A Model for Thread and Memory Placement on NUMA Systems

by

Justin Funston

B.Sc. Computer Science, Gonzaga University, 2010

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

January 2018

Abstract

The problem of placement of threads, or virtual cores, on physical cores in a multicore system has been studied for over a decade. Despite this effort, we still do not know how to assign virtual to physical cores on a non-uniform memory access (NUMA) system so as to meet a performance target while minimizing resource consumption. Prior work has made large strides in this area, but these solutions either addressed hardware with specific properties, leaving us unable to generalize the models to other systems, or modeled much simpler effects than the actual performance in different placements.

An interdependent problem is how to place memory on NUMA systems. Poor memory placement causes congestion on interconnect links, contention for memory controllers, and ultimately long memory access times and poor performance. Commonly used operating system techniques for NUMA memory placement fail to achieve optimal performance in many cases.

Our contribution is a general framework for reasoning about workload placement and memory placement on machines with shared resources. This framework enables us to automatically build an accurate performance model for any machine with a hierarchy of known shared resources. Using our methodology, data center operators can minimize the number of NUMA (CPU+memory) nodes allocated for an application or a service, while ensuring that it meets performance objectives. More broadly, the methodology empowers them to efficiently "pack" virtual containers on the physical hardware. We also present an effective solution for placing memory that avoids congestion on interconnects due to memory traffic and additionally selects the best page size that balances translation lookaside buffer (TLB) effects against more granular memory placement. The solutions proposed can significantly improve performance and work at the operating system level so they do not require changes to applications.

Lay Summary

Modern server-class computer hardware is becoming increasingly complex as hardware designers scale core counts. This hardware runs important applications like scientific simulations, databases, and machine learning, and represents a significant portion of the worlds electricity usage. Software must be carefully designed and optimized to get the most out of such hardware, otherwise performance can suffer and energy is wasted. This work presents insights and analysis into how software interacts with modern server-class hardware, and proposes techniques and algorithms to automatically optimize software for it. The solutions proposed can significantly improve performance and work at the operating system level so they do not require changes to applications.

Preface

The research chapters of this dissertation (Chapters 2–5) span multiple related projects done in collaboration, all of which are previously published or currently under peer-review. Chapter 2 is the culmination of the projects and represents the bulk of my personal contributions.

At the time of this writing, the work in Chapter 2 is under peer-review as:

- Justin Funston, Maxime Lorrillere, David Vengerov, Baptiste Lepers Jean-Pierre Lozi, Vivien Quema, and Alexandra Fedorova. A Practical Model for Placement of Workloads on Multicore NUMA Systems. Submitted to *the 13th European Conference on Computer Systems*.

I was the lead investigator responsible for research direction, experiment design, data analysis, and solution design. Maxime Lorrillere conducted the experiments in Section 2.5 and Section A.1, implemented the fast memory migration mechanism in Section 2.5, and wrote Section 2.5 of the manuscript. I wrote the majority of the rest of the chapter's manuscript and conducted all other experiments in the chapter. Other co-authors provided technical and editorial advice.

Chapter 3 is a modified version of previously published work. It is ©2012 IEEE, reprinted with permission from:

- Justin R. Funston, Kaoutar El Maghraoui, Joefon Jann, Pratap Pattnaik, and Alexandra Fedorova. 2012. An SMT-Selection Metric to Improve Multi-threaded Applications Performance. In *Proceedings of the 26th International Parallel & Distributed Processing Symposium (IPDPS 12)*. IEEE, Washington, DC, USA, 1388–1399. https://doi.org/10.1109/IPDPS.2012.125

I was the lead investigator responsible for research direction, experiment design, data analysis, and solution design, and I wrote the majority of the manuscript. Co-authors provided technical and editorial advice.

Chapter 4 is a modified version of previously published work. It is ©2013 ACM, reprinted with permission from:

- Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 13)*. ACM, New York, NY, USA, 381–394.
https://doi.org/10.1145/2451116.2451157

Mohammad Dashti, Fabien Gaud, and I jointly conducted the initial investigation into NUMA effects, including the discovery of the importance of congestion over locality (reported in Section 4.1 and Section 4.2). I designed and implemented the page-level replication mechanism described in Section 4.3.3 with debugging help from Fabien Gaud. I wrote the sections of the manuscript relevant to my contributions. Fabien Gaud designed and implemented the Carrefour algorithm. All co-authors including myself provided technical and editorial advice.

Chapter 5 is a modified version of previously published work:

- Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC 14)*. USENIX Association, Berkeley, CA, USA, 231–242.

I conducted the initial investigation into NUMA and large pages, including the discovery and analysis of the hot page and page-level false sharing problems (reported in Section 5.2 and Section 5.3.1), and wrote the sections of the manuscript relevant to my contributions. Fabien Gaud and Baptiste Lepers designed and implemented the Carrefour-LP algorithm. All co-authors including myself provided technical and editorial advice.

Table of Contents

List of Tables

List of Figures

Glossary

CMP	Chip multiprocessing
HPE	Hardware performance event, a.k.a. hardware performance counter
IBS	Instruction-based sampling
IC	Interconnect
IPC	Instructions per cycle
ML	Machine learning
NUMA	Non-uniform memory access
SMP	Symmetric multiprocessing
SMT	Simultaneous multithreading
THP	Transparent huge pages
TLB	Translation lookaside buffer

Acknowledgments

I give my warmest thanks to my advisor Dr. Fedorova for her peerless guidance and for helping me achieve my potential as a researcher. I would also like to thank all of my collaborators and co-authors for their essential hard work and insight.

I would like to thank my family for all of their love and support.

Lastly I would like to thank IBM for the internship that led to the research in Chapter 3, and Oracle Labs and the British Columbia Innovation Council for funding the work of Chapter 4.

Chapter 1

Introduction

Data centers use 2% of the electricity consumed in the United States [85] and 3% of the world's [17] electricity. These data centers run important applications such as scientific simulations, data analytics, web servers, and databases, and the applications often have strict performance requirements. In order to keep power consumption under control hardware designers have introduced systems with continually increasing core counts and power control features to reduce energy usage when parts of the system are idle. Using a smaller number of larger systems (i.e. systems with more cores) can be 25%–33% more power efficient than using twice as many systems of half the size (based on our own experiments, vendor specifications [3], and previous studies [13]).

Increasing core counts, though, come at the cost of increased hardware complexity and architectural trade-offs made for scalability. Applications and system software must be designed and optimized with the hardware architecture in mind, otherwise performance can suffer. This dissertation focuses on how system software can optimize the performance of applications running on the large server-class systems typically used in data centers, and on how data center operators can make reliable performance trade-offs for better server utilization or reduced power consumption. The first step in doing so is choosing the level of parallelism an application should use (in other words, how many cores it runs on). Due to scalability limits or load patterns, it is not always beneficial to give all the cores of system to a single application. Additionally, in cloud environments it is common for customers to pay for a set number of cores upfront, so the problem of determining how many cores to use is moot in that case. A comprehensive solution on how to determine the level of parallelism for an application is beyond the scope of this dissertation, but the primary contribution of Chapter 3 is a solution for it applicable to simultaneous multithreading (SMT) hardware.

The second step after the number of cores has been determined is to choose *which* cores to use. The choice of cores to use, which we call a *placement*, affects what hardware resources are available to the application which in turn affects performance and power usage. Section 1.1 gives an overview of the relevant hardware resources and how they affect applications. The key complicating factor for selecting a placement is that different applications have different needs for hardware resources. The primary contribution of Chapter 2 is a practical model for workload placement. Our model does not require modifying applications, can be easily adapted to various architectures, and can be used to predict the performance of different placements. Performance prediction, as opposed to simply finding the best performing placement, is crucial because it allows trade-offs to be made. Most of the extensive previous research in the area does not attempt to predict performance, only handles a single type of hardware resource (for example, only considering beneficial cache sharing but not taking into account SMT sharing of functional units or other hardware resource sharing), requires expert knowledge and manual tweaking to adapt to a new architecture (usually because one would need to write carefully designed micro-benchmarks for new hardware resources), or is not deployable online because new applications would require several offline profiling runs. Table 1.1 provides an overview of significant related work and their respective short-comings compared to our solution described in Chapter 2. A detailed description of related work is provided in Section 2.2.

Predicting the performance of workload placements allows for one especially important trade-off. For many applications, using fewer hardware resources will only have a minimal or moderate effect on performance. If data center operators or cloud server providers can predict these cases and the performance impact is within acceptable tolerances, then they can use remaining hardware resources to pack additional workloads or applications onto the same server. This increases utilization and efficiency and ultimately reduces energy usage. Section 2.5 evaluates specific scenarios where our workload placement solution can increase server utilization while maintaining performance goals.

The final step after workload placement is memory placement. Large multicore systems often have a non-uniform memory access (NUMA) architecture (described in more detail in Section 1.1), which means that memory placement can also affect performance. Better memory placement will reduce contention for hardware resources which in turn reduces memory access times. Chapters 4 and 5 describe our solution to the memory placement problem, which provides significant performance benefits over standard techniques.

	Predicts Performance	**Multiple Hardware Resources**	**Easily Adapted**	**Deployable Online**
Chapter 2's Solution	Y	Y	Y	Y
Pandia [49]	Y	Y	N	N
SMiTe [105]	Y	Y	N	Y
Bubble-Flux [101]	Y	Y	N	Y
Asymsched [59]	N	N	Y	Y
DINO [107]	N	N	Y	Y
Thread Clustering [93]	N	N	Y	Y

Table 1.1: Summary of related work and their capabilities

1.1 Background

Modern server-class systems have a complex hierarchy of shared resources, which is a necessity to scale them to high core counts. Figure 1.1 shows an example NUMA system. At the lowest level of the hierarchy is a hardware context (also known as a hardware thread or, confusingly, as a core on some architectures). The hardware context contains everything required to track the execution state of a software thread such as the program counter and registers. When a software thread is assigned to a hardware context it can execute on the CPU.

At the next level of the hierarchy is the physical core. A physical core en-

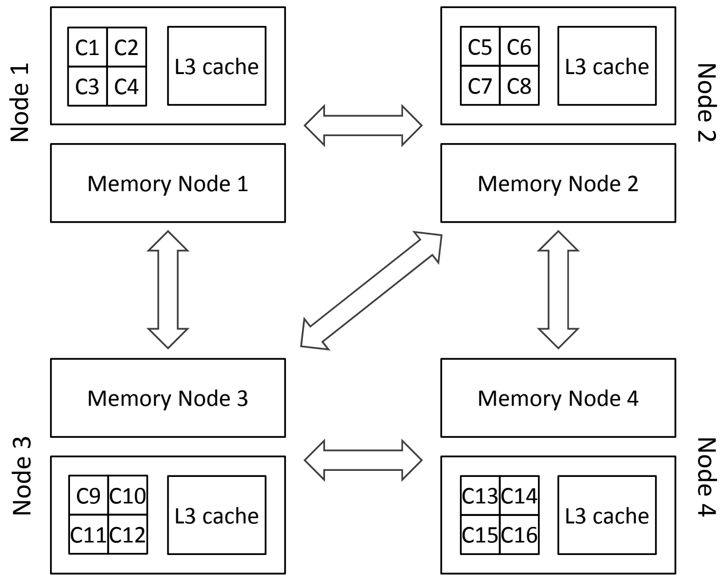


Figure 1.1: A modern NUMA system, with four nodes and four cores per node. ©2013 ACM, reprinted with permission from [36].

compasses data and instruction caches, functional units (the ALU, branch units, load/store units, etc.), and the instruction decode and dispatch units. Simultaneous multithreading (SMT) is a feature that places multiple hardware contexts onto a single physical core. The physical core's resources are shared between hardware contexts, although on some architectures some of the resources may be duplicated for each context (for example, each hardware context might get its own L1 cache but have to share the L2 cache with other contexts). In general, SMT can improve the overall throughput of the system but each software thread has lower performance than if it is able to execute alone on the physical core because of the competition for core resources. If the application heavily shares data between threads and benefits from very low latency communication between threads then SMT can actually improve individual threads' performance (the kmeans benchmark on our AMD test system, shown in Figure A.4, is one such instance).

At the highest level of the hierarchy is the non-uniform memory access (NUMA) node. A NUMA node contains a set of cores and an associated set of locally connected memory. Programs can access memory on any NUMA node from any core transparently without special consideration. The operating system decides which NUMA node to allocate memory on. Memory accesses to the local memory have a lower latency. NUMA nodes are connected by interconnect (IC) links, so memory accesses to remote memory travel over the IC links and have a higher latency. Cores on the same NUMA node typically share an L3 cache, the memory controller, and the IC links. So, if all the cores on a NUMA node are performing very memory-intensive computations then the bottleneck will likely be the shared memory subsystem and performance will suffer.

If software does not take into account all of these shared resources then contention for the resources results in poor performance, sometimes resulting in slowdowns of $2\times$ or more! The following chapters analyze the performance effects in detail and propose practical and effective solutions.

Chapter 2

A Model for Placement of Workloads on Multicore NUMA Systems[1]

Data center operators balance two objectives: providing a satisfactory experience for the customer and efficiently allocating the hardware that runs customer virtual instances or operator services. We believe that hardware provisioning should not be a guessing game. No one should have to allocate more hardware than needed just out of the fear that providing less could violate a service-level objective.

Unfortunately, mapping threads, or virtual CPUs (vCPUs), to physical cores on multicore NUMA systems is still like playing a guessing game. Modern NUMA systems consist of several nodes that share various resources. Cores within a node may share SMT pipelines and caches. Nodes themselves share an interconnect, which may be asymmetric, providing higher bandwidth for some links than for others and for some directions of communication than for others (e.g., Figure 2.2).

The challenge of placing workloads on a NUMA system has to do with complex interactions between the workload and the hardware. Our solution, presented in this chapter, consists of two main contributions. First, is an **abstract machine model** (Section 2.3) that abstracts hardware resources and provides a practical way

[1]This chapter is an expanded version of work currently under peer-review as [45]

to represent placements. Second, we build on the abstract machine model to build a **performance prediction model** (Section 2.4) that predicts the performance of important placements.

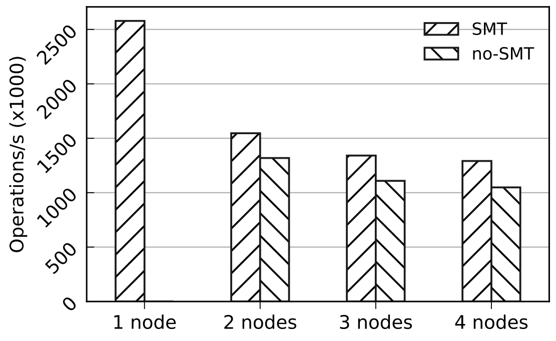
Attribution: Maxime Lorrillere conducted the experiments in Section 2.5 and Section A.1, implemented the fast memory migration mechanism in Section 2.5, and wrote Section 2.5 of the manuscript. I wrote the majority of the rest of the chapter's manuscript, was responsible for all areas of research, and conducted all other experiments in the chapter.

2.1 Introduction & Motivation

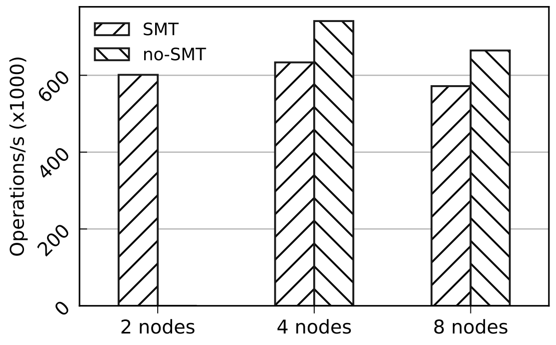
Consider Figure 2.1a, which shows performance of MongoDB's WiredTiger key-value store [4] on a NUMA Intel machine. This workload, which runs a BTree search using 24 threads, runs almost twice as fast when all of the threads are placed on a single node, as opposed to being spread across two or more nodes. On Figure 2.1b, on the other hand, we see that the same workload configured with 16 threads on an AMD NUMA system runs much faster when it has four nodes at its disposal, rather than two[2]. On the Intel system, this application prefers having all of its threads co-located on a single NUMA node and using SMT because it enjoys lower communication latencies. On the AMD machine, on the other hand, it suffers from contention for shared resources when all of its threads are crowded on a small number of nodes. How can a data center operator know, save for trying all possible placements, that to achieve the best possible performance this container should be placed on a single node on an Intel system but on four nodes (without SMT) on an AMD system? This is an example of questions that we aim to answer in this work.

The observations detailed in the previous paragraph are not new and were studied in our community for more than a decade [21–23, 36, 41, 46, 48, 49, 56, 57, 59, 69, 87, 101, 105, 107, 108]. Yet, as we elaborate in Section 2.2, we still do not have a method for deciding how a particular *placement*, a mapping of vCPUs to physical cores, on an arbitrary multicore system will affect performance of an

[2]With eight cores/node we could not fit all threads on a single node without creating contention for CPU cycles.



(a) WiredTiger, Intel



(b) WiredTiger, AMD

Figure 2.1: Throughput of the WiredTiger key-value store on two NUMA systems according to the number of nodes.

unknown workload.

We propose ***a general framework for reasoning about workload placement on machines with shared resources***. Our methodology is not tied to a particular machine, specific shared resources or certain hardware performance events. We abstract a multicore machine as a collection of shared resources, called *scheduling concerns*, where each concern produces a *score* indicating how many threads use the resource in a given placement. Vectors of scores uniquely identify *distinct placements*: placements that differ with respect to resource usage. Our abstraction relies on important simplifications (see Section 2.2.2) that enable us to dramatically reduce the number of distinct placements from billions of all possible to dozens, and makes the problem tractable.

We develop a methodology to automatically build a model for predicting performance in all distinct Pareto-efficient placements, given a specification of a machine's scheduling concerns and observations of the workload at runtime. Our model predicts performance to within 5% of actual on average. With the resulting model users can make decisions such as "give application X as few NUMA nodes as possible while making sure that its throughput remains above Y operations per second". Our framework is flexible: a previously unmodeled shared resource can be added to the model as a new scheduling concern and the model is updated automatically. *We do not have to manually redesign the model for every new machine.*

Many performance models similar to ours used hardware performance events (HPE), observed on a *single* placement as model features [12, 46, 56, 59, 101, 107]. ***Our results show that single-placements HPEs cannot produce a reliable performance model on complex modern systems.*** Our solution uses actual *performance observations* on two different placements as model input features and does not require selecting hardware performance events that *may* correlate with performance[3].

Our method requires running a workload for a short period of time in two different placements before it can generate performance predictions, so it can be used online. Nevertheless, migrating the workload from one placement to another can be costly, because we may need to migrate memory between NUMA nodes. To evaluate this effect, ***we analyze and improve on migration overheads in Linux.*** Our results should help potential users decide when our method is viable for an online deployment.

2.2 Background & Related Work

2.2.1 State of the Art

Workload placement on multicore systems has been explored for over a decade. Early studies examined contention between single-threaded applications for a specific resource, such as the SMT instruction pipeline [46, 87], or shared caches and memory controllers [43, 69, 101, 107]. Later work extended the techniques to

[3]HPEs, such as instructions/cycle may be used for performance measurements, but any other application-specific metric is acceptable.

multithreaded workloads and to additional resource combinations, such as SMT and shared caches [105], memory controllers and the shared interconnect [36, 59]. While laying a crucial foundation for our work, these prior techniques did not provide a general solution for reasoning about such systems.

For instance, the DIO algorithm [107] showed us how to avoid interference for shared memory controllers. A simplified version of the algorithm is as follows: monitor the memory-intensiveness of running threads using HPEs for a sampling period, sort threads by memory-intensiveness, and then place threads on domains sharing memory resources such that the aggregate memory-intensiveness is balanced on domains. This algorithmic approach returns a new thread placement it considers to be optimal. In the absence of unaccounted for resources affecting the performance of the thread placement, the algorithm improves performance. Note that it does not try to predict how much performance will improve, and only handles one set of shared resources.

The AsymSched algorithm of Lepers et al. [59] showed us how to place applications on machines with asymmetric interconnects, by using knowledge of the topology and the application's traffic patterns to place threads on nodes with good interconnect connectivity.

Other previous scheduling algorithms pursued a goal of placing threads that share data near each other (e.g. sharing a cache or on the same NUMA node). Tam et al. [93] provided a scheduling algorithm aimed at maximizing the benefit of thread sharing. It is similar in approach to DIO. Thread sharing is measured online and then the algorithm enacts a new thread placement that optimizes the overall sharing on the system. As with DIO, it does not attempt to predict performance.

Techniques used in prior work did not allow for automatic combination of several models. Every contention model required manual design: careful selection of hardware performance events [36, 43, 56, 59, 107] or even manual crafting of artificial "probe" workloads or "Rulers" [101, 105] that must be run side-by-side with the target workloads to determine their sensitivity to contention.

Most existing scheduling algorithms have a common structure. As inputs, they take knowledge of the hardware and some online metrics from the application (usually obtained from HPEs). Then as an output they suggest a thread placement that should provide optimum performance with respect to the resources they individually account for.

A comprehensive solution should account for all the hardware resources that matter to thread placement, but combining existing algorithms is difficult. There is no baseline for comparing the importance of one algorithm over another. If the algorithms suggest conflicting thread placements, then it is necessary to balance the performance effect of each algorithm against the other, but as discussed the algorithms can at best only say whether a given placement is better or worse than another. They cannot predict the performance effect so the scheduler has no way to know which algorithm should be followed or if a compromise is best.

One could try to develop a new algorithm that incorporates the ideas of the original algorithms. It would require extensive manual testing and tuning to determine the best way to balance the concerns, would likely be sensitive to differing workloads and hardware, and would require expert knowledge that covers both of the original algorithms. All of these are serious downsides to the approach.

The state of Linux scheduler exemplifies these difficulties. The Linux scheduler attempts to address many thread placement concerns, but it does so in an adhoc way with many heuristics and hacks. In one of our previous works we discovered four major bugs in the Linux scheduler, which resulted in a 22% to 138x performance impact in extreme cases [61].

System identification is a well-studied area in the field of control systems [30, 31, 60]. The goal of system identification is to model a system's outputs given some number of inputs. For our purposes, the inputs would be the workload placement and the dynamic application behavior and the output we want to model is the application's performance. The main difficulty is that system identification generally assumes that the inputs are easily measurable and real-valued, for example, an input voltage. This does not apply in our case of workload placement. Deciding exactly what to use as inputs to our model and how to quantify them is an important and difficult problem on its own, and is discussed in the following sections. For this reason we approached the problem with the field of machine learning in mind, rather than system identification.

