- Liu, Q., Moretó, M., Jiménez, V., Abella, J., Cazorla, F. J., & Valero, M. (2013). Hardware support for accurate per-task energy metering in multicore systems. *ACM Transactions on Architecture and Code Optimization, TACO*, 10(4), 34
- Liu, Q., Jiménez, V., Moretó, M., Abella, J., Cazorla, F. J., & Valero, M. (2014a). Per-task energy accounting in computing systems. *Computer Architecture Letters*, 13(2), 85–88

1.6 Dissertation Organization

This dissertation is divided into eight different chapters. Chapter 2 describes the related work relevant to this thesis. Chapter 3 provides a motivation showing the impact of hardware actuators. Chapter 4 describes and characterizes the platforms used in this thesis. It also contains the experimental methodology. Chapter 5 presents an adaptive prefetching mechanism to improve single-threaded performance. Chapter 6 shows a system-wide solution that shifts prefetching bandwidth across workloads. Chapter 7 focuses on resource sharing for large-scale computing facilities. It describes the trade-offs in the design of such a system, and analyzes different sensors and actuators that can be used. Finally, Chapter 8 provides the main conclusions of this thesis and directions for future work.

2 Related Work

A significant amount of research on adaptive solutions for resource management has been conducted over the years. As computing systems continue to share resources at finer granularities, more opportunities for such solutions appeared. Adaptive resource management can be entirely implemented in hardware or they might rely on hardware/software cooperation. We provide a detailed overview of the related work in the field of adaptive resource management. Moreover, as this thesis presents two adaptive solutions for controlling the prefetching engine, we also include a list of relevant prior work on data prefetching.

2.1 Hardware Solutions for Resource Management

Multicore processors typically share the last level cache among the different cores. Therefore, applications running on such processors compete for space in the last level cache. Suh et al. [121] present a dynamic cache partitioning scheme. Their solution adds extra hardware but the control algorithm is actually implemented at the operating system level. Qureshi & Patt [101] enhance the previous work by eliminating any effect on the partitioning algorithm due to interferences between the actual applications sharing the cache. Moreto et al. [87] present a cache partitioning technique but they focus on achieving QoS for the co-running applications. Parallel applications running on a multicore processor need to maintain cache coherency among the different cores. Cache coherency traffic competes for the available bandwidth in the interconnection bus. Martin et al. [82] use an adaptive bandwidth snooping protocol. Depending on the number of processors and the running workloads, the protocol selects a different policy to keep cache coherency, effectively optimizing the usage of the interconnection bus.

If a processor supports multithreading, resource sharing occurs in the processor's pipeline too. Threads running on a particular core will share multiple resources such as the instruction fetch buffers, issue queues or reorder buffer. Tullsen et al. [124] present different instruction fetch policies that try to maximize the usage of resources in the pipeline. Cazorla et al. [23], Choi & Yeung [26] show fetch policies aiming to increase throughput or provide quality of service (QoS).

2.2 Sampling-Based Online Adaptive Systems

Offline adaptive systems typically rely on application profiling to optimize future executions of a particular application. This approach allows for the usage of computationally expensive optimization schemes, but its utility is limited when different inputs are used or when the application mix running on a system changes over time. Online adaptive systems, on the other hand, monitor running applications and make optimization decisions while the applications are running. Many online adaptive systems use a two-step approach. During the first step sensors are read to obtain measurements for the metric of interest (e.g., performance, power consumption or temperature). Then, in the second step an optimization decision is made based on the information obtained in the first step.

Isci et al.[55], Sarikaya et al.[106] present solutions that uses performance monitoring counters (PMCs) to predict application phases. These solutions then use phase information to guide dynamic power management schemes. Tikir & Hollingsworth[123] develop an online page migration scheme that characterizes the memory access pattern of an application, and then moves pages to memory local to the processor that accesses them most frequently. Petrica et al.[98] present a runtime that optimizes system's efficiency by enabling and disabling parts of the processor's pipeline in a fine-grain approach.

Lu et al.[79] relies on performance monitoring counters and dynamic instrumentation to insert prefetching instructions in a running application. Adl-Tabatabai et al.[3] present a similar system for JIT-based runtimes. Schneider et al.[108] use PMCs to optimize object spatial locality in a generational garbage collector.

2.3 Solutions Exposing Custom Sensors and Actuators to the Software

Most current, general-purpose processors expose different sensors (or counters) to the operating system or userspace. For instance, performance monitoring counters allow software to obtain a significant amount of information related to applications' usage of the different subunits in the processor. This information can be used to analyze the behavior of the applications or find their performance bottlenecks. Processors and other components in a computer system may expose power consumption and temperature sensors too. Processors can also expose actuators so that the operating system or the end user are able to alter the behavior of the processor. Two commonly used actuators let the processor's frequency and sleep states to be controlled.

Prior research on resource management has proposed exposing new sensors or actuators that enable the implementation of more complex adaptive schemes. Zhou et al.[133] present a memory allocation adaptive solution that uses the page miss ratio curve to guide its decisions. Their proposal adds extra hardware to collect the page miss ratio curve and it exposes that information through a new sensor that the operating system has access to. Suh et al.[120] utilizes a set of novel hardware counters that expose information about the isolated miss-ratio for each process running on the system. This information can be used to guide scheduling decisions or to dynamically partition the cache. Merten et al.[84] propose sensors that expose applications' hot spots to the operating system. An adaptive scheme could use these sensors to optimize the performance in the hot spots. Yasin[131] adds extra events to the performance monitoring unit of a processor to accurately find the bottlenecks of an application running on that processor. Nagarajan & Gupta[91] expose interprocessor dependence information and build a software solution that efficiently detects mispeculation and improves application reliability. Boneti et al.[16] implement and characterize a kernel module to access the hardware thread prioritization mechanism present in the IBM POWER5 processor. A follow-up work, presents an adaptive solution to balance parallel applications using the thread prioritization mechanism.[17] Bhattacharjee & Martonosi[14] present a sensor that exposes thread criticality predictions to the operating system. Using the measurements obtained from that sensor they present an adaptive scheme to reduce thread imbalance as well as another scheme to improve energy efficiency in barrier-based parallel applications.

2.4 Thread Mapping

Although thread mapping might apparently not look like a traditional actuator—in the sense of a knob that can be tweaked—the way threads are mapped to cores or hardware contexts has a large impact on the exact resource sharing among these threads. For instance, the interference level between threads sharing a last-level cache might be very different depending on the particular mapping that the operating system chooses. Also, threads running on multithreaded processors such as the IBM

POWER7 may have different access to pipeline's resources depending on the multithreading level being used.[111] Therefore, thread mapping might affect performance in this case too.

Radojković et al.[102] use a statistical approach based on extreme value theory to optimally assign tasks in multithreaded processors. Tang et al.[122] study the impact of thread-to-core mappings in terms of cache and bus bandwidth sharing. They also present an adaptive approach to thread mapping in a data center, resulting in significant performance gains. Lo et al.[78] show an adaptive solution to safely colocate latency-sensitive applications on a data center.

2.5 General Prefetching

There is a significant record of past research in data prefetch (e.g., Baer & Chen[7], Fu et al.[41], Jouppi[63], Palacharla & Kessler[95], Smith[113]). Most of the initial proposals were based on sequential prefetchers, which rely on applications exhibiting spatial locality. Although sequential prefetchers work effectively in many cases, there are applications with non-sequential data access patterns that do not benefit from sequential prefetching. This has motivated the research on more complex prefetchers that try to capture the non-sequential nature of these applications. Cooksey et al.[27], Ebrahimi et al.[34], Roth et al.[105], Wang et al.[127], Yang & Lebeck[130] study prefetch techniques targeting pointer-based applications. Joseph & Grunwald[62] study Markov-based prefetchers and present solutions to limit the bandwidth devoted to prefetching. Solihin et al.[114] use a user-level memory thread in order to prefetch data, delivering significant speedups even for applications with irregular accesses. Emma et al.[35], Srinivasan et al.[118] present limit studies and prefetch analytical models.

Most general-purpose processors contain a prefetch engine based on some of these works. Our solution is orthogonal to them since their objective is to improve the accuracy of the algorithms implemented in a single prefetcher.

2.5.1 Filtering Useless Prefetches

Several works that attempt to reduce the number of useless prefetches sent to memory have been presented in the past (Charney & Puzak[25], Lee et al.[69], Lin et al.[73], Mowry et al.[88], Mutlu et al.[90], Zhuang & Lee[134]). Even when these filtering techniques are used, applications still have different prefetch-efficiency degrees. Therefore, our bandwidth shifting mechanism can be complementary used to further improve performance.

2.6 Local Adaptive Prefetching

Using a prefetch engine that implements a fixed algorithm is suboptimal since the prefetch-efficiency of applications may change during the different phases of execution. Several adaptive solutions that attempt to dynamically change the prefetch configuration exist in prior research (Dahlgren et al.[28,29], Nesbit et al.[94], Srinath et al.[116]). The objective of these solutions, however, is not to maximize global system performance. They are either designed for single-threaded processors or they only attempt to locally increase performance of individual cores. Therefore, while they may improve performance for a particular core, system performance may decrease. On the contrary, in this thesis we present a solution that maintains a global system view and increases performance for the whole system.

2.7 CMP-Aware Adaptive Prefetching

With the advent of CMP processors, interaction between threads must be taken into account when designing a prefetch system. Ebrahimi et al.[33,32] study the effect of thread-interaction on prefetch, and propose techniques to design prefetch systems that improve throughput or fairness. Despite the similarities to the solutions presented in this thesis, their solution requires costly extra hardware—amounting to multiple kilobytes—whereas ours work with most modern general-purpose processors.

Liu & Solihin[74] study the impact of prefetching and bandwidth partitioning in CMPs. But their work only presents an analytical model and no mechanism to exploit their observations is included.

2.8 Adaptive Prefetching Solutions for Real Systems

Although there is a significant number of studies on prefetching based on simulators, there are very few works that deal with hardware-based measurement and characterization. Wu & Martonosi[128] characterize the prefetcher of an Intel Nehalem processor and provide a simple algorithm to dynamically control whether to turn the prefetcher on or off. Their study, however, is solely oriented towards reducing intra-application cache interference without taking actual system performance into consideration. Liao et al.[72] construct a machine learning model that dynamically modifies the prefetch configuration of the machines in a data center (based on Intel Core2 processors). Although they improve performance for some applications by enabling/disabling prefetch, their work only focuses on how to improve the performance for a single application without taking into account the performance/bandwidth trade-offs that appear when multiple applications are executed concurrently.

2.9 Per-Task Energy Accounting

Large-scale computing facilities are vast infrastructures with high operation costs. Any possible optimization that improves their efficiency can translate into a considerable cost reduction. Several proposals focus on improving data centers' energy efficiency (Moreira & Karidis[86], Karidis et al.[66], Meisner et al.[83]). Many of these proposals advocate for energy-proportional systems, in which the benefits of energy accounting are higher than in current systems.

Several of these works focus on either reducing power consumption when the system is idle or improving efficiency by consolidating more virtual machines in the same hardware. To this end, Moreira & Karidis[86] leverage workload heterogeneity to better schedule the workloads onto the computing resources, thus increasing resource usage. Nathuji & Schwan[92] propose a mechanism to connect the low-power mechanisms available in the hardware with the power management requests and hints made by an operating system running within a virtual machine.

Accounting users, tasks and virtual machines for the energy they actually consume is orthogonal to the aforementioned proposals. On the one hand, the potential to adapt to the workloads' heterogeneity increases with per-task energy accounting. On the other hand, energy accounting brings benefits by itself as shown earlier.

Kansal et al.[65] present initial steps for an accurate energy-accounting mechanism. Their goal is to develop a better power-capping mechanism in the presence of multiple virtual machines on one node. However, more research is necessary to obtain a more accurate mechanism for use not only for power consumption estimation, but for billing users according to their energy consumption as well. For instance, their proposal uses simple ways to split static power consumption and power consumption caused by virtual machine interferences, among virtual machines.

To obtain better accuracy, hardware and operating system support is necessary. Bertran et al.[13] present an energy-accounting system for small-sized systems. Our work focus on large-scale computing facilities, where other types of solutions are likely needed.

3 Motivation: Impact of Hardware Actuators

Workloads running on a multicore processor compete for different shared resources. Due to workload variability, the usage of these resources might be significantly unbalanced. This fact has important implications on performance, power consumption and energy efficieny. In this chapter we characterize these implications and we analyze the potential to improve system efficiency of solutions based on hardware/software cooperation.

As process technologies advance, the trend is to have more cores per chip, where each core can further increase the amount of concurrent threads via SMT. This has been the case for processors such as the IBM pSeries (POWER6[68], POWER7[111], POWER8[112]) and the Intel Xeon.[18] While CMP processors provide better performance per watt ratios than monolithic architectures, the power dissipation continues to be a key performance limiter also for multithreaded architectures. Consequently, power and thermal characteristics of processors are one of the primary design constraints, and motivate an active research area.

Energy, power and thermal management are of paramount importance in many environments ranging from embedded to High Performance Computing (HPC) systems. In the former case, improvements in battery capacity simply have not kept pace with ever-more-powerful processors, limiting device use to short time periods. In the latter case, supercomputers and data centers provide huge amounts of computation power (necessary, for instance, for weather forecasting, climate research, molecular modeling and other areas), with very high associated energy costs. A study from the U.S. Environmental Protection Agency (EPA) estimates that national energy consumption by servers and data centers will reach more than 100 billion kWh annually and representing $7.4 billion in electricity cost.[36] In HPC systems, besides the heat dissipation issue, the increasing power consumption leads to additional problems in the power delivery and energy costs that account for a considerable percentage of the expenses of a data center. It is certain that managing and reducing the temperature and power consumption is a critical problem that must be addressed in all levels of computing systems, from the application layer to the hardware. As an example, most of the latest processors available in the market employ several techniques to reduce power consumption.[40,93] From the OS perspective, the Linux kernel also implements features to reduce power.[96,110,119]

In this chapter we explore the power and thermal behavior of various power management techniques provided by the IBM POWER6 processor and evaluate their impact at multiple levels:

Hardware level We demonstrate the impact of POWER6's hardware-thread prioritization mechanism on power consumption. Our results show that workload-aware manipulation of thread priorities improves system's energy-delay product by as much as 25%. We also show the power and thermal characteristics of the nap mode, and the combined effect of employing the nap mode and hardware thread priorities. These evaluations show very significant benefits, reducing core temperature up to 26% and total system power consumption by 25%.

OS level We explore the effectiveness of power and thermal management techniques present in modern OS for the POWER architecture: the tickless mechanism and the idle power manager. We demonstrate the benefits of these approaches and their dependence on other system components such as timer interrupt periods.

Application level We characterize system behavior with a set of microbenchmarks and SPEC CPU2006 benchmarks. We correlate power and temperature with performance counters and derive a model capable of estimating system power consumption with an average error under 4.5%. We also look at the potential benefits of hardware-aware OS scheduling. A JS22 system includes two POWER6 chips, each of which is a CMP/SMT chip. In such a system, thread placement affects both performance and power consumption. By placing threads in a workload- and package-aware manner, we can achieve significant energy improvements, without incurring significant performance degradation, with a 3.7X reduction in energy-delay product.

We use such multi-level characterization as motivation examples to show the potential benefits of hardware/software cooperation on system efficiency.

3.1 The IBM POWER6 Processor

POWER6 is a dual-core chip where each core can run in two-way SMT mode. The design is optimized for the server market and it features a mostly in-order pipeline that supports high frequencies in excess of 4 GHz. Despite being an in-order processor it supports limited out-of-order execution for floating point operations. The processor microarchitecture blocks are shown in Figure 3.1. Each core has a 64 KB L1 instruction and data cache. Cores have a 4 MB private L2 cache connected to the L3 controller and to the memory controller through the symmetric multiprocessor (SMP) interconnect fabric. The

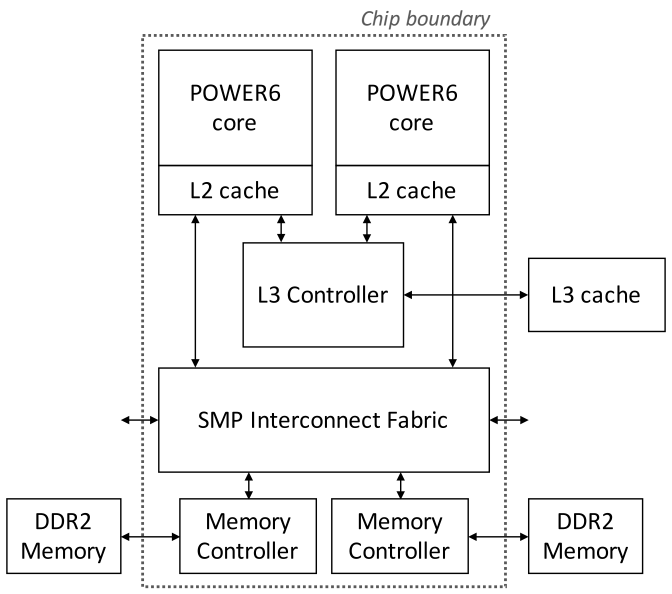


Figure 3.1: POWER6 microarchitecture overview.

optional off-die L3 cache is shared by both cores. Depending on the configuration, each chip has one or two memory controllers that interface to the DRAM memory. Our system, being an entry-level one, does not have the L3 cache and it contains only one memory controller.

POWER6 systems integrate a thin hypervisor layer that abstracts the real hardware and provides the capability of running several virtual machines simultaneously on the same physical resources. This virtualization mechanism is completely transparent and does not require any modification of the guest OS. But, collaboration between the guest OS and the hypervisor has significant benefits for improving chip utilization and throughput as well as for effective power management.

In this thesis we are particularly interested in the interaction between the guest OS and the hypervisor for effective power and performance management. POWER6 implements specific methods for the guest OS to release hardware threads and cores when there are no runnable processes available. This is done by invoking the `cede_processor` hypervisor call, which enables the hypervisor to dispose of the hardware thread and assign the resources to another virtual machine or to employ power management techniques on the unused resources. In our environment we run only one virtual machine, thus the hypervisor performs one of the following operations when the cede_processor is invoked from a given hardware thread: (i) If the other hardware thread on the same core is under use, the hypervisor turns off the hardware thread and puts the core in single thread (ST) mode. This effectively assigns more hardware resources to the running process, thus improving single-threaded performance. Moreover, while this mode does not directly target at reducing power, as several functional units are not utilized during ST mode, overall power consumption also decreases. (ii) If hypervisor has already turned off the other hardware thread in the core (i.e., the core is already in ST mode), the hypervisor puts the core in *nap mode*. Next we describe this power-saving mode and another hardware knob that we use in this thesis for exploring hardware/software cooperation possibilities.

Nap Mode This per-core, low-power mode turns internal clocks off and restricts the operation of the functional units in the core. Reducing active power consumption by turning clocks off reduces temperature as well, which further reduces leakage power. In this thesis we show the effect of nap mode on both system power consumption and core temperature.

Thread Priorities The POWER6 processor employs a thread priority mechanism—through software/hardware co-design—that controls the instruction decode rate for each hardware thread with eight priority levels (POWER ISA[99], Boneti et al.[16]). The software-controlled priorities range from 0 to 7, where 0 means the thread is switched off and 7 means the thread is running in single thread mode (i.e., the other thread is off). A thread's software-controlled priority is enforced by the hardware at decode stage, which determines the actual number of decode cycles assigned to the hardware thread. In general, the higher the priority of a thread with respect to the other thread on the same core, the higher the number of decode cycles assigned to the thread. Consequently, the thread with a higher priority receives more resources and can obtain higher performance.

Table 3.1: METbench and SPEC CPU2006 temperature/power results when executing one thread. Power and temperature are relative to the measured values when the system is idle.

	cpu_int	ld_l1	ld_l2	ld_mem	st_mem	cpu_fp	h264ref	bzip2	gcc	dealII	lbm	cactusADM	mcf	milc	soplex
Types	INT	INT	INT	INT	INT	FP	INT	INT	INT	FP	FP	FP	INT	FP	FP
T_{avg} (%)	19.8	12.5	13.0	10.2	14.6	10.2	22.7	20.7	16.8	19.8	15.5	21.4	14.8	15.5	15.5
P_{avg} (%)	6.0	5.1	5.6	6.4	9.4	4.1	7.8	7.4	7.3	7.6	13.1	10.0	7.7	9.4	8.3
Aggregated Performance Counters															
IPC	1.32	0.26	0.034	0.0020	0.018	0.47	1.16	0.79	0.44	0.66	0.39	0.85	0.12	0.19	0.32
L1 MPKC	0.0	0.0	32.5	1.94	3.62	0.0	11.0	8.9	5.8	5.9	29.5	29.3	5.58	8.3	8.2
L2 LD MPKC	0.0	0.0	0.0	1.94	0.0	0.0	0.00	0.05	0.66	0.25	0.25	0.06	1.20	1.84	0.95
L2 ST MPKC	0.0	0.0	0.0	0.0	3.61	0.0	0.02	0.13	0.16	0.02	5.6	0.51	0.05	0.46	0.37

The main motivation of the software-controlled priority is to address instances where biasing thread performance is desirable because one thread is not really progressing or because it requires some level of quality of service. For example, the Linux kernel implementation for POWER6 reduces the priority of the idle process or of any process spinning on a lock. By doing so, more hardware resources are given to the other running thread. Moreover, depending on the application, software-controlled priorities can significantly improve both throughput and execution time (Boneti et al.[16]). In this thesis, we also show how software-controlled priorities can be used for improving system efficiency.

3.2 Effect of Workload Characteristics

Power and thermal behavior of computing systems strongly depend on the dynamic characteristics of workloads. To characterize the effect of workload characteristics on POWER6, we conduct several experiments with different applications from METbench (see Boneti et al.[16]) and SPEC CPU2006 (see Henning[48]) benchmark suites. While, in general, power and thermal behavior change with the amount of activity in the system, there is not a single factor that directly reflects the power consumption of the system. It is rather a combination of application characteristics as the usage level of the different parts in a CPU and the memory access rate. We present measured power and thermal characteristics for METbench and SPEC CPU2006 benchmarks in Table 3.1. The table shows measured average temperature, T_{avg} (percentage over the baseline), average system power, P_{avg} (percentage over

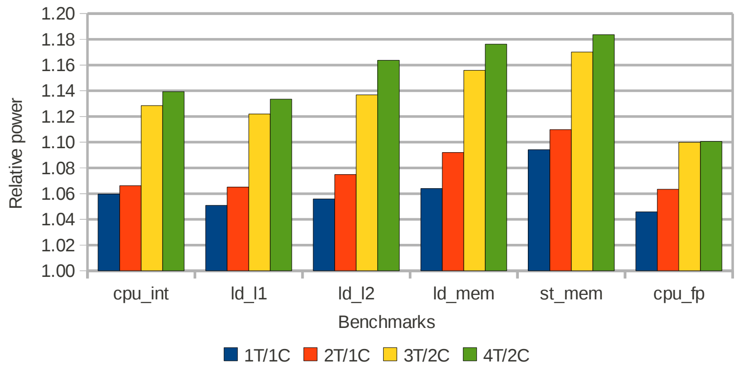


Figure 3.2: METbench power consumption for different number of threads (T) and cores (C).

the baseline), IPC, L1 misses per kilo-cycle (L1 MPKC), and L2 load and store misses per kilo-cycle (L2 LD MPKC and L2 ST MPKC) for each benchmark. We use temperature and power consumption when the system is idle as the baseline.

The results in Table 3.1 show the strong influence of different workload characteristics on power and thermal behavior. We observe strong deviations among benchmarks in terms of their power and thermal behavior and their associated performance metrics. Next we look at specific benchmark categories and derive the relations between major workload features and their impact on power and temperature.

CPU-BOUND BENCHMARKS We see that high-IPC and CPU-bound benchmarks generally lead to higher core temperatures. Among METbench, cpu_int has the highest IPC and a core temperature that is 7-9% higher than the other microbenchmarks. Within SPEC CPU2006, the benchmarks that cause higher core temperatures are h264ref, bzip2 and cactusADM. These three benchmarks also present the highest IPC among SPEC CPU2006. *

*METbench microbenchmarks are designed to exercise a single resource in the system at a time. In contrast, SPEC CPU2006 stress different parts of the system at once. Therefore, SPEC CPU2006 benchmarks typically consume more power and reach higher temperatures than METbench.

Memory-bound benchmarks While benchmarks that are CPU-bound achieve higher temperatures, they do not consume the most power. As Table 3.1 shows, memory intensive benchmarks generally consume more power. This is because of accesses to main memory, which incur a significant power cost. Among the microbenchmarks, ld_mem and especially st_mem are the workloads with the highest power consumption. st_mem consumes more power because, as opposed to the case of ld_mem, evicted L2 lines are dirty and a write-back operation must be performed. This additional access to main memory increases power consumption. Power consumption for ld_mem does not differ significantly from the other microbenchmarks when only one process is used. But as Figure 3.2 shows, the power gap gets larger with increasing number of threads. For the SPEC CPU2006 benchmarks we see a similar trend. Memory-intensive benchmarks like milc and lbm consume more power than the rest. For instance, relative to the baseline, lbm consumes 5.3% more than h264ref, with significantly lower temperature in comparison. Core temperature is generally low for memory-intensive benchmarks as they spend most of the time waiting for data from the main memory.

mcf is a low-IPC benchmark with a considerable amount of L2 cache misses per kilo-cycle and with similar characteristics to milc. However, the power consumption of mcf is considerably smaller (1.7% less). The most significant difference between them is the number of L2 store misses per kilo-cycle, which is 10 times higher for milc. As we have seen before in Figure 3.2, accessing main memory because of a store operation leads to higher power consumption. Accordingly, lbm, which has the highest number of L2 store misses, also shows the highest power consumption among the evaluated benchmarks.

FP benchmarks An interesting application in this category is cpu_fp. Despite having a medium IPC (0.47), it achieves the lowest core temperature. We have several hypothesis for this behavior. First, a POWER6 core contains one on-chip thermal sensor (OCTS) and multiple digital thermal sensors (DTS) distributed along the core, near expected hot spots. Our setup, however, only allows us to access the OCTS, as the DTS can only be accessed by firmware. The OCTS is not located very close to

Table 3.2: METbench results for 2 threads (mixed workloads). Power and temperature are relative to the idle system.

	cpu_int, cpu_fp	cpu_int,ld_l1	cpu_int,ld_mem
Cores	1	1	1
T_{avg} (%)	18.2	19.6	17.1
P_{avg} (%)	6.6	7.2	8.2
Aggregated Performance Counters			
IPC	1.77	1.56	1.31
L1 MPKC	0.00	0.00	1.97
L2 LD MPKC	0.00	0.00	1.95
L2 ST MPKC	0.00	0.00	0.00

the floating-point unit (FPU), and thus it might not be able to measure temperature in the FPU as accurately as the DTS can do. Second, cpu_fp strictly executes scalar floating-point operations. Therefore, it is neither using the vector multimedia extension (VMX) unit nor the decimal FP unit (DFU). This may lead to a relatively low power density—effectively avoiding the formation of hot spots. Having access to the DTS would allow us to verify these hypothesis. Unfortunately, only the firmware is allowed to access these sensors. This observation, however, only concerns to highly specialized microbenchmarks such as the ones we are using—where in general only a few units in the core are used. More generic benchmarks such as SPEC CPU2006 present a more uniform usage of the different units in a core. In this case, the OCTS would effectively return temperature measurements that are closely related to the largest hot spots on the core.

Table 3.2 shows the impact of heterogeneous workload mixes. We observe that co-scheduling a computation-intensive benchmark and a memory-intensive one leads to both high core temperature and high power consumption. For example, considering cpu_int and ld_mem, one thread continuously performs arithmetic operations while the other exercises the memory subsystem. In contrast, a more homogeneous mix, such as cpu_int and cpu_fp leads to lower power consumption. In this case the core temperature is higher as the core is more stressed.

Figure 3.2 also shows an important characteristic of multicore processors—and POWER6 in particular. As we increase the number of used cores from one to two, we observe a significant jump in power consumption. This is due to the fact that a second core has to exit the nap mode to serve the

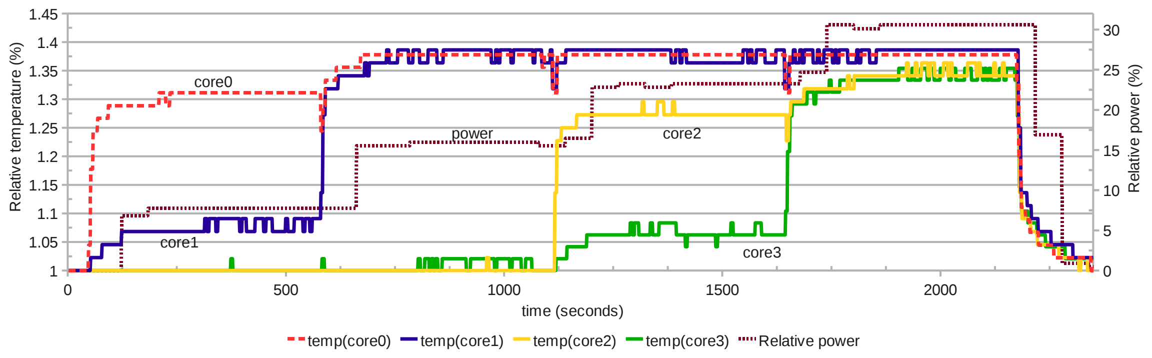


Figure 3.3: Several copies of cpu_int are used to create an incremental execution (2, 4, 6 and 8 hardware threads). The values are relative to the power and temperature measurements when the system is idle.

threads. We demonstrate in the following sections that every core that leaves the nap mode adds a constant power increment of approximately 5% to the system power consumption. We will refer to this increment as P_{AC} in the following sections.

3.3 Effect of Core Usage

In this section we execute several copies of a microbenchmark from the METbench suite in an incremental way. First we execute 2 copies on contexts 0 and 1 (using one core), then 4 copies on threads 0, 1, 2 and 3 (using two cores), and so on until 8 copies (using all four cores). We refer to each of these steps as an *execution step*. Each execution step is roughly 9 minutes long (360 iterations[†]).

CPU-bound benchmark Figure 3.3 shows the results of running an increasing number of cpu_int copies. In the figure we notice that power remains quite stable in the intervals between execution steps. Power noticeably increases when two more copies of cpu_int are started and a new core is used. We observe the first increment around one minute after the program is started. This is because of the 1-minute access granularity of the Thermal and Power Management Device (TPMD).

[†]METbench can iterate a benchmark a certain number of times in order to obtain better stability in the results.

Table 3.3: IPC, aggregated IPC and power consumption for the incremental execution of multiple processes. The power consumption values are normalized with respect to the ones obtained when the system is idle.

(a) cpu_int

#threads	IPC per thread								aggregated IPC	P_{avg} (%)
	chip0				chip1					
	core0		core1		core2		core3			
	0	1	2	3	4	5	6	7		
2	0.8482	0.8483							1.6965	8.3
4	0.8480	0.8480	0.8489	0.8489					3.3938	16.5
6	0.8478	0.8478	0.8489	0.8489	0.8485	0.8485			5.0904	23.9
8	0.8480	0.8480	0.8481	0.8481	0.8487	0.8487	0.8486	0.8486	6.7868	31.2

(b) ld_mem

#threads	IPC per thread								aggregated IPC	P_{avg} (%)
	chip0				chip1					
	core0		core1		core2		core3			
	0	1	2	3	4	5	6	7		
2	0.0017	0.0017							0.0034	10.6
4	0.0011	0.0011	0.0011	0.0011					0.0044	17.0
6	0.0010	0.0010	0.0010	0.0010	0.0016	0.0016			0.0072	26.2
8	0.0011	0.0011	0.0011	0.0011	0.0011	0.0011	0.0011	0.0011	0.0088	33.0

From this experiment we find out that for every two new copies of cpu_int, system power consumption increases approximately 7.6%. Another observation is the interaction between cores within the same chip. In Figure 3.3, around $t = 50$ seconds, we see that the temperature of core 1 increases approximately 8% when core 0 starts executing the benchmark. Later, around $t = 600$ seconds the temperature of core 0 further increases 7% when core 1 starts running. This is due to the lateral heat conduction between the cores within the chip. As the two chips are physically separated, we do not see any inter-chip effect in temperature.

As cpu_int is not using any shared resources between the cores, the aggregated throughput does not reduce as we increase the number of used cores. This can be seen in Table 3.3a. The IPC is stable around 0.85 per thread and the aggregated throughput increases linearly with the number of threads being executed.

Memory-bound benchmark ld_mem continuously executes load instructions that miss in all levels of the cache hierarchy. Therefore, it always needs to go to main memory to get the data. As shown in Table 3.3b, its IPC is much lower than the one measured for cpu_int. In terms of power consumption, as we have previously observed, memory-intensive workloads typically consume more power than computation-intensive loads. This behavior is seen again when comparing the incremental executions of cpu_int and ld_mem.

It is interesting to notice the reduction in IPC as more ld_mem threads are run. For instance, by comparing the cases where two threads (on the same core) and four threads (on the same chip) are executed—the first two rows of Table 3.3b—IPC for the first thread in the core 0 decreases approximately 36%. This suggests that there is contention in the shared hardware resources between cores. Looking at the results for six threads, we observe that the IPC for contexts four and five is approximately the same as it was in the case of two threads for contexts zero and one. The drop in IPC occurs within a chip when going from two to four contexts. Thus, the contention occurs within the chip—probably in the memory controller or the SMP interconnect fabric (as both L1 and L2 cache are private to each core).

3.4 PMC-Based Power Model

The possibility to obtain temperature and power measurements is a useful feature in POWER-based systems. But some configurations may not include the external microcontroller responsible to obtain these measurements (TPMD). Moreover, in some systems it may not be possible to access the console that provides the temperature and power measurements. Accessing the console is typically only available to administrators and regular users do not have access to it. Yet, it is beneficial for users to understand the power and thermal behavior of their applications. Another situation where a power model is useful is when implementing adaptive policies in the OS that manage power consumption—

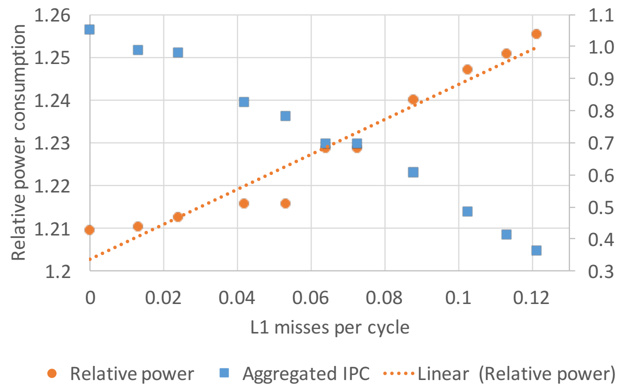
since direct access to TPMD from the OS is not possible, OS could rely on a power model to improve its decisions in terms of power consumption.

In this section, we present a model based on performance counter (PMC) data to estimate system power consumption. Since performance counter data is available and accessible by the OS, an analytical model in this form can alleviate all the shortcomings highlighted previously. This model follows a similar approach as the one presented in Bircher & John[15]. The model presents a good accuracy and it only relies on performance counters. No extra hardware support is required. Moreover, since the set of necessary events is minimal, it is simple to implement it in runtime systems that take decisions based on performance counters data.

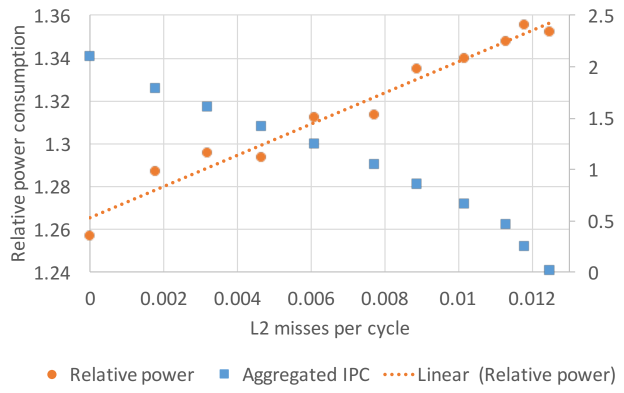
Similarly to the aforementioned work we select a group of PMC that captures the activity in different components of the system (CPU, memory, disk, etc.). In our case, we concentrate on the CPU and memory parts since, as Bircher & John[15] shows, there is not a significant variation in power consumption due to activity in other components (95% of the dynamic power consumption is due to CPU and memory activity). The selection of the right set of events for the model relies on a hybrid scheme, where expert knowledge and pruning techniques based on statistical analysis are utilized. The scheme leads to a set of events that obtain significant accuracy at predicting system power consumption.

The power consumption due to activity in the chip is modeled by using IPC and the number of L1 load misses per cycle (L1LDMPC). The memory system contribution to the power consumption is modeled by using the number of L2 (load and store) misses per cycle (L2LDMPC and L2STMPC). As our system does not have an L3 cache, every miss to the second level cache goes to main memory. Thus, L2 misses per cycle are good indicators of memory power consumption. Bircher & John[15] do not differentiate between load and store misses. But, as discussed in Table 3.1, benchmarks with a high count of L2 store misses consume more power than other type of workloads. Because of it, our analytical model includes L2 store misses to improve accuracy.

As the TPMD allows us to only measure total system power consumption, it is important to under-



(a) L2 accesses effect



(b) Memory accesses effect

Figure 3.4: Effect of accesses to L2 cache and main memory on system power consumption. Power values are relative to the idle system consumption. Several instances of the same benchmark are executed to create a higher power delta. The regression line is just an approximation to show the increasing trend.

stand the power behavior of different components and whether a linear model of these components is sufficient. For this purpose, we use two microbenchmarks that can vary the miss rate both for L1 and L2 from zero to the point where the access rate to L2 and memory saturates. Figure 3.4a displays the power consumption variation as the L1 miss ratio grows, which provides an insight on the L2 cache power contribution to the system power consumption. Figure 3.4b shows similar information for L2 misses reflecting the memory power contribution to the system power. In both figures, we observe that power consumption grows linearly as the number of misses increase. Thus, we define the model as a linear combination of the different factors that contribute to system power consumption (see Equation 3.1).

$$\begin{aligned} P = N_{AC} \times P_{AC} + \alpha \times IPC + \beta \times L1LDMPC \\ + \gamma \times L2LDMPC + \sigma \times L2STMPC \end{aligned} \tag{3.1}$$

Power is predicted as a percentage over the baseline when the system is idle (i.e., no user-process is being executed and the cores spend most of the time in the nap mode). In the characterization step in Section 3.2 we observe that for each core that exits the nap mode there is an increase in power consumption (P_{AC}). N_{AC} is the number of active cores, so multiplying it by P_{AC} gives the power consumption of all the active cores in the system.

We conduct several descriptive statistic tests for the parameters in the data set (e.g., normality test for residuals and non-presence of non-random patterns in the residuals). We also look at the significance of the parameters and their correlation to the response variable.

It is important to note that the coefficients obtained from the linear regression are subject to change if the size or type of the components in the system change (e.g., installed memory). Motivated by this fact, we follow two different approaches to train the model. (i) The first one (*METbench training*) relies on METbench data to train the model. Since running METbench is five times shorter than ex-

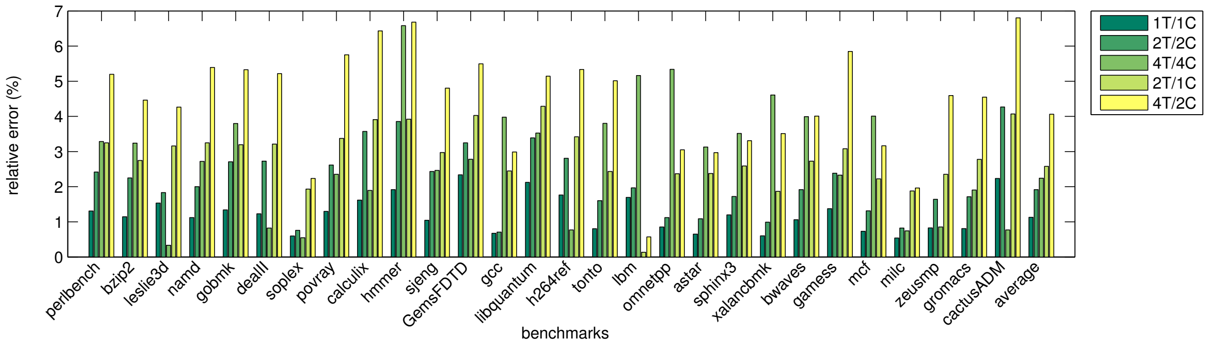


Figure 3.5: Estimation accuracy for the power model trained with METbench data only, for different number of threads (T) and cores (C). For instance, 4T/2C means 4 threads are run on 2 cores (using SMT capabilities). The error is computed as: $\frac{|measured-predicted|}{measured} \times 100$.

ecuting SPEC CPU2006, the amount of time to collect training data for a new model is considerably reduced. Thus, we only use METbench results to train the model, and we test the model with SPEC CPU2006. In case a new model is required for a different system configuration, we can quickly obtain new data by using METbench and later train a new model with the new data. (ii) The second approach (*shared training*) combines all the data (METbench and SPEC CPU2006) into a pool, and then relies on this dataset to train the model. A more general and accurate model is possible when a heterogeneous set of workloads are used. To cross-validate the model we use *leave-one-out cross-validation* (see Harrell[47]). Leave-one-out cross-validation is a standard statistic technique to estimate the accuracy of a regression model.

METbench training Figure 3.5 shows the relative error between our model estimation and the actual power measurement. Different bars for a benchmark correspond to different CMP (1T1C, 2T2C and 4T4C) and hybrid CMP+SMT configurations (2T1C and 4T2C). As shown in Figure 3.5, most of the benchmarks are predicted with an error equal or less than 5%. The relative error for 1T1C configuration is below 2% for almost all benchmarks. The maximum error occurs for cactusADM when it is run as four processes in SMT mode. We compute the average error using the geometrical mean,

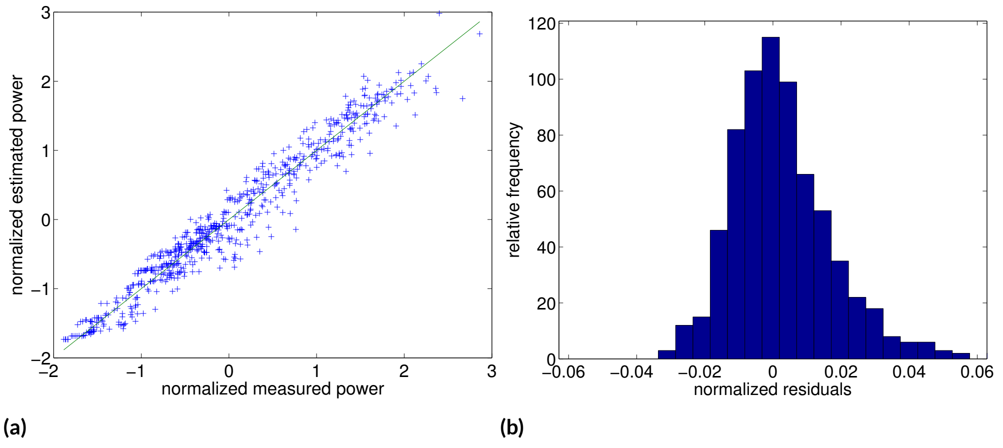


Figure 3.6: Model validation using all the available data (METbench and SPEC CPU2006). (a) shows the normalized measured vs. estimated power. The values are normalized by subtracting the mean of the error and dividing by the standard deviation of the error. (b) shows the residue error distribution with the cross-validation process. The residuals are normalized by dividing to the actual measure value.

being under 4% for all the configurations.

In general, the estimation error increases as more processes run on the system. We observe this effect for both CPU-bound and memory-bound workloads. We attribute it to the accumulation of the errors that are made to predict the power consumption for each core in the system. When both SMT and CMP capabilities are used, the estimation error grows with respect to the CMP case.Nonetheless, the average error for the CMP+SMT case is between 2.5% and 5% for two and four processes, respectively.

Shared training By combining data from METbench and SPEC CPU2006 to train the model, we capture wider resource usage patterns. Thus, we obtain a model that could potentially predict unobserved data points in a more accurate way. Figure 3.6a shows the normalized measured vs. estimated power consumption. Model predictions are considerably close to the real measurements for most of the data points. The residual distribution with the cross-validation process, shown in Figure 3.6b, resembles a normal distribution, with mean $\mu = -7.2 \cdot 10^{-5}$. Only 4.6% of the individuals are out of the confidence interval $[\mu - 2\sigma, \mu + 2\sigma]$. The error is under 6% for all the cross-validation steps.

Overall, both of the approaches that are used to construct the model obtain quite accurate results, with errors less than 6%. This level of accuracy is sufficient for users to study the power consumption behavior of their applications. In addition, this level of accuracy is also attractive for OS to implement such a model to deploy optimization policies.

3.5 Improving Efficiency Through Hardware Knobs

3.5.1 Low Power Mode

POWER6 employs several power reduction techniques for idle cores. Here we quantify the effects of these capabilities. Specifically we look at the effects of thread prioritization and enabling *nap* mode via the cede_processor function call in the kernel. We consider four power management policy combinations: *(i) No power saving* represents the baseline behavior without any power management. In this case all calls to cede_processor and HMT_xxx[‡] have been disabled. *(ii) HMT enabled* only enables hardware thread prioritization. Enabling the calls to HMT_xxx allows the snooze loop—a small active waiting loop that runs before deciding to put a core into nap mode—to be executed with low priority. *(iii) CEDE enabled* only enables the calls to cede_processor so that the cores can go into nap mode, disabling the clock for most of the circuits inside the core. This policy does not rely on hardware thread priorities. *(iv) Both enabled* enables calls to both cede_processor and HMT_xxx, and thus is the most aggressive power management policy.

Table 3.4 shows power and temperature characteristics observed for the idle system with these four policies. With *no power saving* policy, all the cores reach the highest temperature and the highest total system power consumption. We will consider these values as the baseline for this section, showing

[‡]Prioritization functions such as HMT_medium, HMT_low and HMT_very_low.

Table 3.4: Temperature and power savings when the system is idle using different low-power-saving mechanisms in the processor. Values are normalized to the first configuration.

		1 No power saving	2 HMT enabled	3 CEDE enabled	4 Both enabled
Temp savings (%)	core 0	0	8.2	24.6	26.2
	core 1	0	6.8	22.0	23.7
	core 2	0	8.6	22.4	24.1
	core 3	0	9.4	21.9	23.4
Power savings (%)		0	8.7	23.3	24.3

the reduction compared to this baseline for the rest of configurations. *HMT enabled* mode reduces the activity within the core and both power consumption and temperature are considerably reduced. Core temperatures and system power decrease 7-9% and 8.7% respectively using only hardware thread prioritization. We see much more dramatic improvements with the *CEDE enabled* policy. Although we prevent the processor from reducing thread priorities in the snooze loop, higher power savings are achieved by enabling POWER6 nap mode. Compared to the baseline configuration, core temperatures and system power consumption are reduced by 22-25% and 23.3%, respectively. Finally, applying both power management mechanisms (*both enabled* policy) further reduces system power consumption by 1%. This shows limited improvements for an idle system with hardware thread priorities when nap mode is enabled. However, nap mode can only be enabled when both threads in a core are idle, whereas thread prioritization does not have such restriction. When only one thread is idle, prioritization can give more resources to the other thread, increasing both performance and energy efficiency. Overall, combining nap mode and thread prioritization significantly reduces the energy consumption when the processor is in idle mode.

3.5.2 Linux Tickless Kernel

When a core is idle, the OS tries to put it in low power mode. One challenge in this power management scheme is the interrupt behavior of the system. Any interrupt received in low-power state forces the CPU to go back to active state to handle the interrupt. Timer interrupts are the most common

Table 3.5: Timer interrupts for tickless and non-tickless kernel configurations (HZ=100 and HZ=1000). Power and temperature are normalized to the first configuration.

		tickless 100	non-tickless 100	tickless 1000	non-tickless 1000
total ticks/s		30	399	39	3993
temp increase %	core 0	0	0	0	2.3
	core 1	0	0	0	0
	core 2	0	0	0	2.3
	core 3	0	0	0	2.1
power increase %		0	0.46	0.46	2.75

interrupts received by a CPU and local timers fire interrupts periodically. Moreover, if the CPU is idle, the timer interrupt handler does not perform any operation, while still forcing the CPU to wake up. Thus, reducing the amount of interrupts delivered to idle cores increases the time cores stay in low power mode and improves overall system power efficiency. The tickless kernel mechanism (present in version $\geq$ 2.6.21) reduces the effect of timer interrupts by disabling the periodic timer interrupts when a CPU is idle (see Siddha et al.[110]). In practice, instead of programming the local timer to expire every 100 ms (default value for the periodic timer), the kernel programs the timer to expire in the next, non-periodic, timer event (e.g., a software timer programmed by a task that has called the sleep() system call). Therefore, the tickless mechanism avoids disturbing an idle CPU unless there is real work to do.

We measure the effect of the tickless mechanism on temperature and power while the system is idle. As there is no activity in the system, the number of *external* interrupts is negligible. Thus, the system is mainly disrupted by *timer* interrupts (tick events).

We use four kernel versions to evaluate the impact of the tickless mechanism, as shown in columns of Table 3.5. We build tickless and standard tick-based kernels with different tick rates (timer events per second): 100Hz (default value for a server configuration) and 1000Hz. We measure idle core temperature and system power in all these configurations. For this section we choose configuration 4 in Table 3.4 as the baseline for all power and temperature results. For that configuration the system is idle and both low-power mechanisms analyzed in the previous section are active, leading to the minimum

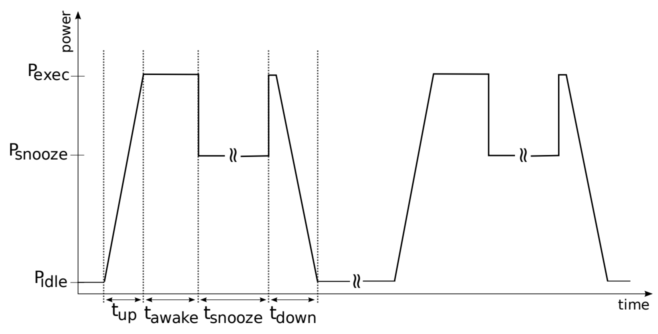


Figure 3.7: Power spikes due to tick time events

power consumption and core temperature.

There is a significant difference in the number of ticks among the different kernel configurations (row 1 of Table 3.5). The number of ticks per second in a non-tickless system is much higher than in a tickless system, increasing by 13X (from 30 to 399) for the 100Hz kernel and 102X (from 39 to 3993) for 1000Hz kernel. These results show how a tickless kernel effectively reduces the number of times that cores have to wake up from their idle state to handle each of these interrupt requests.

Temperature and power results in Table 3.5 show the power and thermal effects of the tickless mechanism. While the first three configurations (tickless-100, non-tickless-100 and tickless-1000) do not show any significant variation, the last one (non-tickless-1000) has a power consumption 2.75% higher than the rest, with a slight increase in temperature. The number of timer events per second is much higher in this configuration (10X compared to non-tickless-100 and 54X compared to tickless-1000). As the number of timer events per second grows, cores are more disrupted and cannot stay much in the nap mode. This is not as significant for the non-tickless-100 kernel due to the smaller number of ticks generated by the lower resolution timer.

Figure 3.7 depicts the interrupt timing behavior in more detail. Each of the spikes in the figure represents an expiration of the tick timer. When the system is idle it consumes P_{idle} (configuration 4 in Table 3.4) and on every tick timer expiration the following actions are carried out:

- The core wakes up from nap mode to active mode. This transition takes t_{up} µs. Behle et al.[11] show that t_{up} fits in the context switch delay, that is, in the order of few microseconds. Our measurements show that for the POWER6 processor, t_{up} equals 4 µs. As we have seen in Table 3.4 (configuration 1), during this period the system power consumption is the highest among the four configurations shown. P_{exec} represents the absolute power for that configuration.

- Once in active mode, we have to account for the time it takes the interrupt handler to run and to go from user to kernel mode and vice-versa, t_{awake}. In the interrupt handler, the OS checks whether there is any job to do. As Gioiosa et al.[42,43] show, both steps take in the order of few microseconds (1-3µs). We assume 3 µs. During this period the power consumption remains at P_{exec}.

- In an idle system most of the time the OS just continues in the idle loop and enters the snooze delay loop checking if a context switch is needed. As the hardware priority is reduced when entering the snooze delay loop, the system power consumption goes down to P_{snooze} (Table 3.4 shows 8.7% reduction over P_{exec}). This phase lasts for t_{snooze}, which by default is 100 µs. Changing the hardware priorities requires executing an *OR* operation, so we assume a delay of 0 µs.

- Finally, the system goes back to nap mode in a transition that takes t_{down} µs. Our results show that for the POWER6 processor, t_{down} equals 4 µs. During this period, the power consumption increases again up to P_{exec} and gradually decreases to P_{idle}.

The effect of ticks on power consumption is represented by Equation 3.2, where t_{total} is the observation period and $\#ticks$ is the number of ticks occurred during that period.

$$\begin{aligned} P = \Big(& [(t_{up} + t_{awake} + t_{down}) \times (P_{exec} - P_{idle}) \\ & + t_{snooze} \times (P_{snooze} - P_{idle})] \times \#ticks \\ & + t_{total} \times P_{idle} \Big) / t_{total} \end{aligned} \tag{3.2}$$

We now apply Equation 3.2 to understand the low impact of the tickless mechanism (especially for HZ=100). Table 3.4 displays the power consumption for the idle loop using different configurations. These measurements are conducted when all four cores are in the same state. Therefore, for the rest of this analysis we will assume that all cores process the tick-timer expiration at the same time. If we considered expirations independently, their number would be higher but system power consumption would be significantly lower as only one core would be active at a time. Thus, both analysis would lead to similar results.

For non-tickless-100 we have close to 100 tick-timer expirations per second in the whole system. Using Equation 3.2, the computed power consumption in this scenario is 0.24% over P_{idle}. For the case of non-tickless-1000, there are approximately 1000 wake-ups per second, leading to a power consumption of 2.3% over the baseline. Both results are close to the actual measurements in Table 3.5.

Overall, we conclude that the tickless mechanism does not significantly reduce the power consumption for a standard tick resolution (HZ=100) as the number of times the cores exit the nap mode is not enough to noticeably increase the power consumption during the period of one second. This observation may change for other systems with the following characteristics: 1) the time to go from/to low-power (t_{up} and t_{down}) mode is longer. This may happen in processors in which low-power modes introduce changes in the supply voltage, in which case t_{up} can be much longer; 2) the difference between P_{exec} and P_{idle} is large; or 3) t_{snooze} is relatively long. Such formulation of the interrupt behavior can help evaluate different kernel configurations for POWER6 systems without the need to deploy them in an actual system.

3.5.3 Effect of Hardware Thread Priorities

Boneti et al.[16] demonstrated that the hardware prioritization mechanism in POWER processors can improve system throughput. We look at hardware prioritization from a power and energy efficiency angle. We show the effect of applying this mechanism in a energy-aware manner and present use cases where thread prioritization can improve not only system throughput, but also system efficiency. We present only a subset of the multiple priority levels in POWER6 as we are more interested in showing their possible use to improve energy efficiency, rather than doing an extensive characterization.

In Section 3.5.1 we show that by using hardware thread prioritization, power consumption for an idle system can be reduced up to 9%. In that case performance is not a major concern since the system is solely running the idle loop. When a system is executing workloads, however, hardware thread prioritization cannot be blindly used to reduce power consumption in a performance-agnostic manner. Careful consideration of power-performance trade-offs is needed to choose the appropriate priority levels. We can use hardware thread prioritization in a workload-aware manner to reduce power consumption and increase system throughput, thus improving efficiency.

Table 3.6a shows the results of executing a high-IPC application (h264ref) together with a low-IPC, memory-intensive one (lbm). With the standard priority configuration (4,4) power consumption is 16% over the baseline. If the priority configuration is changed to (5,4)—so that the priority of the high-IPC workload is increased—system power consumption is slightly reduced as less memory requests are performed by lbm. Moreover, aggregated IPC increases as more computational resources are given to h264ref, thus obtaining a better relative energy-delay product (EDP) (see Brooks et al.[19]). It is important to notice that in this case the individual IPC for lbm is not drastically reduced (approximately only 11%). In the most extreme configuration (6,1) power is further reduced and performance

Table 3.6: Power results using prioritization for a single core. Power values are normalized to consumption when idle. EDP and ED^2P normalized to configuration (4,4).

(a) Mixed workload (h264ref and lbm)

Priorities	3,4	4,4	5,4	6,1
IPC				
h264ref	0.32	0.55	0.72	1.15
lbm	0.36	0.35	0.31	0.01
Aggregated	0.68	0.9	1.03	1.16
P_{avg} (%)	15.1	16.1	15.1	8.7
EDP (relative)	1.73	1	0.75	0.56
ED^2P (relative)	2.29	1	0.65	0.43

(b) Effect of priority (1,1)

Benchmarks	cpu_int		ld_mem	
Priorities	1,1	4,4	1,1	4,4
Aggr. IPC	0.07	1.80	0.0030	0.0034
P_{avg} (%)	3.9	6.9	8.7	9.6
EDP (relative)	642.9	1	1.2	1
ED^2P (relative)	16508.8	1	1.5	1

is increased again. But this comes at the expense of significantly reducing the performance of the memory-intensive workload (lbm).

We consider *priority one* as a special case for power management. Table 3.6b characterizes the effects of this priority mode. It shows the results of executing a CPU-bound (cpu_int) and a memory-bound (ld_mem) workload with priorities (4,4) and (1,1). We notice that the effect of hardware thread prioritization depends on the characteristics of the workload. For instance, running ld_mem with priority (1,1) does not significantly affect its IPC, as it is an extreme low-IPC memory-bound benchmark. Power consumption is also not significantly affected as this benchmark consumes most of the power in the memory subsystem. For a high-IPC workload such as cpu_int the behavior is completely different. Power consumption decreases 3%, at the expense of reducing the IPC from 1.8 to 0.07. In general, the higher the core activity, the higher the power reduction obtained with priority one and the higher the performance impact.

One major advantage presented by this priority-based power/performance management scheme is the ability to make "small" changes to the system behavior to achieve desired power-performance targets. Unlike most adaptation schemes that expose drastically different operating points, the prioritization-based approach can provide small shifts in power and performance with very small impact to runtime behavior. Another advantage of this mechanism is its very short latency until the applied power management actions take effect. The response time of this mechanism is dramatically faster compared to external mechanisms such as dynamic voltage and frequency scaling (DVFS) (see Floyd et al.[40]). Therefore, hardware thread prioritization can be used as a fast and flexible initial response in the case of a thermal/power emergency.

3.5.4 Thread Placement

With the arrival of SMT and CMP architectures, ensuring fairness between the different running processes has become an important issue. Several techniques such as scheduling domains, load balancing and cache affinity have been implemented in actual operating systems.

Job scheduling techniques have also been used in order to reduce power consumption. For instance, Linux provides a setting (sched_mc_power_savings) that attempts to save power consumption by grouping several processes onto a single chip, therefore leaving other chips idle. An analogous setting (sched_smt_power_savings) exists to consolidate several processes into a single core (see Srinivasan et al.[119]). But the kernel does not currently schedule threads based on their resource usage nature.

In this section we study the effect of thread placement on power consumption. Given a set of processes, there are different possible ways of assigning them to hardware threads, considerably varying the impact on power and performance. In order to analyze this impact we conduct several experiments where multiple processes are executed with different core usage patterns. The second row in Tables 3.7a and 3.7b shows the core usage pattern used. The usage pattern is encoded in a binary form where the first four digits correspond to the first chip and the last four ones correspond to the second chip. Within a chip, the first two digits refer to the thread contexts in the first core and the last two refer to the thread contexts in the second core. For instance, the binary pattern 1000 1000 means that the first

Table 3.7: Effect of core configurations on power and performance. Power is normalized to the idle power. EDP and ED^2P values are normalized to the best configuration within each group (2 or 4 threads).

(a) h264ref

	1	2	3	4	5	6
Pattern	1100 0000	1010 0000	1000 1000	1111 0000	1100 1100	1010 1010
HW threads	2	2	2	4	4	4
Cores	1	2	2	2	2	4
P_{avg} (%)	9.6	13.3	12.8	20.2	19.3	29.4
IPC	1.75	2.33	2.34	3.51	3.51	4.68
EDP	1.74	1.01	1	1.65	1.64	1
ED^2P	2.32	1.02	1	2.18	2.14	1

(b) lbm

	1	2	3	4	5	6
Pattern	1100 0000	1010 0000	1000 1000	1111 0000	1100 1100	1010 1010
HW threads	2	2	2	4	4	4
Cores	1	2	2	2	2	4
P_{avg} (%)	15.1	17.9	22.0	22.0	29.4	34.9
IPC	0.41	0.44	0.76	0.42	0.83	0.88
EDP	3.24	2.88	1	3.97	1.08	1
ED^2P	6.01	4.98	1	8.33	1.14	1

thread context in the first core in both chips is used to execute one process.

CPU-BOUND WORKLOAD Table 3.7a shows the effect of thread placement for 2 and 4 instances of the CPU-bound benchmark h264ref. The first thing we notice is that SMT configurations (columns 1, 4 and 5) present lower power consumption with respect to the other scheduling options using the same number of threads. For example, the configuration on column 1 reduces power consumption by 3.2% with respect to the configuration in column 2. Analogously configuration 5 is 7.8% better than configuration 6. As h264ref is CPU-bound, however, running both processes in SMT mode on the same core affects performance (up to 25% slowdown). The energy-delay product is worse for these configurations as the small power reduction does not make up for the loss in performance. Li et al. [71] obtained similar conclusions.

More interestingly, the power consumption remains the same between using 2 cores in a single chip (configuration 2) and using one core in each chip (configuration 3). We expected that in configuration 2, the second chip would be in low power mode most of the time, leading to a power consumption reduction. But the POWER6 saves power at the core level, without any extra reduction when a whole chip is idle. Therefore, what really matters is the number of idle cores and not whether they are in the same chip or not. The same behavior can be observed when using 4 threads in configurations 4 and 5. If the processor were able to reduce the power consumption when a whole chip is idle, it would certainly be possible to consolidate several processes onto one chip in order to reduce total energy consumption.

Memory-bound workload For memory-intensive workloads the situation clearly changes. As they are not bounded by pipeline resources, executing 2 threads on the same core in SMT mode does not significantly hurt the performance. Comparing the IPC for configurations 1 and 2 in Table 3.7b, we observe that IPC reduces only 6.8% (from 0.44 to 0.41). The same behavior is observed for configurations 5 and 6, where four threads are run and IPC decreases 5.7%.

lbm is a memory-intensive application and it saturates the memory bandwidth of the first chip—as it is shown in Section 3.3. As each chip has a dedicated memory controller, distributing the processes across both chips will better use the available bandwidth to memory, compared to consolidating them onto one chip. In Table 3.7b we can observe that the performance nearly doubles when we go from single chip configurations (1, 2 and 4) to double chip ones (3, 5 and 6).

Effects on scheduling Recent versions of Linux use scheduling domains for representing the CPUs hierarchy with a tree-based shape. In our system, at the first level there are the chips in the system. The second level has the cores belonging to the chips from the previous level. The third level contains the HW threads or contexts for every core.

When using the default behavior, the Linux scheduler tries to distribute the threads throughout all the cores in the system, avoiding to run two threads on the same core unless it is not possible (i.e., there are more running threads than cores in the system). As we have seen, running two threads in SMT mode is not very efficient especially when the threads are CPU-bound. Linux prevents putting threads into the same core as long as there are free ones available. If the sched_smt_power_savings flag is active, however, Linux will group processes without considering the nature of the workloads. This may degrade overall performance and energy efficiency. We have also seen that when using processors with power-saving techniques at the core level, grouping threads on the same chip—leaving the other idle—introduces no benefit. In this case, sched_mc_power_savings would not lead to a power reduction, but it might degrade performance if the workloads are memory-intensive.

We analyze the effect of grouping threads into a single core/chip in terms of performance and energy efficiency. In general, a major source of slowdown between threads is sharing the caches. In our setup, the L2 cache is private to each core so threads do not suffer any slowdown due to cache sharing whether they are placed on different chips or on the same chip (on different cores). But there are other resources shared at the chip level that have to be taken into account for memory-bound workloads. In this scenario, multi-chip configurations are much more efficient in terms of energy-delay product with reductions up to 2.9X (configuration 3 vs. 2) and 3.7X (configuration 4 vs. 5) as shown in Table 3.7b. Thus, the decision on whether to consolidate tasks onto the same core/chip cannot be static. It depends on the low-power capabilities of the underlying architecture and the characteristics of the workloads.