Dwyer et al. used an automated model-building methodology, where automatically selected features (from all HPEs available on the machine) were fed into a variety of machine-learning models [41]. However, the model predicted a rather simple outcome: a performance degradation when a target workload was co-scheduled with an interfering one, and not the performance in different placements. Consistent with our finding that HPEs observed in a single placement are poor model features, Dwyer's study reported rather poor prediction accuracy in many cases.

A recent system, Pandia [49], made significant advances. It accurately predicts relative performance of different workload placements on multicore NUMA machines. It can also predict relative performance of an application with different numbers of threads, but such predictions in Pandia require performance observations of six runs with different thread counts, which is difficult to do online because most real applications cannot easily reconfigure their thread count on demand. Despite addressing many limitations of previous work and producing remarkably accurate performance predictions, fundamentally Pandia still relies on the machine-specific modelling methodology that prevents easily transferring results to other systems. Pandia's authors capture factors that contribute to performance, such as cache contention, latency of communication, and load balancing, in a set of machine-specific equations. If the model had to be adapted to another machine, for example one with an asymmetric interconnect, the equations would have to be manually reformulated.

We believe that investing that much effort into designing new models for every new type of hardware puts an unreasonable burden on system engineers. Essentially, while there is significant existing theory for scheduling with respect to time-sharing [86, 100], there is not any developed theory or framework for space-sharing. Instead, we sought a future-proof methodology that uses easily available information about a machine's shared resources and automatically builds an accurate performance model.

2.2.2 Assumptions and Limitations

To make our methodology robust and extensible we make a few simplifying assumptions, which we describe here.

Identically scored placements yield identical performance. As we explain in Section 2.3, a placement is identified by how many vCPUs share each hardware resource. We refer to the degree of sharing for a resource as the *score*; a vector of scores thus identifies a placement. Placements with identical score vectors are deemed to yield identical performance for a given workload. This statement assumes that our machine model must be aware of all shared resources that might yield variations in performance. This assumption is made by most solutions in this space, because one can only model the factors of which one is aware. A radically different approach would be a statistical technique that searches for an optimally performing placement by trying a sufficient number of random placements [76]. Unfortunately, the best known techniques require trying *thousands* of placements and assumes that performance in all placements fits a Generalized Pareto distribution — an assumption that does not hold in our case.

A workload is encapsulated in a virtual container. This assumption sits well with many data centers that use virtualization for a variety of reasons. In our case, it makes the problem easier to solve than if we viewed a workload as an amorphous collection of threads. With a thread-centric view, the degree of concurrency can unpredictably change, either because of application logic or because of OS scheduling decisions. When it changes, performance can be affected because of intra-application scaling issues or because of the change in pressure on shared resources. Combining both effects in a single model is cumbersome: we concluded that for robustly accurate predictions that do not require offline runs we need to train a separate model for each feasible number of vCPUs in a container. Managed cloud environments present their offerings as a menu of virtual instances with a fixed number of vCPUs per instance. For example, AWS offers a dozen instances with the number of different core counts limited to ten [5]. So we can feasibly train a separate model for each machine and each vCPU count. We are *not* interested in finding the optimal number of threads or vCPUs for the workload; for that, users can leverage other tools [49, 90].

We also assume that each vCPU of the container is performing roughly the same type of work. This assumption is not very strict though; we have workloads in our test set that have database maintenance threads or garbage collection threads in addition to worker threads and these workloads still work well with our model.

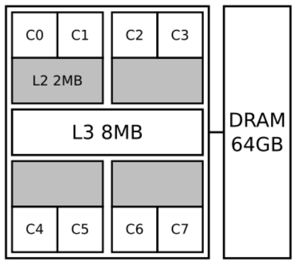
A NUMA node is a unit of resource allocation. We assume that vCPUs are allocated to containers in units of entire NUMA nodes. That is, given a virtual

Concern	**Score**	**Resources**	**Cost?**	**Inverse Perf?**
L2/SMT	Number of L2 caches in use	L2 cache, instruction fetch and decode, and floating point units	Y	Y
L3	Number of L3 caches in use	L3 cache, memory controller, and bandwidth to DRAM	Y	Y
Interconnect	Aggregate bandwidth between nodes in use	Interconnect bandwidth	N	N

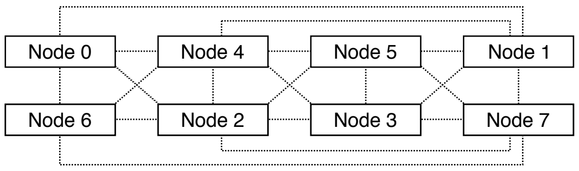
Table 2.1: Scheduling concerns used on our AMD test system (shown in Figure 2.2).

container the goal is to decide across how many NUMA nodes to spread the container; we do not co-locate different containers on the same node. We rely on this observation to make the problem tractable. Modelling contention among different containers on the same NUMA node is much more difficult. On the other hand, if multiple containers on the same machine are mapped to different NUMA nodes, they can co-exist without interference if the nodes used for different containers do not share interconnect links, which is a configuration easy to enforce. We confirmed that this property holds experimentally and the results are shown in A.1.

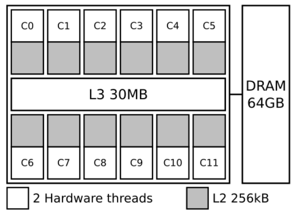
We consider only balanced placements. A balanced placement is one where the number of vCPUs is evenly divisible by any number of shared resource units considered for placement. For instance, if we have shared L3 caches on the system, we will only consider placements where the number of vCPUs sharing each L3 cache is equal. Uneven sharing can cause unpredictable performance effects on the workload, for example by creating stragglers, so we choose to not model these effects. Since we are already assuming that resources will be allocated to containers in multiples of NUMA nodes the balance assumption is not very limiting.



(a) AMD Opteron 6272 node



(b) AMD interconnect



(c) Intel Xeon E7-4830 v3 node

Figure 2.2: The two systems used in this chapter. The first is a quad AMD Opteron 6272. It has eight NUMA nodes (schematically shown in Figure 2.2a) connected with an asymmetric interconnect (Figure 2.2b) and a total of 64 cores. Pairs of cores share the instruction front-end, L2 cache, and floating point units. The second system is a quad Intel Xeon E7-4830 v3 with four NUMA nodes (Figure 2.2c) and 96 hardware threads (12 physical cores per node with SMT). The interconnect (not shown) is symmetric.

2.3 Abstract Machine Model

A major obstacle to a solution to the virtual core placement problem is the sheer number of possible placements. For 16 virtual cores on a 64 core system the number of possible placements is the combinations of 16 objects chosen from a set of

64, which is on the order of 10^{14}.

It is essential to exploit the symmetry in the system to reduce the number of placements to a manageable number. By this we mean that for most types of shared resources it does not matter *which* shared resources are being used but *how much* of the shared resources is available to the workload, and having a way to quantify this would allow us to eliminate the vast majority of placements by only considering those that are actually relevant with respect to performance.

We tackle this with the concept of *scheduling concerns*. A single scheduling concern is responsible for a single hardware resource, or an inseparable set of hardware resources that affect the performance of thread placements. The primary purpose of a scheduling concern is to provide a numerical score when given a thread placement as input. The score represents the utilization of the particular resource and it only depends on the vCPU placement, not the dynamic behavior of a workload. A simple example is an "L2 cache" resource. The scheduling concern for the L2 cache measures the utilization of L2 caches on a system where there are multiple L2 caches shared by sets of cores, so the score would simply be the number of L2 caches in-use by vCPUs. The score remains constant with respect to the symmetry of the hardware resource the concern encompasses. So, two placements might use completely different NUMA nodes but if they use the same number of L2 caches then they will both have the same L2 cache score. A vector of numeric scores for all scheduling concerns uniquely identifies each placement that is distinct with respect to sharing of resources.

There are two additional pieces of information a scheduling concern needs in order to identify the important placements. The first is whether the concern's score is proportional to the user's cost, which is the case for resources like NUMA nodes because fewer nodes (lower score) means more containers can be packed onto a system. If a lower score for a resource only meant worse performance, we could simply discard placements with a lower score for that resource (all other scores being equal) from our list of important placements. But since we want users to be able to make cost-performance trade-offs, placements with lower scores but potentially lower cost could still be relevant. The second piece of information needed by a scheduling concern is whether the resource encompassed by a concern can ever have an inverse relationship with performance. For some resources, like the L2 cache, a higher score is usually better, but for some workloads such as those showing cooperative cache sharing, a smaller score (using fewer L2 caches) may actually improve performance. For other resources, like the shared interconnect described below, a lower score will never improve performance and would not result in a lower cost for the user, so we can safely ignore placements with lower scores when everything else is equal.

In practice, a single scheduling concern may cover multiple shared resources because some resources are inseparable with respect to thread placement. Threads sharing a physical core via SMT typically share a cache, the instruction front-end, and functional units. In cases like this, a single scheduling concern is still sufficient.

Our AMD test system (shown in Figure 2.2) has multiple NUMA nodes, an asymmetric interconnect, and a form of SMT. For this system we developed the scheduling concerns shown in Table 2.1. For the L2/SMT and L3 concerns, the score for a particular placement can be calculated directly from information provided by the operating system. The OS also provides information on the interconnect topology, but it is simpler and more accurate to measure the aggregate bandwidth with a benchmark for each possible combination of nodes (we use *stream* [67] for our measurements). Each concern is relatively simple, easy to implement, and can be developed independently. Since it does not require a performance expert, we envision the specification of scheduling concerns being provided as part of system BIOS.

It is easy to see how scheduling concerns could be developed for other hardware resources. For example, a system with asymmetric CPUs could have a CPU concern where the score is the frequency of the CPUs in use, or a concern that accounts for some nodes being closer to I/O links, where the score is 1 if the nodes in use are near I/O links and 0 otherwise.

Next, from the concerns and hardware topology we need to derive the *important placements*. An important placement must have three properties: **(1)** conform to our balanced assumption, **(2)** be feasible: i.e., not assign more than one vCPU to a single hardware thread, and **(3)** not be superseded by a strictly better placement.

Given a score s and the number of vCPUs v, the balance property is encoded as $v \bmod s = 0$, and the feasibility property is encoded as $v/s \leq Capacity$, where capacity is the number of hardware threads available in a single instance of the resource: e.g. there are eight hardware threads per L3 cache on our AMD test system. We also define the *Count* of a concern as the total number of that resource on the system, so our AMD test system has an *L2Count* of 32 for example. The first step in generating important placements is generating the possible scores that satisfy the balance and feasibility requirements individually. This is done for each scheduling concern that can affect cost or have an inverse relationship with performance. For our AMD test system this step is shown in Algorithm 1.

The next step is computing possible packings of the system, i.e. combinations of placements that fill the entire machine. Recall that we already know how many vCPUs each instance/container will use (based on our assumptions in Section 2.2.2). Depending on the specific placement used, a container may or may not use all available NUMA nodes. If it does not use all available NUMA nodes, then more containers can be packed onto the system. This step computes the packings based on the possible scores generating in the previous step. Specifically, we are using the score of the scheduling concern that corresponds to the highest level of the hierarchy and our unit of resource allocation, which in our case is the L3 scheduling concern. The packings are generated with a recursive method shown in Algorithm 2.

Next, as shown in Algorithm 3, packings that are duplicates and packings that are not Pareto-efficient with respect to the interconnect score are filtered out (since the interconnect concern does not affect cost and cannot have an inverse relationship with performance). Because the L2 and L3 scores can affect cost or have an inverse relationship with performance, placements are not filtered based on them. Lastly, the placements that make up the remaining packings are “expanded” by calculating which L2 scores are feasible for the number of nodes in use, and then the complete placement is added to the list of important placements. If the system in question has a deeper hierarchy, with another scheduling concern below the L2 concern, for example, then the L2 scores would be expanded first and then the scheduling concern below it would be expanded based on the expanded and feasible L2 scores. In general, “score expansion” starts at the highest level of the hierarchy not including the scheduling concern used in generating packings, and then goes downward to the lowest level.

As an example of Pareto-efficient packing, on our AMD test system we know

Algorithm 1 Generating possible L2 and L3 scores

```
L3Scores = List()
for $i \leftarrow 1, L3Count$ do
    if $v/i \leq L3Capacity \land v \bmod i = 0$ then
        L3Scores.append(i)
    end if
end for
L2Scores = List()
for $i \leftarrow 1, L2Count$ do
    if $v/i \leq L2Capacity \land v \bmod i = 0$ then
        L2Scores.append(i)
    end if
end for
return L3Scores, L2Scores
```

we need to keep the four-node placement that uses nodes $\{2,3,4,5\}$ because it is the four-node placement with the highest interconnect score. Therefore the placement using nodes $\{0,1,6,7\}$ is also an important placement and will be kept because it is the placement that can be packed with the best four-node placement. Continuing, suppose that we consider a four-node placement that uses nodes $\{0,1,4,5\}$. If we were to use this placement at runtime, the remaining set of four nodes, potentially used for another workload, is $\{2,3,6,7\}$. Both of these placements have poor interconnect scores, in part because there is a two-hop distance between nodes $\{0,5\}$ and nodes $\{3,6\}$. Instead, we can pack the machine with a better combination of four-node placements: $\{0,2,4,6\}$ and $\{1,3,5,7\}$. Using this observation, the vectors for placements $\{0,2,4,6\}$ and $\{1,3,5,7\}$ will be kept over the worse pair of four-node placements.

After this process is complete, we are left with the important placements. For our AMD system we have 12 of them: two 8-node placements (one sharing L2 caches and one not), two 2-node placements (with the best and second-best interconnect score), and eight 4-node placements (half sharing L2 caches, half not, and various interconnect scores relevant for packing). Our Intel test system (Figure 2.2), on the other hand, only uses an L2/SMT concern and an L3 concern. With 24 virtual cores per container, it has seven important placements which are all of the placements that satisfy the balance and feasibility constraints: a one node

Algorithm 2 Generating packings of placements

```
Packings = List()
procedure MakePackings(L3Scores, NodesLeft, CurrentPacking)
    for all $L3S$ in L3Scores do
        if $L3S >$ len(NodesLeft) then
            continue
        end if
        for all $n$ in Combinations(NodesLeft, L3S) do
            Remaining = NodesLeft - $n$
            NewPacking = CurrentPacking.append($n$)
            if len(Remaining) $> 0$ then
                MakePackings(L3Scores, Remaining, NewPacking)
            else
                Packings.append(NewPacking)
            end if
        end for
    end for
end procedure
return Packings
```

placement sharing L2 caches, two 2-node placements, two 3-node placements, and two 4-node placements.

2.4 Performance Predictions

Automatic model-building techniques learn how to map a set of features describing data to a predicted outcome. The outcome we would like to model is a vector of performance values in all important placements, relative to a baseline placement. For example, if there are three important placements, and the performance in the second and third is 20% and 30% better than that in the first baseline placement, the performance vector will be: $[1.0, 0.8, 0.7]$. Our data elements are executions of workloads in different placements, and the features are some metrics describing the execution.

One way to frame the problem is to predict a *performance category*. That is, assuming that our target workloads can be categorized according to their performance vectors, we can train the model to predict the category and then use the

Algorithm 3 Generating important placements

```
Nodes = range(0, L3Count)
Packings = MakePackings(L3Scores, Nodes, List())
Remove duplicates from Packings
for all ($a$,$b$) in Permutations(Packings, 2) do
    if L3 Scores in $a \neq$ L3 Scores in $b$ then
        continue
    end if
    $aIC$ = Sorted interconnect scores of $a$ placements
    $bIC$ = Sorted interconnect scores of $b$ placements
    ToRemove = True
    for $i$ in range(0, len($aIC$)) do
        if $aIC[i] > bIC[i]$ then
            ToRemove = False
        end if
    end for
    if ToRemove then
        Remove $a$ from Packings
    end if
end for
ImportantPlacements = List()
for all Placements $p$ in Packings do
    $n \leftarrow L2Count/L3Count$
    $L3S$ = L3 Score of $p$
    for all L2S in L2Scores do
        if $n \cdot L3S \geq L2S$ then
            ImportantPlacements.append($p$)
        end if
    end for
end for
return ImportantPlacements
```

category's average vector as the predicted outcome.

We begin by presenting an implementation of this idea (Section 2.4.1). Our first method runs a workload in two or three configurations in order to narrow down the performance category to which it belongs. Although this is not the final method we use, it demonstrates the fundamental benefits of using actual observations of performance as predictive features of the model. Our final method (Section 2.4.2) uses performance observations in two configurations as input features into a machine learning (ML) model. Section 4.4 shows that relying only on hardware performance events (HPE) that may correlate with performance is not nearly as effective as using the measurements of actual performance.

2.4.1 Predicting Performance Categories

Although there is an infinite number of possible workloads and a complex interaction between scheduling concerns, we observed that workloads fall into a relatively small number of categories where each category has very similar performance vectors and is representative of a specific relationship to scheduling concerns. For example, workloads that are not memory intensive and are not adversely affected by sharing SMT contexts would belong to the same category (where thread placement does not matter). Another category would be one where using fewer NUMA nodes and fewer physical cores greatly hurts performance, and so on.

To discover these categories, we use the clustering algorithm k-means, which uses performance vectors as the elements. It partitions the elements into *k* clusters so as to minimize the within-cluster sum of squares (i.e., Euclidean distance) between the vectors. We select the value of *k* that maximizes the average Silhouette coefficient [6, 79] over all data points, which is the standard practice in the field.

The set of applications we experimented with are drawn from the NAS Parallel Benchmark suite [11], Parsec suite [20], the Metis map-reduce benchmarks [65], and BLAST [7]. Also included are the Linux kernel compile gcc benchmark, two Spark graph workloads, TPC-C [95] and TPC-H [96] on Postgres and a WiredTiger [4] BTree benchmark. Workloads were run using `lxc` containers and configured to use 16 vCPUs on the AMD system and 24 vCPUs on the Intel system (Fig. 2.2). Within containers, the number of application threads is set so as to achieve $>70\%$ CPU utilization on each core, typical of what is done in practice.

The workloads on our Intel test system were clustered into six groups. Figure 2.3 shows the performance by placement for the workloads found in three example clusters. Considering the similarity of performance curves in each cluster, it is clear that clustering has worked well in this case. An inspection of the remaining clusters and the clusters produced on our AMD test system showed similar results.

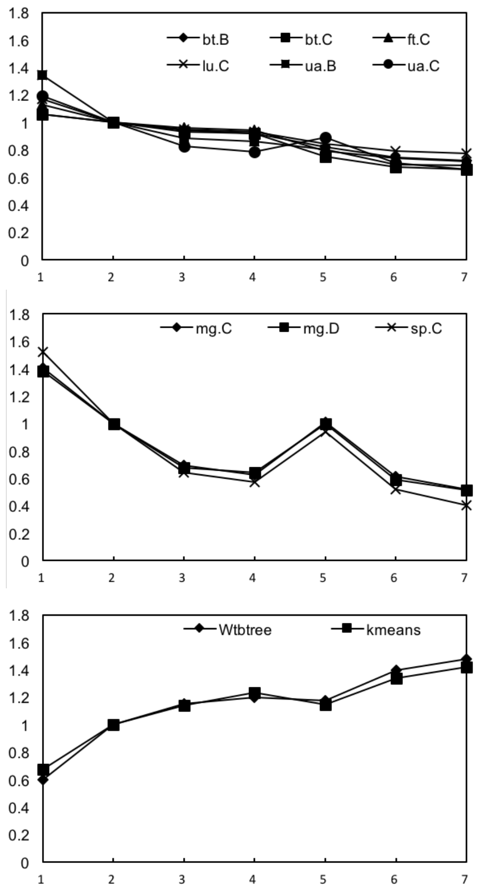


Figure 2.3: Performance by placement for workloads in three example clusters on Intel. The x-axis shows the placement ID. The y-axis shows performance relative to the baseline placement (#2).

This procedure helps to determine the quality and completeness of the training workloads. If k-means cannot create good clusters, then the training set could be incomplete. In our case the quality of clusters is good overall, but some clusters, e.g., the one with *kmeans* and *WTbtree*, contains only two elements and could benefit from adding other workloads.

It also leads to an intuitive method for making performance predictions: run the workload in several placements by trying them during the first few seconds of the execution without interrupting the workload, measure the performance in each, and based on those results use the process of elimination to determine the category to which it belongs. Once we know the category we can use category's average performance per-placement as the predictions.

The workload must first be run on the baseline placement; our performance measurements are all normalized to this placement so it is required but does not provide any information about the workload. Every placement tried thereafter helps narrow down the category to which the workload belongs (if the performance is outside of the range of performance values seen during training for a particular category then we can conclude it does not belong to that category) until only the workload's predicted category remains. The full algorithm for determining a workload's category, which we call the *iterative method*, is shown in Algorithm 4.

For an arbitrary system, the worst-case number of measurement placements needed to complete Algorithm 4 is $min(n-1,k)$ where n is the number of important placements and k is the number of clusters. This worst-case happens when only a single cluster can be eliminated per measurement placement. Real-world systems are likely to have a much better worst-case though, such as our test systems which only require three placements in the worst-case. This is because there are likely to be some correlations between clusters and correlations between scheduling concerns. For example consider a system that has cluster A, cluster B, and some number of other clusters. Cluster A prefers having more L3 caches and more L2 caches, and because memory intensiveness affects both L2 and L3 cache preferences there is no cluster that prefers more L3 caches and prefers fewer L2 caches. Cluster B prefers more L2 caches but does not care how many L3 caches it has. A single measurement placement showing that a workload prefers fewer L2 caches eliminates both cluster A and cluster B. Similarly a single placement could eliminate both cluster A or B and clusters that prefer fewer L2 caches. In this way we are likely to avoid the theoretical worst-case on real systems.

On our test systems the iterative method requires trying only two or three placements (including the baseline) to determine a workload's category and produces fairly accurate predictions. For example, 29 out of the 41 workloads on our Intel test system have predictions within 6% of the actual performance, but the other workloads have at least one placement that has a prediction error around 20%.

The key insight we draw from the iterative method is that performance measurements from multiple placements have very high predictive power due to the fact that workloads tend to belong to distinct categories with respect to their performance behavior. This motivates our approach of using performance measurements as inputs into a ML model, which is described in the next section.