Mixed workload A scheduler that is aware of the workload characteristics can use this information to increase the system performance and/or reduce the power consumption. Table 3.8 shows the results of executing a mixed workload consisting of several h264ref and lbm processes.[§] Comparing configurations 5 and 6 we observe that the latter is a heterogeneous workload mix at the chip level (each

[§]In this case, the patterns are composed of Hs and Ls, standing for h264ref and lbm, respectively.

Table 3.8: Effect of core configurations for a mixed configuration (h264ref and lbm). Power is normalized to the idle power. EDP and ED^{2}P values are normalized to the best configuration within each group (2, 3 or 4 threads).

	1	2	3	4	5	6
Pattern	HoLo oooo	Hooo Looo	LoLo Hooo	LoHo Looo	HoHo LoLo	HoLo HoLo
HW threads	2	2	3	3	4	4
Cores	2	2	3	3	4	4
P_{avg} (%)	21.1	19.7	26.2	26.6	34.9	32.6
IPC	1.54	1.56	1.60	1.93	2.78	3.06
EDP	1.05	1	1.45	1	1.23	1
ED^{2}P	1.07	1	1.74	1	1.36	1

chip executes a CPU-bound and a memory-bound workload), whereas the former is a homogeneous mix at the chip level. This will affect both performance and power consumption. The performance of configuration 6 is 10% better and the power consumption is 2.3% less. This leads to a 18.7% improvement in EDP and 26.3% in ED^{2}P.

An even more noticeable situation is seen in configurations 3 and 4. As in the previous case, placing both memory-bound workloads on the same chip limits their performance, without decreasing the total system power consumption. Thus, by co-scheduling the high-IPC and the memory-intensive workloads on the same chip we can reduce the interference between them, boosting the performance and reducing the energy consumption (1.7X improvement in the ED^{2}P).

Effects on scheduling Current implementation of the Linux scheduler does not take into account workload characteristics. This means that the scheduler may fail to achieve the optimal performance and/or the minimum energy consumption. For instance in Table 3.7b the scheduler may choose either configuration 2 or 3, as none of them uses SMT. If the former configuration is chosen, a 5X ED^{2}P deterioration will be experienced. In Table 3.8 the scheduler may choose either configuration 3 or 4, leading to a 1.7X ED^{2}P worsening. These results show the importance of considering workload characteristics and interaction in order to make more efficient scheduling decisions.

3.6 Conclusions

The different characterizations in this section show the benefits of using hardware/software cooperation to improve system efficiency. In the following chapters of this thesis we will focus on a specific hardware knob—data prefetching—and we will develop and evaluate policies that improve system efficiency by adaptively controlling the prefetching engine.

While the scope of the following sections is more specific, this section shows how hardware and software cooperation can enhance resource sharing. By doing so, system efficiency—based on different metrics such as performance or power consumption—is improved. As processors include more shared resources—as POWER7 does—the impact of hardware and software cooperation will increase, reinforcing the findings in this section.

4

Platform Characterization and Methodology

The techniques and solutions presented in this thesis have been implemented and evaluated on real computing platforms. This section describes in detail the platforms used in the different parts of this thesis as well as the methodology used to evaluate the proposals.

4.1 The IBM POWER7 Processor

The IBM POWER7 (Sinharoy et al. [111]) processor is an out-of-order design manufactured using 45 nm Silicon-On-Insulator technology. Figure 4.1 shows the microarchitecture of a POWER7 processor.

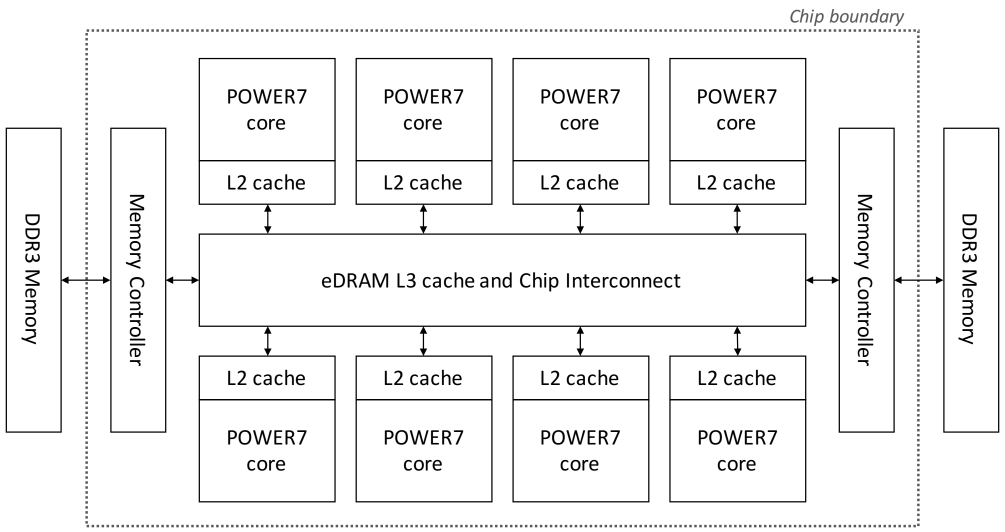


Figure 4.1: POWER7 microarchitecture overview.

A chip contains eight cores and each core can run up to four threads. A core can switch between single-thread (ST), two-way SMT (SMT2), and four-way SMT (SMT4) execution modes. Each core contains a 32 KB four-way set-associative L1 instruction cache and a 32 KB eight-way set-associative L1 data cache. Cores also contain a 256 KB L2 cache. The processor contains an on-chip 32 MB (embedded DRAM) L3 cache. Each core has a private 4 MB portion of the L3 cache, but a given core can also access the rest of portions from other cores—with higher latency.

POWER7 supports four virtual memory page sizes: 4 KB, 64 KB, 16 MB and 16 GB. Large pages provide multiple benefits such as a reduced number of page faults and TLB misses, and they allow prefetching to have a larger reach (data prefetching cannot cross page boundaries). Linux—the OS used in this thesis' evaluations—currently uses a 64 KB default page size for IBM POWER systems.

4.1.1 POWER7 Prefetcher

Implemented within the load-store unit (LSU), the data prefetching unit (DPU) contains twelve prefetch request queues (PRQs) plus associated logic that are capable of detecting and prefetching load, store, and load-to-store streams (see Cain & Nagpurkar[20], Hur & Lin[50], POWER ISA[99] for more details). As in previous POWER implementations, the DPU is able to detect sequential storage reference patterns, but is augmented with a "stride-N" logical subunit. The stride-N subunit detects streams which have regular access patterns, but do not fetch from consecutive cache lines in memory. The DPU can detect strides up to 8 KB in length, with a 32B granularity. The detection is handled in a four entry buffer which examines the stride between the address of the current cache line miss and those from the previous four cache line misses. When a pattern is detected, a data stream is created in a PRQ. From this point forward, the data stream is treated just like any other data stream, with the distinction that subsequent prefetch requests may fetch from non-consecutive cache lines. The detection hardware is unique compared to traditional stride-N approaches in that the pattern can be detected across multiple load instructions in the code sequence. This is required for proper detection of unrolled loops and complex conditional load structures.

The POWER7 DPU uses two types of prefetches to optimize the retrieval of data via prefetching. L3-prefetches prefetch data from memory (or other caches) into the L3 cache and L1-prefetches prefetch data into the L1 data cache. The core generates both types of prefetches to optimally cascade data from high latency DRAM into the L3 and from the L3 into the L1 data cache. POWER7 DPU is programmable and allows users to set different parameters (knobs) that control its behavior and determine the aggressiveness of the prefetch engine: i) *prefetch depth*, how many lines in advance to prefetch, ii) *prefetch on stores*, whether to prefetch store operations, and iii) *stride-N*, whether to prefetch streams with a constant stride larger than one cache block. The L1 and L3 prefetchers cannot be independently controlled. Prefetch settings are controlled via the data stream control register (DSCR). The Linux kernel exposes the register to users through the sys virtual filesystem (Mochel[85]), allowing them to set the prefetch setting on a per-thread basis.

Table 4.1a describes the possible prefetch configurations and introduces the notation that will be used throughout the chapter. Prefetch depth can take values from 2 (*shallowest*) to 7 (*deepest*). Ad-

Table 4.1: Notation used in this chapter for referring to prefetch configurations. We use tags (W/S) to indicate whether *prefetch on stores* (W) or *stride-N* (S) are enabled. Prefetch depth can be set to *default* (D) or to any value in the range 2-7 (*shallowest-deepest*). The special configuration where depth is 001 turns off the prefetcher (O). Table b shows some examples with this notation.

(a) Notation

Shortname	DSCR value	Description
O	xx001	Off (prefetch disabled)
D	xx000	Default depth
2	xx010	Shallowest
3	xx011	Shallow
4	xx100	Medium
5	xx101	Deep
6	xx110	Deeper
7	xx111	Deepest
W	x1xxx	Prefetch on stores
S	1xxxx	Stride-N

(b) Examples

Shortname	Depth	Prefetch on stores	Stride-N
D	Default	No	No
WD	Default	Yes	No
SD	Default	No	Yes
SWD	Default	Yes	Yes
S2	Shallowest	No	Yes
S3	Shallow	No	Yes
7	Deepest	No	No
S7	Deepest	No	Yes
SW7	Deepest	Yes	Yes

ditionally, there are two special values that can be used: 001b (O) and 000b (D). The former disables the prefetcher, while the latter is the system-predefined default depth. In POWER7 the default depth corresponds to depth 5 (*deep*). This value is automatically selected when the system boots (Abeles et al.[1]). *Prefetch on stores* (W) and *stride-N* (S) can only be enabled or disabled—they are disabled in the default configuration. Therefore, the default configuration corresponds to configuration 5 (using our notation). Every knob in the prefetcher can be independently configured. Table 4.1b shows some examples of the possible combinations that can be formed by setting values for each prefetch knob. Additionally, it shows how the shortnames that we use along the chapter are constructed.

4.1.2 Performance Monitoring

The POWER7 processor has a built-in performance monitoring unit (PMU) for each hardware thread that provides instrumentation to aid in performance monitoring, workload characterization, system characterization and code analysis. There are six thread-level Performance Monitor Counters (PMC)

Table 4.2: Description of performance events.

Name	Description
PM_CYC	Processor cycles
PM_INST_CMPL	Instructions that completed
PM_DATA_FROM_L3MISS	Demand load from L3 miss
PM_L3_MISS	L3 miss
PM_L3_LD_MISS	L3 demand load miss
PM_L3_CO_MEM	Total L3 castouts that went to memory
PM_L3_PREF_MISS	Total L3 prefetches sent from core that miss L3 prefetch directory
PM_MEM0_RQ_DISP	The memory controller has dispatched a read operation
PM_MEM0_WQ_DISP	The memory controller has dispatched a write operation
PM_MEM0_PREFETCH_DISP	The memory controller has dispatched a read for a prefetch operation

in a PMU. PMC1 - PMC4 are programmable, PMC5 counts non idle completed instructions and PMC6 counts non idle cycles. The thread level and core level instrumentation have access to a rich set of performance events (close to 550) that cover essential statistics such as miss rates, unit utilization, thread balance, hazard conditions, translation related misses, stall analysis, instruction mix, L1 I cache and D cache reload source, effective cache counts and memory latency counts. Table 4.2 shows the description of the events that we use in the different adaptive prefetching solutions.

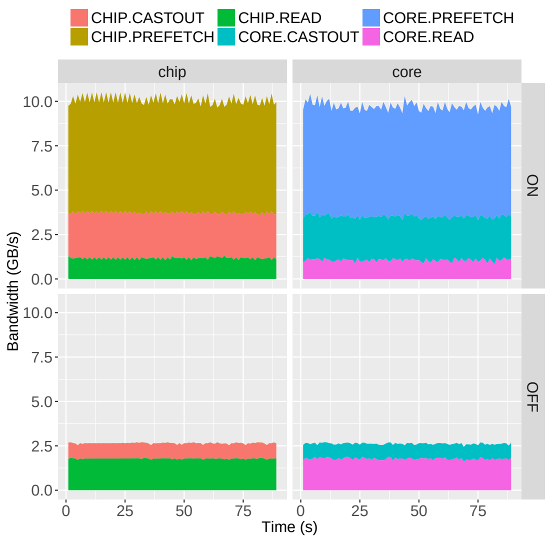
In this thesis we need to measure both thread performance and memory bandwidth consumption. In order to measure thread performance we use instructions per cycle (IPC). To collect bandwidth measurements, two different approaches exist. The first one (shown in Equation 4.1) computes memory bandwidth for the total chip.

$$
\begin{aligned}
ChipBW &= 128 * 8 * (ChipRead + ChipWrite + ChipPref) \qquad (4.1)\\
ChipRead &= PM\_MEM0\_RQ\_DISP - PM\_MEM0\_PREFETCH\_DISP\\
ChipWrite &= PM\_MEM0\_WQ\_DISP\\
ChipPref &= PM\_MEM0\_PREFETCH\_DISP
\end{aligned}
$$

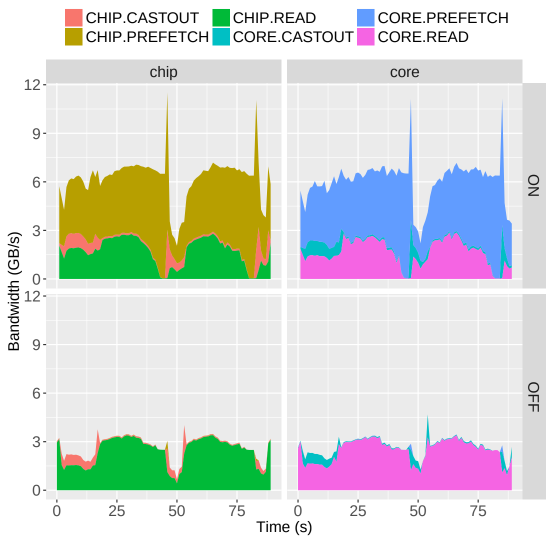
We use this metric, for instance, to determine whether the memory bandwidth is saturated. Sometimes we also require to measure memory bandwidth at a finer granularity. Equation 4.2 shows how to measure the memory bandwidth consumed by each core in the system.

$$
\begin{aligned}
CoreBW &= 128 * (CoreRead + CoreWrite + CorePref) \qquad (4.2)\\
CoreRead &= PM\_DATA\_FROM\_L3MISS + (PM\_L3\_MISS - PM\_L3\_LD\_MISS)/2\\
CoreWrite &= PM\_L3\_CO\_MEM/2\\
CorePref &= PM\_L3\_PREF\_MISS/2
\end{aligned}
$$

The fraction corresponding to the read traffic is composed of demand loads that miss in the L3 cache and reads with intention to modify (RWITM). Some of the events are scaled down by a factor of two because they originate in a different frequency domain within the processor. Ideally we would like to measure per-thread memory bandwidth consumption. Unfortunately some of the necessary events for measuring memory bandwidth are not collected in the core but in the nest (the L3 slice corresponding to each core). This is also the case for other processors (see Intel[52]). While there are ways to estimate the contribution of each thread within a core to the core's memory bandwidth consumption, enabling support to independently measure bandwidth-related events for each thread



(a) GemsFDTD

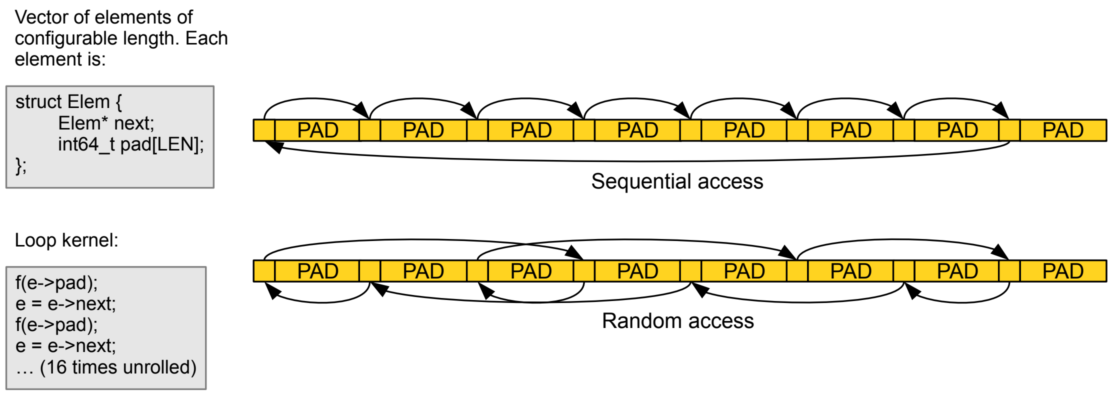


(b) mcf

Figure 4.2: Measured bandwidth for two SPEC CPU2006 benchmarks.

might significantly improve the accuracy of the measurement. We believe this thesis may encourage processor's designers to enhance the features of their PMU so that more efficient adaptive management techniques can be envisioned.

Figure 4.2 shows the result of measuring memory bandwidth for two benchmarks from the SPEC CPU2006 suite (GemsFDTD and mcf). The figure displays bandwidth measured at the chip and core level for two different prefetching settings: 1) enabled (shown in the top row), and 2) disabled (shown in the bottom row). Both benchmarks are memory intensive and their memory bandwidth consumption significantly increases when prefetching is enabled. As the figure shows, measuring memory bandwidth with either method (per-chip and per-core) results in very similar measurements.


Vector of elements of configurable length. Each element is:

```
struct Elem {
        Elem* next;
        int64_t pad[LEN];
};
```

Loop kernel:

```
f(e->pad);
e = e->next;
f(e->pad);
e = e->next;
… (16 times unrolled)
```

Figure 4.3: Microbenchmarks description. The microbenchmarks perform an array traversal either in sequential or random order. The distance between accesses is a configurable parameter. Depending on function f, the accesses to every array element can be loads, stores or both.

4.1.3 Impact of Prefetch Settings on Microbenchmarks

In order to understand the behavior of the multiple knobs available in POWER7's prefetcher, we use several microbenchmarks and characterize their effect on performance, memory bandwidth and power consumption.

Real applications present phases and significant dynamic variations during their execution, complicating the task of fine-grain architectural characterization. In addition to that, OS interferences and asynchronous I/O services further complicate the analysis. Microbenchmarks with well-defined characteristics simplify this problem by allowing us to understand the behavior of the different architectural components in isolation.

We developed a set of synthetic microbenchmarks that stress the prefetcher, caches and memory subsystem in different ways. By using them, we can understand the behavior of the prefetcher and its interaction with the rest of the memory hierarchy. The basic structure for all the microbenchmarks consists of an array traversal following a given order and bringing lines from a given point in the memory hierarchy to levels closer to the CPU. Figure 4.3 shows the implementation details of the microbenchmarks as well as two access patterns (sequential and random traversal). Each element of the array is composed of a pointer to the following element—the next element will depend on the type of traversal—and a padding area. The length of the padding area will determine how consecutive lines are accessed. For instance, if the size of the element structure equals the size of a cache line, every step in the traversal will touch a line. This design, when applied within a sequential traversal, will bring adjacent lines from the memory to the low-level caches. If the padding size is bigger, however, two consecutive accesses will not touch adjacent lines. Although sequential prefetching does not help with this access pattern, we will see that if stride-N is enabled, the prefetcher is able to improve performance.

In this characterization we use three microbenchmarks based on the scheme presented in Figure 4.3: seq-bench, seq-bench-stride and rnd-bench. The first one performs sequential accesses to consecutive cache lines. The second one is similar but the stride between two accesses is larger than a cache line. This creates an access pattern that a sequential prefetcher cannot identify. Finally, the last one performs random accesses, therefore it does not benefit from prefetching. All the microbenchmarks are mainly composed of memory load operations, and they are heavily memory-bound workloads.

Performance Results Figure 4.4 shows the results of running an increasing number of threads (from 1 to 32) under different prefetch configurations for different prefetch depth values. The left part of the figure shows per-thread IPC and memory bandwidth for seq-bench. This workload accesses consecutive memory blocks, and hence prefetching helps in this case. Prefetch depth significantly affects performance too, with the deepest configuration (7) achieving a 2.6X speedup over the shallowest one (2) for the single-thread case. As more threads run, memory bandwidth consumption significantly increases, and prefetch depth does not make a significant difference after eight threads are running in the system—with the exception of depth 2. After that point, the effect of prefetch depth is somehow limited, but there is still a large performance gap between enabling and disabling prefetching. If more

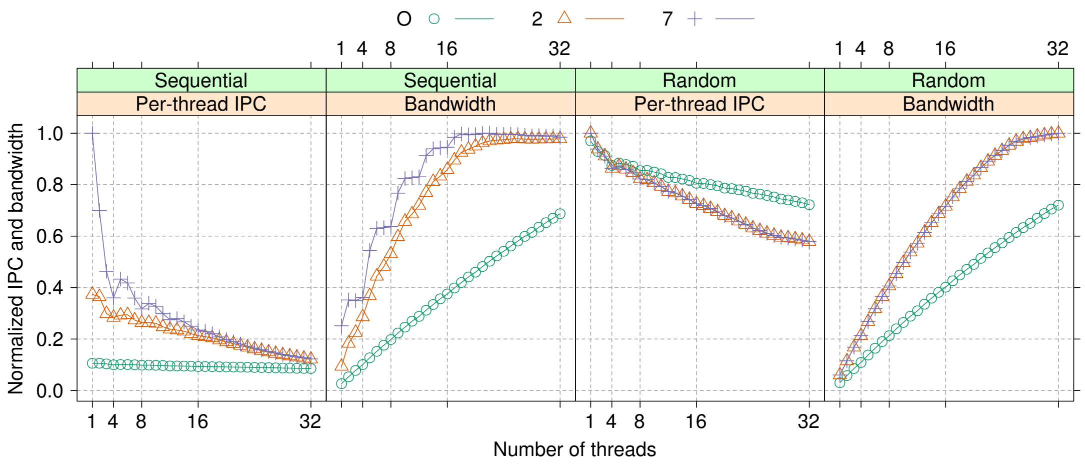


Figure 4.4: Prefetch depth effect characterization. Both sequential and random microbenchmarks are used to show the effect of prefetch depth on performance and memory bandwidth. Threads are bound to contexts in an increasing order (the first four threads go to the first core, the next four ones go to the second core, and so on). Values are normalized to the maximum value observed in each plot.

threads continue to be added, at some point memory bandwidth saturates and performance asymptotically converge to the same performance as when the system is not using prefetching. Although this example helps us to understand the effect of prefetch depth on both performance and memory bandwidth, we must bear in mind that it is an extreme case since the workload is mainly composed of operations that continuously access memory.[*] When more realistic workloads (e.g., SPEC CPU2006) are used, pressure on bandwidth is not so high, and prefetch depth keeps helping beyond the early saturation point seen in this example.

The right part of Figure 4.4 shows the same experiment with a benchmark that accesses memory positions in random order. In this case, prefetching cannot help since the workload's access pattern is not sequential. In fact, if prefetching is enabled, bandwidth consumption increases up to 1.5X compared to the case when prefetch is disabled. Upon encountering a cache miss, the prefetcher sends

[*]POWER7-based systems contain *two* memory controllers, but ours is a low-end system with only *one* controller. This could explain why bandwidth is saturated with fewer threads.

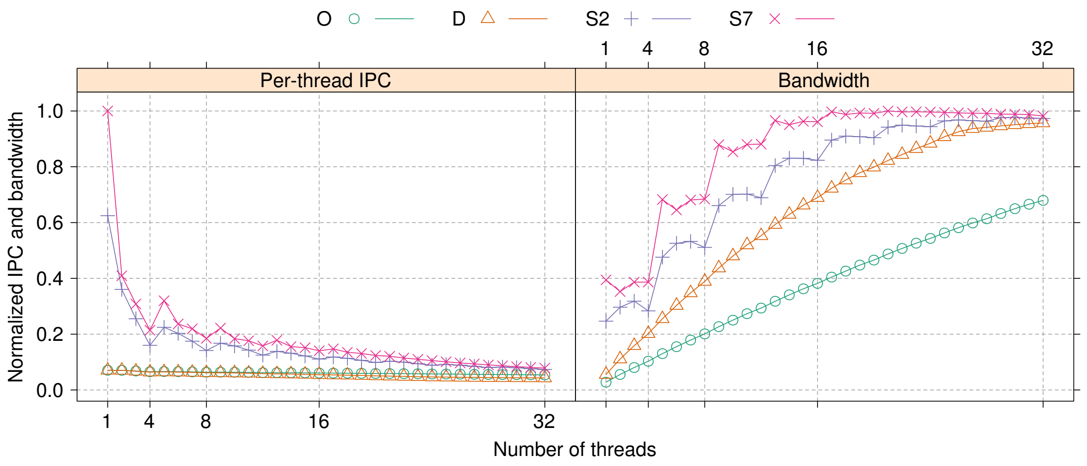


Figure 4.5: Stride-N and prefetch depth effect characterization. A sequential strided microbenchmark is used to show the effect of stride-N and prefetch depth on performance and memory bandwidth. Threads are bound to contexts in an increasing order (the first four threads go to the first core, the next four ones go to the second core, and so on). Values are normalized to the maximum value observed in each plot.

an L3 prefetch for the next cache line. Since those prefetches are useless, they do not contribute to increase performance, but they actually increase memory bandwidth and create more cache conflicts. Because of all these factors, disabling prefetch actually provides the best performance, especially when the number of threads increases.

Figure 4.5 shows the effect of stride-N for several choices of prefetch depth on the seq-bench-stride microbenchmark. We only display the results for depths 2 and 7 in order to ease the comprehension of the figure—the remaining depth values would lie in between 2 and 7. As it was expected, the default prefetch configuration (D) does not improve performance for this benchmark. Since accesses to memory are sequential, but they are not to adjacent cache lines, sequential prefetching does not help. The performance for the default configuration (D) is exactly the same as when prefetch is disabled. As we can see in Figure 4.5, however, the default configuration consumes significantly more bandwidth than turning prefetch off (O), without obtaining any performance benefit. Once stride-N is enabled (configurations S2 and S7), the prefetcher is able to identify the strided access pattern, and a significant

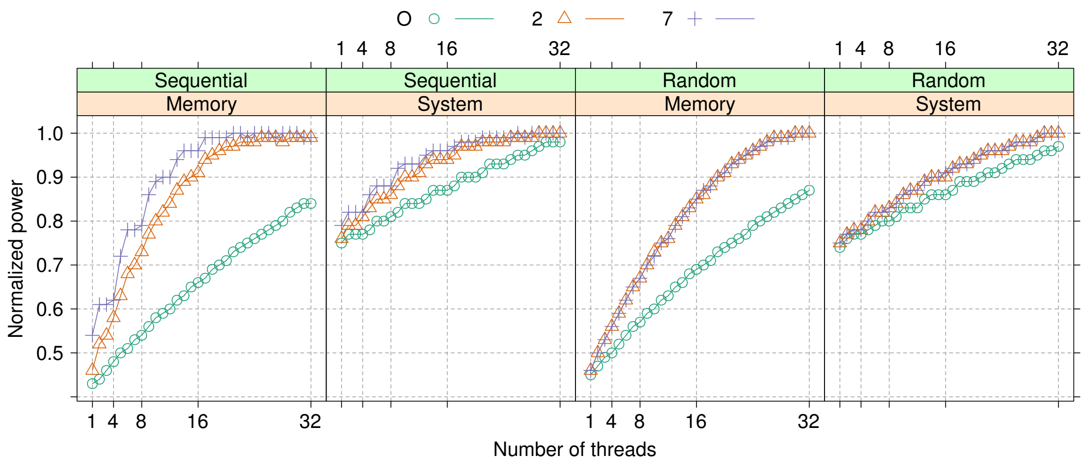


Figure 4.6: Memory and total system power consumption both for sequential and random microbenchmarks. Threads are bound to contexts in an increasing order (the first four threads go to the first core, the next four ones go to the second core, and so on). Values are normalized to the maximum value observed in each plot.

speedup is achieved. The effect of prefetch depth is similar to the one observed for seq-bench (Figure 4.4): when the number of threads is low, increasing prefetch depth achieves a significant speed up. But, as the thread count increases, the impact of prefetching considerably reduces.

Power Consumption Results Figure 4.6 shows both memory and total system power consumption for the same experiments shown in Figure 4.4. In all the cases, power consumption is significantly lower when prefetch is disabled. For seq-bench there is up to 30% memory power consumption difference between enabling and disabling prefetch. In terms of total system power consumption, the difference is still very significant (up to 10%). We must remember, however, that this power consumption reduction comes at the cost of a significant decrease in performance (see Figure 4.4). We computed energy efficiency using *energy-delay product*, and the results show that when both performance and power consumption are taken into account, disabling prefetch is not an efficient decision for seq-bench-like workloads.

Power consumption results for rnd-bench are similar to the ones observed for seq-bench. In this case the maximum observed difference between enabling and disabling prefetch is 18% for memory power consumption (5% for total system power). But as Figure 4.4 shows, a benchmark with a random access pattern does not benefit from prefetch, and performance is typically better when prefetch is disabled. Therefore, this case is a win-win situation. Disabling prefetch both improves performance and reduces power consumption, boosting system efficiency.

Power consumption results for the case when stride-N is enabled are similar to the ones presented in Figure 4.6. Because of that, they are not presented in this thesis.

Overall, we have seen that prefetch depth can significantly influence performance as long as memory bandwidth is not under a lot of pressure. When bandwidth gets saturated due to a large amount of demand loads, prefetching does not help as much anymore. Additionally, if a workload generates many useless prefetches, bandwidth consumption will increase, which may hurt system performance. We have also seen that prefetching typically increases power consumption. When prefetch is useful, power consumption increases along with prefetch aggressiveness. When prefetches are not useful, they may decrease system performance and waste power at the same time. All these observations are useful to understand the results with real benchmarks in the following sections.

4.1.4 Impact of Prefetch Settings on SPEC CPU2006

In the previous section we have studied the effect of prefetch settings on performance and power consumption for a set of microbenchmarks. In this section we conduct a similar study with more realistic workloads, using the SPEC CPU2006 benchmark suite.

Prefetching affects workloads in different ways, depending on their nature. Some experience a significant speedup when prefetch is used, while others are totally insensitive. We classify benchmarks in four different groups (see Table 4.3), according to the way prefetching affects their performance when running in single-thread mode on our POWER7 system: i) *prefetch-insensitive (PI)*; this type of benchmark is insensitive to prefetching. It does not suffer any significant performance variation no

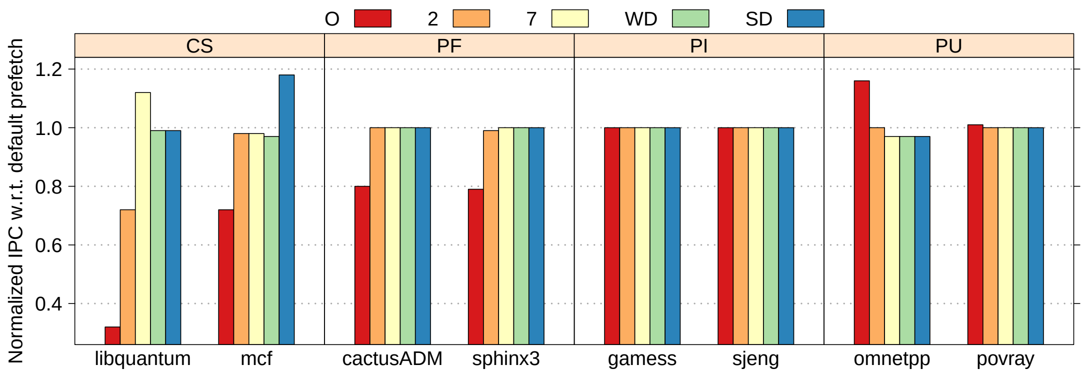


Figure 4.7: Effect of prefetching on performance for single-threaded runs. Multiple prefetch configurations are used in order to show the effect of each prefetch knob: depth (2-7), prefetch on stores (WD), and stride-N (SD)—refer to Table 4.1 for notation on prefetch configurations.

matter whether prefetching is enabled or not. Additionally, the various configurations (e.g., depth, stride-N and prefetch-on-stores) do not affect its performance (e.g., sjeng and gamess), ii) *prefetch-friendly (PF)*; enabling prefetching positively affects the performance of the benchmarks in this group. But, they are not affected when the prefetch setting is varied (e.g., zeusmp and cactusADM), iii) *config-sensitive (CS)*; for benchmarks in this group performance also increases when prefetching is enabled. Moreover, changing the prefetch configuration affects their performance too (e.g., enabling stride-N improves performance with respect to the default configuration; this is the case for mcf and milc), and iv) *prefetch-unfriendly (PU)*; for this type of benchmark, enabling prefetching negatively affects its performance (e.g., omnetpp and povray).

Figure 4.7 shows the performance for SPEC CPU2006 benchmarks—representatives of each different class—running in single thread mode under several prefetch settings. We use the default prefetch configuration (D) as the baseline to normalize IPC. The figure visualizes the impact of prefetching on the different classes that we use to classify the benchmarks (Table 4.3 contains the exact classification for all the SPEC CPU2006 benchmarks).

In terms of power consumption, Figure 4.8 shows CPU and memory power consumption for the

Table 4.3: Benchmark classification based on how their performance is affected by the prefetch settings when running in single-thread mode.

Classes	Benchmarks		
Prefetch-insensitive	perlbench gromacs sjeng astar	bzip2 namd h264ref xalancbmk	gamess gobmk tonto
Prefetch-friendly	gcc dealII GemsFDTD sphinx3	zeusmp calculix lbm	cactusADM hmmer wrf
Config-sensitive	bwaves leslie3d	mcf soplex	milc libquantum
Prefetch-unfriendly	omnetpp	povray	

same benchmarks appearing in Figure 4.7. The values are normalized to the ones obtained with the default prefetch configuration. In terms of CPU power, there is not too much variation when using the most aggressive prefetch setting (SW7). When disabling prefetch (O), CPU power consumption decreases up to 3% (for libquantum). The reduction of memory power consumption is much more significant—especially for config-sensitive, prefetch-friendly and prefetch-unfriendly benchmarks. For instance, memory power consumption for libquantum decreases 15%—at the expense of reducing its IPC more than 60%, though (see Figure 4.7). Perhaps the most interesting cases are povray and especially omnetpp. For the latter, disabling prefetching reduces memory power consumption 10% while, at the same time, performance increases close to 20%. That is a win-win situation, caused by avoiding useless bandwidth consumption due to inefficient prefetches. Power consumption is decreased for povray too, although in this case the reduction is more modest (5%).

Overall, as results in Figures 4.7 and 4.8 show, an adaptive prefetching mechanism could tune the prefetcher for every particular benchmark in order to find the prefetch configuration that leads to its optimal performance, potentially saving power consumption at the same time.

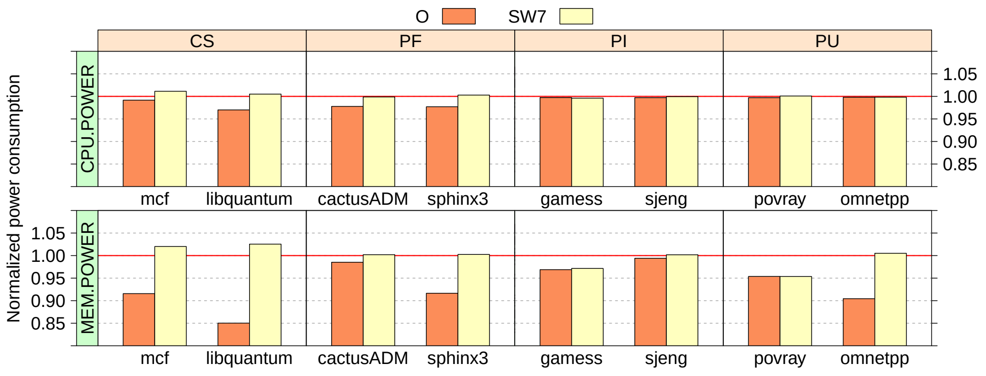


Figure 4.8: Effect of prefetching on CPU and memory power consumption for single-threaded runs. The values are normalized to the ones obtained with the default prefetch configuration.

4.2 Methodology

We use an IBM BladeCenter PS701 to conduct most of the experiments in this thesis—this is the case for all the experiments involved in the evaluation of the adaptive prefetching solutions. The system contains one POWER7 processor running at 3.0 GHz and 64 GB of DDR3 SDRAM running at 800 MHz. The maximum bandwidth achievable by this system is approximately 40 GB/s. The operating system is SUSE Linux Enterprise Server 11 SP1. We use IBM XL C/C++ 11.1 and IBM XL Fortran 13.1 compilers to compile all the SPEC CPU2006 benchmarks. We disable compiler-generated prefetch instructions in order to avoid interactions between these instructions and the hardware prefetcher. Although compiler-generated prefetches (Callahan et al. [21], Mowry et al. [88]) may improve the performance of some applications, because of their static nature, they are not a suitable instrument for dynamically adapting to the mix of applications running on a system. Our solution is actually orthogonal to software prefetching. The interaction between hardware and software prefetching has already been studied in the past (Cain & Nagpurkar [20]). We also use Graph500 (Murphy et al. [89]) and SPECjbb2005 [115] for evaluating the presented solutions. Graph500 is a representative example of a new class of server applications: analytics. The Java VM is IBM J9 VM build 2.4. We run all benchmarks until completion. Each benchmark may run for a different amount of time. Because of this, we restart benchmarks that finish early until all benchmarks have fully completed their execution at least once (see Vera et al.[125]). For collecting information from the performance counters we use *perf*, the official implementation in the mainstream Linux kernel (see Carvalho de Melo[22]). The default page size in Linux for POWER is 64 kilobytes. This helps prefetching since it is not necessary to restart the streams after crossing the boundary of relatively small 4 kilobytes pages.

Power measurements are obtained using the IBM Automated Measurement of Systems for Temperature and Energy Reporting (AMESTER) software (Floyd et al.[38], Lefurgy et al.[70]). The software connects to the EnergyScale microcontroller to download real-time power, temperature, and performance measurements of the POWER7 microprocessor and server. The software samples sensors at 1-ms granularity. By using this software we can access multiple sensors in the system, making it possible to sample total system power, chip power and memory power.

An IBM BladeCenter JS22 is used to conduct the experiments in chapter 3. This system contains two POWER6 processors running at 4.0 GHz. Each processor is a dual-core, two-way SMT chip. Thus, the system presents eight logical CPUs to the hypervisor and the OS layer. Our system does not include an off-chip L3 cache. Therefore, the last level of cache in our system is the 4MB L2 cache private to each core. Only one memory controller per processor is available in our configuration. The amount of DRAM memory is 15GB. The system runs SUSE Linux Enterprise 10 SP2 with a 2.6.28 kernel patched with perfmon2 3.8 in order to access the performance counters. The rest of the environmental setup is similar to the one described for the POWER7 system.

5 Adaptive Prefetching: Improving Per-Application Performance

5.1 Introduction

Different workloads typically benefit from different prefetching settings. Execution phases within a workload might as well be sensitive to the specific prefetching configuration being used. In this chapter we present a runtime-based adaptive prefetch mechanism capable of improving performance via dynamically setting the optimal prefetch configuration, without the need for a priori profile information. We evaluate the performance benefits of adaptive prefetching. Our adaptive scheme increases performance up to 2.7X and 1.3X compared to the default prefetch configuration for single-threaded and multiprogrammed workloads, respectively. We also show that our mechanism is able to reduce memory power consumption in some cases. In addition to using the SPEC CPU2006benchmark suite we also evaluate the impact of our adaptive prefetch mechanism on a Java server-side workload (SPECjbb2005). For that benchmark, adaptive prefetching is able to both improve performance by 21% and reduce memory power consumption by 22%. We also study the implementation of such an adaptive prefetch scheme within the OS kernel. After implementing our adaptive mechanism into the Linux kernel, we have observed similar performance improvements to those obtained by the userspace implementation.

5.2 Adaptive Prefetching

In Section 4.1.4 we have seen that different applications derive maximum performance benefit from different prefetch settings. In that approach, users need to profile applications prior to running them in order to determine the best prefetch setting for each application. We refer to this method as the *best static configuration* approach or, simply, the static approach. In that approach, a priori profiling yields the optimal prefetch configuration for a given application, and all future runs of this application would use this optimal configuration to achieve its efficiency target. Note that in that approach, the prefetch configuration is statically fixed for the duration of the application run. A truly dynamic adaptation of the data prefetch algorithm presents the promise of two potential benefits: (i) users would be able to avoid the per-application profiling step; and, (ii) dynamic phase changes within the same application would trigger adaptation of the prefetch parameters in order to further maximize the targeted efficiency metric.

5.2.1 Basic Adaptive Algorithm

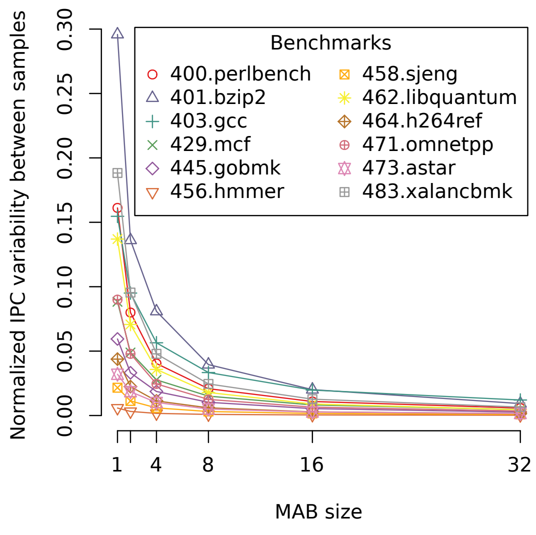
In its simplest form, our adaptive solution is composed of two different phases: 1) an *exploration phase* where the solution evaluates the performance of the different prefetch settings for each application running on the system, and 2) a *running phase* where the best performing setting found for each application is used. This process occurs periodically so that our solution can adapt to application phase changes. Algorithm 1 describes in detail the behavior of our solution.

Algorithm 1 Base adaptive prefetch algorithm.

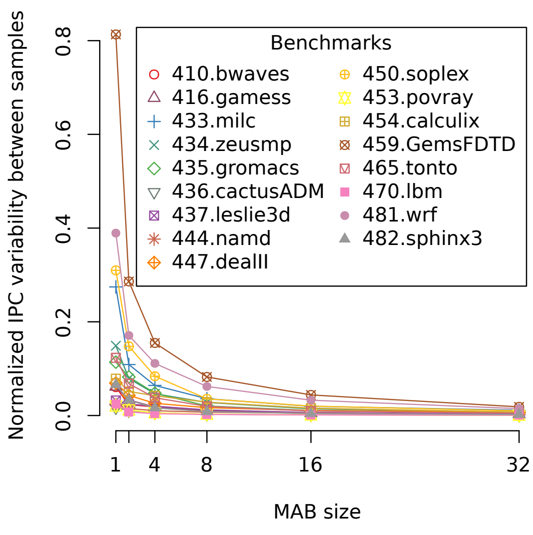
```
 1: for all $t$ in $threads$ do
 2:     for all $ps$ in $pref\_settings$ do
 3:         set_prefetch(cpu($t$), $ps$)
 4:         wait $T_e$ ms
 5:         $ipc[ps]$ = read_pmcs()
 6:     end for
 7:     $best\_ps = arg\,max_{ps}(ipc)$
 8:     set_prefetch(cpu($t$), $best\_ps$)
 9: end for
10: wait $T_r$
```

Algorithm 1 contains two configurable parameters, T_s and T_r. The former specifies the interval length to be used during the exploration phase (line 4). The latter is the amount of time that the best settings found during the exploration phase will be used before a new exploration phase starts (line 10). In our implementation we use the interval lengths $T_e = 10$ ms and $T_r = 100$ ms. This granularity is a good compromise between adaptability and overhead. It is actually a typical value for sampling-based approaches in the OS and runtime realms (Isci et al.[54]). For instance, the Linux kernel allows the user to choose the granularity of the timer tick from 1 ms up to 10 ms. A finer granularity would introduce a significant overhead in the system.

This first algorithm is the base for the other two presented in this chapter. But it suffers from two



(a) SPEC INT



(b) SPEC FP

Figure 5.1: Effect of changing the buffer size on inter-sample IPC variability. IPC variability (see Equation 5.1) is normalized to the average IPC for each benchmark.

potential problems: the effect of phase changes and the impact of "inefficient" prefetch settings (for a particular workload). Next, we examine and present solutions for these two problems.

5.2.2 Impact of Phase Changes

It is well-known that applications present phases during their execution (Denning[30]). They actually present phases at different levels, ranging from the microsecond to the millisecond level (some phases may even last for some seconds). Our adaptive mechanism periodically samples performance for different settings, attempting to find the best setting for that particular interval. We must, however, take care of possible phase changes that may occur between different samples in the exploration phase. Otherwise, we could attribute a performance change to the effect of a given prefetch setting when the real reason is an underlying phase change between measurements.

In order to alleviate this problem we use a moving average buffer (MAB)[2] that keeps the last m IPC samples for every prefetch setting and thread under control of the adaptive prefetch runtime. We then compare the performance of prefetch settings by using the mean of the values in the buffer, instead of using individual measurements. We evaluate the effect of using buffers of different sizes on the IPC variability between consecutive samples. Figure 5.1 shows the normalized IPC variability as we increase the buffer size from 1 (i.e., no buffer is used) up to 32. IPC variability is computed with the following equation:

$$variability = \frac{1}{n-1} \sum_{i=1}^{n-1} |IPC_{i+1} - IPC_i| \tag{5.1}$$

where IPC is an array with all the n IPC samples for a given workload execution. Variability is then normalized to the average IPC for every workload. For clarity reasons the figure is split into two. Figure a contains the results for SPEC INT benchmarks and Figure b does so for SPEC FP benchmarks. As it can be seen in the figure, most of the benchmarks present a small to moderate variation when MAB is not used. A few of them (bzip2, perlbench, wrf and GemsFDTD), however, have quite a high variation. As an example, let us examine bzip2. The average IPC variability between consecutive samples is 30% when MAB is not used. As the buffer size increases the variability is reduced, reaching 2% for a buffer containing the last 32 samples. By using a moving average buffer, we are able to significantly reduce the impact that phase changes may have on the exploration step of the adaptive prefetch mechanism.

Algorithm 2 presents the new version of the algorithm, using the moving average buffer. The algorithm is very similar to the one presented in the previous section. The only differences are on lines 5, 6 and 8, where the buffer is actually used. The operation of pushing a new sample into the buffer (line 5) is implemented using a circular buffer. Thus, when the buffer is full and a new sample is added, the oldest one is removed from the buffer.

Algorithm 2 Adaptive prefetch with MAB

```
1: for all $t$ in $threads$ do
2:     for all $ps$ in $pref\_settings$ do
3:         set_prefetch(cpu($t$), $ps$)
4:         wait $T_e$ ms
5:         push($ipc\_mab[t, ps]$, read_pmcs())
6:         $ipc\_mean[ps]$ = mean($ipc\_mab[t, ps]$)
7:     end for
8:     $best\_ps = arg\,max_{ps}(ipc\_mean)$
9:     set_prefetch(cpu($t$), $best\_ps$)
10: end for
11: wait $T_r$
```

5.2.3 Impact of “Inefficient” Prefetch Settings

The base adaptive prefetch algorithm iterates along a set of prefetch settings during the exploration phase. After the exploration phase is over, the runtime lets the threads run for a certain amount of time with the best setting found. Depending on the workload, there could be a significant performance variation between different settings used in the exploration phase. For instance, for bwaves, disabling the prefetch reduces its performance 78% with respect to the best setting. Such a significant slowdown may actually impact overall performance if the exploration phase is executed too often. Therefore, for this particular workload disabling prefetch would be an inefficient prefetch setting (it is important to mention that an inefficient setting for one workload may be the best one for another workload; thus settings’ efficiency is workload-dependent).

In order to quantify the effect of inefficient settings, we model the expected performance, $\widehat{IPC}$, based on the ratio of exploration and running phases’ length. We use the following equation for the

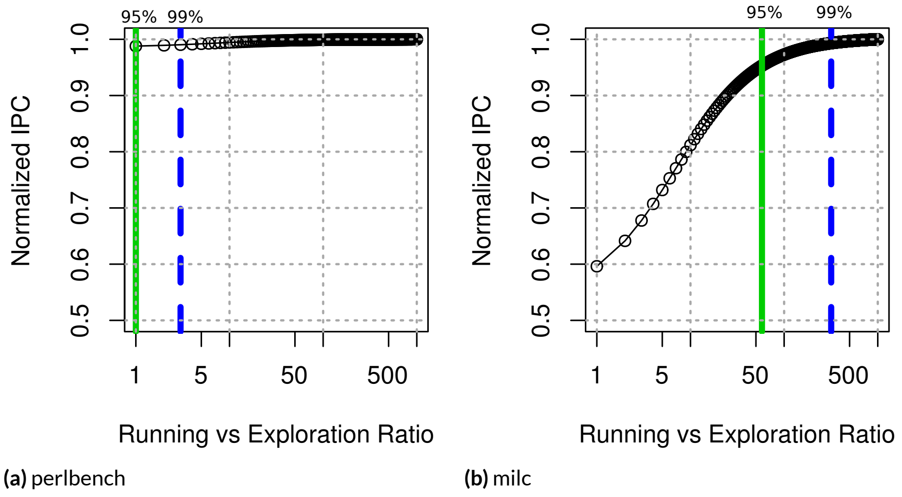


Figure 5.2: Effect of exploration/running ratio on expected performance. Values are normalized to the maximum values observed for each workload.

model:

$$\widehat{IPC} = \sum_{i=1}^{n} \frac{L_e}{L_t} \times IPC_i + \frac{L_r}{L_t} \times \max_i(IPC_i) \tag{5.2}$$

where L_e and L_r are the lengths of the exploration and running phases, respectively, and $L_t = L_e + L_r$. IPC is a set containing the average IPC values for each prefetch setting for a given workload.

Figure 5.2 shows the expected performance for two different types of workloads as the ratio L_r/L_e increases. The solid, green vertical line determines the running-exploration ratio such that the expected performance is within 5% of the best achievable performance (i.e., if there is no exploration phase and the best prefetch setting is used during all the interval). The dashed, blue vertical line is equivalent to the previous one, but it marks the point where the expected performance is within 1% of the best achievable performance. We use two benchmarks, perlbench and milc, to construct an illustrative example. The results for all the other SPEC CPU2006 benchmarks are similar to either one of these two.

Figure 5.2a shows the results for perlbench. This workload is mostly insensitive to prefetching and, thus, the expected performance follows a very flat curve. In order not to lose more than 1% of performance, it suffices to set a running phase four times longer than a single exploration interval. For these types of workloads, since they do not really suffer from inefficient prefetch settings, the running-exploration ratio is not so important. This totally changes for a different type of workload such as milc. Figure 5.2b shows the results for this workload. In this case the curve is not flat anymore. Indeed if we are not willing to pay a performance drop bigger than 5% we must use a running phase at least 50 times longer than a single exploration interval. For a tighter 1% bound, the ratio would increase up to approximately 400. Using such a large value for all the possible workloads would imply a drastic reduction in the number of times that an exploration phase is triggered. Thus, the adaptability of our mechanism would be significantly reduced.

In order to avoid this issue we decided to introduce a new feature in our adaptive prefetch scheme. This feature removes inefficient prefetch settings from the set containing all the settings to be tried during the exploration phase. We call this feature *prefetch setting drop*. Settings are "dropped" for a certain amount of time based on their inefficiency and then, they are considered again to be selected in a future exploration phase. The exact number of iterations, IT_i, that a given setting, i, will be dropped is given by the following equation:

$$IT_i = DF \times |MAB| \times \left(\frac{max_i(IPCi)}{IPC_i} - 1 \right) \tag{5.3}$$

where DF is the *drop factor*, $|MAB|$ is the size of the moving average buffer and the last term is a measure of the slowdown experienced when using setting i. If the performance for setting i is equal to the best performance observed, the last term becomes zero and the setting is actually not dropped at all, so it will be used in the next exploration phase. The slowdown term in the last equation penalizes inefficient settings proportionally to the measured slowdown. Thus, settings that significantly devi-

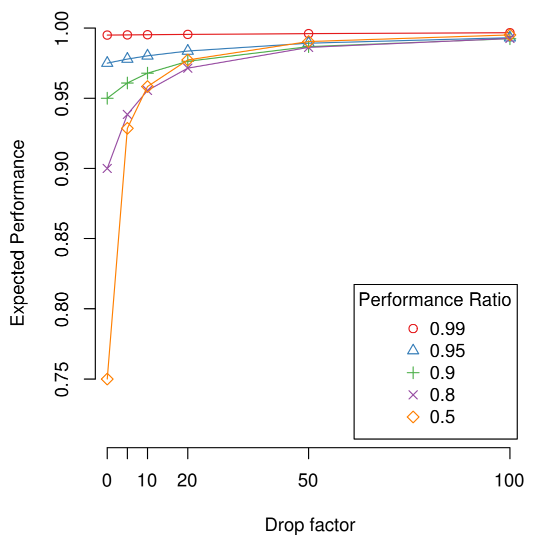


Figure 5.3: Effect of drop factor on expected performance. Values are normalized to the best possible performance.

ate from the best setting's performance will be penalized more than the others. The equation drops settings proportionally to the size of the moving average buffer too. After a setting is dropped, its MAB is reset, because by the time the setting is considered again in the exploration phase, the contents of the buffer may not be valid anymore. Moreover, the adaptive mechanism does not give a prediction for a setting until its associated buffer is full—doing so would be equivalent to not using a buffer. Therefore, $|MAB|$ exploration phases are necessary before the algorithm can decide whether a prefetch setting that has just been reconsidered again for inclusion continues to be an inefficient setting and, consequently, must be dropped once more. The bigger the size of the moving average buffer, the more potentially harmful effect that an inefficient setting may have. Thus, Equation 5.3 includes a term that drops settings proportionally to the size of the moving average buffer.

In Equation 5.3 the drop factor, DF, is the only parameter that the adaptive prefetch mechanism's designer or the end-user must select a value for. Its value will depend on the workloads that the end-user will ultimately execute on the system. Based on mathematical performance modeling and empirical analysis, it is possible to select a default value for that parameter. We use a similar approach as we did to determine the effect of the exploration-execution ratio on performance. In this case, we model the effect of changing the drop factor on the expected performance. We use the following equation to model the impact on performance of different drop factor values for the case of two prefetch settings:

$$\widehat{IPC_i} = \frac{t_1}{T} \times IPC_{best} + \frac{t_2}{T} \times \alpha IPC_{best} \tag{5.4}$$

where t_1 and t_2 correspond to the amount of time that setting one and two are respectively selected. Their values are $|MAB| + IT_i$ and $|MAB|$, respectively. Finally, T is the total interval time $(t_1 + t_2)$ and α is the reduction in performance of setting two compared to the first one. Figure 5.3 shows normalized expected performance for several drop factor values, for the case of two prefetch settings. One of the settings corresponds to the best setting in a given interval (performance = 1.0). We include results for different performances (α) for the second setting, ranging from 1% to 50% slowdown.

As it can be observed in Figure 5.3, settings that are close to the best one do not reduce performance significantly and, thus, they do not need to be dropped for a long time—if at all. As the performance of the second setting decreases, the impact on performance becomes much more noticeable. For instance, if the performance for the second setting is 50% compared to the best setting, not using the drop feature would lead to an estimated overall performance of 75% compared to when just the best setting is used. As the drop factor increases, the impact of inefficient settings clearly reduces and the expected performance tends to converge to the performance obtained with the best prefetch setting.

Algorithm 3 presents the latest version of the adaptive prefetch mechanism, both including the moving average buffer and the drop feature. As it can be seen there is no running phase in this algorithm. The running phase is not necessary anymore since the drop feature removes inefficient settings, thus, allowing us to perform a continuous exploration. Before trying a prefetch setting, the algorithm decrements the number of drop iterations for that setting, and it only actually considers the setting if it is not dropped (lines 3-4). In lines 13-18 the algorithm computes the number of iterations that

Algorithm 3 Adaptive prefetch with MAB and inefficient setting drop.

```
1: for all $t$ in $threads$ do
2:     for all $ps$ in $pref\_settings$ do
3:         $drop\_iter[t, ps] = \max(0, drop\_iter[t, ps] - 1)$
4:         if $drop\_iter[t, ps] = 0$ then
5:             set_prefetch(cpu($t$), $ps$)
6:             wait $T_e$ ms
7:             push($ipc\_mab[t, ps]$, read_pmcs())
8:             $ipc\_mean[ps]$ = mean($ipc\_mab[t, ps]$)
9:         end if
10:    end for
11:    $best\_ps = arg\,max_{ps}(ipc\_mean)$
12:    set_prefetch(cpu($t$), $best\_ps$)
13:    for all $ps$ in $pref\_settings$ do
14:        if $drop\_iter[t, ps] = 0$ then
15:            $SL = (ipc\_mean[best\_ps]/ipc\_mean[ps] - 1)$
16:            $drop\_iter[t, ps] = DF \times |MAB| \times SL$
17:        end if
18:    end for
19: end for
```

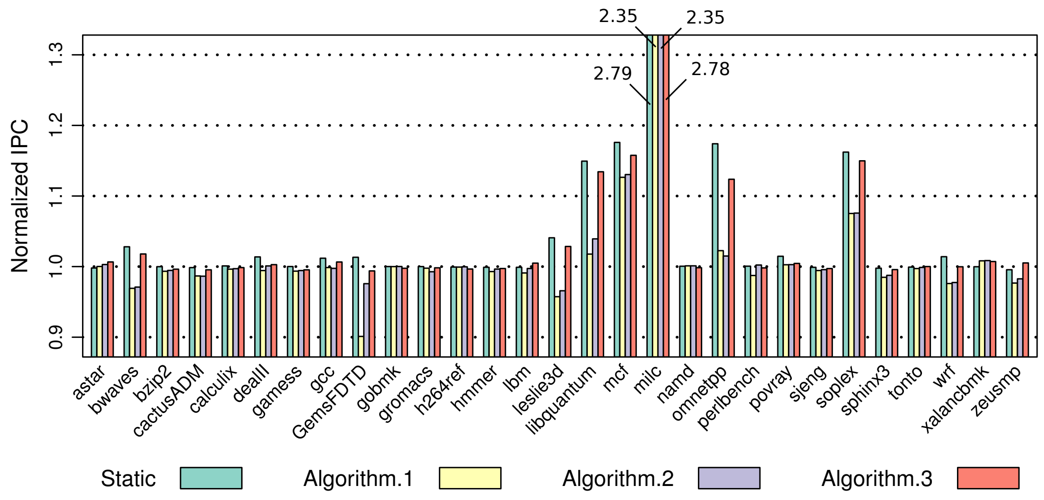


Figure 5.4: Performance results for single-threaded workloads normalized to the ones obtained with the default prefetch configuration.

inefficient settings will be dropped. We select $DF = 100$ based on the previous analysis and on empirical evaluation, obtaining good performance for all the benchmarks both for single-threaded and multiprogrammed workloads, as we will see in the next section.

5.3 Results

In this section we evaluate the performance benefits of using adaptive prefetching. We use both single-threaded workloads as well as multiprogrammed workloads composed of random SPEC CPU2006 benchmark pairs.

5.3.1 Single-Threaded Workloads

Figure 5.4 shows the results for single-threaded workloads. We present results for all the SPEC CPU2006 benchmarks. Performance values are normalized to the ones obtained with the default prefetch configuration. As it can be seen in the figure, many of the benchmarks do not experience any performance variation. That is especially true for prefetch-insensitive workloads. In that case, neither the best static nor the adaptive approaches improve performance. It is important to notice that while the first and the second algorithm may experience a performance decrease compared to the default configuration—due to, for instance, inefficient settings—, that is not the case for the third algorithm. Algorithm 3 does not perform worse than the default configuration for any of the benchmarks. That is an important observation, since otherwise it may not be "safe" to unconditionally enable adaptive prefetching.

We can observe the effect of the moving average buffer especially in the case of GemsFDTD. This benchmark is the one that suffered the most from inter-sample variability (see Figure 5.1). By using a MAB we can reduce the impact of IPC variability between samples and improve performance.

If we look at config-sensitive workloads we observe that adaptive prefetching performs nearly as good as the best static approach. SPEC CPU2006 benchmarks present little variability in terms of which prefetch setting they most benefit from along their execution. Because of this, it is typically not possible for dynamic prefetching to beat the static approach—we look at this in more detail in Section 5.3.2. The speedups obtained with the adaptive scheme are, however, very significant (in the order of 15% for mcf, soplex and libquantum). In the case of milc we observe a large speedup of 2.7X. While all those workloads benefit from prefetch and they see their performance increased when the right setting is selected for them, omnetpp behaves in a completely different way, and it actually benefits from disabling prefetch. By profiling this benchmark we have seen that it spends a significant percentage of its execution time traversing a heap. A heap is a tree-like data structure, and when traversing it, accesses between nodes are separated by a variable stride. This access pattern is very difficult for a sequential prefetcher, even if stride-N is enabled. In fact, prior research already showed that omnetpp does not benefit from prefetch[34,69]. When prefetch is disabled during all the execution (static approach), performance for omnetpp increases 17%. Adaptive prefetching detects that, and turns off prefetch most of the time, significantly improving performance too.

Figure 5.5 shows CPU and memory power consumption results for all the benchmarks when run-

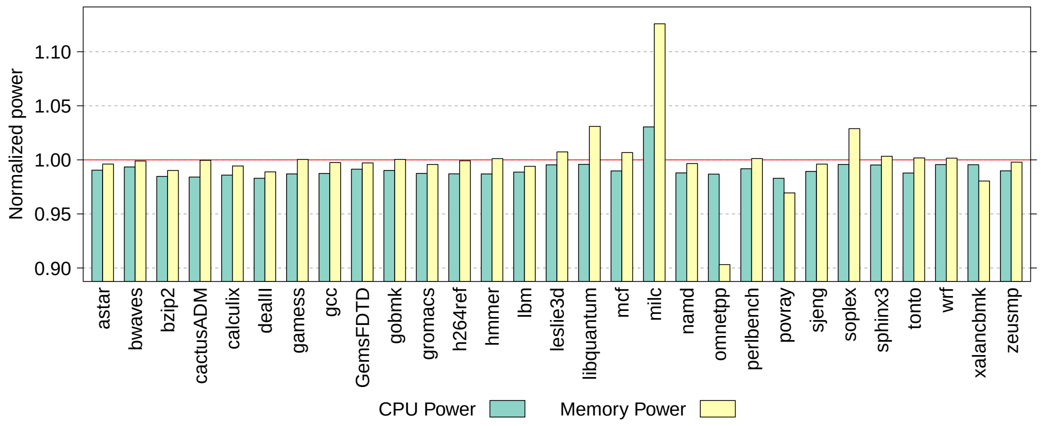


Figure 5.5: CPU and memory power consumption results for single-threaded workloads using Algorithm 3. The values are normalized to the ones obtained with the default prefetch configuration.

ning under Algorithm 3. CPU power consumption is slightly lower for all benchmarks except for milc. That benchmark experiences a 2.8X speedup when running under adaptive prefetching. Selecting the right prefetch setting reduces the impact of cache misses, increasing both CPU and memory activity in a very significant manner. In terms of memory power consumption, prefetch-insensitive and prefetch-friendly benchmarks do not experience any variation, consuming the same amount for both the default configuration and adaptive prefetching. Performance for config-sensitive benchmarks increases when the right prefetch setting is used. That extra performance implies accessing memory more intensively. Because of that, memory power consumption increases. It significantly does for milc (up to 15%) and more modestly for libquantum and soplex (up to 3%). In all the cases, the performance increase surpasses the increment in power consumption. For omnetpp, power consumption actually decreases under adaptive prefetching. Our mechanism effectively detects that disabling prefetching is the best setting for that benchmark. By doing so, useless bandwidth consumption is eradicated, reducing memory power consumption in turn. We also observe a memory power reduction for povray. Being a prefetch-unfriendly workload, disabling prefetch helps as well, reducing power consumption 3%. Another interesting case is xalancbmk. That benchmark presents two different phases. During the first one, prefetching—especially when stride-N is enabled—significantly helps. In the second one, disabling prefetch is slightly better in terms of performance. Doing so also reduces bandwidth consumption to some degree. That reduction translates into a 2% memory power decrease for adaptive prefetching compared to the default setting.

Overall, the significant speedups for single-threaded workloads, together with the fact that performance does not decrease when compared to the default configuration, converts adaptive prefetching into a very useful mechanism to improve performance for memory intensive workloads. Additionally, memory power consumption is reduced for prefetch-unfriendly workloads such as omnetpp and povray, adding further value to our adaptive solution.