Algorithm 4 Iterative method for determining performance category

```
for all Cluster $c$ in Clusters do
    for each Placement $p$ do
        min = min. performance within Cluster for $p$
        max = max. performance within Cluster for $p$
        Interval of c for placement = [min, max]
    end for
end for
Excluded = List()
Run baseline placement and measure performance
while len(excluded) $<$ len($c$)-1 do
    Pop next placement from $p$
    Run next placement and measure performance
    for each Interval for placement do
        if Performance is outside interval then
            Append cluster belonging to interval to excluded
        end if
    end for
    if Performance is outside all intervals then
        Find closest interval and exclude all other clusters
    end if
    if All clusters excluded then
        Return no cluster
    end if
end while
return cluster not in excluded
```

2.4.2 Predicting Performance with Machine Learning

While the method of predicting performance categories using the process of elimination is intuitive and robust, it uses a category's average as the predicted outcome, which is a rather rough measure. We found that we can achieve higher accuracy with a more refined modeling technique. Our final approach directly predicts performance vectors using a machine learning model, a multi-output *Random Forest* regressor. In other words, the model produces one predicted output per placement. As inputs, the model uses performance measurements observed in two different placements: a baseline and one additional. These placements were selected during the model training as those yielding the highest accuracy.

The random forest model has significantly better accuracy for this problem than other regression models like neural networks, support vector machines, and nonlinear regression. Many regression models rely on an assumption that the data fits a particular shape (for example, logarithmic or polynomial) but the data in our case has many instances of step-function and piece-wise behavior. A random forest requires no assumption of the shape of the data. Neural networks do not have this assumption either but they tend to require much more training data to be effective.

An alternative to regression is a search-based machine learning approach. In this case, a machine learning algorithm could continually try placements until the goal is reached. The benefit of this approach is that potentially it would not need to rely on training workloads. A major downside though is that it would require many more placements to be tried which requires migrating memory and would have a prohibitive performance cost. For this reason we focused on the regression approach.

Finally, to evaluate our new approach relative to the best known practices we also train a multi-output *Random Forest* regressor using HPEs observed in a single baseline placement as input features. The best baseline placement was identified during training.

Modern machines have many hundreds of HPEs, some more than 1000 [103]. Sampling that many online cannot be done without large sampling errors. One option is to include the list of important HPEs in the specification of each scheduling concern. This assumes that the engineer providing the specification somehow knows which counters would work best — an assumption we found to be not in the spirit of maximally automating the prediction process. Another approach is to obtain measurements with all possible HPEs during training and use feature selection methods to identify the best predictors, similarly to [41]. This approach is automatic, but increases the training time from hours to weeks. For example on our Intel machine with nearly 1000 performance events, the time to measure all counters for our training set while ensuring acceptable sampling error would amount to 66 days.

Instead we used a combination of the manual and automatic approaches. We started with a set of plausible features (41 HPE derived metrics on our Intel test system and 25 on our AMD test system) covering cache, memory, TLB, interconnect, and pipeline behavior, which are metrics commonly used in similar work. We then used *Sequential Forward Selection* [39, 54] (SFS) to pick the best ones. SFS involves iterating over all potential features, selecting the one that improves prediction accuracy the most, and then repeating this process until prediction accuracy no longer improves. On the AMD test system the SFS process results in four features: the L3 cache misses per cycle, the L1 cache miss rate per instruction, the TLB misses per cycle, and the cache sharing percentage (calculated from the states of cache lines as they are evicted). On the Intel test system, three features were selected: the percentage of stall cycles where at least one memory request was pending, the cache sharing percentage (calculated from the states of cache lines as they are inserted), and the L2 cache misses per instruction caused by the prefetcher.

We also tried adding both performance measurements and HPEs as model inputs, but this did not improve accuracy at all on our AMD system and only marginally improved accuracy for two workloads on our Intel system. In the end we had two model variants to compare: the first one used as inputs the actual performance measurements observed in two placements, and the second one used only the HPEs observed in a single placement.

2.4.3 Results

The full per-benchmark performance prediction results for the AMD test system are provided in the appendix in Figure A.4 and for the Intel test system in Figure A.5. The figures for the benchmarks discussed in this section are also provided here for ease of reference.

The results are per-application cross-validated. For example, when training the model that will be used for predicting a Spark workload neither the data from *spark-cc* (a Spark connected components algorithm run on the LiveJournal database) nor *spark-pr-lj* (a PageRank algorithm run on the LiveJournal database) is included

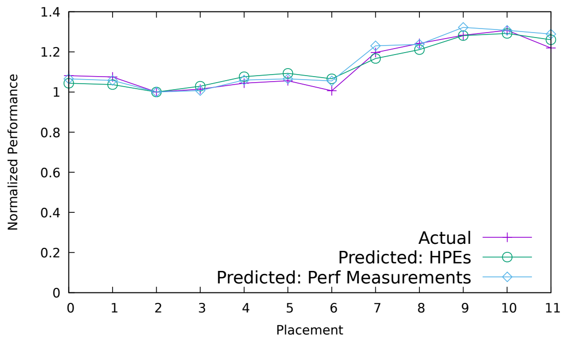


Figure 2.4: Prediction results for *postgres-tpcc* on the AMD test system.

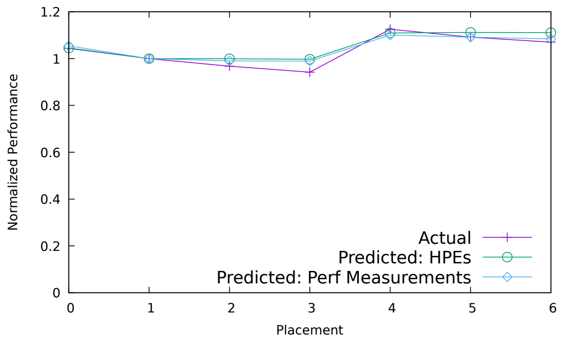


Figure 2.5: Prediction results for *spark-pr-lj* on the Intel test system.

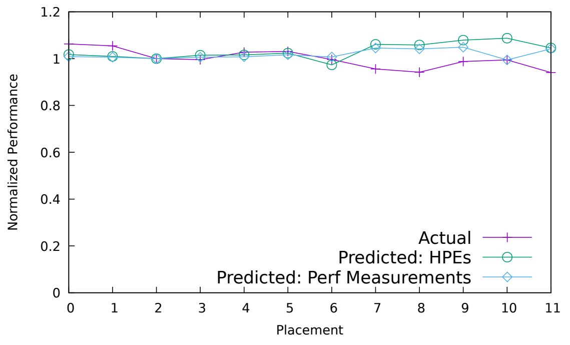


Figure 2.6: Prediction results for *kmeans* on the AMD test system.

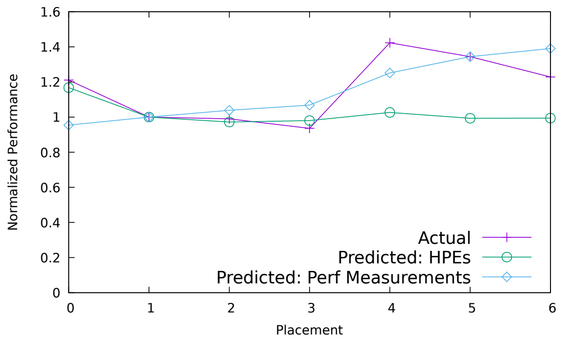


Figure 2.7: Prediction results for *canneal* on the Intel test system.

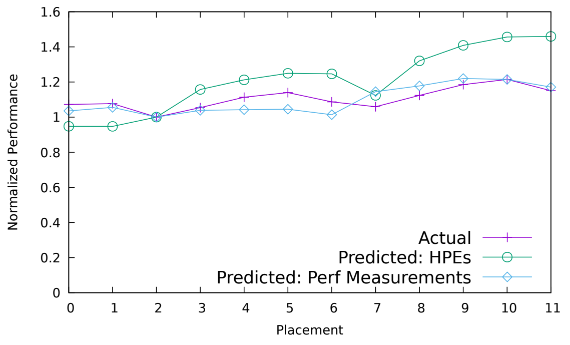


Figure 2.8: Prediction results for *dc.B* on the AMD test system.

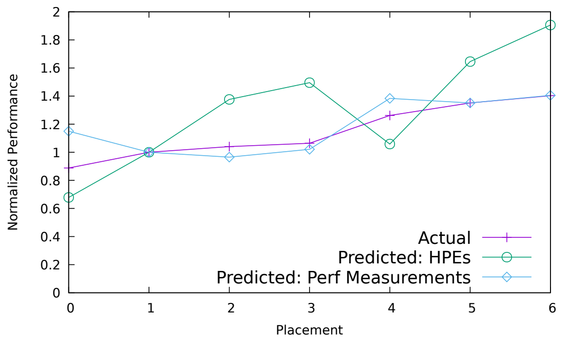


Figure 2.9: Prediction results for *ft.C* on the Intel test system.

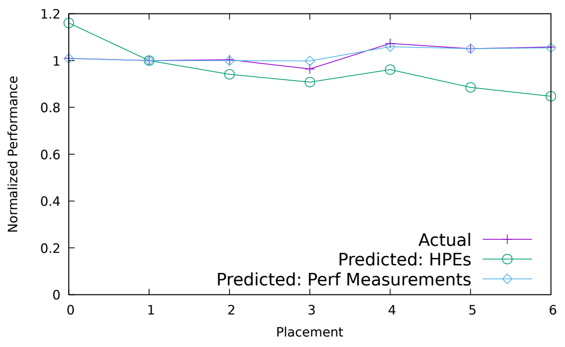


Figure 2.10: Prediction results for *freqmine* on the Intel test system.

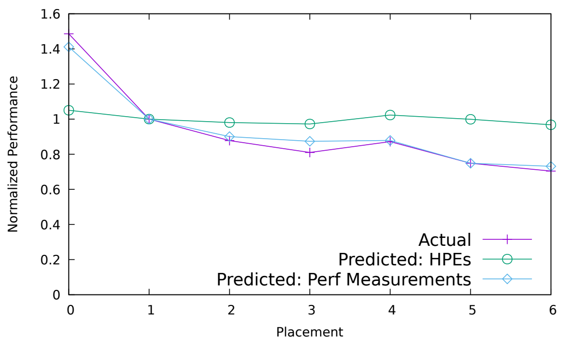


Figure 2.11: Prediction results for *kmeans* on the Intel test system.

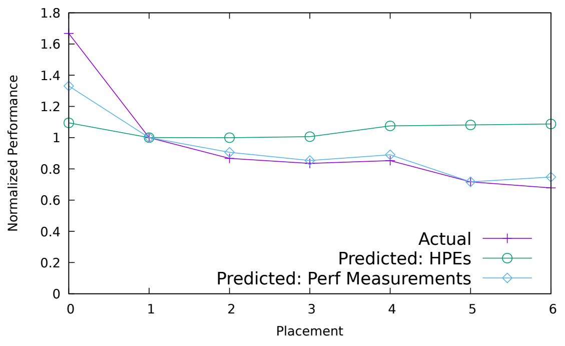


Figure 2.12: Prediction results for *WTbtree* on the Intel test system.

in the training.

Overall the accuracy when using only the actual performance measurements as model features is high. The predicted performance is within 4.4% of actual on average on the AMD system, and within 6.6% on Intel. Postgres running the TPCC benchmark (*postgres-tpcc*) on the AMD test system (Figure 2.4) and *spark-pr-lj* on the Intel test system (Figure 2.5) are examples where the predictions are very accurate.

There are a few cases where the training set did not include any workloads that behaved similarly to the predicted benchmark, which results in poor predictions. For example *kmeans* on the AMD system (Figure 2.6), which was the only benchmark in our training set that preferred SMT, or *canneal* on the Intel system (Figure 2.7). As explained in Section 2.4, we can identify weaknesses in the training set using the performance clusters, so it is straightforward to figure out where extra time and effort could be spent on adding new training workloads.

Prediction accuracy when using only the HPEs from a single placement was a lot less reliable. On the AMD system it produced good results overall, but the accuracy was still noticeably worse for *dc.B* (Figure 2.8) compared to the model variant that relied only on actual performance measurement. The real shocker is the results on Intel, where the model relying only on HPEs produced many poor predictions. It completely missed the performance trend for *ft.C* (Figure 2.9) and *freqmine* (Figure 2.10), produced errors of over 40% for *kmeans* (Figure 2.11) and *WTbtree* (Figure 2.12), and is noticeably worse for several other workloads. Using both the actual performance measurements and HPEs yielded small accuracy improvements on the Intel system, but made no difference on the AMD system.

An example of why HPEs observed in a single placement could have poor predictive power, and one of the reasons why the Intel system produced worse predictions, is predicting the effect of inter-thread communication latency. There is a huge latency difference for communication between a single-node placement and placements including more than one node. For some applications, reduced inter-thread communication latency when all threads are running on a single node has a major performance impact, as is the case for *WTbtree*. Separating the sensitivity to latency from overall memory intensiveness (which can be measured by the cache miss rate) is difficult to do with HPEs. Similarly, it is also very difficult to determine if a workload's working set will fit in a given number of L3 caches by only measuring HPEs on a single placement. ***We conclude that, contrary to prior belief, single-placement HPE observations are not reliable features for modelling performance on multicore NUMA systems.***

2.5 Using the Model in Practice

There are many ways in which data center operators can use our model. To illustrate one potential use case we set up a scenario where the user would like to pack as many instances of a given virtual container into a physical server while respecting a performance target. To assess the overheads, we measure the costs of container migration on our test systems (the overhead of prediction inference is negligible).

2.5.1 A Potential Use Case

In our experiment we use virtual containers of three types: one running WiredTiger with a B-tree search workload, another running Postgres with the TPC-H workload, and another running Spark with the PageRank workload on a LiveJournal database. We have many containers of each type, and our goal is to pack as many of them as we can per physical server without violating a performance goal.

The performance goal can be specified in terms of an application-level metric such as transactions per second or a generic metric such as instructions-per-cycle. The placement policy is agnostic to the metric used and only requires that the application make this metric available at runtime. For clarity of presentation, we simply set the performance goals to correspond to 90%, 100% and 110% of the performance observed in the baseline placement.

We compare four hypothetical container placement policies. The first policy, referred to as **ML**, is based on our techniques. It decides how many nodes to allocate to the container based on performance observations in two placements and the model presented in the previous section. It runs the workload in two placements during the first few seconds of the execution without interrupting the workload, and then migrates it into the best predicted placement. To separate various aspects of performance, the results shown here do not include the migration overhead; it is studied separately in the next section. The second policy, **Conservative**, is a naïve policy that allocates the entire machine to each instance, allowing only one instance per machine. The third policy, **Aggressive**, is another simple policy that fills the system with as many instances as possible, maximizing machine utilization at the risk of performance violations. For example, our AMD system allows up to four 16-core instances and our Intel system up to four 24-core instances. Neither Conservative nor Aggressive pin vCPUs to cores, allowing Linux to perform the mapping in the way it wishes, and possibly creating unneeded contention. As an alternative, we also evaluate a more sophisticated fourth policy, **Smart-Aggressive**. This policy is similar to Aggressive, except each instance is pinned to the best minimum set of nodes, which we define as having the highest interconnect bandwidth. This policy requires an analysis of the interconnect topology in order to find the correct set of nodes.

We could not make a fair comparison to any other method presented in earlier work. As we explained in Section 2.2, most earlier models targeted very different systems and most did not predict performance vectors, so we could not apply them directly.

We evaluate the policies by measuring how many instances of the same workload they were able to pack per machine (higher is better) and the degree of violation of the performance goal as the percent of the target (lower is better). All workloads were run using `lxc` containers and configured to use 16 vCPUs on the AMD system and 24 vCPUs on the Intel system. Figures 2.13 and 2.14 show the results for the three container types. The bars show the number of instances packed (left y-axis), while the "stars" shows the deviation from the target performance goal, expressed as percentage (right y-axis).

The ML policy always meets the performance goal while in most cases packing more instances per machine than the conservative scheduler. The conservative policy not only wastes resources, but also, surprisingly, may cause performance target violations (Figs. 2.13a and 2.14b), because the Linux scheduler may map vCPUs unevenly to shared resources, causing contention where it could be avoided.

The aggressive policy packs a maximum possible number of containers per machine, at the cost of performance target violations, up to 46% with WiredTiger on AMD, and 43% with Spark on Intel. It is surprising that even when the aggres-

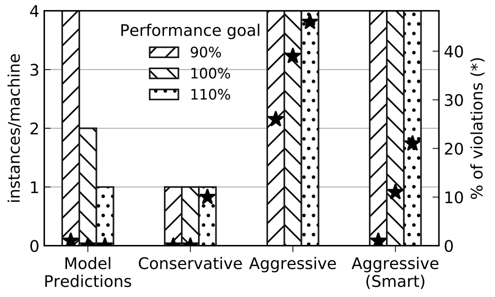
Benchmark	Memory (GB)	Fast Migration (s)	Default Linux (s)
BLAST	18.5	3.0	5.9
canneal	1.1	0.3	3.9
fluidanimate	0.7	0.3	2.3
freqmine	1.3	0.3	4.2
gcc	1.4	0.3	2.8
kmeans	7.2	1.5	6.5
pca	12.0	2.8	10.0
postgres-tpch	26.8	5.8	117.1
postgres-tpcc	37.7	14.9	431.0
spark-cc	17.0	3.7	139.9
spark-pr-lj	17.1	3.8	137.0
streamcluster	0.1	0.1	0.4
swaptions	0.01	0.1	0.0
ft.C	5.0	1.3	19.4
dc.B	27.3	5.4	51.7
wc	15.4	3.4	19.5
wr	17.1	3.6	18.9
WTbtree	36.3	6.3	43.8

Table 2.2: Migration performance on the AMD system, compared to the default Linux migration method. The amount of memory includes processes' memory and the page cache associated with the container.

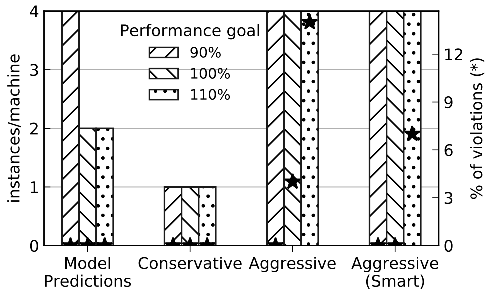
sive policy packs the same number of containers per machine as the model-based policy, it still often reports a higher violation percent. That is because this policy allows virtual containers to share NUMA nodes. Smart-aggressive addresses this shortcoming, but even that policy can cause performance violations (e.g., 20% for WiredTiger on AMD), because it does not take into account all ways in which workload placement might affect performance.

2.5.2 Memory Migration Overhead

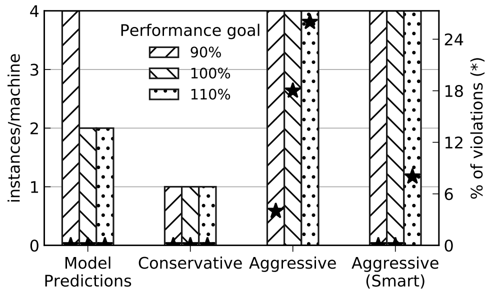
Memory migration in Linux is known to be inefficient [59]. Our migration technique is based on that proposed by Lepers et al. [59]. Their method freezes the application, parses its memory map and migrates pages in parallel using a collec-



(a) WiredTiger

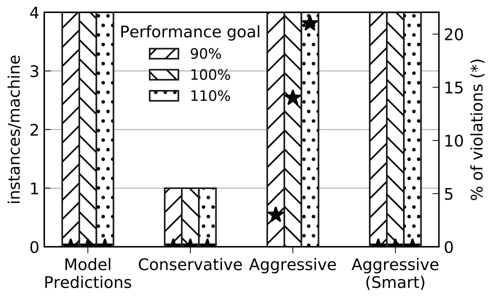


(b) Postgres (TPC-H)

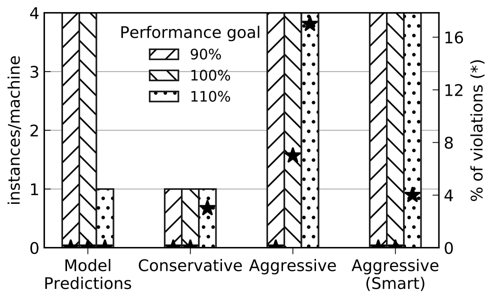


(c) Spark (PageRank)

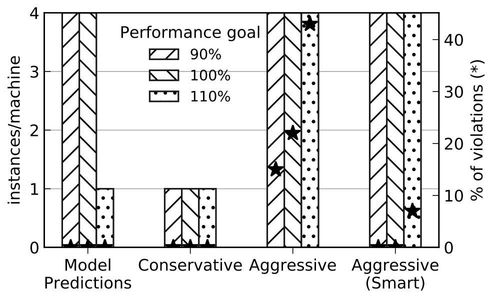
Figure 2.13: Instances packed per machine (left y-axis) and performance goal violation (as %, right y-axis) on the AMD system.



(a) WiredTiger



(b) Postgres (TPC-H)



(c) Spark (PageRank)

Figure 2.14: Instances packed per machine (left y-axis) and performance goal violation (as %, right y-axis) on the Intel system.

tion of worker threads. We improve on the method in [59] by reducing the locking overhead when migrating shared pages and by creating mechanisms for migrating a page cache, which is not migrated at all by either Leper's method or by default Linux. Table 2.2 show the total time needed to migrate the workloads used in §2.4. Note that Default Linux migration time does not include the time needed to migrate the page cache, because this is not supported, and yet this can be a very large fraction of migration overhead (93% with BLAST, 75% with TPC-C and 62% on TPC-H). The overhead of migration highly depends on the workload: a few big files in the page cache are more likely to generate contention during migration than many small files, because each file has its own memory map. Processes sharing memory will incur a higher per-page migration cost, as is the case with Postgres. Overall, we found that migrating a large amount of memory can be done in a few seconds, and is order of magnitude faster than default Linux migration method (38$\times$ faster for Spark). One exception is Postgres TPC-C, whose migration takes 14.9 seconds. This benchmark has an atypically large number of processes and threads: 167 and 33067 respectively! Linux's mechanism for updating the `cpuset`, which changes when migration is performed, has per-task overhead and is very slow for an application with such an extreme task count.

A drawback of this method is that it requires freezing the container during migration in order to reduce contention on some critical kernel locks. As a result, it is not suitable for interactive latency-sensitive workloads. In this case, we have the option of not freezing the container and instead throttling the bandwidth given to the migration process so as to reduce the impact on the running application. Thus, the migration takes more time but with a smaller impact on the running container. Using this method, the overhead of migration for the WiredTiger workload was between 3% and 6%, and the migration took 60 seconds[4].

Overall, we observe that the migration overhead is proportional to the amount of memory used by the container, except in cases with extremely high thread counts. Using the container's memory footprint, the user can estimate whether the cost of migration would permit an online deployment of the container placement algorithm, or if it is preferable to use the model offline for placement of recurring

[4]WiredTiger is the only workload we could evaluate in the interactive mode because it is the only one able to report performance during the execution.

jobs.