5.3.2 Composite Workloads

As shown in the previous sections, our adaptive scheme is able to find, without user intervention, the best prefetch setting for all SPEC CPU2006 benchmarks, with similar performance speedups to the best static approach. For an application that benefits from multiple "best" prefetch settings over its full execution period, however, the dynamic approach generally performs better. We use the term *intra-workload prefetch setting sensitivity* to refer to the degree of potential improvement that applications may have due to benefiting from multiple prefetch settings within their execution. In the previous sections we have pointed out that a single SPEC CPU2006 benchmark does not benefit from multiple prefetch settings, thus they have a low intra-workload prefetch setting sensitivity.

Figure 5.6 shows the sensitivity for all benchmarks. We compute the sensitivity as the ratio of time where a prefetch setting different from best static setting obtains a better performance compared to the best static one. Most benchmarks present a very low sensitivity (under 5%). And the only three benchmarks with relatively higher sensitivity only experience a slight increase in their performance during less than 15% of their execution. Therefore, we conclude that SPEC CPU2006 benchmarks do not present a high intra-workload prefetch setting sensitivity.

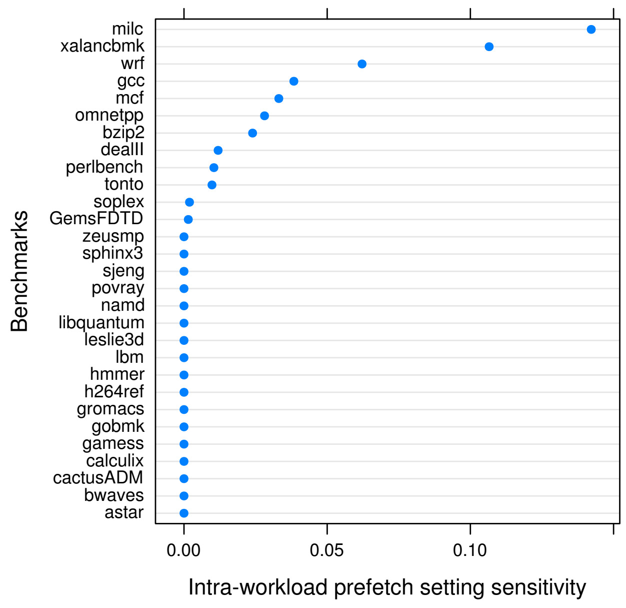


Figure 5.6: Intra-workload prefetch setting sensitivity for all the SPEC CPU2006 benchmarks.

It is, however, conceptually easy to imagine the existence of applications that would benefit from different prefetch settings during their execution. For instance, a scientific application that retrieved a large amount of data from the Internet, uncompressed the data and finally processed it, would have three very different macro-phases. Moreover, each one of these phases may benefit from a different prefetch setting. In such a scenario, the best static approach could easily perform worse than a dynamic mechanism.

Table 5.1: Performance increase for adaptive prefetching compared to the static approach for composite workloads.

Workload	IPC speedup (%)
bwaves-omnetpp	9.1
mcf-omnetpp	8.9
milc-omnetpp	10.5
libquantum-omnetpp	7.7

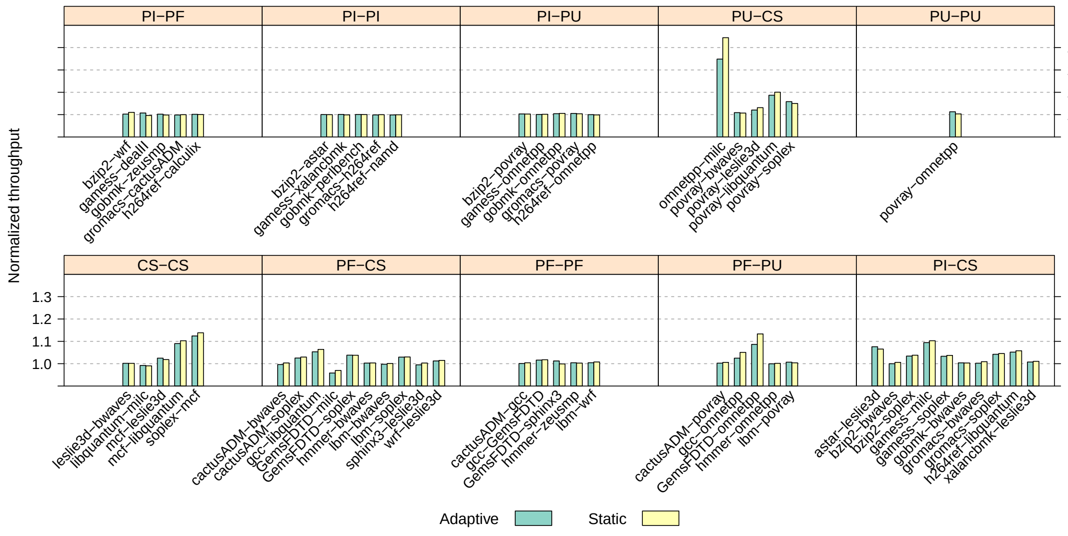


Figure 5.7: Performance results for both the static and adaptive approaches for mixed-workloads. Each workload is composed of two different benchmarks from different classes (PI=prefetch-insensitive, PF=prefetch-friendly, PU=prefetch-unfriendly, CS=config-sensitive). Four copies of each benchmark are run at the same time. Results are normalized to the ones obtained with default prefetching.

In order to demonstrate the potential benefits of an adaptive scheme compared to a static one, we construct some *composite workloads* by stitching together two SPEC CPU2006 benchmarks, one running after the other. Table 5.1 shows the speedup obtained by adaptive prefetching compared to the best static approach. As we can observe, there are significant performance improvements for workloads with a higher intra-workload prefetch setting sensitivity. As these results show, the adaptive prefetch mechanism is able to find the best prefetch setting for each of the macro-phases, thus increasing performance compared to a static approach.

5.3.3 Multiprogrammed Workloads

In this section we compare adaptive prefetching against the default configuration and the static approach for multiprogrammed workloads. Since, as we have seen in Section 5.3.1, the performance for Algorithm 3 is much better than the other two, in this section we only show results for the third algorithm. The results in Figure 5.7 are normalized to the case where all the benchmarks run with the default prefetch setting. We construct random pairs in such a way that all the benchmark types listed in Table 4.3 are represented. Each workload is composed of eight threads, four from a benchmark class and four from the other class. Each thread runs on a different core. We show results for five random workloads for each class combination except for PF-CS and CS-CS where we use ten random workloads since the result space and the performance variability are larger for these combinations. For PU-PU there is only one result, since there are only two benchmarks in PU class.

Looking at the results we observe that, as it was the case with single-threaded workloads, there is not too much difference in performance for workloads composed of prefetch-insensitive or prefetch-friendly benchmarks (PI-PI, PI-PF or PF-PF classes). For config-sensitive workloads, however, we observe very significant speedups (over 10%) for some pairs. Throughput goes up to 30% for the pair omnetpp-milc. In this case the adaptive mechanism disables prefetch for omnetpp and enables stride-N for milc, boosting the performance of both workloads. It is also important to notice that virtually in no case the performance achieved by the adaptive prefetch mechanism is lower than the baseline (using default prefetch for all the threads). The only two cases where this happens are for the pairs GemsFDTD-milc (in PF-CS class) and libquantum-milc (in CS-CS class). The reason for these results is the small absolute IPC for milc. When it runs together with other higher-IPC benchmarks, the total throughput may not increase that much—it may actually decrease—when using the adaptive approach. If we look at the individual IPC values, however, the results show that the adaptive mechanism actually improves performance. Let us examine the GemsFDTD-milc case in more detail. For that pair, adaptive prefetching worsens total throughput 4% compared to the baseline. When using the baseline, IPC values are 0.61 and 0.18 for GemsFDTD and milc, respectively. Adaptive prefetching selects different prefetch settings, and the IPC values change to 0.37 and 0.34 for the same benchmarks. These results show that GemsFDTD suffers a 35% slowdown, but the speedup for milc is almost 2X, easily compensating the slowdown for GemsFDTD. In addition to throughput, we have used other metrics such as the

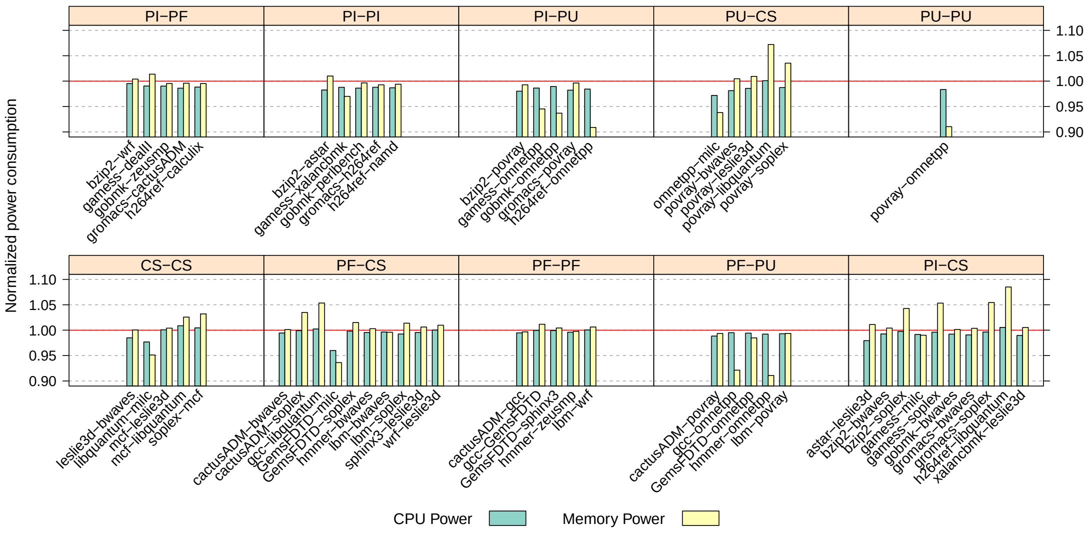


Figure 5.8: CPU and memory power consumption results for the adaptive approach for mixed-workloads (same pairs as in Figure 5.7). Values are normalized to the ones obtained with the default prefetch configuration.

harmonic speedup in order to obtain performance measurements that combine both throughput and fairness between threads in each pair. Our results show that the adaptive mechanism always obtain a better performance compared to the baseline when using the harmonic speedup metric.

We observe that the static approach always obtains a performance equal or slightly higher than the adaptive one. As we pointed out in the previous section, virtually no SPEC CPU2006 benchmark benefits the most from more than a single prefetch setting. In such a case, the static approach always obtains the best possible performance. With our adaptive scheme, however, the user gets the benefit of autonomic performance boost across all workloads (compared to the default configuration), without the need to invest into a priori characterization of each and every workload.

Figure 5.8 shows CPU and memory power consumption for the same set of pairs that we used in Figure 5.7. As it was the case with single-threaded experiments, power consumption does not significantly vary for pairs where both benchmarks are either prefetch-insensitive or prefetch-friendly. Config-sensitive benchmarks, such as libquantum and soplex, experience significant speedups when the right prefetch setting is selected by our adaptive mechanism. That extra performance is delivered through an increase in memory bandwidth usage, and therefore, memory power consumption increases too. As Figure 5.8 shows, memory power consumption can increase up to 10% for these kinds of benchmarks. An interesting example is milc; this benchmark considerably benefits from enabling stride-N, resulting in a significant performance increment. As we can see in Figure 5.8, the pairs containing milc experience a power consumption reduction when they run under our adaptive mechanism. Adaptive prefetching enables stride-N most of the time for milc, effectively capturing that benchmark's access pattern, and increasing prefetching efficiency. When we use the default prefetch configuration, the prefetcher fails to capture the strided pattern, and bandwidth consumption due to demand loads increases. Yet, (useless) prefetches are still generated, thus consuming memory bandwidth—and increasing power consumption in turn. Actually, we already observed this effect for seq-bench-stride microbenchmark (see Figure 4.5). Finally, we also observe how adaptive prefetching is able to reduce power consumption for prefetch-unfriendly benchmarks. Pairs where omnetpp appears, experience memory power consumption reductions close to 10%. These results demonstrate the potential of our adaptive prefetch scheme, not only at improving performance, but at reducing memory power consumption as well.

5.3.4 Java Business Workloads

So far we have evaluated our adaptive prefetch mechanism using SPEC CPU2006—a benchmark suite mainly composed of HPC simulation kernels and some integer workloads. Those are, however, just a fraction of the representative workloads running on real systems. Therefore, in addition to SPEC CPU2006, we have also evaluated our mechanism using SPECjbb2005[115], a server-side, Java business application that models a three-tier client/server system. This type of application is commonly used nowadays in areas such as banking, wholesale suppliers or data warehousing.

In all the experiments, we run SPECjbb using eight warehouses—each warehouse is executed by a

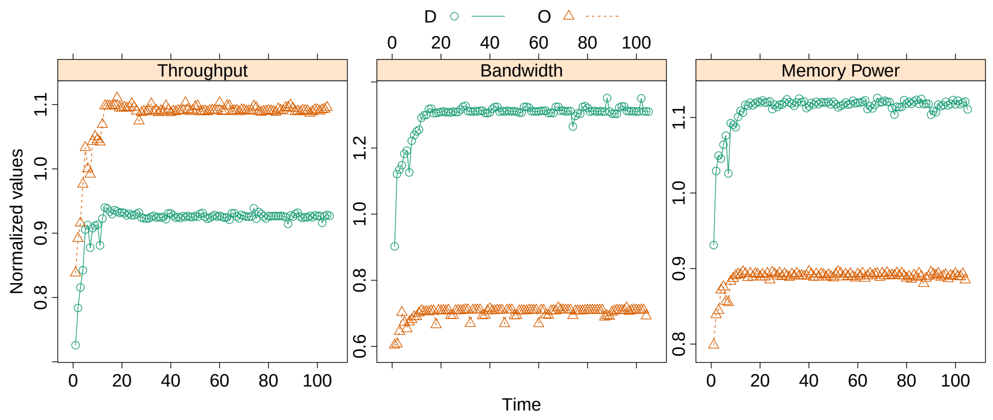


Figure 5.9: Performance and power characterization for SPECjbb2005 along its execution for eight warehouses (i.e., threads). Individual thread values are first aggregated, and then they are normalized, dividing them by the mean of all the samples. In this way we keep the same ratio between both prefetch configurations as in the original values.

different thread. Thus, we have eight threads in total, mapping each one of them onto a different core. We have tried other numbers of warehouses, obtaining similar results. Typical SPECjbb executions consist of multiple steps where the number of warehouse is increased from 1 to the number of CPUs in the system. The reason to do that is to study how the system scales as more warehouses are executed. In our case, however, we are not studying the scaling capabilities, but just how different prefetch settings affect performance and power consumption for SPECjbb. Thus, we just execute the last step—when all the cores are used.

Figure 5.9 shows the results of executing SPECjbb with different prefetch configurations: default (D) and off (O). As we can observe in the figure, throughput increases 19% when prefetching is disabled compared to using the default prefetch configuration. SPECjbb is a prefetch-unfriendly benchmark, thus benefiting from disabling data prefetching—just in the same way omnetpp (from SPEC CPU2006) does too. In the same figure we also notice that bandwidth consumption increases 56% when prefetching is enabled. Since SPECjbb is a prefetch-unfriendly benchmark, that extra bandwidth consumption is basically wasted due to inefficient prefetches. Even if the increased bandwidth consumption does not translate into extra performance—the opposite is actually true in this case—,

Table 5.2: Throughput and memory power consumption evaluation for SPECjbb2005. Results are normalized to the ones obtained with the default prefetch setting.

	Throughput	Power Consumption
Static	22.4%	-23.1%
Adaptive	21.1%	-21.9%

more frequent accesses to the memory subsystem incur into a significant memory power consumption overhead (22% increase).

In such a scenario, adaptive prefetching has the potential to both improve performance and reduce power consumption at the same time. That is a very much desired win-win situation. We have evaluated the impact of using our adaptive prefetch mechanism while running SPECjbb. Table 5.2 contains the results, showing total throughput and memory power consumption. All the values are normalized with respect to the ones obtained when using the default prefetch configuration. The static approach, as expected, significantly increases performance by 22.4% and reduces power consumption by 23.1%. Our adaptive prefetch mechanism effectively detects that disabling prefetching is the optimal choice for this benchmark, and it obtains similar results: 21.1% performance speedup and 21.9% power reduction.

5.4 OS-Based Implementation

The presented implementation of the adaptive prefetch is based on a user-level runtime. Compared to an OS implementation, a user-level runtime provides the maximum flexibility and portability. An OS-based implementation would provide several advantages, though. For instance, the overhead for reading performance counters as well as for changing the DSCR register would be reduced, since it would not be necessary to change the privilege mode to do so.

Therefore, besides evaluating the runtime-based mechanism, we studied the implementation of adaptive prefetch within the Linux OS. For that purpose, we have implemented OS-based adaptive prefetch algorithms similar to the runtime-based ones.

Algorithm 4 OS-based implementation of Algorithm 1

```
1: $ct$ = get_current_running_thread()
2: if $mode = EXPLORATION$ then
3:     $perf[ct, curr\_ps[ct]]$ = read_ipc()
4:     if $curr\_ps[ct] \neq$ last_ps() then
5:         $curr\_ps[ct]$ = next_ps($curr\_ps[ct]$)
6:         set_dscr($ct$, $curr\_ps[ct]$)
7:     else
8:         $best\_ps = arg\,max_{ps}(perf[ct])$
9:         set_dscr($ct$, $best\_ps$)
10:        $run\_quantum[ct] = RUN\_QUANTUM$
11:        $mode = RUNNING$
12:    end if
13: else if $mode = RUNNING$ then
14:    $run\_quantum[ct] = run\_quantum[ct] - 1$
15:    if $run\_quantum[ct] = 0$ then
16:        $curr\_ps[ct]$ = first_ps()
17:        set_dscr($ct$, $curr\_ps[ct]$)
18:        $mode = EXPLORATION$
19:    end if
20: end if
```

We rely on the *timer interrupt* in order to divide the execution of threads into intervals containing exploration and running phases. At each timer interrupt a reference to the thread running on the current context is first obtained (see Algorithm 4). Then the behavior of the algorithm depends on the current phase: i) If the exploration phase is active, the performance for the current prefetch setting (*curr_ps*) is recorded and the next setting is selected (lines 5-6). In case no more settings are available, the algorithm starts the running phase, after selecting the best setting found during the exploration phase (lines 8-11). ii) If the running phase is active, the running quantum is first reduced (line 14). That quantum determines how long a running phase will be. A larger value will reduce the effect of inefficient prefetch settings at the expense of a coarser adaptability.

Using OS-based algorithms we have observed similar results to the ones obtained at user-level.

These promising results encourage us to further pursue this path. We leave, however, the exploration of other OS-based adaptive schemes for future work.

5.5 Conclusions

Prefetching engines in processors are getting more sophisticated over time. While designing a new processor it is not easy to select a prefetching setting that performs well under all workloads that may later run on the processor. In response to this, processor manufacturers are exposing multiple knobs that users can tweak in an attempt to improve their workloads performance. But, doing so typically requires a costly profiling step to determine the best prefetching setting for a particular workload. Moreover, when the workload set changes overtime, the profile results might not be useful anymore. Therefore, this manual approach does not scale in a scenario where systems are shared among multiple users and workload consolidation is becoming pervasive.

In this chapter we present an adaptive prefetch mechanism capable of boosting performance by leveraging on prefetching knobs. We evaluate its impact on performance for single-threaded and multiprogrammed workloads, showing that significant speedups can be obtained with respect to the default prefetch setting. We compare the adaptive scheme to an approach where applications are first profiled and the best prefetch setting found is used for future executions. Our dynamic approach, however, frees users from profiling every application in order to find the best static prefetch setting.

6 Bandwidth Shifting: Improving System-Wide Performance

6.1 Introduction

As newer systems become capable of running a larger thread count, effectively sharing the available bandwidth to memory is becoming even more important. Total bandwidth continues to increase through multiple architectural improvements. But bandwidth per thread is actually becoming scarcer in newer systems. Therefore in this section we place the focus on a solution that balances the bandwidth usage of the different workloads running on the system. This approach will attempt to maximize the utilization of memory bandwidth, potentially improving system performance and/or reduc-

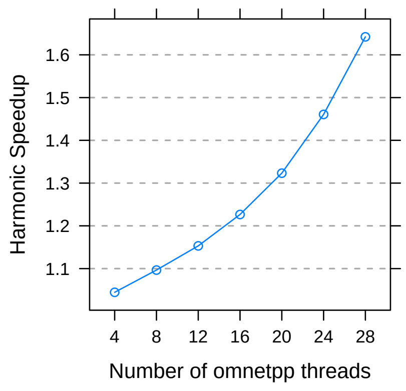


Figure 6.1: Effect of bandwidth shifting on system performance when a prefetch-efficient benchmark (bwaves) and a prefetch-inefficient one (omnetpp) run together. The X axis shows the number of omnetpp threads (x). The number of bwaves threads is $32 - x$.

ing power consumption (e.g., by turning off the prefetcher for applications that are not amenable to prefetching). To the best of our knowledge, this solution is the first one that addresses prefetch bandwidth management for CMP processors *without requiring hardware support*. Because of its design, our solution should work on any multicore system with a programmable prefetch engine—most modern processors allow users to control prefetching in different ways.

Figure 6.1 shows an illustrative example of the effect of bandwidth shifting on system performance. In this example we run two benchmarks—bwaves (prefetch friendly) and omnetpp (prefetch unfriendly). For every execution (represented as a tick in the X axis) we run 32 processes in total: x omnetpp copies and $32 - x$ bwaves copies. We compute the system speedup using the harmonic speedup between two configurations: 1) both benchmarks using the most aggressive prefetch setting, and 2) bwaves keeps using the most aggressive setting, but prefetching is disabled for omnetpp. Our bandwidth shifting mechanism would effectively shift prefetch resources from the prefetch-friendly to the prefetch-unfriendly benchmark. As the number of omnetpp copies increases, the benchmark keeps adding pressure to the available memory bandwidth thus taking bandwidth away from bwaves. If we shift bandwidth between both benchmarks by disabling prefetching for omnetpp, we observe very significant speedups. This is especially noticeable as the number of omnetpp copies increases, since prefetches issued for that benchmark saturate the bandwidth to memory. When we intelligently shift bandwidth between the applications, for 28 omnetpp threads the system speedup exceeds 60%. As Figure 6.1 demonstrates, there is ample room for an intelligent bandwidth shifting mechanism that takes bandwidth resources away from prefetch-inefficient workloads, and gives those resources to more efficient workloads.

In this chapter we first introduce a metric that estimates prefetch usefulness for a given thread based solely on performance counters commonly available in current processors. Then we present a novel bandwidth shifting mechanism capable of significantly improving system performance by taking bandwidth away from benchmarks that do not use prefetching in an efficient way and giving it to prefetch-efficient benchmarks. The mechanism does not require any hardware support, and it is able to obtain up to 18.5% speedup (10-11% on average). We also study the impact of bandwidth shifting in extreme cases where one benchmark is highly prefetch-efficient and the other uses prefetching inefficiently. Our results show that bandwidth shifting achieves much larger speedups (>1.6X). Finally we evaluate the impact of the bandwidth shifting mechanism on power consumption too.

6.2 Effect of Prefetching on Performance and Bandwidth

In order to study the potential for a bandwidth shifting mechanism and to better understand the inherent trade-offs to such a mechanism, in this section we look at the effect of prefetching on performance and bandwidth for different mix of benchmarks running concurrently on a system.

Figure 6.2 shows the throughput and memory bandwidth consumption for a subset of the SPEC CPU2006 benchmarks (the rest of the benchmarks in the suite have a similar behavior to one of the benchmarks shown in the figure). The results show throughput and bandwidth values for an increasing thread count, from 4 threads (using only one core) to 32 threads (using all 8 cores). Results are

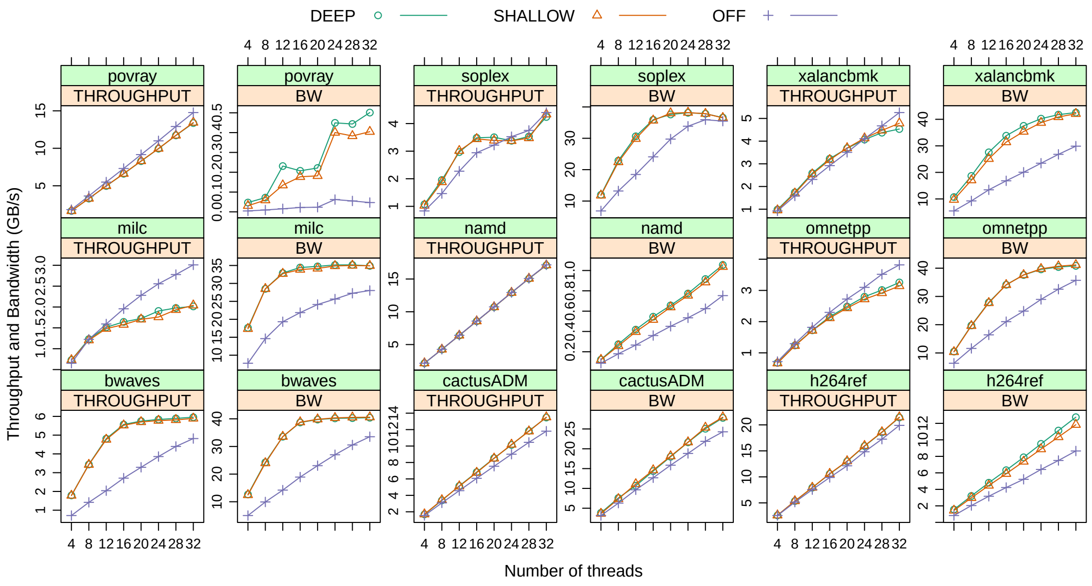


Figure 6.2: Throughput and memory bandwidth consumption characterization for a subset of the benchmarks. This subset is representative of all the benchmarks used in this chapter (i.e., the curves for the benchmarks not shown here match one of the benchmarks shown in the figure).

shown for three different prefetch configurations: DEEP, SHALLOW and OFF. In the first two settings prefetching is enabled, but with various aggressiveness configurations. For DEEP the prefetcher uses the longest prefetch distance available, while for SHALLOW it uses the shortest one. Setting OFF simply turns off the prefetcher.

As the figure shows, bandwidth and performance of certain benchmarks saturate and level off when we use more than 16 threads. In this study we use a low-end POWER7 system, where the maximum available bandwidth is 40GB/s per socket and the DRAM clock frequency is 800MHz. These results might differ for higher-end systems, which have more than double this memory bandwidth per socket. It should be noted that our goal is to evaluate the benefits of the memory bandwidth shifting idea, not to measure the effectiveness of the existing data prefetch mechanisms in this particular IBM POWER7 machine.

Some benchmarks use prefetching in a very efficient way. High performance computing (HPC) applications such as bwaves and cactusADM are good representatives of prefetch-efficient workloads. Their speedup when prefetching is enabled is linearly proportional to the extra bandwidth consumption. Prefetching is critical for these applications to obtain high performance. Other benchmarks such as h264ref benefit from prefetching, but the performance benefit they obtain does not compensate the extra bandwidth utilized. Benchmarks like this one do not utilize prefetching as efficiently as benchmarks such as bwaves do.

Benchmarks such as xalancbmk, milc and soplex are high bandwidth consumers—all of them reaching 40 GB/s (the bandwidth limit in our system) when the thread count is high. When the thread count is relatively small, prefetching increases performance since memory bandwidth is not saturated. As the number of threads running on the system increases, bandwidth to memory becomes saturated, and at some point, prefetching stops being useful. After a certain thread count (which depends on the benchmark) prefetching may degrade performance.

Other benchmarks such as omnetpp and povray simply do not benefit from prefetching for any thread count. Even if the total bandwidth consumption of povray is very low, Figure 6.2 shows that the bandwidth consumption significantly increases (in relative terms) when we enable prefetching. Performance, on the contrast, decreases when prefetching is turned on. omnetpp has a similar behavior, but this benchmark consumes a large amount of bandwidth. It saturates the available bandwidth with useless prefetches that do not benefit itself, and degrade the performance of other workloads running on the system. Another benchmark with a similar behavior is namd. When prefetching is turned off, however, its performance increase is barely noticeable.

Our findings show that the efficiency of prefetching on applications significantly varies depending on the access patterns of these applications. Bandwidth saturation is another important parameter that determines the effectiveness of prefetching—an application that benefits from prefetching when there is plenty of bandwidth available might be negatively affected by prefetching when bandwidth is scarce. All these observations suggest that a dynamic—online—mechanism, such as the one we are presenting in this chapter, is required in order to use bandwidth in a more efficient way.

Because we do not observe significant differences when varying the prefetch aggressiveness, in the design of the bandwidth shifting mechanism we only consider two settings: ON and OFF. [*] We use DEEP as the configuration when prefetching is enabled (ON).

6.3 Intelligent Bandwidth Shifting

Our intelligent bandwidth shifting mechanism dynamically takes prefetch resources away from prefetch-inefficient threads and gives those resources to more efficient threads, effectively shifting bandwidth between the threads running on the system. Giving that extra bandwidth to the threads that use prefetching efficiently leads to system (global) speedups.

In order to decide which threads use prefetching efficiently, we must first define a metric to estimate the prefetch usefulness (PU) level for a given thread. We define that metric as:

$$PU = \frac{IPC_{on}/BW_{on}}{IPC_{off}/BW_{off}} \tag{6.1}$$

where IPC_{on} and IPC_{off} are the instructions per cycle when the prefetch is on and off, respectively. Similarly, BW_{on} and BW_{off} refer to the memory bandwidth consumption for the same configurations. All these values are dynamically obtained while applications are running on the system by sampling per-thread IPC and bandwidth from the performance monitoring unit (PMU) in POWER7. The rationale behind this metric is to compare the increase in performance to the increase in bandwidth when going from prefetching disabled to enabled. The theoretical range of values for this metric is $(0, 1]$. On the one hand, workloads with a prefetch usefulness close to 0 experience a very significant

[*] Using SHALLOW instead of DEEP makes a difference when the system runs few threads in single-threaded mode. But, we are interested in more realistic cases where many threads run on the system.

performance decrease or a large increase in bandwidth consumption (without a proportional increase in performance) when prefetching is enabled. On the other hand, prefetch-efficient workloads that obtain a prefetch usefulness equal to 1 have a proportional increase in performance and bandwidth when prefetching is turned on. This implies that for every unit of bandwidth consumed by prefetching there is a linear increase in performance. This indeed is the upper limit for prefetch usefulness. As a proof, let us consider a memory-bound program that traverses an array of size N bytes. With no prefetching it takes T_{off} seconds to execute. Therefore, bandwidth consumption is N/T_{off} bytes/second. Let us now assume a perfect prefetch engine (in terms of coverage, accuracy and timeliness). Such a prefetcher does not waste any data, thus only N bytes are moved from memory into the processor as well. The difference is that in this case data is not transferred because of demand misses, but because of prefetch actions. When prefetching is used, the time the program takes to complete is T_{on} ($T_{on} < T_{off}$). Bandwidth consumption is N/T_{on} bytes/second. Since execution time is inversely proportional to IPC we have the following:

$$PU = \frac{IPC_{on}/IPC_{off}}{BW_{on}/BW_{off}} = \frac{T_{off}/T_{on}}{BW_{on}/BW_{off}} = \frac{T_{off}/T_{on}}{\frac{N/T_{on}}{N/T_{off}}} = 1 \tag{6.2}$$

This case represents the upper limit since in any other case where the prefetch engine moved useless data, bandwidth would proportionally increase more than performance and prefetch usefulness would be less than 1.

We explored the potential of extending the PU metric with some extra information such as estimators of cache pollution. But we decided to keep the metric as simple as possible for three reasons: 1) to increase the portability across different platforms, 2) due to the fact that in some systems obtaining an estimate of cache pollution may not be feasible or it may require reading a significant number of events from the performance monitoring unit (PMU)—incurring a high cost, and 3) because based on empirical observation we concluded that the effect of bandwidth saturation was a much more important

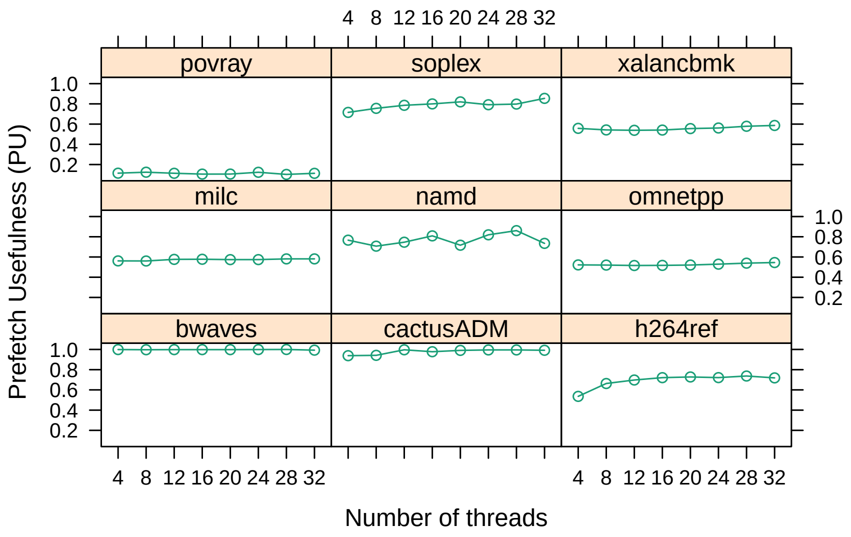


Figure 6.3: Prefetch usefulness characterization for the benchmarks shown in Figure 6.2.

factor to be addressed than cache pollution. Although we do not directly measure cache pollution or cache interference between threads, our algorithm dynamically recomputes PU for every thread running on the system. Therefore, our mechanism naturally adapts to application phases and changes in the thread mix. Our approach is also compatible with using extra information that might be potentially available in future processors.

Figure 6.3 shows a prefetch usefulness characterization for the benchmarks shown in Figure 6.2. Prefetch-efficient benchmarks such as bwaves and cactusADM consistently reach a prefetch usefulness of 1 for any number of threads (that is the largest prefetch usefulness that a benchmark could obtain). Benchmarks such as soplex and h264ref make a moderately efficient usage of prefetching. Other benchmarks such as omnetpp, milc and xalancbmk do not use prefetching in an efficient way. Therefore, it is typically better to take prefetching bandwidth away from them and to give it to more efficient workloads when bandwidth is scarce. Finally, povray is by far the most inefficient benchmark in terms of

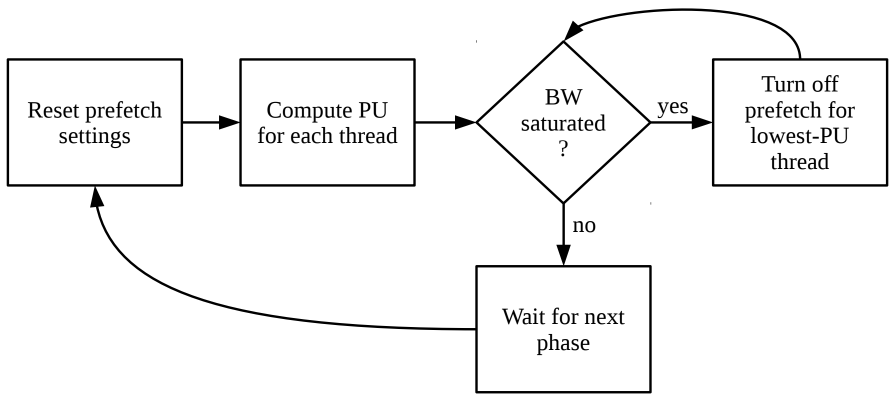

Reset prefetch settings
Compute PU for each thread
BW saturated ?
yes
Turn off prefetch for lowest-PU thread
no
Wait for next phase

Figure 6.4: Base bandwidth shifting algorithm.

prefetching usage. Because of its low bandwidth consumption, however, this benchmark does not negatively affect other benchmarks running on the system.

6.3.1 Mechanism Description

Figure 6.4 shows the base implementation of the intelligent bandwidth shifting algorithm. It uses a fully online approach—no offline profiling step is required at all. The algorithm behaves in an iterative way. At the beginning of an iteration, the prefetch setting for every thread is reset using the most aggressive prefetch configuration. After that step, the algorithm computes the prefetch usefulness for every thread, and keeps the results in a table. [†] It does so by sequentially turning on and off prefetching for each thread, and measuring IPC and bandwidth in both configurations. By doing this process in a sequential manner, our mechanism is able to indirectly account for the interferences between threads at the different levels of the cache hierarchy. The algorithm samples performance counters with 1ms granularity. Therefore, computing prefetch usefulness for all the threads takes 64ms. The distance between two sampling steps or phases is 100ms. This sampling granularity may seem coarse

[†]We actually do not just store the last read sample. Instead, we use an exponentially-weighted moving average (EWMA) in order to limit the noise coming from micro-phases during the execution. This approach is similar to using a moving average buffer as we did in the previous chapter.

compared to hardware-based solutions, but it is common for software-based ones. In fact, the Linux kernel cannot sample PMCs at a granularity smaller than 1ms (when sampling events from different groups). While hardware-based solutions are able to exploit the dynamic behavior of shorter phases, real applications present phases lasting just a few nanoseconds all the way up to the multi-seconds range. Therefore, our solution is able to adapt to the longer application's phases. But, more important, it can adapt to changing conditions in the system (e.g., when new threads are spawned and the workload mix changes). The sampling overhead is negligible since the runtime spends most of the time sleeping. We conducted tests where the runtime sampled PMCs but did not take any bandwidth shifting action. The results showed no measurable slowdown.

This is a common sampling granularity for OS and runtime adaptive solutions. Finer granularities might not be accurate and they may create significant overhead in the system. In the next step the algorithm checks the total bandwidth consumption in the system. If the threads running on the system do not saturate the total bandwidth capacity[‡], the algorithm does nothing and it just waits for the next phase or iteration. If bandwidth is saturated, shifting bandwidth from low to highly-efficient threads will typically improve system performance. Our mechanism therefore turns prefetch off for the thread with the lowest prefetch usefulness. The algorithm then checks whether the bandwidth is still saturated. While it is, the algorithm will keep turning prefetch off for the running threads, based on their prefetch usefulness—going from low to high values.

Our bandwidth shifting mechanism is implemented as a runtime that monitors the running threads (reading the PMCs) and controls the prefetchers (through an OS-interface exposed in sysfs). The scheme, however, is not restricted to this implementation. For instance, an OS-level implementation would also be possible.

[‡]Bandwidth is determined to be saturated once it reaches 90% of the peak achievable bandwidth. We have conducted experiments and determined that this threshold is the turning point where system performance degrades if prefetch bandwidth is not carefully managed.

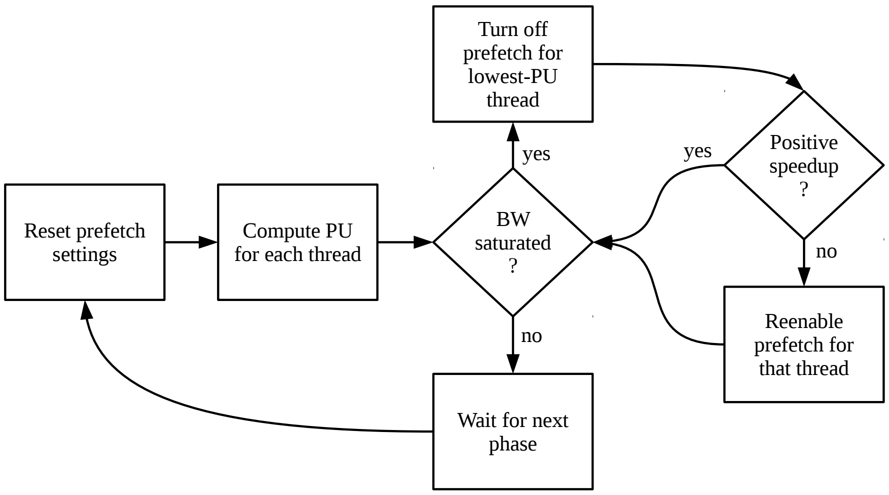

Turn off prefetch for lowest-PU thread
yes
yes
Positive speedup ?
Reset prefetch settings
Compute PU for each thread
BW saturated ?
no
no
Reenable prefetch for that thread
Wait for next phase

Figure 6.5: Enhanced bandwidth shifting algorithm with a guard mechanism.

6.3.2 Guard Mechanism

Our bandwidth shifting solutions computes prefetching usefulness (PU) for each thread, and then uses that information to shift bandwidth resources from threads with lower PU to threads with higher PU. In some circumstances, however, the available bandwidth capacity left unused after prefetching is turned off for the lowest-PU thread cannot be used by the other threads. This might happen for instance if threads with better PU cannot generate a higher rate of prefetches or demand loads to fill all the extra bandwidth capacity. In such a scenario, the thread that had prefetching disabled experiences a slowdown, and performance for the other threads remains the same—effectively leading to a global performance slowdown. This behavior might occur because of the intrinsic nature of the workloads—they might be already generating memory accesses as fast as necessary—or because of some hardware limitation. Threads running on a system share hardware resources in the memory hierarchy, and this limits their individual peak bandwidth (e.g., there is a limit on the number of simultaneous prefetch streams that threads can allocate).

Current hardware does not expose such information to the software. Yet, it is essential to prevent our bandwidth shifting solution from taking a decision that may lead to a system performance decrease. We thus extend our base algorithm with a guard mechanism that increases performance up to 33% compared to the base algorithm. Figure 6.5 shows the new version of the algorithm, including the guard mechanism. The behavior for the first steps is equivalent to the previous version of the algorithm. When the algorithm decides to turn prefetching off for a given thread, however, system performance is measured before and after disabling prefetching for that thread. In case there is a negative global speedup, prefetching is restored for that thread—and the decision to turn it off again is not considered again until the next iteration. Otherwise, the algorithm behaves as the base one and prefetching is kept turned off until another iteration starts.

6.4 Results

In this section we evaluate the performance and power impact of our bandwidth shifting mechanism. Our mechanism targets improving system performance but it considers individual thread performance as well. Therefore, we use harmonic speedup (HS) as the metric to measure performance since it both captures individual thread performance and global system performance. We compute harmonic speedup using the following equation:

$$HS = \frac{\#threads}{\sum_{i}^{\#threads} \frac{time_{i,bw-shifting}}{time_{i,baseline}}} \tag{6.3}$$

where $time_{i,bw-shifting}$ is the execution time for thread i when the bandwidth shifting mechanism is applied, and $time_{i,baseline}$ is the execution time for thread i when using the baseline prefetch configuration. In all our evaluations we run all the threads until the last one completes its execution. Before reaching that point we re-execute threads that finish earlier, thus keeping the number of threads constant during the execution. We use a configuration where prefetching is enabled for all the threads running on the system as the baseline for the evaluation of bandwidth shifting. In all the following figures we use this baseline for computing the harmonic speedup and for normalizing the power consumption.

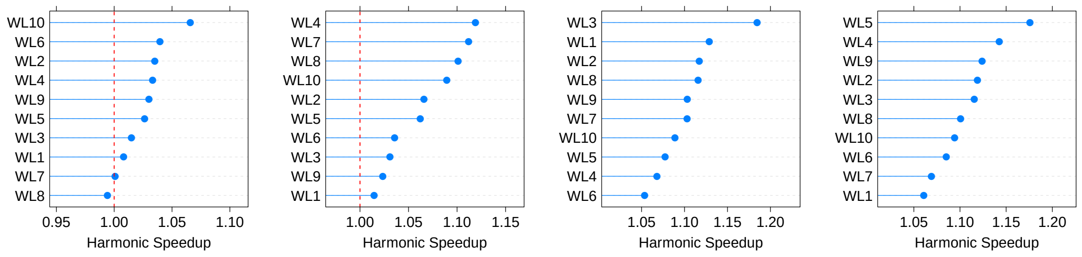
6.4.1 Random Workloads

We use workloads composed of multiple randomly-selected benchmarks to evaluate the benefits of using our bandwidth shifting mechanism. We construct workloads containing eight benchmarks each. In total 32 threads are executed since we use all four SMT contexts, effectively running 4 threads per core (we run 4 copies of the same benchmark in each core). Evaluating the bandwidth shifting with all the possible combinations of eight benchmarks is not feasible due to the vast exploration space. Instead, we create multiple groups of workloads with different characteristics in terms of memory bandwidth usage and prefetch efficiency. We use these groups to show the benefits of bandwidth shifting under different scenarios. We construct such groups by constraining the benchmarks that can be part of the different workloads. We construct the following four groups:

Random Workloads in this group are constructed in a purely random way—no constraints are enforced. This translates into workloads where the bandwidth consumption is typically not high and prefetch efficiency varies across all the possible range. We use this group to demonstrate that our bandwidth shifting mechanism does not degrade performance for workloads that, a priori, should not significantly benefit from such a mechanism.

MI-PE-high Workloads in this group are memory intensive and they contain a high percentage of benchmarks that are prefetch efficient.

MI-PE-mix Workloads in this group are memory intensive and they are composed of benchmarks with mixed prefetch efficiency levels.



(a) Random **(b)** MI-PE-high **(c)** MI-PE-mix **(d)** MI-PE-low

Figure 6.6: Performance results for randomly-constructed workloads.

MI-PE-low Workloads in this group are memory intensive and they are composed of benchmarks that are not prefetch efficient.

Table 6.1 shows the exact benchmarks used in every workload for all the workload groups. In the following figures we utilize the workload name as displayed in this table to identify individual workloads.

Performance Evaluation

Figure 6.6 shows the performance impact of using our bandwidth shifting mechanism for all the workload groups. We study the benefits of using bandwidth shifting independently for each group.

Random group Workloads in this group typically do not contain more than one prefetch inefficient benchmark. Since they are also composed of multiple benchmarks with low memory bandwidth consumption, the speedups obtained when using the bandwidth shifting mechanism are not large. Except for workload 10, speedups are always below 5%—the average speedup is 2.5%. Workload 10 is one of the workloads with highest bandwidth consumption in this group. It additionally contains two inefficient benchmarks (milc and astar). Because of this, the bandwidth shifting mechanism is able to obtain a 7% speedup. We observe a small performance degradation for workload 8. The slowdown

Table 6.1: Benchmark combinations used for creating the random workloads.

(a) Random

WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
sjeng	dealII	tonto	GemsFDTD	gromacs	GemsFDTD	perlbench	libquantum	h264ref	milc
tonto	milc	namd	milc	Graph500	soplex	lbm	zeusmp	mcf	zeusmp
zeusmp	libquantum	cactusADM	h264ref	soplex	milc	bwaves	gobmk	soplex	astar
hmmer	zeusmp	soplex	h264ref	bzip2	soplex	wrf	gromacs	hmmer	GemsFDTD
h264ref	calculix	namd	perlbench	xalancbmk	calculix	omnetpp	xalancbmk	libquantum	gcc
gobmk	leslie3d	povray	sphinx3	gamess	h264ref	perlbench	dealII	Graph500	soplex
hmmer	GemsFDTD	omnetpp	soplex	bwaves	xalancbmk	gamess	mcf	hmmer	lbm
h264ref	bzip2	mcf	sjeng	sjeng	tonto	dealII	wrf	wrf	mcf

(b) MI-PE-high

WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
leslie3d	astar	gcc	milc	lbm	Graph500	astar	sphinx3	xalancbmk	Graph500
xalancbmk	GemsFDTD	soplex	milc	lbm	mcf	leslie3d	lbm	libquantum	leslie3d
xalancbmk	mcf	libquantum	omnetpp	astar	libquantum	bwaves	sphinx3	gcc	Graph500
leslie3d	xalancbmk	GemsFDTD	leslie3d	lbm	Graph500	omnetpp	sphinx3	Graph500	xalancbmk
mcf	milc	gcc	astar	astar	leslie3d	milc	milc	mcf	bwaves
gcc	libquantum	GemsFDTD	bwaves	bwaves	gcc	astar	milc	gcc	milc
gcc	sphinx3	omnetpp	omnetpp	xalancbmk	GemsFDTD	astar	GemsFDTD	GemsFDTD	omnetpp
leslie3d	mcf	leslie3d	libquantum	sphinx3	xalancbmk	libquantum	milc	mcf	libquantum

(c) MI-PE-mix

WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
xalancbmk	omnetpp	omnetpp	gcc	milc	Graph500	gcc	GemsFDTD	sphinx3	omnetpp
astar	sphinx3	omnetpp	astar	sphinx3	soplex	leslie3d	Graph500	astar	leslie3d
mcf	bwaves	milc	gcc	milc	Graph500	astar	milc	lbm	astar
soplex	milc	sphinx3	Graph500	GemsFDTD	astar	milc	Graph500	milc	xalancbmk
milc	soplex	bwaves	leslie3d	libquantum	xalancbmk	mcf	omnetpp	astar	Graph500
omnetpp	mcf	milc	Graph500	astar	libquantum	astar	gcc	GemsFDTD	Graph500
Graph500	xalancbmk	sphinx3	omnetpp	gcc	GemsFDTD	leslie3d	Graph500	milc	sphinx3
lbm	xalancbmk	milc	mcf	gcc	soplex	milc	astar	leslie3d	lbm

(d) MI-PE-low

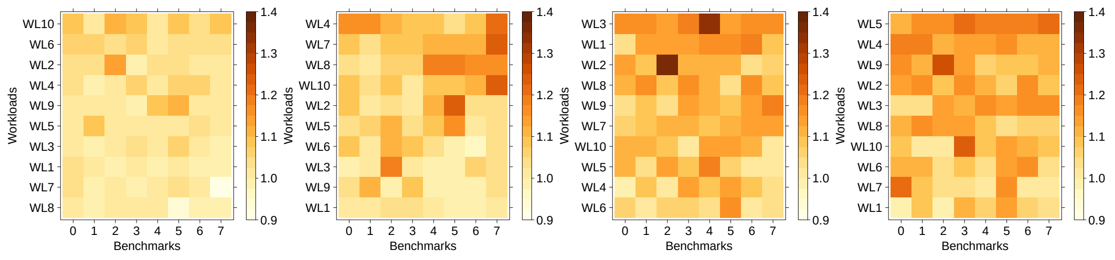
WL1	WL2	WL3	WL4	WL5	WL6	WL7	WL8	WL9	WL10
gcc	GemsFDTD	xalancbmk	milc	gcc	astar	libquantum	milc	milc	leslie3d
astar	Graph500	gcc	milc	omnetpp	omnetpp	astar	leslie3d	GemsFDTD	xalancbmk
gcc	milc	omnetpp	omnetpp	lbm	xalancbmk	milc	milc	libquantum	gcc
Graph500	Graph500	GemsFDTD	astar	milc	astar	gcc	milc	omnetpp	bwaves
leslie3d	omnetpp	milc	omnetpp	omnetpp	xalancbmk	xalancbmk	Graph500	xalancbmk	omnetpp
Graph500	gcc	astar	GemsFDTD	mcf	Graph500	bwaves	xalancbmk	milc	Graph500
omnetpp	Graph500	sphinx3	Graph500	omnetpp	libquantum	xalancbmk	Graph500	milc	astar
mcf	astar	omnetpp	milc	milc	gcc	gcc	soplex	astar	omnetpp

is, however, less than 0.5%. No other workload suffers any performance degradation, so we can effectively consider that our bandwidth shifting mechanism works efficiently even in cases where the potential benefits are not expected to be large.

Memory intensive, high prefetch efficiency Most workloads in this group contain less than three prefetch-inefficient benchmarks. But compared to the previous group, there is more room for effectively shifting bandwidth from prefetch-inefficient benchmarks to other benchmarks that use prefetching more efficiently. A 6.5% average speedup reflects that observation. In this group no workload suffers a performance degradation—the minimum speedup is 1.5% (workload 1). Workloads 4 and 7 contain multiple copies of prefetch-inefficient, memory-intensive benchmarks such as milc, omnetpp and astar. The bandwidth shifting mechanism obtains speedups slightly over 10% for these two cases.

Memory intensive, mixed prefetch efficiency Workloads in this group contain approximately 50% prefetch-inefficient benchmarks. The potential for bandwidth shifting in this scenario seems ample and the results confirm that intuition. The average speedup for this group of workloads is 10%. This is a very significant performance increase, especially considering that we are conducting our experiments on a real machine with all the software stack running on it. The highest speedup (18.5%) is achieved for workload 3. That workload contains a large number of prefetch-inefficient benchmarks plus some highly efficient ones. In such a scenario, the bandwidth shifting mechanism obtains the best results by giving the valuable bandwidth to the benchmarks that better use it. All the workloads experience speedups when run under the control of the bandwidth shifting mechanism. The smallest speedup is 5.4%.

Memory intensive, low prefetch efficiency Workloads in this group are composed of approximately 70% prefetch-inefficient workloads. It may seem as if bandwidth shifting could poten-



(a) Random **(b)** MI-PE-high **(c)** MI-PE-mix **(d)** MI-PE-low

Figure 6.7: Individual speedups for results in Figure 6.6. Each square in a plot displays the individual speedup for a benchmark within a workload. The speedup degree is shown with a color scale, going from light colors—lower speedup—to dark colors—higher speedup.

tially obtain higher speedups for this group compared to the previous one. When most benchmarks in a workload are prefetch-inefficient, however, there are not so many good candidates to shift the bandwidth to. Nonetheless, the bandwidth shifting mechanism performs very successfully in this group too. The average speedup is 11%. The minimum and maximum speedups are 6% and 18%, respectively.

According to these results, we conclude that the bandwidth shifting mechanism works efficiently across a wide range of different scenarios. Even if, as expected, large speedups are not obtained for workloads with a low degree of memory intensity and few prefetch-inefficient benchmarks, performance for these workloads is not degraded in virtually any case. When we use bandwidth shifting under the presence of memory-intensive, prefetch-inefficient workloads, such a dynamic mechanism certainly helps at improving performance—the maximum speedup obtained being 18.5%.

Fairness Evaluation

The main goal of this chapter is to present a mechanism for improving global system performance by intelligently allocating prefetch bandwidth among the different applications. For such kind of mechanism, however, it is important to assess its impact on fairness. Because of the nature of our mechanism,

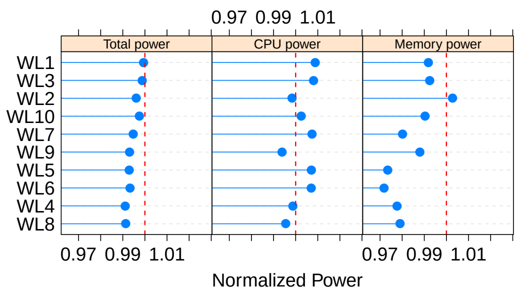
Table 6.2: Summary of individual speedups for the different workload groups.

	Random	MI-PE-high	MI-PE-mix	MI-PE-low
Min. speedup	0.92	0.97	0.98	0.99
Max. speedup	1.15	1.25	1.36	1.27
Avg. speedup	1.03	1.07	1.11	1.11

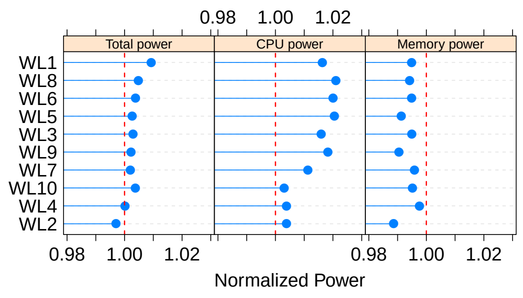
even if the global performance increases, some applications may experience speedups while others may experience slowdowns. Therefore, we study the impact of our mechanism on the fairness among the running benchmarks. In order to do that, we look at the individual speedups experienced by all the benchmarks composing the workloads.

Figure 6.7 contains a heat map showing individual benchmark speedup for each workload group. Lighter colors represent lower speedups—or even slowdowns—and darker ones stand for higher speedups. For instance, the row labeled WL3 in Figure 6.7b shows the individual speedups for all benchmarks in workload 3 (from the MI-PE-high group). Such workload contains a heterogeneous mix of workloads: some such as libquantum and leslie3d are prefetch-efficient while omnetpp is very inefficient. Other benchmarks such as gcc, GemsFDTD and soplex are sensitive to prefetching but to a much lesser degree. Our mechanism shifts bandwidth away from omnetpp so that libquantum and leslie3d can benefit from that extra bandwidth, obtaining 18 and 4% speedups, respectively. At the same time omnetpp benefits as well, since turning prefetch off for that benchmark avoids the generation of a high number of useless prefetches. In the figure we observe a dark square—representing the high speedup for libquantum—and two smaller speedups—omnetpp and leslie3d. The rest of the benchmarks experience $\pm 1\%$ speedups.

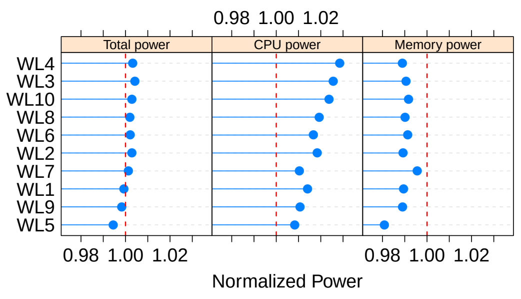
Looking at Figure 6.7 we observe that in general there is not a high degree of unfairness. Some benchmarks such as bwaves and libquantum consistently obtain very significant individual speedups up to 36%—most of the darkest squares in the figure are related to these two benchmarks. Other benchmarks experience a slowdown since bandwidth is taken away from them, but the maximum



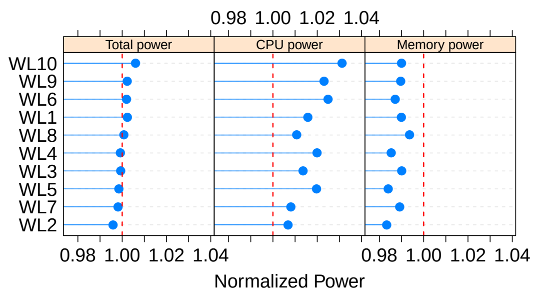
(a) Random



(b) MI-PE-high



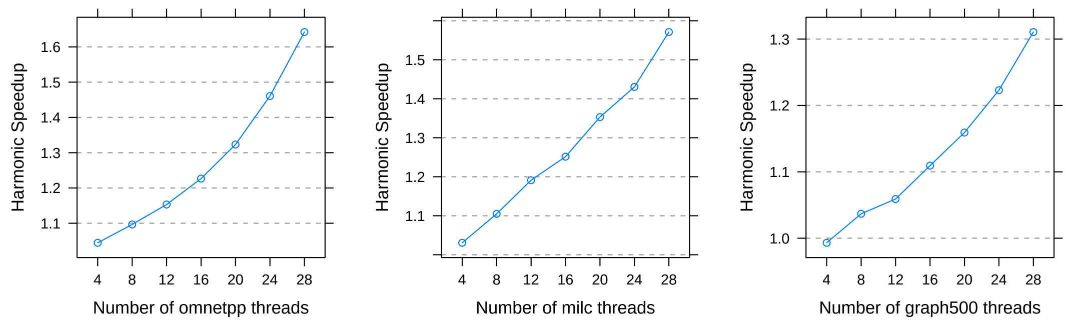
(c) MI-PE-mix



(d) MI-PE-low

Figure 6.8: Power consumption results for random workloads. Values are normalized to the case where the most aggressive prefetch setting is used for all the benchmarks.

performance degradation is 8% in the most extreme case—below 3% in the average case. Table 6.2 shows these observations. It contains a summary of the individual speedups obtained for each workload group. The results show that our bandwidth shifting mechanism—with the help of the guard feature—achieves to maintain a good level of fairness between the different benchmarks running on the system. As future work, we plan to extend the guard mechanism to keep fairness under a certain threshold if desired.

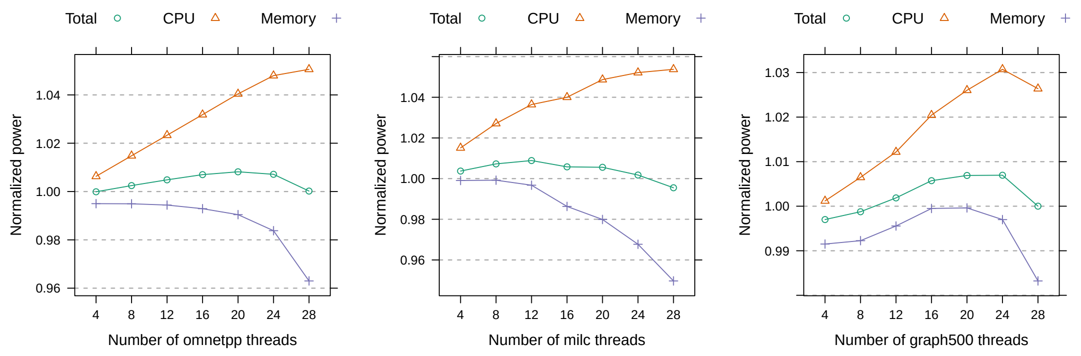


(a) bwaves-omnetpp **(b)** bwaves-milc **(c)** bwaves-Graph500

Figure 6.9: Performance results for an increasing number of copies of prefetch-inefficient benchmarks.

Power Consumption Evaluation

Figure 6.8 shows the power consumption impact of using our bandwidth shifting mechanism for all the workload groups. In general, total power consumption is not significantly affected—all values are within 1% of the power consumption obtained when the bandwidth shifting mechanism is not utilized. For virtually all workloads memory power consumption decreases (up to 3%). The reason behind that reduction is that the bandwidth shifting mechanism effectively turns off prefetch for those benchmarks that are not efficiently using prefetching. Even if prefetch-efficient benchmarks benefit from that extra available bandwidth—performance actually increases for them—in some cases, these benchmarks might not be able to utilize all the bandwidth freed by the shifting mechanism. On the other hand, CPU power consumption typically increases for most benchmarks. By shifting bandwidth to the more efficient benchmarks, useless prefetches and cache misses due to cache pollution are avoided. That increases CPU utilization for the benchmarks running on the system while at the same time it increases power consumption in the cores.



(a) bwaves-omnetpp **(b)** bwaves-milc **(c)** bwaves-Graph500

Figure 6.10: Power consumption results for an increasing number of copies of prefetch-inefficient benchmarks. Figure 6.9 shows the performance results for the same set of experiments.

6.4.2 Limit Studies

In this section we look at the potential of bandwidth shifting in more extreme cases where we run two benchmarks with very different characteristics together. Figure 6.9 shows the results of these experiments. In all the cases we run one benchmark that uses prefetching in a very efficient way (bwaves) and a benchmark that is very prefetch-inefficient (omnetpp, milc and Graph500). It is important to note that even if we only use bwaves as the representative of prefetch-efficient benchmarks, similar results are obtained with other benchmarks such as leslie3d or libquantum.

In Figure 6.9 the X axis shows the number of copies running for the prefetch-inefficient benchmark (x). In all the cases we run 32 copies in total, therefore the number of copies for the prefetch-efficient benchmark is $32 - x$. All the combinations show a similar trend: when the number of prefetch-inefficient threads is low (4) most of the bandwidth is consumed by the prefetch-efficient benchmark. Therefore, there is limited potential for bandwidth shifting in this scenario. In all cases the speedup is well below 10%. As we increase the number of prefetch-inefficient threads, however, the pressure on bandwidth increases and the impact of bandwidth shifting on performance is much more significant. In the most extreme case—when 28 omnetpp copies are run—bandwidth shifting achieves a speedup over 1.6X.

While we have seen very significant speedups in the case of randomly constructed workloads, the potential for bandwidth shifting is even higher for mixed workloads with applications that exploit prefetching in a very efficient way and applications that are prefetch unfriendly.

Figure 6.10 shows the power consumption for the experiments shown in Figure 6.9. In terms of total power consumption, the variation due to using bandwidth shifting is rather small (<1%). If we look at the CPU and memory power consumption we observe higher power variations in the order of 4-5%. As it was the case with random workloads, memory power consumption tends to go down when bandwidth shifting is used. This decrease is much more significant as the number of prefetch-inefficient threads gets larger. For instance, in Figure 6.10a we observe that when 28 omnetpp copies are run, memory power consumption goes down 4% with respect to the case where no bandwidth shifting is used. The reason for that is a total bandwidth decrease. As bandwidth shifting turns off prefetching for omnetpp, its bandwidth consumption is almost halved. The only 4 bwaves copies running on the system cannot consume enough bandwidth—even when prefetching is enabled. Since the total bandwidth to memory decreases, and active power consumption in DRAM is proportional to bandwidth consumption, using bandwidth shifting reduces memory power consumption. The same trend can be observed for the other two experiments.