2.6 Summary

Modern multicore systems have a complex hierarchy of shared resources and performance can vary wildly depending on how virtual CPUs are mapped to hardware contexts. Operators waste resources and money by using conservative and suboptimal placement policies.

We have shown a solution to this problem using a methodology to abstract a system's shared resources, identify important placements, and predict their performance. We presented a method for predicting performance based on multi-output regression. The best accuracy is achieved when observations of actual performance on two placements are used as model features. Hardware performance events, conventionally used as features for predictive models, turned out to be not as predictive as previously thought. Our method can lead to very significant advantages in machine utilization while keeping performance guarantees.

CPU architecture is continually changing, often by sharing resources between cores in new ways, in order to continue scaling the core count. AMD's newly introduced Zen architecture [32] has L3 cache sharing separate from sharing the memory controller. Intel's Haswell-E architecture has asymmetric links between NUMA nodes through its cluster-on-die feature [71], which has unique performance implications different from other asymmetric architectures. The flexibility of our methods means that they can be used on systems like these or future architectures without significant retooling by an expert.

Chapter 3

An SMT-Selection Metric[1]

SMT is an architectural technique used to improve the overall performance of a wide range of applications [97]. It is designed to improve CPU utilization by exploiting both instruction-level parallelism and thread-level parallelism. Chapter 2 addressed scheduling with respect to SMT in conjunction with other scheduling concerns, but it assumed that the number of vCPUs a workload uses is already known beforehand. This chapter, on the other hand, focuses on how to choose the level of parallelism of an application running on an SMT system. In other words, *Given a multithreaded application, will performance improve if additional hardware contexts are available via SMT as we increase the number of threads?* Precisely, we are considering situations where the number of physical cores is fixed, the number of hardware threads/contexts used per physical core varies (that is, different SMT-levels), and the number of software threads is set to the number of available hardware threads. Because some applications scale poorly to higher thread counts and because the utilization benefit of SMT is application dependent, using SMT can sometimes hurt performance significantly in this scenario.

We propose an SMT-selection metric (*SMTsm*) to answer this question, and it is the primary contribution of this chapter. It does not require any changes to applications or operating systems and incurs low overhead. The metric is measured online while applications are being run. Our metric-based approach relies on HPEs and measures the tendency of a workload to run better or worse in more

[1]This chapter is a modified version of work previously published in [46]

hardware contexts. SMTsm can be easily integrated in any user-level scheduler or even kernel schedulers to provide insights and intelligent decisions about the right SMT level to be used by a workload. SMTsm can be measured periodically and hence allows adaptively choosing the optimal SMT level for a workload as it goes through different phases. We also show how this metric can be used in a scheduler or a user-level optimizer to help guide scheduling decisions.

The rest of the chapter is organized as follows: Section 3.1 provides background information and motivating examples, Section 3.2 describes the SMT-selection metric and its rationale. Section 3.3 presents the experimental methodology adopted using two processor architectures. Performance evaluation is presented in Section 3.4. Section 3.5 describes ways the SMT-selection metric can be used. Finally, related work is examined in Section 3.6 and concluding remarks and future work are discussed in Section 3.7.

3.1 Background & Motivation

With SMT, the processor handles a number of instruction streams from different threads in the same cycle. The execution context, like the program counter, is duplicated for each hardware thread, while most CPU resources, such as the execution units, the branch prediction resources, the instruction fetch and decode units and the cache, are shared competitively among hardware threads. In general the processor utilization increases because there are more instructions available to fill execution units and because instructions from other hardware threads can be executed while another instruction is stalled on a cache miss. Since the threads share some of the key resources, it is performance-efficient to schedule threads with anti-correlated resource requirements.

Several studies have shown that SMT does not always improve the performance of applications [51, 66, 81]. The performance gains from SMT vary depending on a number of factors: The scalability of the workload, the CPU resources used by the workload, the instruction mix of the workload, the cache footprint of the workload, the degree of sharing among the software threads, etc. Figure 3.1 shows the performance of three benchmarks with and without SMT (4-way SMT) on the 8-core POWER7 microprocessor. We first run the application with eight threads at

SMT1. Then we quadruple the number of threads and enable SMT4. Note that for Equake, SMT4 degraded the performance of the application, while it improved the performance of EP. MG's performance was oblivious to whatever SMT level was used.

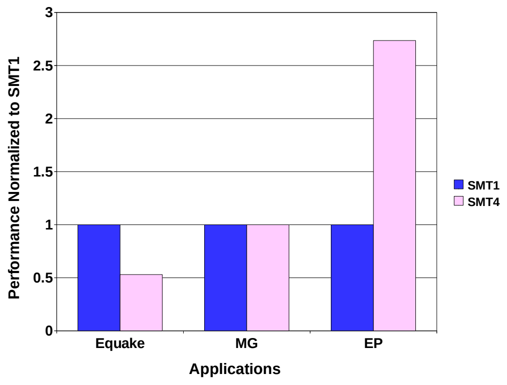


Figure 3.1: Comparison of performance with SMT1 vs. SMT4 for three applications on an 8-core POWER7 system. Each application is run alone in a separate experiment. The application uses eight threads under SMT1 and 32 threads under SMT4 and threads are bound to their own hardware contexts.

In general, workloads that benefit from SMT contain threads that under-utilize certain processor resources as well as threads are able to make use of those resources. Reasons why such ”symbiotic” situations occur include:

1. *A large number of cache misses*: For a non-SMT processor, when an instruction miss occurs, no more instructions are issued to the pipeline until more instructions have been brought to the instruction cache. A similar situation happens in the case of a data cache miss, the stream of instructions ceases execution until the missing data is brought to the cache. Such situations could result in delays ranging from tens to hundreds of cycles. SMT enables one or more other hardware threads to execute their instruction streams when such delays occur; hence, maximizing the use of the processor pipeline.

2. *Long chains of instruction dependencies*: Inter-instruction dependencies limit the instruction-level parallelism of applications. Based on the layout of the

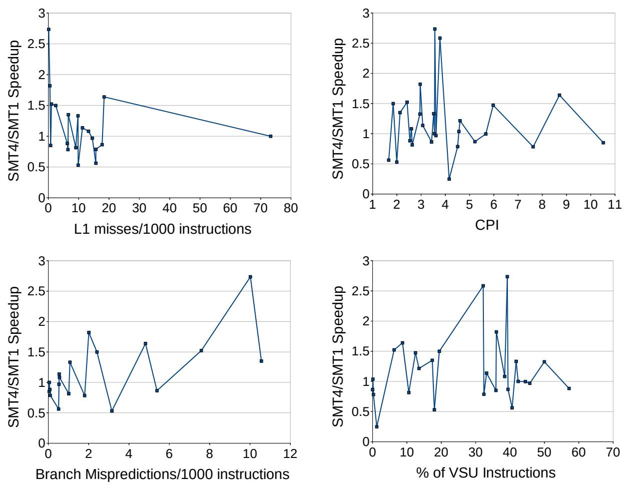


Figure 3.2: Speedup on SMT4/SMT1 plotted against cache misses, CPI, branch-mispredictions, and fraction-of-floating-point/vector instructions for 27 benchmarks on the POWER7 processor. Eight threads are used under SMT1 (on eight cores), 32 threads are used under SMT4, and threads are bound to their own hardware contexts.

multiple pipeline stages, compilers attempt to generate independent instructions that can be executed in parallel. When dependencies exist, the next instruction ceases execution until it can receive the results of the previous instruction. So if the workload exhibits very long chains of instruction dependencies, SMT could help to fill the gaps by allowing other independent instruction streams from other threads to execute in the otherwise idle execution units.

3. *A large number of branch mis-predictions*: When the branch history table and the branch target buffer are not large enough to service a large number of branch mis-predictions, the execution units remain idle. This is again another opportunity for SMT to allow other hardware threads to use the execution units while the branch mis-prediction is being resolved.

The workloads in these examples are expected to benefit from SMT, because one or more threads leave resources idle, but other threads have *sufficient diversity in the instruction mix* to put these resources to use. At the same time, if a workload consists of threads that are individually well optimized for a super-scalar processor (e.g., they do not leave resources idle), this workload is not expected to benefit from SMT, because there are no resource gaps to fill.

While SMT allows executing multiple streams of instructions in the same cycle, it also introduces more resource contention among the hardware threads that are co-scheduled on the same core. If any of the shared resources becomes a bottleneck, all threads contending for the resource will suffer, and SMT will not be beneficial. Properties of workloads that create contention for resources include:

1. *A homogeneous instruction mix*: If one or few types of instruction are more common than others, the workload may create contention for the functional unit responsible for this type of instruction. For example, workloads that are floating-point intensive are likely to gain little from simultaneous multi-threading.

2. *Intensive use of the memory system*: Irrespective of instruction mix, a workload stressing the memory system (e.g., because of poor cache locality) may cause memory-related stalls to become even longer and more frequent on an SMT processor due to increased contention for the memory bandwidth. As a result, processor resource utilization could decrease instead of increasing.

In summary, we intuitively understand that workloads that benefit from SMT have threads that under-use resources, which other threads are able to use, while at the same time not creating contention for these resources. At the same time, being able to predict what is the right SMT level to use for a given workload is not a trivial task. This requires a thorough knowledge of both the internals of the workloads and the internals of the hardware they run on. The complexity of predicting the right SMT level increases as the number of supported SMT levels increases. For instance, IBM's POWER7 processor [55] has 4-way SMT multithreading and exposes to applications three different levels: SMT disabled or SMT1 level, 2-way SMT or SMT2 level, and 4-way SMT or SMT4 level. Although this technology provides more flexibility, it also introduces more complexity since the user needs to decide what is the right SMT level for their running application.

In an attempt to see if it is possible to predict performance improvements from SMT by just looking at applications' characteristics, we plotted the speedup obtained at SMT4 vs. SMT1 against four main application metrics on a POWER7 machine: L1 cache misses, branch mispredictions, cycles per instruction (CPI), and fraction of floating point operations. The experiment was conducted using 27 representative multithreaded benchmarks on a POWER7 system (more details about the benchmarks used will be presented in subsequent sections). Figure 3.2, shows that there is no correlation between any of the four metrics and the SMT speedup.

One option for SMT tuning is to compare application performance with and without SMT offline and then use the configuration resulting in better performance in the field. However, this method is not effective if the hardware used in the field is not the same as that used for original testing, and if the application behavior significantly changes depending on the input. Another option is to vary the SMT level online and observe changes in the instructions-per-cycle (IPC), but this method has limited applicability, because not all systems allow changing the SMT level online. Furthermore, IPC is not always an accurate indicator of application performance (e.g., in case of spin-lock contention).

3.2 The SMT-Selection Metric

The rationale behind the SMT-selection metric is based on how well the instructions of a workload can utilize the various pipelines of a processor during each cycle. An ideal workload for SMT would have a good mix of instructions that are capable of filling all available functional units at each cycle. Figure 3.3 shows the pipeline of a generic processor core. In each cycle, the processor fetches from the instruction cache a fixed number of instructions. These instructions are then decoded and buffered. As resources become available, instructions can be dispatched/issued to the various execution/functional units in an out-of-order manner. *Issue ports* are the pathways through which instructions are issued to the various functional units, which can operate independently. If the instructions that are issued consist of a mix of load, store, branch, integer, and floating point instructions and there are little data dependencies between them, then all functional units will

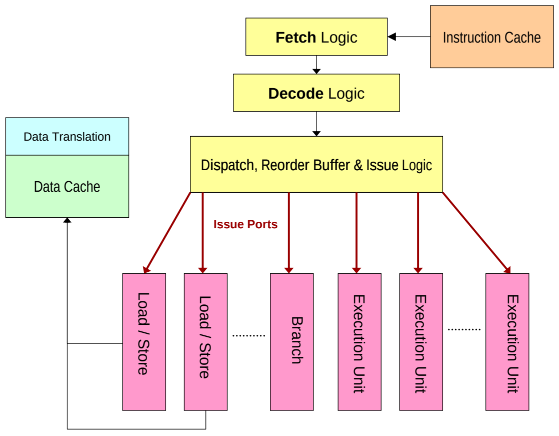


Figure 3.3: A generic processor execution engine.

be able to be used concurrently, hence increasing the utilization of the processor.

We define the term *ideal SMT instruction mix* to mean a mix of instructions that is proportional to the number and types of the processor's issue ports and functional units. With an ideal mix, the processor is able to execute the maximum number of instructions supported. In order for SMT to increase utilization there needs to be instructions available from all the hardware contexts to use as many issue ports as possible. Consider a multithreaded application whose vast majority of instructions are fixed point (integer) instructions. Running the application with more hardware contexts will not help because the fixed point units were already occupied most of the time with one hardware context. On the other hand, if we have an application with an ideal SMT instruction mix, then SMT should improve performance since the processor will have more opportunities to fill all the execution units.

Since SMTsm must be able to predict whether an application benefits from additional SMT resources *as we increase the number of threads*, it must also include some measure of scalability within the application itself. After all, if there are software-related scalability bottlenecks, the application will not run better with increased number of threads irrespective of hardware. We observe that instruction mix, which is crucial for predicting hardware resource utilization in SMTsm, is also a good indicator of software scalability. An application that spends significant time spinning on locks will have a large percentrage of branch instructions and a high deviation from the ideal SMT mix.

Equation 3.1 shows how to calculate the SMTsm metric for the generic processor discussed above, where *Pi* denotes a unique issue port, *N* is the total number of issue ports, *DispHeld* is the fraction of cycles the dispatcher was held due to lack of resources, *TotalTime* is the wall-clock time elapsed, and *AvgThrdTime* is the average time spent by each hardware thread. Smaller metric values indicate greater preference for a higher SMT. The metric consists of three factors: i) the instruction mix's deviation from an ideal SMT instruction mix, ii) the fraction of cycles that the dispatcher was held due to lack of resources, and iii) the ratio of the wall-clock time elapsed to the average CPU time elapsed across all threads. f_{Pi} is the fraction of instructions that are issued to *Pi*. For example, to calculate f_{P1}, the number of instructions issued through port 1 is divided by the total number of instructions.

$$\begin{aligned} SMTsm = (&\sum_{i=0}^{N-1} (f_{Pi} - 1/N)^2)^{1/2} \\ &* DispHeld \\ &* (TotalTime/AvgThrdTime) \end{aligned} \tag{3.1}$$

The second factor of the SMT-selection metric is the fraction of cycles that the dispatcher was held due to lack of resources. The meaning of *resources* is architecture dependent and may include many items but it should primarily refer to the issue queues of the execution units. If the issue queues are filling up to the point where the dispatcher is held, then having additional instruction streams to dispatch from is not going to be useful. This factor is important to have in addition to the instruction mix because it indirectly captures the effect of instruction-level parallelism and cache misses. The number of cycles the dispatcher is held due to resources is easily obtained through HPEs in many modern processors.

The final factor of the metric is the ratio of the wall-clock time elapsed to the average CPU time elapsed per hardware thread. This measures scalability limitations manifested through sleeping or Amdahl's law as opposed to busy waiting.

This factor does not have a direct relationship with SMT preference, but scalability is an important factor to consider since additional software threads are needed to use the available SMT hardware contexts.

In the following subsections, we illustrate how SMTsm metric is measured for two different processor architectures: IBM's POWER7 and Intel's Nehalem Core i7.

3.2.1 SMTsm on IBM's POWER7 Processor

In a given cycle, the POWER7 [55] core can fetch up to eight instructions, decode and dispatch up to six instructions, and issue and execute up to eight instructions. The core has 12 execution units: two fixed point units, two load/store units, four double-precision floating-point pipelines, one vector unit, one branch unit, one condition register (CR) unit, and one decimal floating point pipeline. POWER7 processors support up to 4-way SMT. In other words, up to four hardware contexts can concurrently use the core. If there is only a single software thread running on a core, the processor automatically runs the core at SMT1 which gives the hardware thread access to resources that would be partitioned or disabled at higher SMT levels. Similarly, if there are only two software threads on a core then the core runs at SMT2.

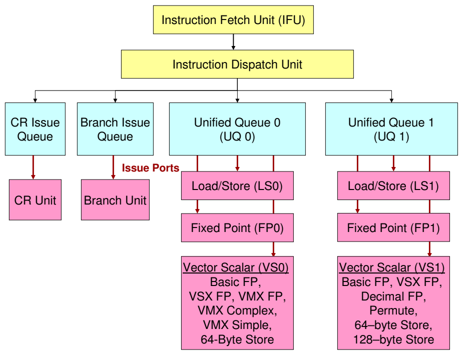


Figure 3.4: IBM POWER7 out-of-order execution engine.

Figure 3.4 shows that an issue port in POWER7 is tied to a type of instruction. For instance, a fixed point instruction always uses a fixed point issue port. There are a total of eight issue ports: 1 port corresponds to a conditional register (CR) instruction, 1 port corresponds to a branch instruction, the remaining 6 issue ports are divided equally between the two unified issue queues, UQ0 and UQ1. Through each UQ, up to one load/store instruction, one fixed point instruction (FP) and one vector scalar (VS) instruction can be issued concurrently. It is important to note here that the CR unit is a special unit. It is tightly tied to the branch unit. It is also not heavily used in general. This unit has been mainly designed to avoid sending the *compare* instructions through the FP unit to avoid tying branch prediction to the FP unit. Therefore, we consider in our metric both the CR and branch units as one execution unit. So, an ideal SMT instruction mix for the POWER7 architecture would consist of 1/7 loads, 1/7 stores, 1/7 branches, 2/7 FP instructions, and 2/7 VS instructions. The loads and stores are separated because they rely on separate resources like the load and store buffers. To measure the second term of the equation (dispatcher held) in POWER7, the hardware performance event *PM_DISP_CLB_HELD_RES* can be used. The SMT-selection metric for the POWER7 processor is shown in Equation 3.2.

$$\begin{aligned} P7SMTsm = {} & ((f_L - 1/7)^2 + (f_S - 1/7)^2 \\ & + (f_{BR} - 1/7)^2 \\ & + (f_{VS} - 2/7)^2 + (f_{FP} - 2/7)^2)^{1/2} \\ & * DispHeld \\ & * (TotalTime/AvgThrdTime) \end{aligned} \tag{3.2}$$

3.2.2 SMTsm on Intel's Nehalem Processor

On the Nehalem Core i7, the number of issue ports equals the maximum number of instructions that can be issued in a cycle (see Figure 3.5). In contrast to POWER7, each of the six issue ports is used for a variety of unrelated instructions [94]. The unified reservation station serves as a single scheduler for all the execution units. It is responsible for assigning instructions to the different execution units. The

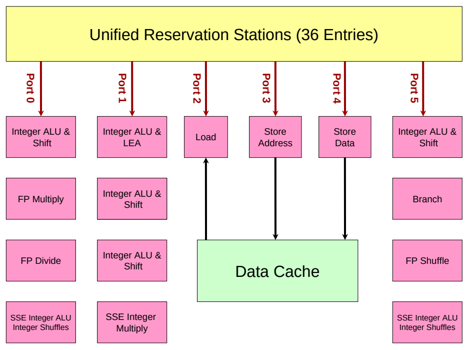


Figure 3.5: Intel Nehalem out-of-order execution engine.

core can issue up to 6 instructions per cycle. Three of them are memory operations (load, store address and store data), and the other three are computational instructions (floating point, branch, and integer operations). Intel's Nehalem core supports 2-way SMT.

Equation 3.3 shows the SMT-selection metric for Intel's Nehalem Core i7 processor. The term f_{Pi} refers to the fraction of instructions that have been issued through port i ($i \in [0,1,2,3,4,5]$). Since the issue ports on Nehalem are not related to a single type of instruction, we simply measure the number of instructions issued to each port. All instructions map to a single issue port, except for integer ALU instructions which map to three ports, so the mix of instructions sent to each issue port is sufficient for calculating the SMT-selection metric. Dispatch held can be obtained using *RAT_STALLS* event with the *rob_read_port* unit mask [52].

$$\begin{aligned} Ci7SMTsm = {} & (\sum_{i=0}^{5}(f_{Pi} - 1/6)^2)^{1/2} \\ & * DispHeld \\ & * (TotalTime/AvgThrdTime) \end{aligned} \tag{3.3}$$

3.3 Experimental Methodology

3.3.1 System Configuration

Experiments were conducted on an AIX/POWER7 system and a Linux/Core i7 (Nehalem) system.

The AIX/POWER7 system uses AIX 6.1.5 and two 8-core POWER7 chips. For the single-chip experiments, the benchmarks were restricted to run on one 8-core chip. The POWER7 CPU is clocked at 3.8 GHz and the system has 64 GB of RAM. The C, C++, and Fortran benchmarks were compiled with IBM XL compiler 11.1.0.6 using these flags: *-O3 -qstrict -qarch=auto -qsimd=auto -q64* and *-qsmp=omp*. The MPI programs use the IBM Parallel Operating Environment version 5.2.2.3. the Java benchmarks use the 64-bit IBM JVM version 1.6.0. The SMT levels on POWER7 can be changed without rebooting the system by running the *smtctl* command with privileged access.

The Linux/Core i7 system uses Linux kernel 2.6.34 with 3GB of RAM and a four cores Intel Core i7 965 clocked at 3.2 GHz with two SMT threads per core. GCC 4.4.5 was used to compile the benchmarks with the flags *-O3 -march=native* and *-fopenmp* where appropriate. Unlike POWER7, the SMT level can only be changed by rebooting and modifying a BIOS setting. In our experiments SMT2 is always enabled in the BIOS. Therefore to simulate SMT1 we only use one software thread per core. This better represents typical use cases because SMTsm is designed to be used dynamically at run-time.

3.3.2 Benchmarks

The experiments use a diverse set of benchmarks to capture the variations in characteristics of various workloads. The benchmarks are drawn from the NAS Parallel Benchmarks (NPB) v3.3.1, the PARSEC Benchmark Suite v2.1, the SSCA2 benchmark, the STREAM synthetic benchmark, the SPEC OMP2001 benchmark suite v3.2 and two commercial benchmarks. Due to compatibility issues we were not able to run all of the benchmarks on the POWER7 system. Due to time constraints, we focused mostly on the POWER7 system, because it supports a higher SMT level than Nehalem; as a result we did not run all of the benchmarks on Nehalem.

A brief description of the benchmarks used is outlined below.

- *The NAS parallel benchmark Suite* [11] is a set of programs that have been initially designed to evaluate the performance of supercomputers. Both the MPI and OpenMP versions were used on AIX/POWER7 but only the OpenMP versions were used on Linux/Core i7.

- *PARSEC benchmarks* [20]: PARSEC stands for Princeton Application Repository for Shared-Memory Computers. It is a set of programs designed to evaluate the performance of Chip-Multiprocessors (CMPs). PARSEC benchmarks mimic multithreaded applications from different fields such as recognition, mining, and large-scale commercial applications. PARSEC does not officially support the AIX operating system, so only a handful of the benchmarks were able to be used on the AIX/POWER7 system.

- *SSCA2 benchmarks* [10]: SSCA, which stands for the Scalable Synthetic Compact Applications, is a computational graph theory benchmark that uses OpenMP. It consists of four kernels with irregular access to a large, directed, and weighted multi-graph. This benchmark is characterized by integer operations, a large memory footprint, and irregular memory access patterns.

- *STREAM* [67] is a synthetic benchmark designed to measure memory bandwidth and also uses OpenMP. To obtain reasonable running times for our experiments, we have increased the the default array size and number of iterations to 4577.6 MB and 1000 respectively.

- *SPEC OMP benchmark Suite* [82] is adapted from the SPEC CPU2000 benchmarks. Its goal is to evaluate the performance of openMP applications on shared memory multi-processors. We have used the SPEC OMP experiments only on the AIX/POWER7 system.

- *DayTrader* [1] is a Websphere benchmark application that emulates an online stock trading system. The application simulates typical trading operations such as login, viewing portfolios, looking up stock quotes, and buying or selling stock shares. The benchmark consists of a websphere front-end, a DB2 database, and a load generator. The DayTrader client is a Linux Intel

Xeon machine running the JIBE (Websphere Studio Workload Simulator), which simulates a specifiable number of concurrent browser clients. We simulated 500 clients for stressing the DayTrader application running on the DayTrader server. This number of clients was sufficient to keep the server continuously busy with waiting requests to be processed.

- *SPECjbb2005* is a Java server benchmark from the Standard Performance Evaluation Corporation [2] based on the TPC-C benchmark specifications. It simulates a 3-tier system in a JVM with emphasis on the middle tier.
- *SPECjbb05-contention* is a custom benchmark derived from SPECjbb2005. The primary change introduced in SPECjbb05-contention is that all worker threads operate on a single warehouse instance instead of each worker thread operating on its own warehouse instance. This introduces synchronization contention that is not present in SPECjbb2005.

3.4 Evaluation

In all of the experiments conducted, the number of software threads used is chosen to be the same as the number of available hardware threads/contexts in the OS instance. For example, in the AIX instance on one 8-core POWER7 chip, 32 software threads were used at SMT4, 16 software threads were used at SMT2, and 8 software threads were used at SMT1. Similarly, on the Linux instance on the 4-core Core i7 machine, 8 software threads were used at SMT2, and 4 software threads were used at SMT1.

The metric was originally developed and tested on the POWER7 system. After finalizing the metric, it was evaluated on the Core i7 machine.

3.4.1 SMT-Selection Metric (SMTsm) Evaluation

Figure 3.6 shows the relationship between the SMT-selection metric measured at SMT4 and the speedup obtained on SMT4 relative to SMT1 on the AIX/POWER7 system. We can see a clear correlation between the metric value and the speedup, and the correlation is strong enough to predict the optimum SMT level in most cases. If we set a threshold close to the value of 0.07 then we can be confident that

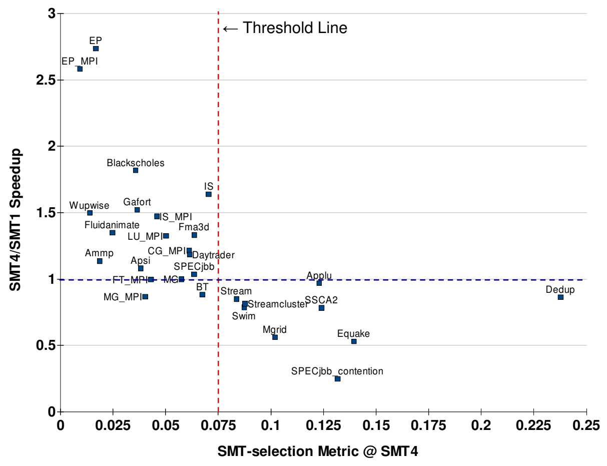


Figure 3.6: SMT4/SMT1 speedup vs. metric evaluated @SMT4 – AIX instance on an 8-core POWER7 chip.

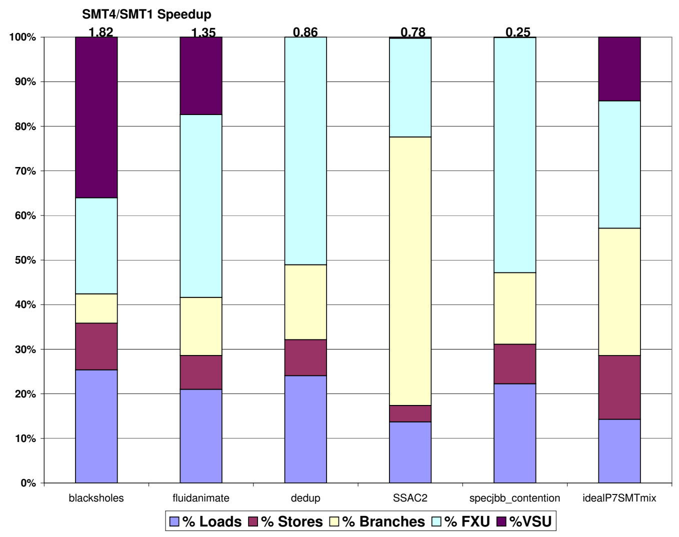


Figure 3.7: Instruction mix of five benchmarks – AIX instance on an 8-core POWER7 chip.

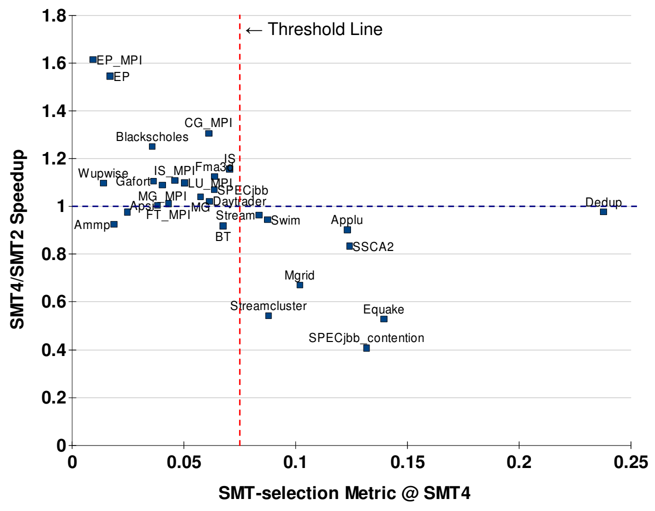


Figure 3.8: SMT4/SMT2 speedup vs. metric evaluated @SMT4 – AIX instance on an 8-core POWER7 chip.

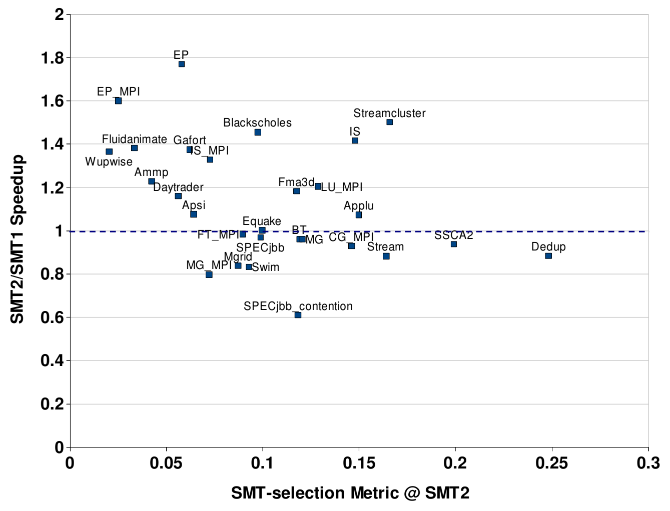


Figure 3.9: SMT2/SMT1 speedup vs. metric evaluated @SMT2 – AIX instance on an 8-core POWER7 chip.

any application with a metric greater than the threshold will perform better or the same at SMT1 than SMT4, and applications with a metric less than the threshold benefit or are not harmed by using SMT4. This is true for 93% of the benchmarks evaluated. Applications that fall to the left of the threshold are likely to prefer SMT4, with only two of the evaluated benchmarks having a metric less than the threshold and performing slightly worse at SMT4.

In Figure 3.7, we clearly see a correlation between the instruction mix and the SMT4/SMT1 speedup. We have selected representative benchmarks from the set of benchmarks studied. As we move from the left of the figure to the right, the speedup going from SMT1 to SMT4 decreases from 1.82 to 0.25, while the instruction mix tends to be more and more dominated by one or two functional units.

The metric versus SMT4/SMT2 speedup on AIX/POWER7 is shown in Figure 3.8. Once again a threshold of 0.07 provides good separation. All of the benchmarks with a metric greater than the threshold prefer SMT2. Three benchmarks have a metric less than the threshold and a speedup less than 1 but greater than 0.9. All of the remaining benchmarks have a metric below the threshold and a speedup greater than 1.

The experiment shown in Figure 3.9 is the same as the previous experiments except it uses the SMT2 over SMT1 speedup. In this case, the SMT-selection metric is not capable of always making an accurate prediction. For metric values below 0.07 or above 0.19, we can predict the optimum SMT level. However, for metric values between 0.07 and 0.19, it is not possible to predict the application's SMT preference.

Figure 3.10 shows the SMT-selection metric compared to the SMT2/SMT1 speedup on the Linux/Core i7 system. In this experiment, a stronger correlation than in any of the AIX/POWER7 experiments is observed, but there is one outlier on the far right which is Streamcluster from PARSEC. With only eight software threads running at SMT2, there is less synchronization contention so only a few of the benchmarks prefer SMT1 over SMT2. In this case there is not much motivation for SMT optimization but the experiment does show that the SMT-selection metric can be adapted to other architectures.

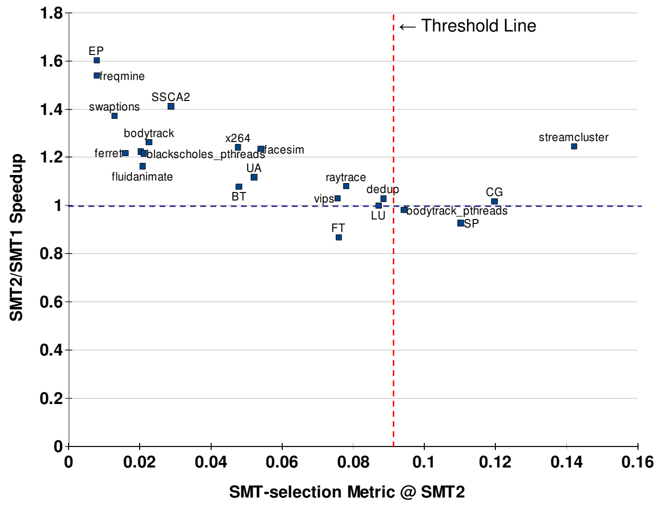


Figure 3.10: SMT2/SMT1 speedup vs. metric evaluated @SMT2 – Linux instance on a quad-core Core i7 system.

3.4.2 SMTsm Evaluation at a Lower-SMT Level

The previous subsection evaluated how well the SMTsm estimated performance speedup is, when the SMTsm is measured at the highest supported SMT level (SMT4). In this subsection, we evaluate the metric when the application is running at a lower SMT level, and the metric is used to predict the speedup at a higher SMT level.

Figures 3.11 and 3.12 show the same experiments presented in subsection 3.4.1 but with the SMTsm measured at the lowest supported SMT level. The experiments did not show a good correlation between the metric and the speedup. This is not surprising, as the metric is not able to foresee scalability limitations caused by more threads at a higher SMT level; the metric is only capable of detecting a slowdown when it is happening. At SMT1 we are not able to accurately capture contention as it was the case at SMT4, so the metric breaks down at SMT1. Therefore, it is important to use the metric at the highest SMT-level available. Moreover, in all SMT-capable processors, the highest SMT-level is always used as the default since many multi-threaded applications benefit from SMT. This motivates further the use of the metric at higher SMT-levels to predict whether going to a lower SMT-level

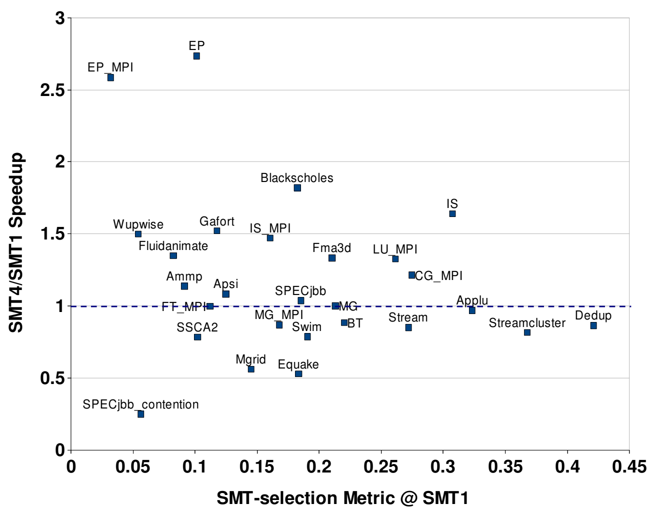


Figure 3.11: SMT4/SMT1 speedup vs. metric evaluated @SMT1 – AIX instance on an 8-core POWER7 chip.

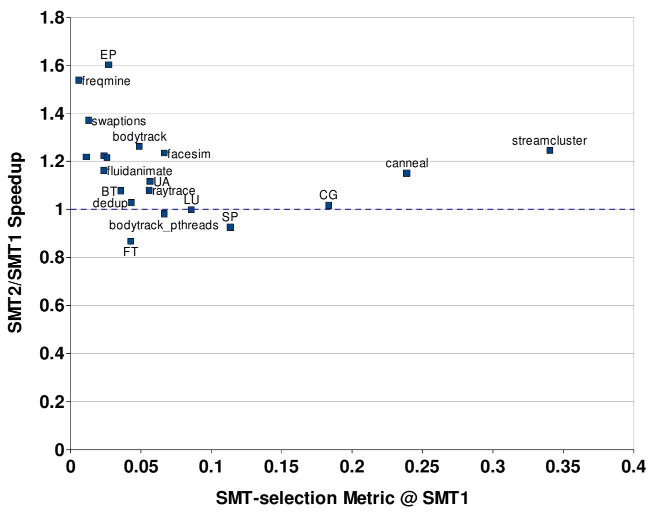


Figure 3.12: SMT2/SMT1 speedup vs. metric evaluated @SMT1 – Linux instance on a quad-core i7 system.

benefits the running workload.

3.4.3 Metric Evaluation Across Chips

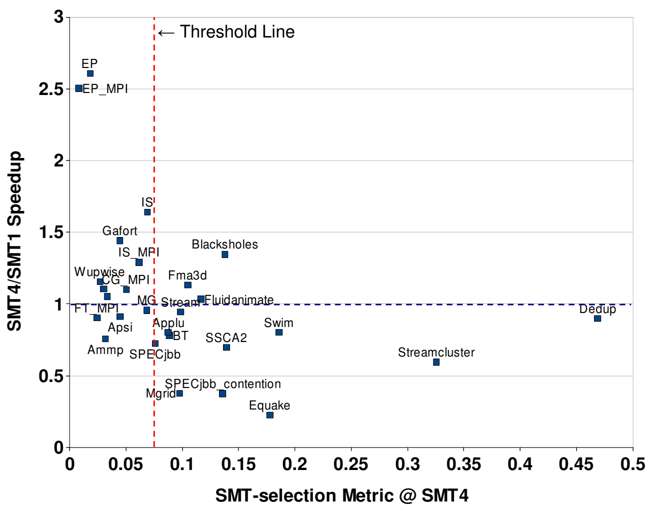


Figure 3.13: SMT4/SMT1 speedup vs. metric evaluated @SMT4 – AIX instance on two 8-core POWER7 chips.

Figures 3.13, 3.14, and 3.15 give the results for the SMTsm prediction experiments on an AIX instance running on a two-chip POWER7 system. For these experiments there are 16 cores, which means 64 software threads are used at SMT4, 32 threads are used at SMT2, and 16 threads are used at SMT1. Using two chips introduces two new variables that the metric must compensate for to remain accurate. First, there is a performance penalty for cross-chip communication, so applications that are more sensitive to NUMA effects may affect the metric differently. Second, the number of running software threads is doubled at all SMT levels compared to the single chip case, so the effect of scalability is amplified.

For the SMT4/SMT1 case presented in Figure 3.13, the results are similar to the SMT4/SMT1 experiment with only one chip. However, there are more benchmarks that are mis-predicted. We also notice, that applications that have a metric near the threshold are more likely to be mispredicted. Another difference with the single chip experiment is that more applications prefer SMT1 over SMT4. This is

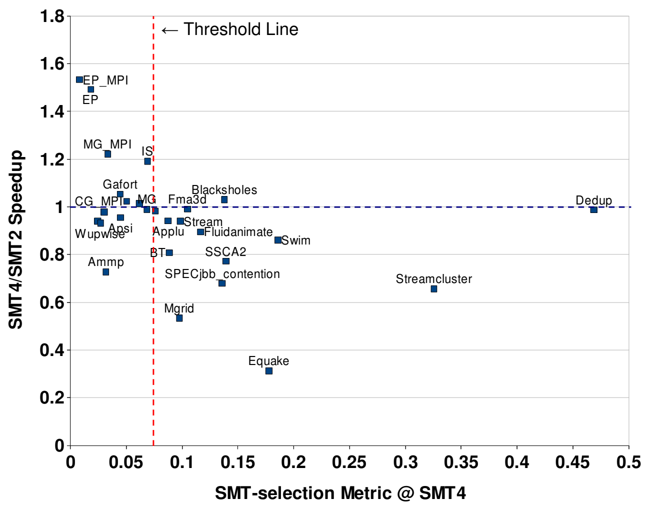


Figure 3.14: SMT4/SMT2 speedup vs. metric evaluated @SMT4 – AIX instance on two 8-core POWER7 chips.

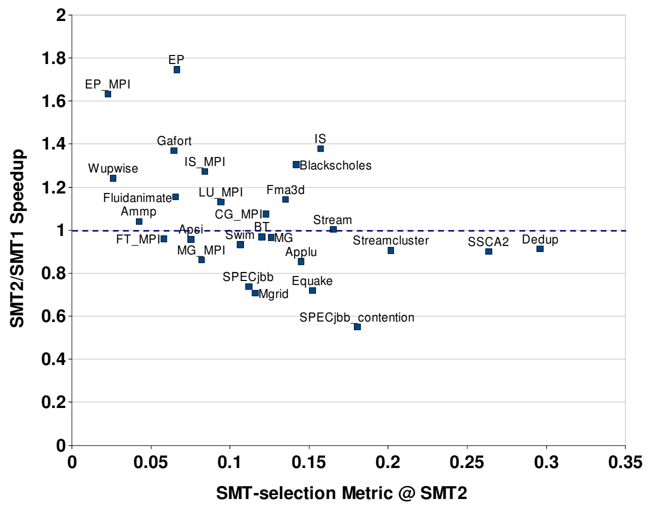


Figure 3.15: SMT2/SMT1 speedup vs. metric evaluated @SMT1 – AIX instance on two 8-core POWER7 chips.

expected since with more software threads, more contention for synchronization resources will be introduced, and hence more scalability limitations.

The SMT4/SMT2 results (Figure 3.14) look better than the SMT4/SMT1 results, but there is still only a small difference in metric values between SMT4-preferring applications and SMT1-preferring applications. Figure 3.15 demonstrates that SMT2/SMT1 prediction is ineffective, the same as in the single chip case.

SMT preference prediction is important for large systems with many cores because more applications will be hindered by SMT as synchronization overhead and contention over CPU resources overtake the benefits of SMT. The results show that the SMT-selection metric is still useful at 16 cores, but more work needs to be done since the metric is less accurate at 16 cores than at 8 cores. One possibility is that the scalability detection aspect of the metric starts to break down with a large number of threads. This is supported by the fact that the metric works better at SMT4/SMT2 prediction with 16 cores, since the change in the number of software threads is smaller than when predicting SMT4/SMT1 speedup.

3.5 Applying the SMT-Selection Metric

The SMT-selection metric can be used by operating systems to guide scheduling decisions. It can also be used by user-level optimizers or application tuners to dynamically adjust the SMT level of the underlying system to improve the performance of running applications.

To use the SMT-selection metric, the formula must first be adapted to the target architecture. In section 3.2, we presented the metric for the IBM POWER7 and Intel Nehalem architectures. The metric can be ported to other architectures in similar ways. The threshold for changing the SMT level needs to be determined for each new system. This can be achieved by running a representative set of workloads, recording the SMT speedups and the observed SMTsm metric values for each workload, as we did in section 3.4. Once the (metric, speedup) values are gathered, the threshold can be obtained automatically using statistical techniques. We describe two methods to obtain a good SMTsm threshold for deciding when a change in SMT level would benefit the performance of a given application.

3.5.1 Using Gini Impurity to Decide on a Good SMTsm Threshold

Gini impurity [77] is a measure of how well separated or clustered a set is. We look for a separator (potential threshold) that leads to the lowest overall Gini impurity as follows:

1. Re-label the (metric, speedup) tuples into the form (metric, i) with $(i \in \{0,1\})$, setting i=0 if the speedup is less than 1, and i=1 if the speedup is greater than or equal to 1.

2. Divide the tuples into 2 sets $\{$L=Left-set, R=Right-set$\}$ based on whether the metric value is to the left or to the right of the separator value.