CPU power consumption goes up when using bandwidth shifting. The reason is again a more effective usage of core resources since the cores do not need to wait so long for memory requests to come back from the last level cache or DRAM. All three experiments show similar results where CPU power consumption increases up to 5%. It is important to note that such a power consumption increase comes with a very significant performance speedup in the order of 1.3-1.6X.

Table 6.3: Decisions taken by the algorithms implemented in adaptive prefetching (AP) and bandwidth shifting (BWS) solutions for two sample executions (WL5 and WL7 in Table 6.1b). The fraction of time where prefetching was off for each workload is shown for both solutions. For adaptive prefetching the setting selected more times is also listed (best AP)—the column is blank if there was not a single setting that dominated the execution.

(a) WL5

Workload	Time Off BWS (%)	Time Off AP (%)	Best AP
lbm	14	0	SW7
lbm	15	0	SW7
astar	82	3	7
lbm	20	0	SW7
astar	87	4	-
bwaves	0	0	SW7
xalancbmk	80	6	-
sphinx3	2	0	7

(b) WL7

Workload	Time Off BWS (%)	Time Off AP (%)	Best AP
astar	66	6	-
leslie3d	6	0	7
bwaves	4	0	SW7
omnetpp	91	1	-
milc	92	0	SW7
astar	68	4	-
astar	74	3	-
libquantum	0	0	SW7

6.5 Comparing Bandwidth Shifting to Adaptive Prefetching

The *adaptive prefetching* solution presented in Chapter 5 is designed to independently find an optimal prefetching configuration for each thread in the system. In spite of that, adaptive prefetching can also improve performance for multi-programmed workloads. That is the case specially for relatively simple workloads where only eight threads—from two different benchmarks—are executed (see Section 5.3). But for more complex workloads such as the ones used to evaluate the *bandwidth shifting* solution presented in this chapter, adaptive prefetching might suffer from a lack of a system-wide perspective.

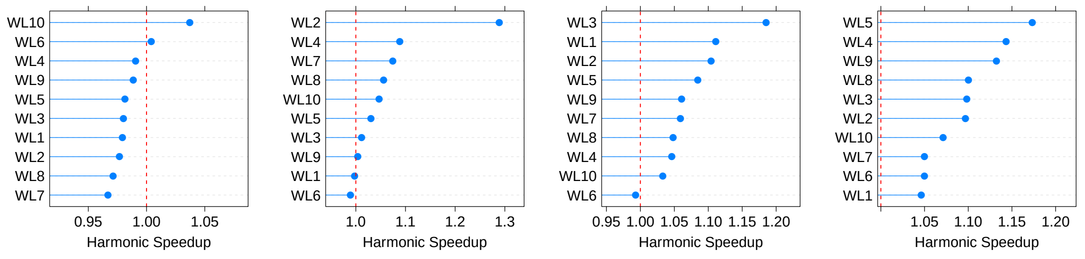
Table 6.3 shows detailed information of the decisions taken by both solutions for two sample executions (WL5 and WL7 in Table 6.1b). For adaptive prefetching, the setting that was selected more times during the execution is also listed. As it can be seen, adaptive prefetching makes good local choices: i) it enables prefetching for stores operations for lbm, a benchmark with a high write bandwidth; ii) it enables stride-N for milc, a benchmark that exhibits a strided memory access pattern, and iii) it selects the deepest setting for leslie3d, bwaves and libquantum, all of them benchmarks that significantly benefit from an aggressive prefetching setting. The rest of the benchmarks do not benefit from a specific setting. That is for instance the case of prefetch-friendly benchmarks—they benefit from prefetching, but once prefetching is enabled, changing the configuration does not affect their performance.

While these decisions might be optimal from a local point of view, they might not be ideal from a global, system-wide perspective. Let us imagine a workload composed of benchmarks that benefit from prefetching, yet with different prefetching efficiency levels. In such a scenario, adaptive prefetching will independently try to optimize the prefetching setting for each benchmark—likely by making prefetching more aggressive for each benchmark. As Table 6.3 shows, adaptive prefetching keeps prefetching enabled for all benchmarks most of the time. When the total memory bandwidth usage is high, however, it might be better to give extra resources to benchmarks with a high degree of prefetching efficiency.

This is exactly what our bandwidth shifting solution does. Benchmarks that are not prefetch-efficient such as milc, omnetpp, astar or xalancbmk have prefetching disabled up to 92% of their execution time. That leaves more bandwidth available to prefetch-efficient benchmarks such as bwaves, leslie3d or libquantum, thus improving global system performance. It is important to note that our bandwidth shifting solution enforces a certain fairness degree as the guard mechanism uses the harmonic mean to compute the global speedup when prefetching is disabled for the least-efficient benchmark (see Section 6.4.1 for details).

Figure 6.11 shows the performance speedup for each workload in Table 6.1 when using bandwidth shifting compared to adaptive prefetching. In general, bandwidth shifting significantly outperforms adaptive prefetching when system-wide performance is considered. This is due to the lack of global awareness of the adaptive prefetching solution, which just tries to optimize single-threaded performance. There are, however, some differences in the results depending on the nature of the workloads evaluated.

In the case of completely randomly selected bechmarks (Figure 6.11a) bandwidth shifting actually



(a) Random **(b)** MI-PE-high **(c)** MI-PE-mix **(d)** MI-PE-low

Figure 6.11: Performance speedup resulting from using bandwidth shifting over adaptive prefetching.

performs slightly worse than adaptive prefetching. But the slowdown is small—in the 1-3% range. We have pinned down the origin of the slowdown to the exploration phase. The most advanced version of adaptive prefetching (see Algorithm 3 in Section 5.2.3) drops inefficient settings based on their inefficiency degree so that they do not significantly hurt performance during the exploration phase. We have not implemented such a technique in our bandwidth shifting solution. Therefore, prefetching-efficient applications suffer a performance degradation every time the exploration phase runs. This, however, could be avoided—or at the very least alleviated—through several means: 1) implementing a drop mechanism similar to the one present in adaptive prefetching; 2) reducing the number of times the exploration phase needs to run by using a phase-detection mechanism for a multicore processor; or 3) eliminating the need of the exploration phase altogether—improving the information provided by the hardware to the software so that the latter can decide the relative prefetching efficiency of the different applications running on the system without the need to explicitly turn prefetching off. We leave the research on this improvements as future work.

In the other three cases—Figure 6.11 (b-d)—bandwidth shifting performs equally or better than adaptive shifting. As a larger percentage of the benchmarks composing the workload groups become more prefetching-inefficient, bandwidth shifting obtains significant speedups in the 5-20% range, with one instance reaching 30% system performance improvement.

Overall, bandwidth shifting is a better solution for managing prefetching resources from the perspective of improving global system performance. Its system-wide awareness is responsible for a better allocation of bandwidth resources—specially when the workload is memory-intensive. Moreover, its performance can be further improved with the aforementioned enhancements.

6.6 Conclusions

Effectively managing memory bandwidth consumption in highly-threaded CMP/SMT systems is becoming paramount. In this chapter we present an intelligent bandwidth shifting mechanism that assigns bandwidth resources to applications in such a way that prefetch-inefficient applications do not waste these resources while prefetch-efficient ones benefit from the extra available resources. To the best of our knowledge, our solution is the first one to address this important problem without requiring hardware support. We assess the impact of our bandwidth shifting mechanism using an extensive evaluation. We obtain very significant speedups—in the order of 10-20% for randomly built workloads and over 1.6X for more extreme cases where one benchmark uses prefetching in a very efficient way while the other is very prefetch-inefficient.

Compared to adaptive prefetching (see Chapter 5) our bandwidth shifting solution uses a global, system-wide approach that achieves a better management of memory bandwidth resources available to applications. By doing so, it obtains signficant speedups compared to both adaptive prefetching and a static approach in most cases.

7 Per-Task Energy Accounting

7.1 Introduction

The previous chapters in this thesis have focused on how to improve system efficiency by leveraging co-operation between hardware and software. Specifically, we have presented adaptive solutions that use hardware sensors and actuators to optimize resource utilization by the different workloads running on a system. We have shown that such solutions can provide significant efficiency improvements. Yet, exploiting the information given by sensors and creating policies that intelligently use hardware actuators is not limited to a processor or a system. Large-scale computing facilities (LSCFs) can also profit from obtaining advanced metrics using sensors and eventually adapt to the specific workloads running on the facility. In this chapter we present our view on how LSCFs can exploit a tighter collaboration between hardware and software in order to optimize system efficiency. In particular we analyze how accurately tracking energy consumption by the different applications and users of such facilities can help to improve efficiency—specially as the trend towards energy-proportional systems continues. We also explore the different types of sensor and actuators required for such endeavour, and what changes are required to existing hardware and software.

7.1.1 Background

Energy and power trends in large-scale computing facilities pose challenges that shape the design of next-generation facilities. The carbon footprint of societal energy consumption levels has seen intense scrutiny in recent years. According to recent statistics, the electricity demand from LSCFs shows the fastest growth among all sectors. Current facilities consume several megawatts—enough to power small towns (Belady & Malone[12]). The US Environmental Protection Agency (EPA) estimates that national energy consumption attributable to servers and data centers will soon reach more than 100 billion kWh annually (EPA[36]). Raghavendra et al.[103] estimate the corresponding electrical cost to be US$30 billion.

The cost of energy is rising, further exacerbating the problem. Recent studies show that power accounts for 13% of the total cost of ownership (TCO) of LSCFs. This cost increases up to 31% if we add the cost for cooling and power infrastructure, becoming the second largest contributor to the TCO after server costs (Hamilton[46]). Yet, while server cost has remained flat over successive generations, energy cost is expected to rise (Barroso[9]), thus increasing the relative cost of energy.

Despite these energy consumption trends, user or task-specific accounting for energy or power consumption is limited. The accounting method applied for user-level billing is usually based simply on the amount of time that a resource is used. But, this method typically does not consider the exact level of resource usage—power consumption attributable to a specific user job is either estimated on the basis of known peak (or nameplate) values for used resources, or a derated estimation for the actual peak power consumption that the system can achieve under a realistic workload (Bean et al.[10]). Nonetheless, this is a rough estimation typically based on average or worst-case behavior. Thus, using a more accurate method such as energy-aware accounting might provide significant benefits.

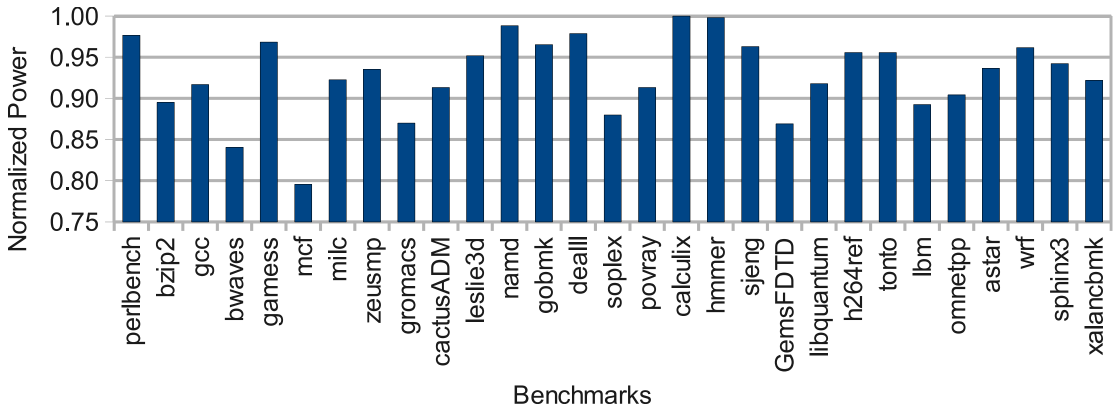
Although accounting based just on usage time and resource type and size is adequate in the present context, where static power dominates the total power consumption in current hardware, there is a clear movement towards *energy-proportional* systems (Barroso & Holzle[8]). In such systems, most of the energy an application consumes—and hence, its cost—is due to its activity. In this scenario, current accounting systems can be neither accurate nor fair. For instance, two customers can incur different utilizations across similarly allocated resources, and yet result in nearly identical usage time. Moreover, the facility owner's cost could vary significantly because of differences in power and energy consumption.

In this chapter we highlight the importance and benefits of using energy-aware accounting and billing on current LSCFs such as data centers. We explore the opportunities, as well as the problems in implementing such technology. We propose an accounting and billing method—based on accurate measurements of actual resource usage levels—that would benefit the typical consumer in terms of (generally) reduced expenses. Additionally, we show that the facility owner's adoption of such accounting metrics would drive up energy efficiency in computing facilities.

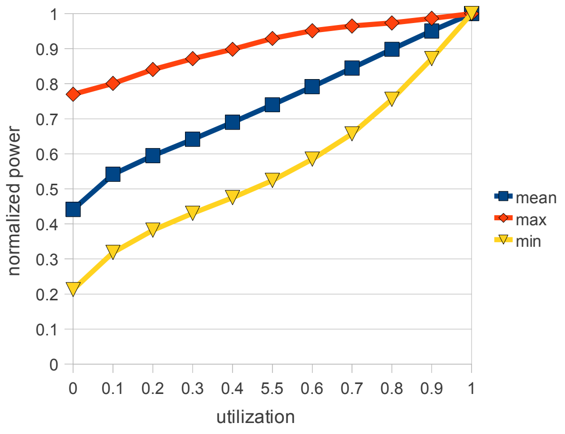
Detailed, technical solutions to the issues and trade-offs presented in this chapter are left as future work. In fact, a new research line in our group is already exploring these solutions (Liu et al.[75,77]).

7.1.2 Motivation Examples

To elaborate on the need for accurate, energy-aware accounting principles, we consider several benchmarks as proxies for the behavior of applications executed by different users. Figure 7.1a shows the results for executing all the SPEC CPU2006 benchmarks on an Intel quad-core, single-socket server system. A 10% variation in power across workloads is typical, with the maximum variation being 20%



(a) SPEC CPU2006



(b) SPECpower

Figure 7.1: Power consumption for SPEC CPU2006 benchmarks measured on an Intel quad-core system (a) and for the available results for SPECpower at several CPU utilization levels (b). *Max* and *min* refer to the most- and least-consuming systems. *Mean* is the average for all the submitted results.

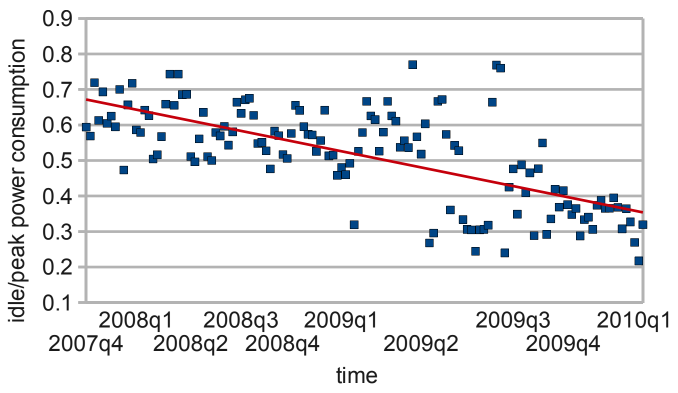


Figure 7.2: Comparison of idle and peak power consumption for SPECpower submitted results.

(between mcf and calculix). So, mcf-like and calculix-like workloads executing for the same length of time on the same platform would incur energy usage levels that actually differ by a margin of 20%. Yet, current accounting and billing practices would treat them equally. Figure 7.1b shows the power consumption at different usage levels for all the submitted SPECpower (Lange[67]) results between 2007 and 2010 avaialble at the SPEC website. This example illustrates variable-demand workloads, showing considerably different power consumption for different CPU usage levels.

These variations are already significant and they will most probably increase in the future, when system vendors build more energy-proportional systems. As idle consumption levels drive down, and peak system power remains constant (or perhaps even higher), the variation in usage-driven power profiles across different workloads is bound to increase in the future. In fact, multiple on-going initiatives are trying to reduce the significance of the static consumption fraction. Techniques implemented in current processors, such as *dynamic voltage and frequency scaling* (DVFS) and *sleep modes* with different depth levels, reduce static energy consumption. Yet for many hardware components, a high fraction of their power consumption is still static regardless of their activity.

Although current systems are not energy-proportional yet, the trend is moving towards this kind of systems. Figure 7.2 shows the ratio of idle power consumption over peak consumption for all the SPECpower results submitted between 2007 and 2010. The data is sorted by submission date and it shows a clear trend to reduce the idle power consumption's significance—a move toward energy-proportional systems. In the presence of truly energy-proportional systems, the static power cost would be almost entirely eliminated, and the dynamic cost would account for most of the energy consumption. Under this situation, all the energy that systems consume will be a consequence of application activity. Thus, considering energy consumption for accounting purposes becomes attractive.

7.2 Benefits of Energy-Aware Accounting/Billing

Users and owners of a large-scale computing facility can benefit from using energy-aware accounting and billing.

7.2.1 User's Perspective

The first benefit for users would be a more accurate and fair billing. Consider the consequences of current billing practices on the user community. Figure 7.3 shows the normalized power consumption as a function of usage for one system submitted to the SPECpower webpage. Under current accounting practices, if the user instance executes for T hours, the billing would effectively be based at the peak power rate, (P_{peak}), where usage is 1 (see Figure 7.3). Thus, the user's bill would be:

$$bill_{conv} = K \cdot P_{peak} \cdot T \tag{7.1}$$

where K is a constant value (measured in dollars per power unit per hour).

If the energy accounting were done accurately, the user would be charged depending on the average resource utilization. For example, in Figure 7.3 we observe that if the average CPU utilization was recorded to be 40%,[*] then the power consumption would decrease by slightly more than 50% and a

[*]A study at Google revealed that most of the servers typically operate at 10-50% utilization (Barroso & Hol-

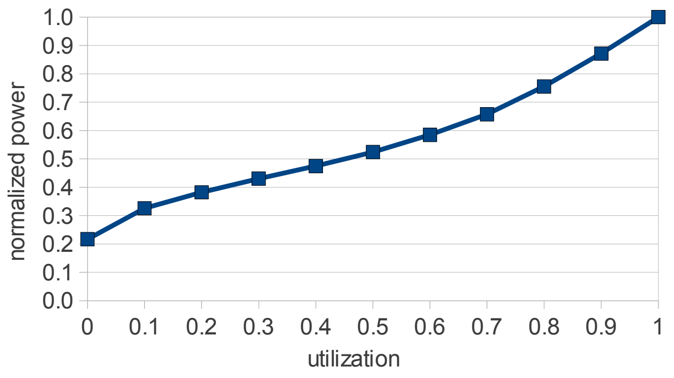


Figure 7.3: Power consumption as a function of usage for a system submitted to the SPECpower webpage. The values are normalized to the consumption when the system is fully utilized. The actual system is a Fujitsu PRIMERGY TX150 S7 server, based on a quad-core Intel Xeon X3470 with 4GB of RAM. Its maximum power consumption is 112 watts when utilization is 100 percent.

fair bill would have been:

$$bill_{fair} = K \cdot P_{40\%} \cdot T \simeq 0.5 \cdot bill_{conv} \tag{7.2}$$

Energy consumption is not the only cost for LSCF; personnel, capital cost and maintenance represent a significant part of the TCO. But, the cost of power plus cooling and power distribution accounts for up to 31% of the TCO (Hamilton[46]). Therefore, a 50% reduction in energy cost translates into a 16% reduction for the user's bill.

Energy-aware billing enables other end-user benefits. For instance, current facilities do not expose power consumption to their users. Exposing power consumption per task or virtual machine would let users understand their applications' power and energy profile, and their power consumption versus execution time trade-off. Thus, users could optimize their applications and deployment configurations to reduce their bill. This *green trend* also benefits the data center owner and society in general—for instance, by reducing the power need of a large-scale computing facility, which in turn reduces its impact on the environment.

zle[8]).

Our approach should not require users to have too-advanced computer science skills to exploit energy-accounting. We envision a runtime system that will help users select proper setups for their applications and the underlying hardware to reduce energy. Energy-accounting, on the other hand, could make users uncertain about the billing they will receive, because it depends on the actual energy the applications use. The facility owner can provide bounds or estimates on the energy that user's applications will consume using profiling (for example, using a mapping function between utilization and cost similar to Figure 7.3).

7.2.2 Owner's Perspective

In addition to the benefits for end-users, there are several reasons why a facility owner should invest in accurate, energy-aware accounting.

Finer-grain precision in allocating and managing cooling resources Today's LSCFs design the cooling infrastructure so it can effectively dissipate the heat produced by the systems under worst-case load scenarios—either based on the sum of nameplate powers across all the facility resources or on a derated estimation of the actual peak power consumption under realistic workloads. As we mentioned earlier, however, servers are typically underused. Therefore, facilities might consider the possibility of reducing cooling costs by underprovisioning the cooling resources, based on typical or observed "peak" workloads in the facility. But, heuristically fixing thermal thresholds could lead to frequently needing to engage performance-throttling mechanisms or to tripping fuses, producing unplanned server outages. Precise energy accounting practices would result in better runtime task allocation and cooling resource allocation to prevent unplanned outages or performance shortfalls.

Safe workload consolidation In prior non-virtualized systems, once a user instance received some physical resources, no other user would be able to share those resources. In such a situation, time is indeed money; so, even if the user instance is not using the allocated resources, it would make sense to charge the user a flat, per-hour "rental" rate, because once a set of resources is tied up, the owner cannot make rental income out of those resources from any other waiting customer.

With the advent of virtualized hardware, the owner can make money from multiple customers sharing a resource. The net resource utilization could then approach 100%, a good business proposition. In this new scenario, the owner has no reason not to move to an energy-aware accounting system based on actual resource usage. Since the total usage across all users approaches 100%, the net effect is that the total bill amount across multiple users sharing the same system basically follows Equation 7.1 again, with the total revenue approaching $K \cdot P_{peak} \cdot T$.

A built-in energy accounting system could guide the workload management system to make scheduling decisions that result in safe, more efficient workload consolidation. For example, let's assume that a system can run N virtual machines simultaneously. When selecting a subset of virtual machines for execution, it is hard to determine whether the power or energy threshold would be exceeded, so the virtual machine manager must be conservative. With per-virtual machine energy accounting, at the time of composing a workload of N virtual machines, we know the power consumption of each VM and thus the workload. Therefore, energy-accounting improves efficiency, and we can consolidate more virtual machines simultaneously. By doing so, we can add more computing nodes and service more customers with the same power budget—a clear benefit for the data center owner.

Reduction in energy costs Motivation for end users to reduce their energy consumption (and bill) will also drive down the total energy consumption incurred by the data center. In a context where energy cost is a significant fraction of the TCO, and electricity price is increasing, the data center owner will welcome any reduction in power consumption.

7.3 Target Facilities

Several types of facilities exist, each with a different business model. Energy-accounting targets multiple facility types, though its potential benefits depends on their characteristics. We consider two major types of facilities:

Private data centers In this case, the facility follows a dedicated provisioning in which some physical nodes (or the slots to place them) are leased to a given user. A user's application can span multiple nodes, and the overall provisioned capacity is dedicated to the deployed applications. In this model, the leased nodes' overall operation and power cost can be attributed to the running applications. Only a per-node energy accounting is needed. Supercomputers in national labs are examples of systems in this category

Dedicated hosting services and colocation facilities Adoption of this type of facility is growing, and we envision clear benefits from energy accounting in these types of facilities. Users are billed according to the number of hours their instances (e.g., virtual machines) are on without considering the detailed compute resource usage profile. Although some parameters—such as data transfer, I/O, and disk space—are used for billing purposes, two instances running for the same number of hours will be charged the same, in terms of wall-clock CPU time, regardless of what the actual CPU and memory usage is. Examples of these types of facilities are Amazon EC2,[4] Google App Engine,[44] or IBM Bluemix.[51]

We can make several considerations when applying energy accounting in virtualized data centers. First, resource providers such as Amazon EC2 provision end users with virtual resources. Here, the direct mapping of the end user applications to actual physical resources is not transparently known. Moreover, because applications are not directly mapped to physical hardware, direct hardware profiling is not generally available at the application level. Instead, the *virtual machine manager* has direct access to hardware profiling and knows when applications are really mapped to hardware. This layer is an appropriate level for implementing energy-accounting capabilities.

Second, virtualization vendors further provide additional resource management vehicles such as resource guarantees, limits and shares. In this case, each application's and virtual machine's contribution to energy consumption depends on provisioned virtual resources, the imposed resource constraints and the underlying resource-sharing mechanism. All these management vehicles are orthogonal to energy accounting. For instance, some applications handle asynchronous events and have hard latency requirements. To deal with this situation, the application or user must reserve resources in advance. From the energy-accounting point of view, this just implies that the user must pay the reserved resources' static power consumption. Once the user's application starts running, it follows our proposed energy-accounting policy.

Finally, many virtualization technologies also employ additional resource optimizations such as *page sharing* across compatible virtual machines, *linked clones* (Antony[6]) with shared based images, *memory overcommitment* and dynamic *memory ballooning* (Waldspurger[126]). These techniques, while improving overall resource use efficiency, also blur the resource and energy usage association with individual applications and end users.

7.4 Energy Accounting Design and Trade-Offs

There are challenges and opportunities associated with energy and power accounting at various granularities in a large-scale computing environment. Some changes are also required, both at the hardware and software levels, to provide accurate energy accounting. The infrastructure required to accurately track power and energy dissipation can vary significantly over the computing spectrum. However, there are several common considerations that apply to all systems.

GRANULARITY VERSUS OVERHEAD A critical point in an energy-accounting system is to decide the level at which energy is tracked. The hardware and software overhead increases for per-user rack and node-level accounting. Within a node, accounting becomes even more challenging. At hardware level, we must decide the area, power and cost overhead of the additional hardware blocks to provide accurate accounting. At the software level, we must decide how much overhead we will allow for tracking energy consumption.

FAIRNESS From the user's perspective, an important principle to follow is that different runs of the same application with the same input exhibit a similar energy profile. This is called the principle of accounting, and it is currently applied to CPU time accounting (Luque et al.[80,81]). In an ideal scenario, the application reaches the same energy-accounting result for the same input, regardless of the applications it is coscheduled with. In reality, however, several factors complicate the ideal case, potentially causing significant variation for repeated runs. Accurate, fair energy-aware accounting and billing should account for this.

7.4.1 Static and Dynamic Power Consumption

To accurately track energy consumption, we must first break down power-related costs between *static* and *dynamic* costs. The former accounts for the power that does not depend on system activity (e.g., the power consumption of an idle machine that is not running any user process, or power distribution unit loses). The latter is related to the extra power consumed when there is user activity on a system. The fraction between static and dynamic power depends on both the system under consideration and the workload itself.

For the dedicated data center case—where users do not share nodes—that distinction is not really necessary, because the total power consumption can be typically measured at the node level.[†] However,

[†]If some external resources (e.g., storage) are shared, some of the following discussion might apply to the

for virtualized data centers, we must estimate the fraction of these components that must be attributed to each virtual machine running on the system.

Static Power Splitting the cost of static power consumption among virtual machines depends on the level at which resources are shared, leading to several possibilities with different associated accuracies and overheads. The easiest solution is to split the static consumption among all the virtual machines mapped to that node either evenly among all them or proportional to each virtual machine's dynamic power consumption (Kansal et al. [65]). If a higher accuracy is desired, we can individually look at the system's components. We need either hardware support to derive the static power consumption or the hardware vendor to provide these values. For instance, current performance-monitoring counters or power sensors are not enough to derive the static power consumption of a system's individual components.

We differentiate two component types on the basis of their nature:

Spatial-sharing In spatially shared components (such as cache or memory) there is a linear relation between the amount of space a virtual machine demands and the cost of static power. If at a given instant a resource with an associated space of M_{total} bits has a static power consumption of S_{total} watts, it can be broken down among N virtual machines as follows: $S_i = (M_i/M_{total}) \cdot S_{total}$, in which $\sum_{i=1}^{N} M_i = M_{total}$ and $\sum_{i=1}^{N} S_i = S_{total}$, where M_i and S_i are the amount of space used and the static consumption incurred by virtual machine i, respectively.

Temporal-sharing Temporally shared components (such as the CPU or hard drive) consume static power proportionally to the duration they are enabled. In this case, we can use an *interval-based accounting* approach. Let's assume we divide the time into intervals of fixed length I. If during a given interval a certain amount of virtual machines access a component, all of its static

accounting of these resources.

power consumption is charged to these virtual machines. The other running virtual machines should not be charged, because we assume that the components can go into a low-power mode if they are not accessed for an interval I. Thus, the static energy consumption for virtual machine i during time interval k—when N_k virtual machines are accessing the component—is $S_{i,k} = S_k/N_k$, where S_k is the static energy consumed by a device during interval k. It follows that the static power charged to virtual machine i after N intervals is $\sum_{k=1}^{N} S_{i,k}$.

We can find components that, depending on their power-saving capabilities, present both spatial and temporal sharing characteristics. In that case, we can apply a hybrid combination of the methodology we discussed in the previous paragraphs.

Dynamic Power Splitting the dynamic power consumption among virtual machines is a complex task that in some cases might require hardware or software support (see Section 7.4.3). We can use several approaches for attributing energy consumption to multiple virtual machines sharing a node. CPU usage is a high-level metric that typically correlates well with power and energy consumption (Barroso & Holzle[8], Kansal et al.[65]). Its main advantage is that it is easy to collect, thus reducing the complexity and the overhead for energy-accounting implementation.

Additionally, if a higher accuracy level is desired, we can estimate energy consumption on the basis of lower-level metrics, such as events occurred in the system. We can use different sources to collect events: performance counters such as instructions per cycle (IPC) and cache misses, or operating system statistics such as I/O operations. Bircher & John[15] demonstrate the high correlation between system events and power consumption. Obtaining these metrics, however, may have higher associated overheads compared to using high-level metrics.

The type of metrics required to estimate power consumption also depends on the workloads being executed within the virtual machines. For instance, in CPU-intensive workloads, high-level generic metrics are generally less useful. CPU usage for these kinds of workloads is close to 100%, rendering CPU usage-based power estimation not so useful. Yet, as Figure 7.1a shows, significant power consumption variation exists among workloads running at 100% CPU usage. We can use workload-specific, high-level metrics—such as transactions per second—but this solution is not portable among different workloads, and it might be challenging to make these metrics visible from outside the virtual machine. Therefore, in the case of CPU-intensive workloads, event-based metrics are a much better fit to accurately estimate energy consumption.

7.4.2 Application Interference and System Activity

In shared environments there is generally interference among virtual machines accessing the same hardware resources. Nowadays, most facilities use processors that can concurrently execute more than one thread—based on chip-level multiprocessing (CMP), simultaneous multithreading (SMT) or a hybrid approach. In these systems, two different virtual machines share certain resources when they are executed at the same time. Although program output will not change, the actions that the system takes to obtain this output could differ compared to when a virtual machine is executed in isolation. For instance, the aggregated memory footprint of both virtual machines can exceed the amount of cache or memory installed in the system, leading to memory or disk accesses that would not occur if the virtual machines ran in isolation. Luque et al. [80,81] show that the interaction between multiple applications running on a CMP can lead to errors in CPU time accounting up to 19%. Including hardware support for tracking intra- and inter-task interferences can reduce the error down to 1%. Similarly, using mechanisms based on tracking per-thread component usage would make energy-aware accounting more precise.

Another source of interferences is system activity caused by housekeeping (e.g., freeing virtual memory and cleaning system logs). Finally, optimizations across virtual machines create interactions among them as well. The challenge here is to determine how to account for the energy that the system consumes considering such interference. Current solutions such as Kansal et al. [65] do not focus on these issues since hardware and operating system support would be necessary to increase the accuracy of their energy-accounting proposal.

7.4.3 Hardware/Software Support for Energy Accounting

As we have shown, several shortcomings exist in obtaining accurate energy accounting with low overhead. But, new hardware support could overcome some of these problems. First, some current systems already let us obtain power measurements at the processor level. A standard, accurate way to obtain similar measurements for a system's most consuming components can greatly enhance the accuracy of energy accounting. Second, an easier way to derive power consumption from performance counters is desirable. Kadayif et al.[64] presented a framework based on performance counters to obtain energy measurements. But, a native hardware implementation will probably prove more accurate. For instance, the IBM POWER7 processor internally uses a power proxy based on more than 50 different architectural events to estimate the power consumption for each core (Floyd et al.[38,39]). Estimates from such proxy can significantly improve the accuracy and granularity of energy accounting solutions. Third, although we can use performance counters as a power-proxy, we cannot use current performance counters to derive static power consumption. For instance, including hardware support to obtain the instruction mix per thread—or alternatively recording unit utilization using PMCs—can significantly increase the accuracy on power consumption estimation. Fourth, as we mentioned earlier, hardware support to overcome application interference can also help to improve the accuracy of energy accounting.