3. Calculate the Gini impurity of the left-set (I_L) and the right-set (I_R) as shown in equations 3.4 and 3.5, where $|L_0|$ denotes the size of the left-set with $i = 0$ (i.e., $speedup < 1$), $|L_1|$ denotes the size of the left-set with $i = 1$ (i.e., $speedup \geq 1$), and $|L|$ is the size of the entire left set ($|L| = |L_0| + |L_1|$). Similar notation is used with the right set.

$$I_L = 1 - \left[\frac{|L_1|}{|L|}\right]^2 - \left[\frac{|L_0|}{|L|}\right]^2 \tag{3.4}$$

$$I_R = 1 - \left[\frac{|R_1|}{|R|}\right]^2 - \left[\frac{|R_0|}{|R|}\right]^2 \tag{3.5}$$

4. Calculate the ***overall Gini impurity*** using equation 3.6.

$$Impurity = \frac{|L|}{|L+R|} \cdot I_L + \frac{|R|}{|L+R|} \cdot I_R \tag{3.6}$$

An impurity of 0 indicates that the set is perfectly separated, i.e. all of the sample points to one side of the separator have a speedup greater than or equal to 1, and all of the remaining points are on the other side. *A high impurity value means that the selected separator is not a good classifier, and vice versa.*

Figure 3.16 shows the results of using Gini impurity to provide a suitable metric threshold value at which to decide on performing a change in SMT level to improve performance, when using SMT4/SMT1 speedup data on POWER7. The dotted vertical lines mark the range of optimal thresholds. The figure also displays two easy ways to observe the qualitative fitness of the SMT-selection metric on a given system and a set of benchmarks. First, is how low the impurity is at its lowest point, which represents how good a prediction can be made. In the figure, the lowest impurity is 0.23 which is good, as verified by the fact that only four of the benchmarks were misclassified with the threshold obtained by this method (refer to Figure 3.6 which has four points to the left of the separator with a speedup below 1). Second, is how large the range of optimal thresholds is. If the range is very small, then a new application with a metric beyond that range is likely to be mispredicted.

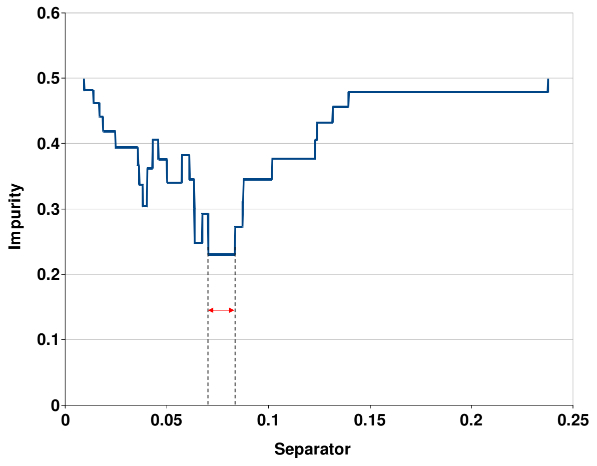


Figure 3.16: Total overall Gini impurity for potential thresholds of the SMTsm metric for SMT4/SMT1 speedup on POWER7.

3.5.2 Using the Average PPI (Percentage Performance Improvement) Method to Decide on a Good SMTsm Threshold

With this method we are trying to estimate how much performance improvement we would obtain if we switched from the default SMT level (e.g., SMT4) to a lower one (e.g., SMT1) as dictated by different thresholds. The threshold with the highest estimated Percentage Performance Improvement (PPI) is deemed the best. In order to do this, for each potential SMTsm threshold value, and for each benchmark, we estimate a PPI value as follows:

- If the benchmark's measured SMTsm value is less than the threshold in question, then its PPI is set to 0. Essentially, this means that the benchmark is not expected to benefit from a lower SMT setting, so its expected PPI from switching to lower SMT level is zero.

- If the benchmark's measured metric value is greater than the threshold value in question, then the PPI is set to $((\frac{1}{SMT4/SMT1speedup} - 1) * 100$. In other words, if the benchmark is expected to benefit from a lower SMT setting based on the current threshold, we calculate PPI as the performance improvement at SMT1 relative to SMT4 (expressed in percent).

Then, we take the average of the PPIs over the whole set of benchmarks as the Y-value to plot against that threshold value. This gives us the average expected performance improvement at each threshold level. Examining this data, we can choose the best threshold – the one that gives us the highest PPI.

Figure 3.17 shows an example of using this method for SMT4/SMT1 performance improvement prediction on POWER7. The results are similar to those using Gini impurity, but this methods provides the following additional benefits:

1. It can be used to easily show how much performance improvement the SMTsm metric can provide. The Gini impurity method only shows that the metric is working, but cannot show potential improvements.

2. It also gives a view of potential PPIs over a range of threshold values. Even though the range of optimal metric thresholds is relatively small in both methods, we can see from Figure 3.17 that there is actually a large range of potential threshold values where we have an average PPI that is greater than 15%. This means that a new application whose metric value falls into this range is not likely to experience a severe negative effect from using this metric value as a threshold for deciding on a change in SMT level.

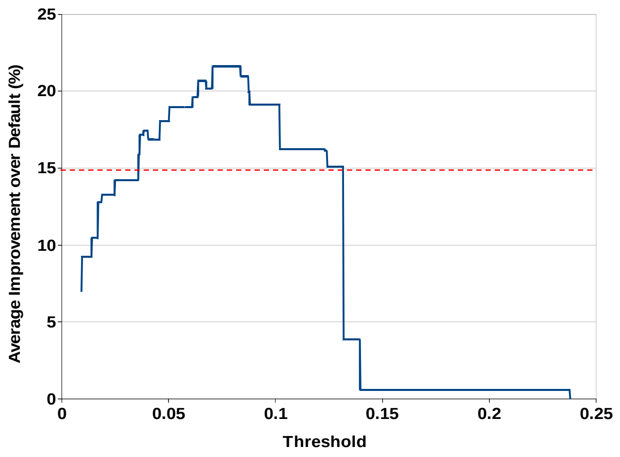


Figure 3.17: Average SMT4/SMT1 percentage performance improvement of all the benchmarks vs. SMTsm values – AIX instance on POWER7.

3. This method can also provide a better threshold than the Gini impurity method in some cases, because Gini impurity does not consider the amount of speedup. For example, there could be a few benchmarks with speedup values just below one, and a benchmark with a very large speedup just to the right of them. In this case Gini impurity would suggest putting the threshold to the left of all the mentioned benchmarks so as to classify more benchmarks correctly, whereas this method would suggest putting the threshold to the right, thereby preserving the large speedup in return for minimal slow-downs in the other benchmarks.

3.6 Related Work

SMT job schedulers and SMT performance characterization comprise the majority of related work. The SMT job schedulers are designed to find high performing (often referred to as symbiotic) co-schedules from a larger selection of running applications. They do not attempt to optimize the SMT level itself like our SMT-selection metric. The SMT performance characterizations do investigate the effect of the SMT level but none of them propose a general metric or algorithm for optimizing it. Additionally, most of the previous work focuses on single-threaded applications while our work studies multi-threaded applications.

Mathis et al [66] evaluate and analyze the effect of SMT2 on the POWER5 CPU with single-threaded applications. To measure the SMT2 gain of an application, they simply run one copy of the application per available hardware thread/context with and without SMT. The authors found that most of the tested applications have a moderate performance improvement with SMT. They also found that applications with the smallest improvement have more cache misses when using SMT. This result is less applicable to multi-threaded applications because the total amount of work and data does not increase as the number of threads increases, like it does when you run more copies of a single-threaded application and because threads of a multi-threaded application may share data.

Ruan et al [80] evaluate and analyze the effect of SMT on network servers but do not attempt any optimization. They found that the overhead of using an SMP-capable kernel sometimes outweighs the benefit of SMT, but this is irrelevant today since all modern server CPUs are at least dual-core. The authors also discovered that SMT can sometimes hurt performance when there is more than one CPU which supports our claim that the SMT level should be optimized.

Snavely and Tullsen [87] describe a job scheduler for SMT systems called SOS (Sample, Optimize, Symbios). The goal of SOS is to choose an effective co-schedule of applications from a pool of ready-to-run applications. SOS has a sampling phase where it tries many different co-schedules and measures a performance predictor metric from hardware counters. Then, it has a symbiotic phase where it runs the co-schedules with the best predicted performance. The authors evaluated several different predictors and found that a high IPC and a low L1 data cache miss rate are both good predictors. They did try a predictor based on the instruction mix, but it only looked at integer and floating point instructions and it did not take into account the mix of execution units. Snavel et al [88] extended SOS to support application priorities. Overall, SOS is effective for finding good co-schedules among many single-threaded applications, but it is not designed to choose the best SMT level for a multi-threaded application.

Settle et al [84] designed a job scheduler similar to SOS in its goals: find optimal co-schedules from a set of single-threaded applications. They use custom hardware performance counters to create a fine-grained view of the cache access patterns of the applications, from which they derive co-schedules with an average of 7% improvement over the default scheduler.

Eyerman and Eeckhout [42] propose an SMT job scheduler that is meant to surmount the shortcomings of SOS. They use a probabilistic model to co-schedule applications without the need for a sampling phase, and it can be configured to optimize for throughput or turn-around time. The major downside of their approach is that it requires specialized CPU counters that are not available on commercial hardware.

Tam et al [93] present a solution for scheduling threads on SMP-CMP-SMT systems. Their goal is to reduce remote cache accesses by scheduling threads together that access the same data. The authors approach this problem by using hardware performance counters to monitor the addresses of memory that cause remote cache accesses and then scheduling together (on the same chip or on the same core) threads that access the same memory. They achieve 5-7% performance improvements for a handful of applications, but their system is not meant to determine the optimal SMT level.

Parallel scalability prediction is also related to our work, since the SMT-selection metric must estimate the effect of scalability when changing the SMT level. Previous work in the area requires many sample data points [37][14] or access to the source code of the application [68], so it is not suited to our purpose. Our approach only uses information available at run-time to detect scalability limitations and is accurate enough for SMT-selection prediction for most applications. Unlike other works, we only attempt to determine if an application is experiencing slowdown due to scalability limitations, i.e. we do not try to predict the upward scalability of an application.

The WASH AMP scheduler [53] is one recent work that predicts application scalability online. It does so by instrumenting locks and measuring the relative time of waiting for locks against total run time. The downside is that instrumentation must be implemented for each parallel environment (WASH AMP specifically targets the Java VM), and for some environments the instrumentation may incur overhead.

3.7 Summary

Simultaneous multithreading can provide substantial benefits in the utilization of CPU resources. However, automatically predicting when SMT fails to provide the expected increase in performance for many applications is still not a well-understood area of research.

This chapter examines a methodology for SMT-level selection. At the heart of our methodology is the SMT-selection metric that is capable of predicting potential change in application performance when the SMT-level is changed. We have shown that it is very difficult to predict SMT preference by just relying on certain parameters like cache misses, branch mispredictions, number of floating point instructions, or CPI.

Our performance evaluation used a large number of multithreaded standard benchmarks that represent a wide range of applications behavior. Our results have shown that the SMT-selection metric was able to predict the correct SMT speedup in 93% of the cases on the IBM POWER7 processor, and in 86% of the cases on the Intel Nehalem processor. The metric can easily be adapted to other architectures once we have a good understanding of the issue ports and functional units used by the target architecture. We have also presented an algorithm based on the Gini impurity that can be used to accurately obtain a range of SMT-selection metric thresholds that can be used by schedulers or application optimizers.

While we tried to capture most of the factors that could impact SMT performance for a general microprocessor, the SMTsm still does not address directly some issues like instruction-level dependencies and relative execution speeds of various instruction types. SMTsm attempts to approximate such effects indirectly through the dispatch-held factor. Studying such effects is the subject of our future investigations. More future work needs to be done to increase the accuracy of prediction, to test the metric on other architectures, to improve the scalability of the metric when applied to a much larger number of cores.

Chapter 4

NUMA Traffic Management through Memory Placement[1]

As we have seen in Chapter 2, NUMA can have a huge effect on performance, and careful thread placement is crucial for performance. By choosing the correct number of nodes and the correct interconnect links, one can balance the latency and bandwidth requirements of an application. The other side of the coin to thread placement is memory placement. A complete solution must both place threads and place memory correctly. We envision that the placement algorithm in Chapter 2 would be used to place threads first, and then our algorithm for memory placement described in this chapter would be used to place memory intelligently.

Optimal performance on NUMA systems can be achieved only if we place memory in consideration of the system's physical layout and the application's characteristics. How to achieve this on modern systems with acceptable overhead is the primary research question of this chapter, and the solution to the problem, described in Section 4.3, is the main contribution.

Attribution: Mohammad Dashti, Fabien Gaud, and I jointly conducted the initial investigation into NUMA effects, including the discovery of the importance of congestion over locality (reported in Section 4.1 and Section 4.2). I designed and implemented the page-level replication mechanism described in Section 4.3.3

[1]This chapter is a modified version of work previously published in [36]

with debugging help from Fabien Gaud. Fabien Gaud designed and implemented the Carrefour algorithm.

4.1 Background

Previous work on NUMA-aware memory placement focused on maximizing locality of accesses, that is, placing memory pages such that data accesses are satisfied from a local node whenever possible. That was done to avoid very high costs of remote memory accesses. Contrary to insights from previous work, we discover that on modern NUMA systems remote wire delays, that is, delays resulting from traversing a greater physical distance to reach a remote node, are *not* the most important source of performance overhead. On the other hand, *congestion on interconnect links and in memory controllers*, which results from high volume of data flowing across the system, can dramatically hurt performance. This motivates the design of new NUMA-aware memory placement policies.

To make these statements concrete, consider the following facts. On NUMA systems circa 1990s, the time to access data from a remote node took 4-10 times longer than from a local node [98]. On NUMA systems that are built today, remote wire delays add at most 30% to the cost of a memory access [25]. For most programs, this latency differential alone would not have a substantial impact on performance. However, fast modern CPUs are able to generate memory requests at very high rates. Massive data traffic creates congestion in memory controller queues and on interconnects. When this happens, memory access latencies can become as large as 1000 cycles, from a normal latency of only around 200. Such a dramatic increase in latencies can slow down data-intensive applications by more than a factor of three. Fortunately, high latencies can be avoided or substantially reduced if we carefully place memory pages on nodes so as to avoid traffic congestion.

In response to the changes in hardware bottlenecks, we approach the problem of thread and memory placement on NUMA systems from an entirely new perspective. We look at it as the problem of *traffic management*. Our algorithm, called *Carrefour*[2], places threads and memory so as to avoid traffic hotspots and prevent

[2]Carrefour– (French) intersection, crossroads.

congestion in memory controllers and on interconnect links. This is akin to traffic management in the context of city planning: popular residential and business hubs must be placed so as to avoid congestion on the roads leading to these destinations.

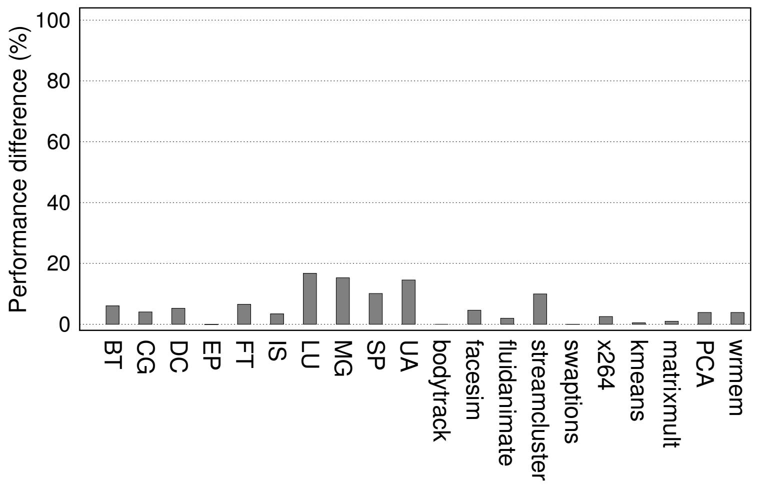
The mechanisms used in our algorithm: e.g., migration and replication of memory pages, are well understood, but the algorithm itself is new. Our algorithm makes decisions based on global observations of traffic congestion. Previous algorithms optimized for locality, and relied on local information, e.g., access pattern of individual pages. We found that in order to effectively manage congestion on modern systems we need an arsenal of techniques that go beyond optimizing locality. While locality plays a role in managing congestion (when we reduce remote accesses, we reduce interconnect traffic), alone it is not sufficient to achieve the best performance. The challenge in designing Carrefour was to understand how to combine different mechanisms in an effective solution for modern hardware.

Implementing an effective NUMA-aware algorithm on modern systems presents several challenges. Modern systems do not have the same performance monitoring hardware that was present (or assumed) on earlier systems. Existing instruction sampling hardware cannot gather the profiling data needed for the algorithm with the desired accuracy and speed. We had to navigate around this problem in our design. Furthermore, the memory latencies that we are optimizing are lower than on older systems, so we can tolerate less overhead in the algorithm.

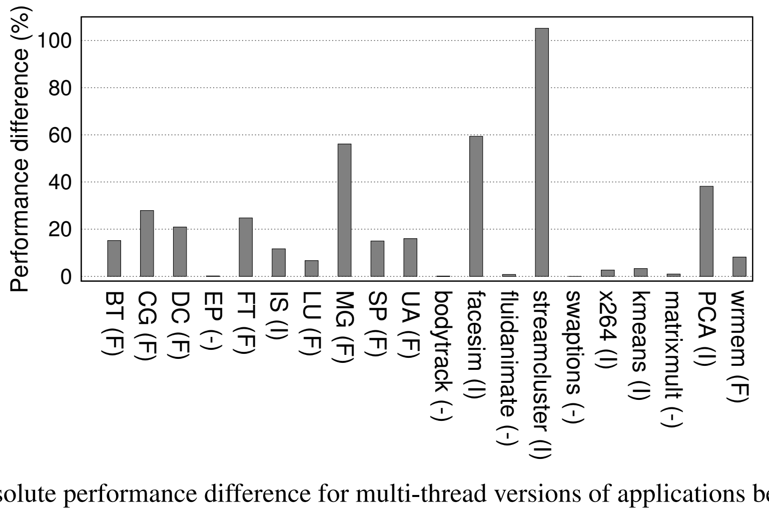
We implemented Carrefour in Linux and evaluated it with several data-centric applications: k-means clustering, face recognition, map/reduce, and others. Carrefour improves performance of these applications, with the largest gain of 3.6$\times$ speedup. When memory placement cannot be improved Carrefour never hurts performance by more than 4%. Existing NUMA-aware patches for the Linux kernel perform less reliably and in general fall short of improvements achieved with Carrefour.

4.2 Traffic Congestion on Modern NUMA Systems

In this section, we demonstrate that the effects of traffic congestion are more substantial than those of wire delays, and motivate why memory placement algorithms must be redesigned. To that end, we report data from two sets of experiments. In



(a) Performance difference for single-thread versions of applications between local and remote memory configurations.



(b) Absolute performance difference for multi-thread versions of applications between First-touch (F) and interleaving (I).

Figure 4.1: Performance difference of applications depending on the thread and memory configuration.

the first set, our goal is to measure the effects of wire delays only. We run applications in two configurations: *local-memory* and *remote-memory*. Under *local-memory*, the thread and its data are co-located on the same NUMA node; under *remote-memory*, the thread runs on a different node than its data. To ensure that wire delay is the dominant performance factor, we had to avoid congestion on memory controllers and interconnects, so we run one application at a time and use only one thread in each application. We do not include applications with CPU utilization less than 30%, because memory performance is not their main bottleneck. The experiments are run on a system described in Section 4.4 as Machine A. We use applications from the NAS, PARSEC and map/reduce Metis suites, also described in Section 4.4.

Figure 4.1(a) shows relative completion time under *remote-memory* vs. *local-memory* configuration. The performance degrades by at most 20% under *remote-memory*, which is consistent with at most 30% difference in local-vs-remote memory latencies measured in microbenchmarks [25].

In the second set of experiments, we want to observe traffic congestion, so we run each application with as many threads as there are cores. Threads are bound to their own cores and threads may access memory from any of the four NUMA nodes. We demonstrate how performance varies under two memory placement policies on Linux, as they induce different degrees of traffic congestion. The first policy is *First-touch* (F): the default policy where the memory pages are placed on the node where they are first accessed. The second policy is *Interleaving* (I), where memory pages are spread evenly across all nodes. Although these are not the only possible and not necessarily the best policies, comparing them illustrates the salient effects of traffic congestion.

Figure 4.1(b) shows the absolute difference in completion time achieved under *first-touch* and *interleaving*. The policy that performed the best is indicated in parenthesis next to the application name; a “-” is shown when the application performs equally well with either policy. We observe that the differences are often much larger than what we can expect from wire delays alone. For Streamcluster, a k-means clustering application from PARSEC, the performance varies by a factor of two depending on the memory placement policy!

To illustrate that these differences are due to traffic congestion, we show in Ta-

	Streamcluster		***PCA***	
	Best (I)	Worst (F)	Best (I)	Worst (F)
Local access ratio	25%	25%	25%	33%
Memory latency	476	**1197**	465	**660**
Mem-ctrl. imbalance	8%	**170%**	5%	**130%**
IC: imbalance, (avg)	22% (59%)	**85%** (33%)	20% (48%)	**68%** (31%)
L3MPKI	16.85	16.89	7.35	7.4
IPC	0.29	**0.15**	0.52	**0.36**

Table 4.1: NUMA traffic congestion effects

ble 4.1 some supporting data for the two applications, *Streamcluster* and *PCA* (a map/reduce application):

Local access ratio: The percent of all memory accesses sourced from a local node.

Memory latency: The average number of cycles to satisfy a memory request from any node.

Memory controller imbalance: The standard deviation of the load across all memory controllers, expressed as percent of the mean. Load is measured as the number of requests per time unit.

Average interconnect (IC) usage: The utilized interconnect bandwidth as percent of total, averaged across all links.

Interconnect (IC) imbalance: The standard deviation of utilization across the links as percent of mean utilization.

L3MPKI: The number of last-level (L3) cache misses per thousand instructions.

IPC: The number of instructions per cycle.

The data in Table 4.1 leads to several curious observations. First, we see that locality of memory accesses either does not change regardless of the memory management policy, or *decreases* under the better performing policy. For *Streamcluster*, most of the memory pages happen to be placed on a single node under *first-touch* (because a single thread initializes them at the beginning of the program). Under *interleaving* the pages are spread across all nodes, but since the threads access data from all four nodes, the overall access ratio is about the same in both configurations. For *PCA*, interleaving decreases the local access ratio and yet *increases* performance. So the first surprising conclusion is that ***better locality does***

not necessarily improve performance!

And yet, the IPC substantially improves (2× for *Streamcluster* and 41% for *PCA*), while the L3 miss rate, as well as L1 and L2 miss rates, remain unchanged. The explanation emerges if we look at the memory latency. Under interleaving, the memory latency reduces by a factor of 2.48 for *Streamcluster* and 1.39 for *PCA*. This effect is entirely responsible for performance improvement under the better policy. The question is, *what is responsible for memory latency improvements*? It turns out that interleaving *dramatically reduces memory controller and interconnect congestion* by alleviating the load imbalance and mitigating traffic hotspots. Rows 5, 6 in Table 4.1 show significant reductions in imbalance under interleaving, and Figure 4.2 illustrates these effects visually for *Streamcluster*. So even without improving locality (we even *reduce* it for *PCA*), we are able to substantially improve performance. And yet, existing NUMA-aware algorithms disregarded traffic congestion, optimizing for locality only. Our work addresses this shortcoming.