Software support can improve energy accounting's accuracy as well. For example, the operating system or the virtual machine manager can help by tracking the time that resources are being used by the operating system itself, without contributing to a direct profit for the user. Also, interaction between the accounting system and the virtual machine monitor can help to track energy usage in the presence of virtual machine optimizations such as those described in Section 7.3. The operating system can also use performance counters to create a profile of each application. Such profile could contain usage information for the different units in the system. This information could later be used to enhance estimates for static and dynamic power consumption, thus improving the accuracy of energy accounting.

7.5 Conclusions

In this chapter we make a case for energy-aware computing under the current context where energy consumption in large-scale computing facilities is increasing and it is becoming a bigger fraction of their TCO. We argue that, in this scenario, introducing energy accounting will benefit both end users and facility owners. Additionally, using energy consumption for accounting purposes can trigger a spiral process that leads to "greener" facilities and reduce the carbon footprint associated with large-scale computing.

The complexity of implementing an energy-aware accounting solution depends on the environment characteristics. The case of dedicated systems is considerably simple. But, multiple research challenges exist for shared environments. Interaction among the different layers in the system (hardware, hypervisor and software) is necessary in order to obtain accurate accounting systems. We discuss several options and trade-offs that are of importance to the design of an energy accounting solution. The results obtained from this exercise have already started a new line of research in our group (see Liu et al.[75,77]).

Finally, we also argue on the importance to continue the trend towards energy-proportional systems. In fact, an energy-aware accounting will benefit from this trend and, at the same time, can accelerate it as demand for "greener" computing grows.

8 Conclusions and Future Work

8.1 Conclusions

Current computing systems use hardware and software cooperation to a certain extent to improve system efficiency. Yet, in many cases actuators exposed by hardware are left unused. Additionally, system efficiency could be further optimized if hardware were to expose more actuators to software. This would further enable software to adapt resource allocation based on application's needs, and improve a certain target metric.

In this thesis we show the potential benefits of using hardware and software cooperation to improve resource allocation and system efficiency in a server-class system. We then implement several adaptive policies that rely on a given hardware actuator—a programmable data prefetching unit—to improve performance and, in some cases, reduce power consumption too.

Our evaluation of different actuators available on an IBM POWER6 system (e.g., nap mode and hardware thread priorities) as well as other forms of hardware and software cooperation (such as tickless kernel and resource-aware thread placement) revealed that significant improvements in system efficiency can be obtained. The observed improvements go from single digit percentage up to close to 4X improvements to ED^2P—depending on the specific mechanism under consideration.

We describe and evaluate the implementation of an adaptive solution to improve application efficiency by dynamically controlling the hardware data prefetcher in an IBM POWER7 processor. Prefetch engines in current server-class microprocessor are getting more and more sophisticated. Specifically, the POWER7 processor contains a programmable hardware data prefetcher, allowing users to control different knobs in order to adapt the prefetcher to workload requirements. We evaluate the impact of our solution on performance for single-threaded and multiprogrammed workloads, showing that significant speedups can be obtained with respect to the default prefetch setting. We also show how our adaptive mechanism reduces power consumption for prefetch-unfriendly benchmarks. We compare the adaptive scheme to an approach where applications are first profiled and the best prefetch setting found is used for future executions. Our dynamic approach, however, frees users from profiling every application in order to find the best static prefetch setting.

Our bandwidth shifting solution goes beyond improving performance for individual applications, and it instead targets global system performance. Our results show that effectively managing memory bandwidth consumption in highly-threaded CMP/SMT systems is of paramount importance. We present an intelligent bandwidth shifting mechanism that assigns bandwidth resources to applications in such a way that prefetch-inefficient applications do not waste these resources while prefetch-efficient ones benefit from the extra available resources. To the best of our knowledge, our solution is the first one to address this important problem without requiring hardware support. We assess the impact of our bandwidth shifting mechanism using an extensive evaluation. We obtain very significant speedups—in the order of 10-20% for randomly built workloads and over 1.6X for more extreme cases where one benchmark uses prefetching in a very efficient way while the other is very prefetch-inefficient.

Although we use POWER6 and POWER7-specific measurements and analysis in this thesis, the basic insights gleaned generally also apply to other (non-POWER) systems that expose hardware actuators—allowing software to implement adaptive policies on top of them. For instance, some Intel processors allow users to enable or disable individual prefetching engines in the processor (see Intel[53]). Certain versions of Intel processors also expose other actuators such as cache partitioning mechanisms. And, the latest pSeries processor from IBM—the POWER8—adds extra knobs to the prefetching unit such that, for instance, the urgency to achieve steady-state prefetching can be programmed by users (see POWER ISA[99]). Similar solutions to the ones developed in this thesis can be applied to these other actuators.

Beyond the case of a single computing system, we also analyze how large-scaling computing facilities could benefit from a larger degree of sensors and actuators exposure to the software. We study the design of a per-task energy accounting system, and we explore the multiple trade-offs in the construction of such a system. The results from our analysis have already spawned a new area of research in our group.

Overall, this thesis shows clear benefits from a strengthened collaboration between hardware and software. By exposing sensors and actuators to the software, the latter can tune resource allocation to the demands of the workloads running on the system. This thesis also implements different adaptive policies that illustrate how system efficiency can be boosted by tightening the degree of hardware and software collaboration.

8.2 Future Work

The adaptive prefetching solutions presented in this thesis target systems running a mix of heterogeneous workloads—similarly to what it occurs in a data center. Balancing of shared resources is specially critical in such environments. But, other types of workloads might benefit from solutions that dynamically control the prefetching unit. A related group at Barcelona Supercomputing Center has recently started to explore the potential of such solutions for parallel applications. Prat et al.[100] have extended the OmpSs programming model (see Duran et al.[31]) so that it can optimize the prefetch configuration for the different phases in parallel applications.

Most of the work in this thesis is conducted on real systems. Developing solutions that actually work on top of real, existing systems has definite advantages, but it might face some challenges as well. For instance, in a simulator-based prototype it is straightforward to measure the traditional metrics used to evaluate prefetching performance: accuracy, coverage and timeliness. On a real system, however, using performance counters to estimate prefetching usefulness presents some difficulties. During the development of our adaptive prefetching solutions multiple events were considered to be used in the design of the prefetching usefulness metric. Unfortunately, the candidates that were found did not behave as expected.

The work conducted during this thesis was done in collaboration with IBM Systems and Technology Group (STG). As part of our feedback, we exposed the shortcomings that we faced to construct an efficient metric to evaluate prefetching usefulness. The newest POWER processor (POWER8) includes new events that help to identify useless prefetches. As future work, it would be very interesting to analyze the potential of using these new events to measure prefetching usefulness in such a way that we could avoid the exploration phase currently present in our adaptive solutions. In case of success, this would be a good example of hardware and software codesign (see Shalf et al.[109]).

Another area of research that would benefit adaptive resource management mechanisms is phase detection and prediction in highly-threaded processors. While there is a significant amount of work for single-threaded processors (Denning[30], Isci et al.[54,55], Sarikaya et al.[107]), the same problem is far from being solved when many threads run on the same system. Yet, if accurate phase detection and prediction for highly-threaded processors was available, adaptive solutions could significantly benefit from it. Triggering a reconfiguration of a given actuator could be done in a smarter way—only when a change in applications' behavior took place.

The solutions presented in this thesis are research prototypes that were used to show the potential benefits of leveraging hardware actuators to improve system efficiency. Undoubtedly, implementing such solutions as a product in either the operating system or some form of runtime is something we also envision as future work.

In the context of per-task energy accounting, the insights gained during this thesis have already spawned a new line of research in our group. Liu et al.[75,76,77] have presented hardware solutions to estimate per-task energy consumption on multi-core processors and DRAM devices. As future work, we expect further models to account energy consumption for other computing components.

From a more general point of view, we would like to extend the work in this thesis by developing similar solutions for other hardware actuators as well as finding opportunities to expose more actuators to the software. Doing so would require some form of coordination in the adaptive software that took care of the potential interferences between the different actuators being handled. These two new areas of research would definitely strengthen the cooperation between hardware and software—making systems more efficient.

References

1. Abeles, J. et al. (2010). *Performance Guide for HPC Applications on IBM POWER 755 System*. IBM. https://www.power.org/events/Power7/Performance_Guide_for_HPC_Applications_on_Power_755-Rel_1.0.1.pdf.

2. Abraham, B. & Ledolter, J. (1983). *Statistical Methods for Forecasting*. Wiley series in probability and mathematical statistics: Applied probability and statistics. Wiley.

3. Adl-Tabatabai, A.-R., Hudson, R. L., Serrano, M. J., & Subramoney, S. (2004). Prefetch Injection Based on Hardware Monitoring and Object Metadata. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI (pp. 267–276).: ACM.

4. Amazon (2015). Amazon Elastic Computed Cloud (Amazon EC2). https://en.wikipedia.org/wiki/Amazon_Elastic_Compute_Cloud.

5. Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., & Weihl, W. E. (1997). Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)*, 15(4), 357–390.

6. Antony, J. (2015). Virtual machine cloning. US Patent App. 14/020,303.

7. Baer, J. L. & Chen, T. F. (1991). An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, SC (pp. 176–186).: ACM.

8. Barroso, L. & Holzle, U. (2007). The Case for Energy-Proportional Computing. *Computer*, 40(12), 33–37.

9. Barroso, L. A. (2005). The Price of Performance. *Queue: Multiprocessors*, 3(7), 48–53.

10. Bean, J., Bednar, R., Jones, R., Jones, R., Morris, P., Moss, D., Patterson, M., Prisco, J., Vinson, W., & Wallerich, J. (2009). Proper Sizing of IT Power and Cooling Loads. Green Grid.

11. Behle, B. et al. (2009). *IBM EnergyScale for POWER6 Processor-Based Systems*. IBM.

12. Belady, C. & Malone, C. (2006). Data center power projections to 2014. In *The Tenth Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronics Systems* (pp. 439–444).

13. Bertran, R., Becerra, Y., Carrera, D., Beltran, V., Gonzalez Tallada, M., Martorell, X., Torres, J., & Ayguade, E. (2010). Accurate energy accounting for shared virtualized environments using PMC-based power modeling techniques. In *Proceedings of the 11th IEEE/ACM International Conference on Grid Computing*, GRID (pp. 1–8).

14. Bhattacharjee, A. & Martonosi, M. (2009). Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA (pp. 290–301).: ACM.

15. Bircher, W. L. & John, L. K. (2007). Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS (pp. 158–168).

16. Boneti, C., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Cher, C. Y., & Valero, M. (2008a). Software-Controlled Priority Characterization of POWER5 Processor. In *Proceedings of the 35th International Symposium on Computer Architecture*, ISCA (pp. 415–426).: IEEE Computer Society.

17. Boneti, C., Gioiosa, R., Cazorla, F. J., & Valero, M. (2008b). A Dynamic Scheduler for Balancing HPC Applications. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC (pp. 41:1–41:12).: IEEE Press.

18. Bowhill, B., Stackhouse, B., Nassif, N., Yang, Z., Raghavan, A., Morganti, C., Houghton, C., Krueger, D., Franza, O., Desai, J., Crop, J., Bradley, D., Bostak, C., Bhimji, S., & Becker, M. (2015). 4.5 The Xeon processor E5-2600 v3: A 22nm 18-core product family. In *International Solid-State Circuits Conference*, ISSCC (pp. 1–3).

19. Brooks, D. M., Bose, P., Schuster, S. E., Jacobson, H., Kudva, P. N., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., & Cook, P. W. (2000). Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6), 26–44.

20. Cain, H. W. & Nagpurkar, P. (2010). Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS (pp. 203–212).

21. Callahan, D., Kennedy, K., & Porterfield, A. (1991). Software Prefetching. In *Processing of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS (pp. 40–52).: ACM.

22. Carvalho de Melo, A. (2010). Performance Counters for Linux. http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf.

23. Cazorla, F. J., Knijnenburg, P. M. W., Sakellariou, R., Fernández, E., Ramirez, A., & Valero, M. (2006). Predictable Performance in SMT Processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7), 785–799.

24. Chandrakasan, A. P., Sheng, S., & Brodersen, R. W. (1992). Low-power cmos digital design. *IEICE Transactions on Electronics*, 75(4), 371–382.

25. Charney, M. J. & Puzak, T. R. (1997). Prefetching and memory system behavior of the SPEC95 benchmark suite. *IBM Journal of Research and Development*, 41, 265–286.

26. Choi, S. & Yeung, D. (2006). Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *Proceedings of the 33rd International Symposium on Computer Architecture*, ISCA (pp. 239–251).: IEEE Computer Society.

27. Cooksey, R., Jourdan, S., & Grunwald, D. (2002). A Stateless, Content-directed Data Prefetching Mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS (pp. 279–290).: ACM.

28. Dahlgren, F., Dubois, M., & Stenstrom, P. (1993). Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *Proceedings of the 22nd International Conference on Parallel Processing*, volume 1 (pp. 56–63).

29. Dahlgren, F., Dubois, M., & Stenstrom, P. (1995). Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *Transactions on Parallel and Distributed Systems*, 6(7), 733–746.

30. Denning, P. J. (1968). The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5), 323–333.

31. Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., & Planas, J. (2011). OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21, 173–193.

32. Ebrahimi, E., Lee, C. J., Mutlu, O., & Patt, Y. N. (2011). Prefetch-Aware Shared Resource Management for Multi-Core Systems. In *Proceedings of the 38th International Symposium on Computer Architecture*, ISCA (pp. 141–152).: ACM.

33. Ebrahimi, E., Mutlu, O., Lee, C. J., & Patt, Y. N. (2009a). Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *Proceedings of the 42nd International Symposium on Microarchitecture*, MICRO (pp. 316–326).: ACM.

34. Ebrahimi, E., Mutlu, O., & Patt, Y. N. (2009b). Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, HPCA (pp. 7–17).

35. Emma, P. G., Hartstein, A., Puzak, T. R., & Srinivasan, V. (2005). Exploring the limits of prefetching. *IBM Journal of Research and Development*, 49(1), 127–144.

36. EPA (2007). *EPA Report to Congress on Server and Data Center Energy Efficiency*. Technical report, U.S. Environmental Protection Agency.

37. Eranian, S. (2006). Perfmon2: a flexible performance monitoring interface for linux. In *Proceedings of the 2006 Ottawa Linux Symposium* (pp. 269–288).

38. Floyd, M., Allen-Ware, M., Rajamani, K., Brock, B., Lefurgy, C., Drake, A., Pesantez, L., Gloekler, T., Tierno, J., Bose, P., & Buyuktosunoglu, A. (2011a). Introducing the Adaptive Energy Management Features of the POWER7 Chip. *IEEE Micro*, 31(2), 60–75.

39. Floyd, M., Ware, M., Rajamani, K., Gloekler, T., Brock, B., Bose, P., Buyuktosunoglu, A., Rubio, J., Schubert, B., Spruth, B., Tierno, J., & Pesantez, L. (2011b). Adaptive energy-management features of the IBM POWER7 chip. *IBM Journal of Research and Development*, 55(3), 8:1–8:18.

40. Floyd, M. S., Ghiasi, S., Keller, T. W., Rajamani, K., Rawson, F. L., Rubio, J. C., & Ware, M. S. (2007). System power management support in the IBM POWER6 microprocessor. *IBM Journal of Research and Development*, 51(6).

41. Fu, J. W. C., Patel, J. H., & Janssens, B. L. (1992). Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, MICRO (pp. 102–110).: IEEE Computer Society Press.

42. Gioiosa, R., Petrini, F., Davis, K., & Lebaillif-Delamare, F. (2004). Analysis of system overhead on parallel computers. In *Proceedings of the 4th IEEE International Symposium on Signal Processing and Information Technology* (pp. 387–390).

43. Gioiosa, R., Sancho, J., Jiang, S., & Petrini, F. (2005). Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (pp. 9–).

44. Google (2008). Google App Engine. https://en.wikipedia.org/wiki/Google_App_Engine.

45. Haensch, W., Nowak, E., Dennard, R., Solomon, P., Bryant, A., Dokumaci, O., Kumar, A., Wang, X., Johnson, J., & Fischetti, M. (2006). Silicon CMOS devices beyond scaling. *IBM Journal of Research and Development*, 50(4.5), 339–361.

46. Hamilton, J. (2010). Overall Data Center Costs. http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx.

47. Harrell, F. E. (2001). *Regression Modeling Strategies: With Applications to Linear Models, Logistic Regression, and Survival Analysis*. Graduate Texts in Mathematics. Springer.

48. Henning, J. L. (2006). SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4), 1–17.

49. Horowitz, M., Alon, E., Patil, D., Naffziger, S., Kumar, R., & Bernstein, K. (2005). Scaling, power, and the future of CMOS. In *International Electron Devices Meeting*, IEDM (pp. 7–15).

50. Hur, I. & Lin, C. (2006). Memory Prefetching Using Adaptive Stream Detection. In *Proceedings of the 39th International Symposium on Microarchitecture*, MICRO (pp. 397–408).: IEEE Computer Society.

51. IBM (2014). Bluemix. https://en.wikipedia.org/wiki/Bluemix.

52. Intel (2014). *Intel ®Xeon ®Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual*. Intel. http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-2600-v2-uncore-manual.html.

53. Intel (2016). *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. Intel. http://www.intel.com/products/processor/manuals.

54. Isci, C., Buyuktosunoglu, A., & Martonosi, M. (2005). Long-Term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro*, 25(5), 39–51.

55. Isci, C., Contreras, G., & Martonosi, M. (2006). Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 359–370).

56. Jiménez, V., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., Cazorla, F. J., & Valero, M. (2015). Increasing multicore system efficiency through intelligent bandwidth shifting. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, HPCA (pp. 39–50).: IEEE.

57. Jiménez, V., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., Bose, P., O'Connell, F. P., & Mealey, B. G. (2014). Adaptive prefetching on POWER7: improving performance and power consumption. *ACM Transactions on Parallel Computing, TOPC*, 1(1), 4.

58. Jiménez, V., Cazorla, F. J., Gioiosa, R., Kursun, E., Isci, C., Buyuktosunoglu, A., Bose, P., & Valero, M. (2011a). Energy-Aware Accounting and Billing in Large-Scale Computing Facilities. *IEEE Micro*, 31(3), 60–71.

59. Jiménez, V., Cazorla, F. J., Gioiosa, R., Valero, M., Boneti, C., Kursun, E., Cher, C., Isci, C., Buyuktosunoglu, A., & Bose, P. (2010). Power and thermal characterization of POWER6 system. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT (pp. 7–18).

60. Jiménez, V., Cazorla, F. J., Gioiosa, R., Valero, M., Boneti, C., Kursun, E., Cher, C., Isci, C., Buyuktosunoglu, A., & Bose, P. (2011b). Characterizing power and temperature behavior of power6-based system. *IEEE Journal Emerging and Selected Topics in Circuits and Systems, JETCAS*, 1(3), 228–241.

61. Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., & O'Connell, F. P. (2012). Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT (pp. 137–146).: ACM.

62. Joseph, D. & Grunwald, D. (1997). Prefetching using Markov Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, ISCA (pp. 252–263).: ACM.

63. Jouppi, N. P. (1990). Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, ISCA (pp. 364–373).: ACM.

64. Kadayif, I., Chinoda, T., Kandemir, M., Vijaykirsnan, N., Irwin, M. J., & Sivasubramaniam, A. (2001). vEC: Virtual Energy Counters. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE (pp. 28–31).: ACM.

65. Kansal, A., Zhao, F., Liu, J., Kothari, N., & Bhattacharya, A. A. (2010). Virtual Machine Power Metering and Provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC (pp. 39–50).

66. Karidis, J., Moreira, J. E., & Moreno, J. (2009). True Value: Assessing and Optimizing the Cost of Computing at the Data Center Level. In *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF (pp. 185–192).: ACM.

67. Lange, K.-D. (2009). Identifying Shades of Green: The SPECpower Benchmarks. *Computer*, 42(3), 95–97.

68. Le, H. Q., Starke, W. J., Fields, J. S., O'Connell, F. P., Nguyen, D. Q., Ronchetti, B. J., Sauer, W. M., Schwarz, E. M., & Vaden, M. T. (2007). IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6), 639–662.

69. Lee, C. J., Mutlu, O., Narasiman, V., & Patt, Y. N. (2008). Prefetch-Aware DRAM Controllers. In *Proceedings of the 41st International Symposium on Microarchitecture*, MICRO (pp. 200–209).: IEEE Computer Society.

70. Lefurgy, C., Wang, X., & Ware, M. (2007). Server-Level Power Control. In *Proceedings of the 4th International Conference on Autonomic Computing*, ICAC (pp. 4–14).: IEEE Computer Society.

71. Li, Y., Skadron, K., Brooks, D., & Hu, Z. (2005). Performance, energy, and thermal considerations for SMT and CMP architectures. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture* (pp. 71–82).

72. Liao, S. W., Hung, T.-H., Nguyen, D., Chou, C., Tu, C., & Zhou, H. (2009). Machine Learning-Based Prefetch Optimization for Data Center Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC (pp. 1–10).: ACM.

73. Lin, W.-F., Reinhardt, S. K., Burger, D., & Puzak, T. R. (2001). Filtering Superfluous Prefetches Using Density Vectors. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, ICCD (pp. 124–132).: IEEE Computer Society.

74. Liu, F. & Solihin, Y. (2011). Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS (pp. 37–48).: ACM.

75. Liu, Q., Jiménez, V., Moretó, M., Abella, J., Cazorla, F. J., & Valero, M. (2014a). Per-task energy accounting in computing systems. *Computer Architecture Letters*, 13(2), 85–88.

76. Liu, Q., Moreto, M., Abella, J., Cazorla, F. J., & Valero, M. (2014b). DReAM: Per-task DRAM energy metering in multicore systems. In *Proceedings of the 20th International European Conference on Parallel and Distributed Computing*, Euro-Par (pp. 111–123).: Springer.

77. Liu, Q., Moretó, M., Jiménez, V., Abella, J., Cazorla, F. J., & Valero, M. (2013). Hardware support for accurate per-task energy metering in multicore systems. *ACM Transactions on Architecture and Code Optimization, TACO*, 10(4), 34.

78. Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., & Kozyrakis, C. (2015). Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA (pp. 450–462).: ACM.

79. Lu, J., Chen, H., Fu, R., Hsu, W.-C., Othmer, B., Yew, P.-C., & Chen, D.-Y. (2003). The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO (pp. 180–190).: IEEE Computer Society.

80. Luque, C., Moreto, M., Cazorla, F., Gioiosa, R., Buyuktosunoglu, A., & Valero, M. (2009a). Itca: Inter-task conflict-aware cpu accounting for cmps. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT (pp. 203–213).

81. Luque, C., Moreto, M., Cazorla, F. J., Gioiosa, R., Buyuktosunoglu, A., & Valero, M. (2009b). CPU Accounting in CMP Processors. *IEEE Computer Architecture Letters*, 8(1), 17–20.

82. Martin, M., Sorin, D., Hill, M., & Wood, D. (2002). Bandwidth adaptive snooping. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA (pp. 251–262).

83. Meisner, D., Gold, B. T., & Wenisch, T. F. (2009). PowerNap: Eliminating Server Idle Power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS (pp. 205–216).: ACM.

84. Merten, M., Trick, A., George, C., Gyllenhaal, J., & Hwu, W.-M. (1999). A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA (pp. 136–148).

85. Mochel, P. (2005). The sysfs Filesystem. *Proceedings of the Annual Linux Symposium*.

86. Moreira, J. E. & Karidis, J. P. (2010). The Case for Full-Throttle Computing: An Alternative Datacenter Design Strategy. *IEEE Micro*, 30(4), 25–28.

87. Moreto, M., Cazorla, F. J., Ramirez, A., Sakellariou, R., & Valero, M. (2009). FlexDCP: a QoS Framework for CMP Architectures. *SIGOPS Operating Systems Review*, 43(2), 86–96.

88. Mowry, T. C., Lam, M. S., & Gupta, A. (1992). Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS (pp. 62–73).: ACM.

89. Murphy, R. C., Wheeler, K. B., Barrett, B. W., & Ang, J. A. (2010). Introducing the Graph 500. *Craig User's Group, CUG.*

90. Mutlu, O., Kim, H., Armstrong, D. N., & Patt, Y. N. (2005). Using the First-level Caches As Filters to Reduce the Pollution Caused by Speculative Memory References. *International Journal of Parallel Programming*, 33(5), 529–559.

91. Nagarajan, V. & Gupta, R. (2009). ECMon: Exposing Cache Events for Monitoring. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA (pp. 349–360).: ACM.

92. Nathuji, R. & Schwan, K. (2007). VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. *SIGOPS Operating Systems Review*, 41(6), 265–278.

93. Naveh, A., Rotem, E., Mendelson, A., Gochman, S., Chabukswar, R., Krishnan, K., & Kumar, A. (2006). Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*, 10(2).

94. Nesbit, K. J., Dhodapkar, A. S., & Smith, J. E. (2004). AC/DC: An Adaptive Data Cache Prefetcher. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT (pp. 135–145).

95. Palacharla, S. & Kessler, R. E. (1994). Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st International Symposium on Computer Architecture*, ISCA (pp. 24–33).: IEEE Computer Society Press.

96. Pallipadi, V. (2007). Cpuidle - Do nothing, efficiently... *Linux Symposium*.

97. Pallipadi, V. & Starikovskiy, A. (2006). The Ondemand Governor. Past, Present, and Future. *Linux Symposium*.

98. Petrica, P., Izraelevitz, A. M., Albonesi, D. H., & Shoemaker, C. A. (2013). Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA (pp. 13–23).: ACM.

99. POWER ISA (2013). *Power ISA™ Version 2.07*. IBM. https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf.

100. Prat, D., Ortega, C., Casas, M., Moreto, M., & Valero, M. (2015). Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7. In *Proceedings of the 6th International Workshop on Adaptive Self-tuning Computing Systems*, ADAPT (pp. 1–6).

101. Qureshi, M. K. & Patt, Y. N. (2006). Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, MICRO (pp. 423–432).: IEEE Computer Society.

102. Radojković, P., Čakarević, V., Moretó, M., Verdú, J., Pajuelo, A., Cazorla, F. J., Nemirovsky, M., & Valero, M. (2012). Optimal Task Assignment in Multithreaded Processors: A Statistical Approach. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS (pp. 235–248).: ACM.

103. Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., & Zhu, X. (2008). No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. *SIGOPS Operating Systems Review*, 42(2), 48–59.

104. Rogers, B. M., Krishna, A., Bell, G. B., Vu, K., Jiang, X., & Solihin, Y. (2009). Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. In *Proceedings of the 36th International Symposium on Computer Architecture*, ISCA (pp. 371–382).: ACM.

105. Roth, A., Moshovos, A., & Sohi, G. S. (1998). Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS (pp. 115–126).: ACM.

106. Sarikaya, R., Isci, C., & Buyuktosunoglu, A. (2010). Runtime workload behavior prediction using statistical metric modeling with application to dynamic power management. In *Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, IISWC (pp. 1–10).

107. Sarikaya, R., Isci, C., & Buyuktosunoglu, A. (2013). Runtime application behavior prediction using a statistical metric model. *IEEE Transactions on Computers*, 62(3), 575–588.

108. Schneider, F. T., Payer, M., & Gross, T. R. (2007). Online Optimizations Driven by Hardware Performance Monitoring. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI (pp. 373–382).: ACM.

109. Shalf, J., Quinlan, D., & Janssen, C. (2011). Rethinking hardware-software codesign for exascale systems. *Computer*, (11), 22–30.

110. Siddha, S., Pallipadi, V., & Ven, A. V. D. (2007). Getting maximum mileage out of tickless. *Linux Symposium*.

111. Sinharoy, B., Kalla, R., Starke, W. J., Le, H. Q., Cargnoni, R., Van Norstrand, J. A., Ronchetti, B. J., Stuecheli, J., Leenstra, J., Guthrie, G. L., Nguyen, D. Q., Blaner, B., Marino, C. F., Retter, E., & Williams, P. (2011). IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3), 1–29.

112. Sinharoy, B., Van Norstrand, J., Eickemeyer, R., Le, H., Leenstra, J., Nguyen, D., Konigsburg, B., Ward, K., Brown, M., Moreira, J., Levitan, D., Tung, S., Hrusecky, D., Bishop, J., Gschwind, M., Boersma, M., Kroener, M., Kaltenbach, M., Karkhanis, T., & Fernsler, K. (2015). IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1), 1–21.

113. Smith, A. J. (1978). Sequential Program Prefetching in Memory Hierarchies. *IEEE Computer*, 11(12), 7–21.

114. Solihin, Y., Lee, J., & Torrellas, J. (2002). Using a User-Level Memory Thread for Correlation Prefetching. In *Proceedings of the 29th International Symposium on Computer Architecture*, ISCA (pp. 171–182).: ACM.

115. SPECjbb2005 (2005). Standard Performance Evaluation Corporation. http://www.spec.org/jbb2005/.

116. Srinath, S., Mutlu, O., Kim, H., & Patt, Y. N. (2007). Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, HPCA (pp. 63–74).: IEEE Computer Society.

117. Srinivas, M., Sinharoy, B., Eickemeyer, R., Raghavan, R., Kunkel, S., Chen, T., Maron, W., Flemming, D., Blanchard, A., Seshadri, P., Kellington, J., Mericas, A., Petruski, A., Indukuru, V., & Reyes, S. (2011). Ibm power7 performance modeling, verification, and evaluation. *IBM Journal of Research and Development*, 55(3), 4:1–4:19.

118. Srinivasan, V., Davidson, E. S., & Tyson, G. S. (2004). A prefetch taxonomy. *IEEE Transactions on Computers*, 53(2), 126–140.

119. Srinivasan, V., Shenoy, G. R., Vaddagiri, S., Sarma, D., & Pallipadi, V. (2008). Energy-Aware Task and Interrupt Management in Linux. *Linux Symposium*, 2.

120. Suh, G., Devadas, S., & Rudolph, L. (2002). A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA (pp. 117–128).

121. Suh, G. E., Rudolph, L., & Devadas, S. (2004). Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing*, 28(1), 7–26.

122. Tang, L., Mars, J., Vachharajani, N., Hundt, R., & Soffa, M. L. (2011). The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA (pp. 283–294).: ACM.

123. Tikir, M. M. & Hollingsworth, J. K. (2004). Using Hardware Counters to Automatically Improve Memory Performance. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC (pp. 46–).: IEEE Computer Society.

124. Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., & Stamm, R. L. (1996). Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA (pp. 191–202).: ACM.

125. Vera, J., Cazorla, F. J., Pajuelo, A., Santana, O. J., Fernández, E., & Valero, M. (2007). FAME: FAirly MEasuring Multithreaded Architectures. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, PACT (pp. 305–316).

126. Waldspurger, C. A. (2002). Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36, 181–194.

127. Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., & Weems, C. C. (2003). Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, ISCA (pp. 388–398).: ACM.

128. Wu, C. J. & Martonosi, M. (2011). Characterization and Dynamic Mitigation of Intra-Application Cache Interference. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS (pp. 2–11).: IEEE Computer Society.

129. Wulf, W. A. & McKee, S. A. (1995). Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23, 20–24.

130. Yang, C. L. & Lebeck, A. R. (2000). Push vs. Pull: Data Movement for Linked Data Structures. In *Proceedings of the 14th International Conference on Supercomputing*, ICS (pp. 176–186).: ACM.

131. Yasin, A. (2014). A top-down method for performance analysis and counters architecture. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS (pp. 35–44).

132. Zagha, M., Larson, B., Turner, S., & Itzkowitz, M. (1996). Performance analysis using the mips r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, SC (pp. 16–16).: IEEE.

133. Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., & Kumar, S. (2004). Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS (pp. 177–188).: ACM.

134. Zhuang, X. & Lee, H.-H. S. (2003). A hardware-based Cache Pollution Filtering Mechanism for Aggressive Prefetches. In *Proceedings of the 32nd International Conference on Parallel Processing* (pp. 286–293).

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. A template that can be used to format a PhD thesis with this look and feel has been released under the permissive MIT (XII) license, and can be found online at github.com/suchow/Dissertate or from its author, Jordan Suchow, at suchow@post.harvard.edu.