Although the two selected applications performed significantly better under interleaving, this does not mean that interleaving is the only desired policy on modern NUMA hardware. In fact, as Figure 4.1(b) shows, many NAS applications fared a lot worse with interleaving. In the process of designing the algorithm we learned that a range of techniques — interleaving, page replication and co-location — must be judiciously applied to different parts of the address space depending on global traffic conditions and page access patterns. So the challenge in designing a good algorithm is understanding when to apply each technique, while navigating around the challenges of obtaining accurate performance data and limiting the overhead.

4.3 Design and Implementation

We begin by describing the mechanisms composing the algorithm: *page co-location*, *interleaving*, and *replication*. Then we explain how they fit together.

4.3.1 The Mechanisms

Page co-location is when we re-locate the physical page to the same node as the thread that accesses it. Co-location works well for pages that are accessed by a single thread or by threads co-located on the same node.

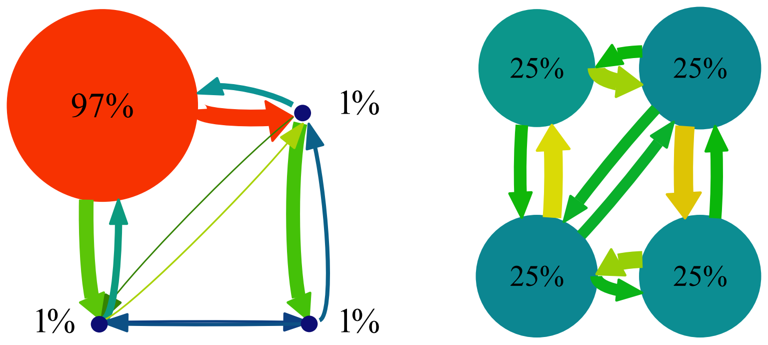


Figure 4.2: Traffic imbalance under first-touch (left) and interleaving (right) for *Streamcluster*. Nodes and links bearing the majority of the traffic are shown proportionately larger in size and in brighter colors. The percentage values show the fraction of memory requests destined for each node. The figure is drawn to scale.

Page interleaving is about evenly distributing physical pages across nodes. Interleaving is useful when we have imbalance on memory controllers and interconnect links, and when pages are accessed by many threads. Operating systems usually provide an interleaving allocation policy, but only give an option to enable or disable it globally for the entire application. We found that interleaving works best when judiciously applied to parts of the address space that will benefit from it.

Page replication is about placing a copy of a page on several memory nodes. Replication distributes the pressure across memory controllers, alleviating traffic hotspots. An added bonus is eliminating remote accesses on replicated pages. When done right, replication can bring very large performance improvements. Unfortunately, replication also has costs. Since we keep multiple copies of the same page, we must synchronize their contents, which is like running a cache coherency protocol in software. The costs can be very significant if there is a lot of fine-grained read/write sharing. Another potential source of overhead is the synchronization of page tables. Since modern hardware walks page tables automatically, page tables themselves must be replicated across nodes and kept in sync. Finally, replication increases the memory footprint. We should avoid it for workloads with large memory footprints for fear of increasing the rate of hard page faults.

Global statistics	
MC-IMB	Memory controller imbalance (as defined in Section 4.2)
LAR	Local access ratio (as defined in Section 4.2)
MAPTU	Memory (DRAM) accesses per time unit (microsecond)
Per-application statistics	
MRR	Memory read ratio. Fraction of DRAM accesses that are reads
CPU%	Percent CPU utilization
Per-page statistics	
Number of accesses	The number of sampled data loads that fell in that page
Access type	Read-only or read-write

Table 4.2: Statistics collected for the algorithm.

4.3.2 The Algorithm

Our memory management algorithm has three components: *measurement*, *global decisions* and *page-local decisions*. The measurement component continuously gathers various metrics (Table 4.2) that later drive page placement decisions. Global and per-application metrics are collected using HPEs with very low overhead. Per-page statistics are collected via instruction-based sampling (IBS) [40], which can introduce significant overheads at high sampling rates. Section 4.3.3 describes how we keep the overheads at bay. Global decisions are based on system-wide traffic congestion and workload properties which determine what mechanisms to use. Page-local decisions examine access patterns of individual pages to decide their fate.

Global Decisions

The global decision-making process is outlined in Figure 4.3.

Step 1: We decide whether to enable Carrefour. We only want to run Carrefour for applications that generate substantial memory traffic. Other applications would not be affected by memory placement policies, so there is no reason to subject them to sampling overhead. This decision is driven by the application's memory access rate (MAPTU – see Table 4.2). Carrefour is enabled for applications with the MAPTU above a certain threshold. The MAPTU threshold is to be determined experimentally and the right setting may vary from system to system.

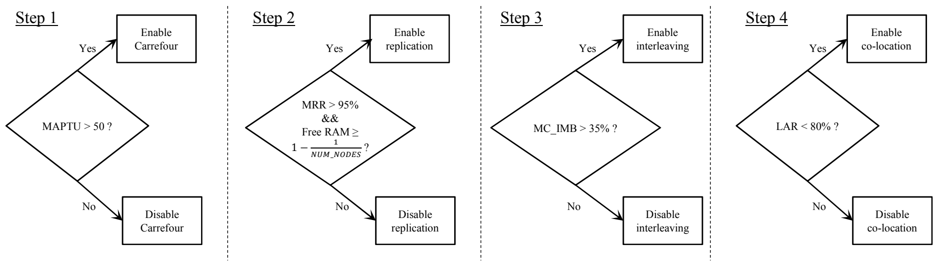


Figure 4.3: Global decisions in Carrefour.

We found the threshold of 50 MAPTU worked well on all hardware we evaluated, and the performance was not very sensitive to its choice. To determine the right MAPTU threshold on a system very different from ours, we recommend running a benchmark suite under different NUMA policies, noting which applications are affected and using the lowest observed MAPTU from those experiments.

Once we decided whether there is sufficient memory traffic to justify running Carrefour, we need to decide which of the available mechanisms, replication, interleaving and co-location, should be enabled for each application given its memory access patterns. The goal here is to choose the most beneficial techniques and avoid any associated overhead. The next three steps take care of this decision.

Step 2: We decide whether it is worthwhile to use replication. Replication risks introducing significant overheads if it forces us to run out of RAM (and causes additional hard page faults) or requires frequent synchronization of pages across nodes (see more discussion in Section 4.3.3). To avoid the first peril, we conservatively enable replication only if there is sufficient free RAM to replicate the entire resident set. That is, the fraction of free RAM must be at least $1 - \frac{1}{NUM\_NODES}$. This is a conservative threshold, because not all pages will be replicated, and not all resident pages will be accessed frequently enough to generate significant page fault overhead if evicted. Evaluating the trade-off between replication benefit and potentially increased page-fault rate was outside the scope of the work. This requires workloads that both benefit from replication and have very large memory-resident sets, which we did not encounter in our experiments.

To avoid the overhead associated with the synchronization of page content across nodes, we do not replicate pages that are frequently written. An application must have the memory read ratio (MRR) of at least 95% in order for its memory pages to be considered for replication[3]. The setting of this parameter can have a very significant effect on performance. While we found that the performance was not sensitive when we varied the parameter in the range of 90-99%, it is always safe to err on the high side.

Step 3: We decide whether to use interleaving. Interleaving improves performance if we have large memory controller imbalance. We enable interleaving if memory controller imbalance is above 35%, but found that the performance was not highly sensitive to this parameter. Applications that benefit from interleaving usually begin with a very large imbalance.

Step 4: We decide whether or not to enable co-location. Co-location will be triggered only for pages that are accessed from a single node, and so it will not exacerbate the imbalance if memory-intensive threads are evenly spread across nodes. Therefore, we enable co-location if the local access rate is slightly less than ideal (LAR $< 80\%$). Performance is not highly sensitive to this parameter; we observed that if this parameter is completely eliminated from the algorithm and co-location is always enabled then the largest performance impact is only a few percent.

Although we expect that optimal settings for the parameters used in the algorithm would vary from one system to another, we found that we did not need to adjust the settings when we moved between the two experimental systems used in our evaluation. Although our systems had the same number of nodes and both used AMD CPUs, they differed in the number of cores per node, the cache-coherency protocol (broadcast vs. directory-based), and one had a higher interconnect throughput than the other. Therefore, it is possible that the algorithm parameters settings are rather stable across all but drastically different systems.

[3]MRR is approximated as fraction of L1 refills from DRAM in modified state, because there is no HPE that provides this quantity precisely per core, as opposed to per-node. Similarly, due to HPE limitations described in Section 4.3.3, it is very difficult to measure the MRR per page. That is why we use the MRR for the entire application.

Page-local Decisions

Carrefour makes page-local decisions depending on the mechanisms enabled: e.g., pages are only considered for replication if replication is enabled for that application. The following explanation assumes that all three mechanisms are enabled.

To decide the fate of each page, we need at least two memory-access samples for that page. If the page was accessed from only a single node we migrate it to that node. If the page is accessed from two or more nodes, it is a candidate for either interleaving or replication. If the accesses are read-only, the page is replicated. Otherwise it is marked for interleaving. To decide where to place a page marked for interleaving, we use two probabilities: $P_{migrate}$ and P_{node}. $P_{migrate}$ determines the likelihood of migrating the page away from the current node. $P_{migrate}$ is the MAPTU of the current node as the fraction of MAPTU on all nodes, so the higher the load on the current node relative to others, the higher the chance that we will migrate a page. P_{node} gives us the probability of migrating a page to a particular node, and it is the complement of $P_{migrate}$ for that node, so Carrefour will migrate the page to the least loaded node.

4.3.3 Implementation

We implemented Carrefour in the Linux kernel 3.6.0. Carrefour measures the selected performance indicators, and with periodicity of one second makes decisions regarding page placement and resets statistic counters. To a large extent, Carrefour relies on well-understood mechanisms in the Linux kernel, such as physical page migration. The non-trivial aspects of the implementation were understanding how to accomplish fast and accurate sampling of memory accesses and navigating around the overheads of replication. We describe how we overcame these challenges in the two sections that follow.

Fast and Accurate Memory Access Sampling

A crucial goal of the algorithm is to quickly and accurately detect memory pages that cause the most DRAM accesses, and accurately estimate the read/write ratio of those pages. To that end, we used Instruction-Based Sampling (IBS): hardware-supported sampling of instructions available in AMD processors. Intel processors support similar functionality in the form of PEBS: Precise Event-Based Sampling. IBS can be configured to deliver instruction samples at a desired interval, e.g., after expiration of a certain number of cycles or micro-ops. Each sample contains detailed information about the sampled instruction, such as the address of the accessed data (if the instruction is a load or a store), whether or not it missed in the cache and how long it took to fetch the data. Unfortunately, every delivered sample generates an interrupt, so processing samples at a high rate becomes very costly. Other systems that relied on IBS performed off-line profiling [57, 75], so they could tolerate much higher overhead than what would be acceptable in our online algorithm.

After experimenting with IBS on our systems, we found that for most applications the sampling interval of 130,000 cycles incurs a reasonable overhead of less than 5%. The desired sampling rate can be trivially derived for new systems: it amounts to experimenting with different sampling rates and settling for the one that generates acceptable runtime overhead.

Our initial decision was to filter out all the samples that did not generate a DRAM access. However, we found that the resulting number of samples was extremely low. Even very memory-intensive workloads access DRAM only a few times for every thousand instructions. That, combined with a low IBS sampling frequency, gave us the sampling rate of less than one hundred thousandth of a percent, and made it very difficult to generate a sufficient number of samples. Furthermore, filtering samples that did not access DRAM caused us to miss the accesses generated by the hardware prefetcher. These accesses are not part of any instruction so they will not be tagged by IBS. For prefetch-intensive applications, we obtain a very small number of samples and a very distorted read-write ratio.

To address this problem, we used two solutions. First, is the adaptive sampling rate. When the program begins to run, we sample it at a relatively high rate of 1/65,000 cycles. If after this measurement phase we take fewer than ten actions in the algorithm (an action is any change in page placement) we switch to a much lower rate of 1/260,000 cycles, which has a negligible performance impact. Otherwise we continue sampling at the high rate.

The second solution was, when filtering IBS samples, to retain not just the data samples that accessed the DRAM, but those that hit in the first-level cache as well.

First-level cache loads include accesses to prefetched data, so we avoid prefetcher-related inaccuracy. On the one hand, considering cache accesses can introduce "noise" in the data, because we could be sampling pages that never access DRAM. On the other hand, Carrefour is only activated for memory-intensive applications, and for them there is a higher correlation between the accesses that hit in the cache and those that access DRAM.

With these two solutions combined, we were able to successfully identify the pages that are worth replicating, while this was nearly impossible prior to introducing these solutions. For example, for *Streamcluster* we used to be able to detect only a few percent of the pages that are worth replicating, but with these solutions in place, we were able to identify 100% of them[4].

However, even though performance became much better (we were able to speed up *Streamcluster* by 26% relative to the default kernel), we were still far from the "ideal" manual replication, which sped it up by more than $2.5\times$. To approach ideal performance, we had to mitigate the overheads of replication, which we describe next.

Replication

Replication has overhead from the following three sources. First, there is the initial set-up cost and slightly more expensive page faults. Modern hardware walks page tables automatically, so a separate copy of a page table must be created for each node. Page faults become slightly more costly, because a new page table entry must be installed on every node. To avoid these costs when we are not likely to benefit from replication, we avoid replication unless the applications has at least a few hundred pages marked for replication[5].

The second source of overhead comes from additional hard page faults if we exceed the physical RAM capacity by replicating pages. As explained earlier, we avoided this overhead by conservatively setting the free memory threshold when enabling replication.

The final and most significant source of overhead stems from the need to syn-

[4]*Streamcluster* holds shared data in a single large array, so it is trivial to detect which data is worth replicating and implement a manual solution to use as the performance upper-bound.

[5]We use the threshold of at least 500 pages. Performance is not highly sensitive to this parameter.

chronize the contents of replicated pages when they are written. This involves a physical page copy and is very costly. Before explaining how we avoid this overhead we provide a brief overview of our implementation of replication.

In Linux, a process address space is represented by a `mm_struct`, which keeps track of valid address space segments and holds a pointer to the page table, which is stored as a hierarchical array. Since modern hardware walks page tables automatically, we cannot modify the structure of the page table to point to several physical locations (one for each node) for a given virtual page. Instead, we must maintain a separate copy of the page table for each node and synchronize the page tables when they are modified, even for virtual pages that are not replicated. Linux dictates that the page table entry (PTE) be locked when it is being modified. We do not make any changes to this locking protocol. The only difference is that we designate one copy of the page table as the *master copy*, and only lock the PTE in the master copy while installing the corresponding PTEs into all other replicas.

When a page is replicated, we create a physical copy on every memory node that runs threads from the corresponding application. We install a different virtual-to-physical translation in each node's page table. We write-protect the replicated page, so when any node writes that page we receive a page protection fault. To handle this fault, we read-protect the page on all nodes except the faulting one, and enable writing on the faulting node. If another node accesses that page, we must copy the new version of the page to that node, enable the page for reading and protect it from writing.

We refer to all the actions needed to keep the pages synchronized as *page collapses*. Collapses are extremely costly, and would occur if we replicate a page that is write-shared. Even with very infrequent writes (e.g., one in 1000 accesses), the collapse overhead could be prohibitively high. With limited capabilities of IBS, we are unable to detect read/write ratio on individual pages with sufficient accuracy. That is why we use the application-wide MRR and disable replication for the entire application if the MRR is low. Furthermore, we monitor collapse statistics of individual pages and disable replication for any page that generated more than five collapses. We allow five collapses because this enables applications to perform some writes for initialization or a phase-change without losing the benefits of replication, but any page with regular (even if infrequent) writes will have replication disabled because the cost of page collapses will almost always outweigh the benefits of replication.

With these optimizations, as well as those described in Section 4.3.3, we were able to avoid replication costs for the applications we tested and approached within 10% the performance of manual replication for *Streamcluster*.

4.4 Evaluation

In this section, we study the performance of Carrefour on a set of benchmarks. The main questions that we address are the following:

1. *How does Carrefour impact the performance of applications, including those that cannot benefit from its heuristics?*
2. *How does Carrefour compare against existing heuristics for modern NUMA hardware?*
3. *How well does Carrefour leverage the different memory placement mechanisms?*
4. *What is the overhead of Carrefour?*

To assess the performance of Carrefour, we compare it against three other configurations:

Linux: A standard Linux kernel with the default first-touch memory allocation policy.

Manual interleaving: A standard Linux kernel with the interleaving policy manually enabled for the application.

AutoNUMA: A recent Linux patchset [34] considered as the best thread and memory management algorithm available for Linux.

The rest of the section is organized as follows: we first describe our experimental testbed. Next, we study single-application scenarios, followed by workloads with multiple co-scheduled applications. We then detail the overhead of Carrefour and conclude with a discussion on additional hardware support that would improve Carrefour's operation.

4.4.1 Testbed

We used two different machines for the experiments:
Machine A has four 2.3GHz AMD Opteron 8385 processors with 4 cores in each (16 cores in total) and 64GB of RAM. It features 4 nodes (i.e., 4 cores and 16GB of RAM per node) interconnected with HyperTransport 1.0 links.
Machine B has four 2.6GHz AMD Opteron 8435 processors with 6 cores in each (24 cores in total) and 64GB of RAM. It features 4 nodes (i.e., 6 cores and 16GB of RAM per node) interconnected with HyperTransport 3.0 links.

We used Linux kernel v3.6 for all experiments. For the AutoNUMA configuration, we used AutoNUMA v27 and disabled PMD scan because we it decreases performance on all applications we measured[6].

Workloads were configured to use one thread per core available on the test system and threads were bound to their own cores.

We used the following set of applications: the PARSEC benchmark suite v2.1 [20], the FaceRec facial recognition engine v5.0 [18], the Metis MapReduce benchmark suite [65] and the NAS parallel benchmark suite v3.3 [11]. PARSEC applications run with the native workload. From the available workloads in NAS we chose those with the running time of at least ten seconds. We excluded applications whose CPU utilization was below 33%, because they were not affected by memory management policies. We also excluded applications using shared memory across processes (as opposed to threads) or memory-mapped files, because our replication mechanism does not yet support this behaviour. For *FaceRec*, we used two kinds of workloads: a short-running one and a long running one (named *FaceRecLong* in the remainder of the paper). The reason why we present two different workloads is to show that Carrefour is able to successfully handle very short workloads (less than 4s on machine B when running with Carrefour). Each experiment was run ten times. Overall, we observed a standard deviation between 1% and 2% for the *Linux*, *Manual interleaving* and Carrefour configurations. *AutoNUMA* has a more significant standard deviation, up to 9% on machine A and 13% on machine B.

[6]We also tested AutoNUMA v28. The performance results are very similar. However, we observed a significantly higher standard deviation (up to 18% on machine A and 23% on machine B) which makes profiling very difficult.

4.4.2 Single-Application Workloads

Performance comparison. Figures 4.4 and 4.5 show the performance improvement relative to default Linux obtained under all configurations for machine A and machine B. Performance improvement is computed as:

$$\frac{DefaultLinux_{time} - System_{time}}{System_{time}} * 100\%,$$

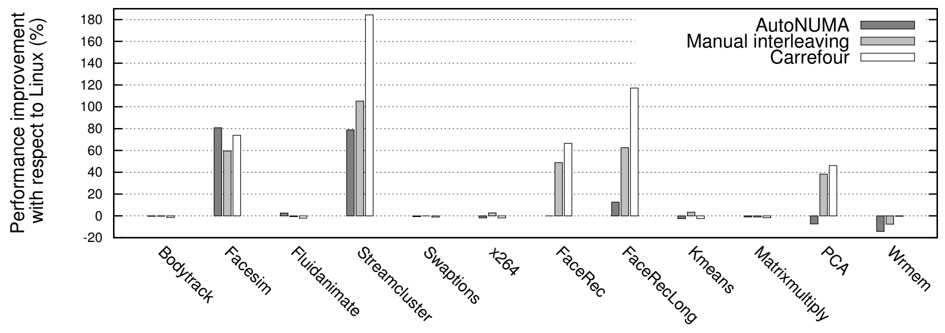
where *System* can be either Carrefour, Manual interleaving or AutoNUMA.

We can make two main observations. First, Carrefour almost systematically outperforms default Linux, AutoNUMA, and Manual interleaving, sometimes quite substantially. For instance, when running *Streamcluster* or *FaceRecLong*, we observe that Carrefour is up to 58% faster than Manual interleaving, up to 165% faster than AutoNUMA, and up to 263% faster than default Linux on machine B. Second, we observe that, unlike other techniques, Carrefour never performs significantly worse than default Linux: the maximum performance degradation over Linux is below 4%. In contrast, AutoNUMA and Manual interleaving cause performance degradations of up to 25% and 38% respectively. Additionally, Manual Interleaving has a very irregular impact on performance. While it does fairly well for PARSEC, NAS applications suffer significant performance degradation (up to 38%) when run with manual interleaving.

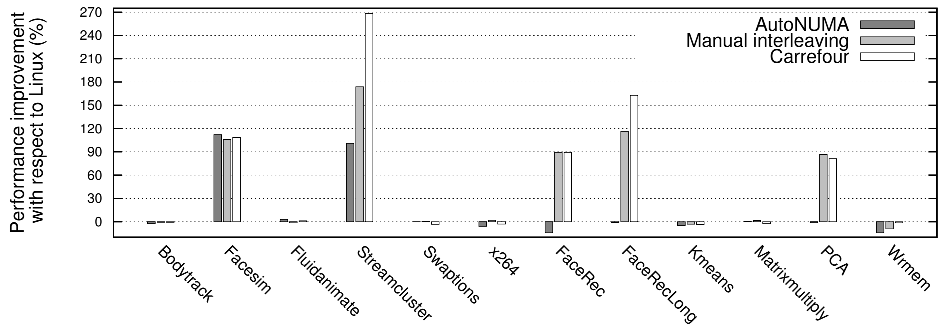
IS is an exception among the NAS benchmarks: Manual interleaving improves its performance, while Carrefour does no better than default Linux. This is because IS suffers from very short imbalance bursts that we are not able to correct due to limited sampling accuracy achievable with low overhead using existing HPEs. Section 4.4.6 discusses hardware support that would help us address this problem.

In order to understand the reasons for the performance impact of the different policies, we study in detail a set of applications whose performance is improved the most by Carrefour. To that end, we present several metrics: the load imbalance on memory controllers (Figure 4.6(a)), the load imbalance on interconnect links (Figure 4.6(b)), the average memory latency (Figure 4.7(a)) and the local access ratio (Figure 4.7(b)).

We draw the following observations. First, Carrefour much better balances



Machine A



Machine B

Figure 4.4: PARSEC/Metis: AutoNUMA, Manual interleaving and Carrefour vs.Default Linux.

the load on both memory controllers and interconnect links than Linux and AutoNUMA. Not surprisingly, Manual interleaving is also very good at balancing the load. Nevertheless, we observe in Figure 4.7(a) that Carrefour induces lower average memory latencies than Manual interleaving, which explains its better performance. To understand why Carrefour reduces memory latencies we refer to Figure 4.7(b), which shows that Carrefour not only balances the load on memory controllers and interconnect links, but also often induces a much higher ratio of local memory accesses than other techniques. This is a consequence of Carrefour's judiciously applying the right techniques (interleaving, replication or co-location) in places where they are beneficial. Interleaving mostly balances the load; replication and co-location in addition to balancing the load improve the local access ratio. Better locality improves latencies in two ways: it avoids remote wire delays

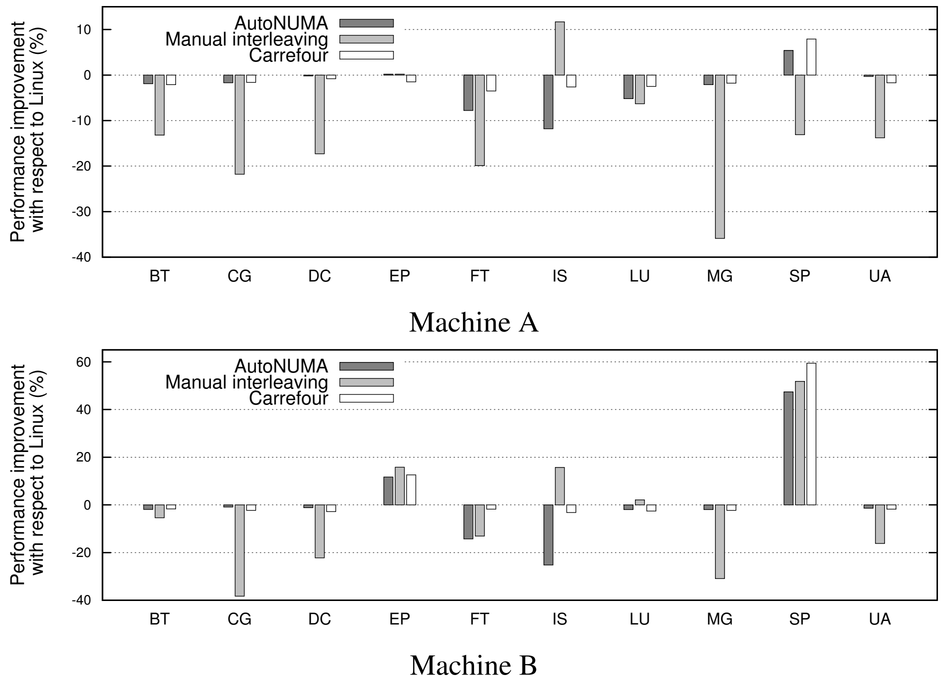
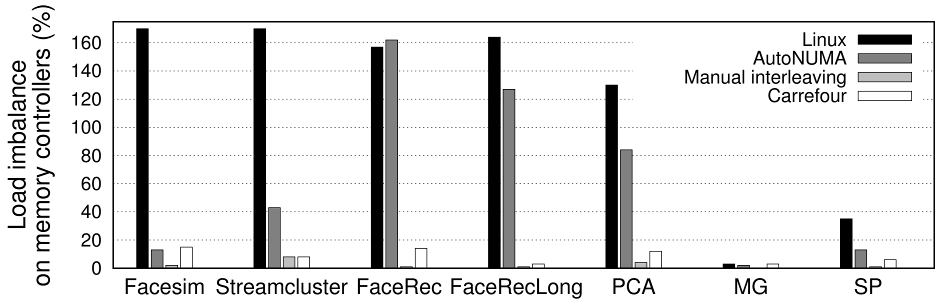


Figure 4.5: NAS: AutoNUMA, Manual interleaving and Carrefour vs. Default Linux.

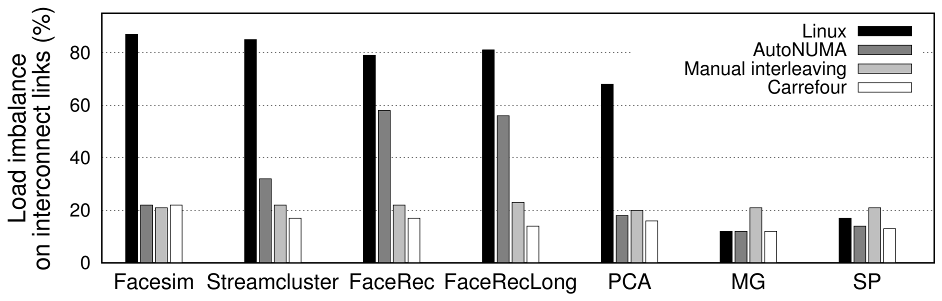
and, most importantly, decreases congestion on the interconnect links.

We also study MG as a representative example of the NAS applications, for which Carrefour does not bring significant improvements over default Linux (but still performs better than AutoNUMA and Manual interleaving in most cases). MG has a low imbalance and a very good local access ratio to begin with. That is why Manual interleaving has a very bad impact on such workloads, significantly decreasing the local access ratio and as a result stressing the interconnect.

Looking inside Carrefour. To better understand the behavior of Carrefour, we show in Table 4.3 the number of replicated pages, the number of interleaved pages, and the number of co-located pages for the chosen benchmarks. These numbers provide a better insight into how Carrefour manages the memory. We observe that *all* three memory placement mechanisms are in use, and that most applications rely on two or three techniques. As discussed previously, MG does not suffer from



(a) Memory controllers

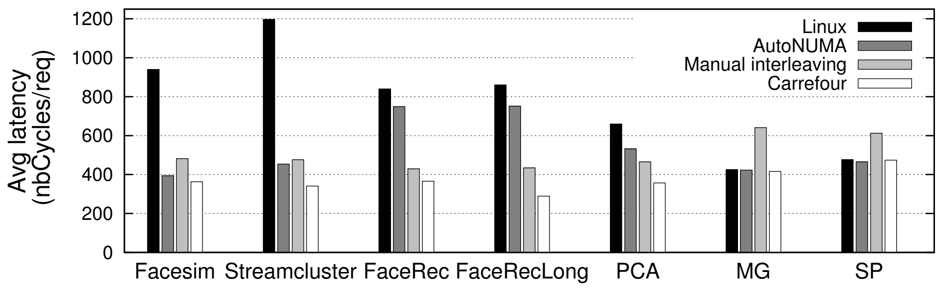


(b) Interconnect links

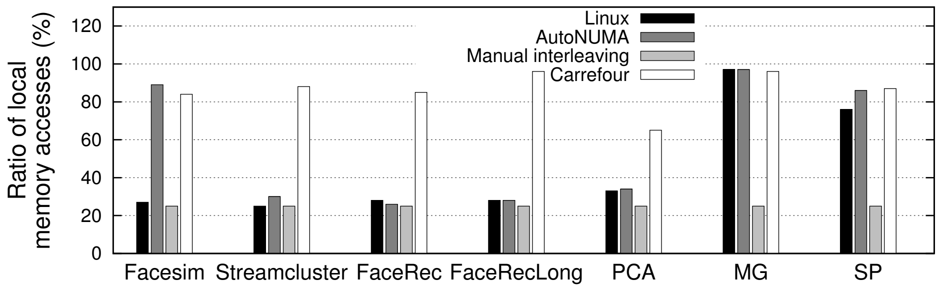
Figure 4.6: Load imbalance for selected single-application benchmarks (machine A).

traffic congestion, so Carrefour does not enable any technique for this application.

A legitimate question we can ask is whether Carrefour always selects the best technique. In Table 4.4, we report the performance improvement over Linux obtained when running a full-fledged Carrefour and when running a reduced version of Carrefour enabling only one technique at a time. We observe that Carrefour systematically selects the best technique. It is also interesting to remark how different techniques work together and how the numbers provided here echo those in Table 4.3. We notice that, for all the studied applications except MG and SP, the combination of several techniques employed by Carrefour outperforms any single technique, even when a given technique has a dominant impact (e.g., for Streamcluster). The slight performance degradation for MG corresponds to the monitoring overhead of Carrefour.



(a) Average memory latency



(b) Local memory access ratio

Figure 4.7: DRAM latency and locality for selected single-application benchmarks (machine A).

	No. replicated pages	No. interleaved pages	No. migrated pages
Facesim	0	431	10.1k
Streamcluster	25.4k	14.5k	858
FaceRec	4k	3	1.3k
FaceRecLong	4.1k	5	1.4k
PCA	31k	33	41.3k
MG	0	0	1
SP	0	305	1.7k

Table 4.3: Number of memory pages that are replicated, interleaved and co-located on single-application workloads (machine A).

	Carrefour	Replication	Interleaving	Co-location
Facesim	74%	-4%	0%	65%
Streamcluster	184%	176%	94%	51%
FaceRec	66%	61%	32%	1%
FaceRecLong	117%	113%	51%	1%
PCA	46%	45%	29%	24%
MG	-2%	-2%	-2%	-2%
SP	8%	-1%	-7%	8%

Table 4.4: Performance improvement over Linux when running Carrefour and the three different techniques individually on single-application workloads (machine A).

4.4.3 Multi-Application Workloads

In this section, we study how Carrefour behaves in the context of workloads with multiple applications that are co-scheduled on the same machine. The goal is to assess that Carrefour is able to work on complex access patterns and to make the distinction between the diverse requirements of different applications. We consider several scenarios based on some of the applications previously studied in Section 4.4.2: (i) *MG + Streamcluster*, (ii) *PCA + Streamcluster*, and (iii) *FaceRecLong + Streamcluster*. We chose these scenarios because they exhibit interesting patterns, which require combining several memory placement techniques in order to achieve good performance.

Each application is run with half as many threads as the number of cores (i.e., 8 threads on machine A, 12 on machine B). With two applications, the workload occupies all the available cores. The threads of each application are clustered on the same node, so each application uses all the cores on two of the four nodes on a machine.

Figure 4.8 shows, for each workload, the performance improvement with respect to Linux for AutoNUMA, Manual interleaving and Carrefour on machine A and machine B. We observe that Carrefour always outperforms AutoNUMA and Manual interleaving, by up to 62% and 36% respectively. Besides, Carrefour also outperforms default Linux, while Manual interleaving hurts MG with a 25% slowdown. Overall, the results obtained with Manual interleaving are closer to the ones

	Carrefour	Replication	Interleaving	Co-location
MG + Streamcluster	2% / 71%	2% / 73%	-5% / 17%	-1% / 6%
PCA + Streamcluster	24% / 57%	18% / 57%	8% / 5%	14% / 1%
FaceRecLong + Streamcluster	53% / 71%	53% / 71%	12% / 9%	12% / 4%

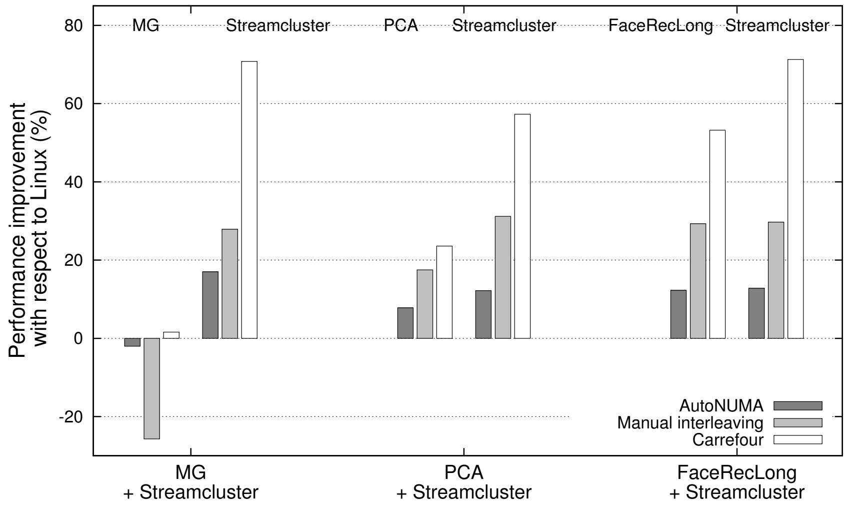
Table 4.5: Multi-application workloads: performance improvement over Linux when running Carrefour and the three different techniques individually (machine A).

of Carrefour compared to the other setups.

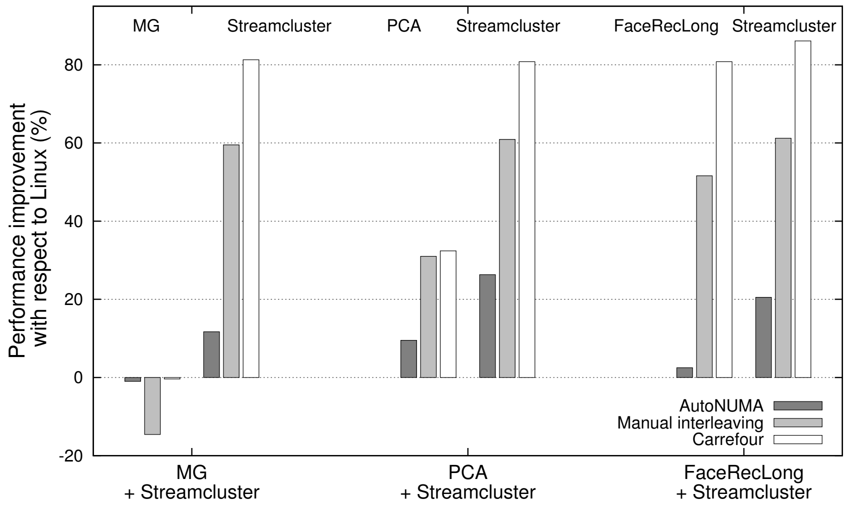
The reason why Manual interleaving performs relatively well in these scenarios is because, with each application using two domains, there is a lot less cross-domain traffic than in the single-application case. Hence there are fewer problems that need to be fixed and there is a smaller discrepancy between the performance of different memory management algorithms.

To explain the results, we show the same detailed metrics as in Section 4.4.2: load imbalance on memory controllers, load imbalance on interconnect links, average DRAM latency and ratio of local DRAM accesses in Figures 4.9, 4.10, 4.11 and 4.12 respectively. The metrics are aggregated for the two applications of each workload. As was the case with the single-application workloads, we see that Carrefour systematically improves the latency of the studied workloads for two reasons: a more balanced load on memory controllers and interconnect links as well as an improved locality for DRAM accesses. Note that, for each workload, the depicted latencies are averaged over the two applications. This explains the small latency variations between Linux, AutoNUMA and Manual Interleaving in the case of MG + Streamcluster.

Finally, we show in Table 4.5 the performance improvement for each application when the effects of only one of the techniques are enabled. We observe that the multi-applications workloads perform as well or better with an arsenal of techniques used in Carrefour rather than with any single technique alone (especially for PCA + Streamcluster).



Machine A



Machine B

Figure 4.8: Multi-application workloads: AutoNUMA, Manual interleaving and Carrefour vs. Default Linux.

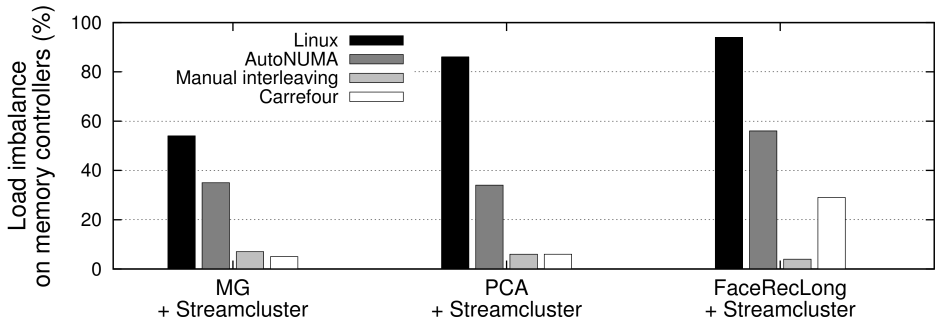


Figure 4.9: Multi-application workloads: load imbalance on memory controllers (machine A).

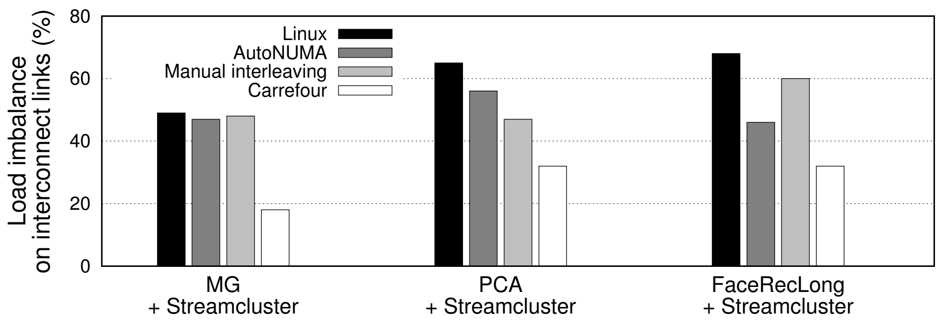


Figure 4.10: Multi-application workloads: load imbalance on interconnect links (machine A).

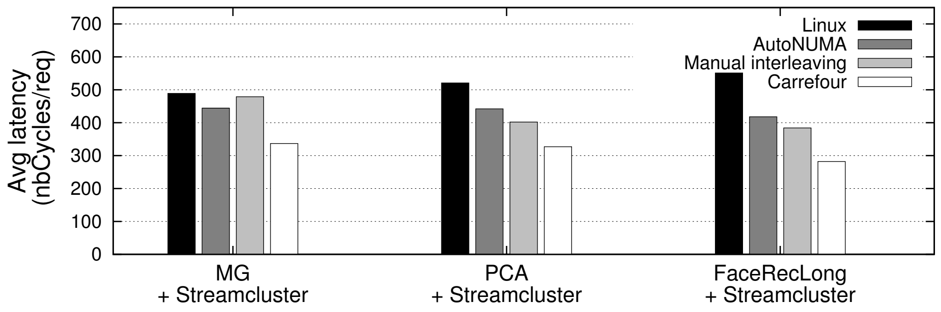


Figure 4.11: Multi-application workloads: average memory latency (machine A).

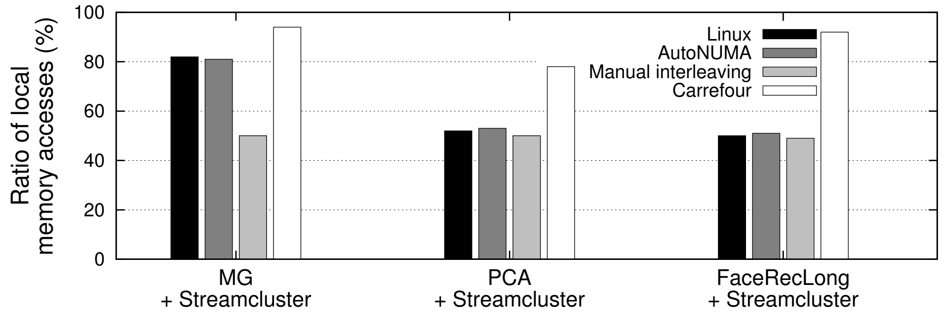


Figure 4.12: Multi-application workloads: local memory access ratio (machine A).

4.4.4 Overhead

Carrefour incurs CPU and memory overhead. The first source of CPU overhead is the periodic IBS profiling. To measure CPU overhead, we compared performance of Carrefour with Linux on those applications where Carrefour does not yield any performance benefits. We observed the overhead between 0.2% and 3.2%. The adaptive sampling rate in Carrefour is crucial to keeping this overhead low. A second and potentially significant source of CPU overhead is replication, if we perform a lot of collapses. A single collapse costs a few hundred microseconds when it occurs in isolation. Parallel collapses can take a few milliseconds because of lock contention. That is why it is crucial to avoid collapses and other synchronization events by disabling replication for write-intensive workloads, as is done in Carrefour.

The first source of memory overhead is the allocation of data structures to keep track of profiling data. This overhead is negligible: e.g., 5MB on Machine A with 64GB of RAM. Carrefour's data structures are pre-allocated on startup to avoid memory allocation during the runtime. We limit the number of profiled pages to 30,000 to avoid the cost of managing dynamically sized structures. The second source of memory overhead is memory replication. When enabled, replication introduces a memory footprint overhead of 400MB (353%), 60MB (210%), 60MB (126%) and 614MB (5%) for Streamcluster, FaceRec, FacerecLong and PCA respectively.

4.4.5 Impact on Energy Consumption

It was observed that remote memory accesses require significantly more energy than local ones [35]. Since Carrefour may both decrease and increase the number of remote memory accesses, we were interested in evaluating its impact on energy consumption[7]. We show the results for selected applications from single-application workloads on Machine A. We report the increase in energy consumption as well as the increase in completion time of all configurations compared to default Linux in Figure 4.13. Completion time increase is computed here as:

$$\frac{System_{time} - DefaultLinux_{time}}{DefaultLinux_{time}} * 100\%.$$

There is a strong relationship between the completion time and the energy consumption: if the completion time is decreased, the energy consumption is also decreased proportionally. As a result, Carrefour saves up to 58% of energy. When no traffic management is needed, Carrefour on its own has a low impact on energy consumption (e.g., 2% on MG).

More generally, we found that the increase of remote memory accesses has little or no impact on the global energy consumption of the machine. For example, Manual interleaving drops the local access ratio of MG from 97% to 25% and thus proportionally increases the number of remote accesses. However, the energy consumption increase is slightly lower that the completion time increase, which indicates that the extra energy overhead of remote memory accesses have no strong impact on overall energy consumption.

4.4.6 Discussion: Hardware Support

We have shown that a traffic management system like Carrefour can bring significant performance benefits. However, the challenge in building Carrefour was the need to navigate around the limitations of the performance monitoring units of our hardware as well as the costs of replicating pages. In this section, we draw some insights on the features that could be integrated into future machines in order to further mitigate the overhead and improve accuracy, efficiency and performance of

[7] We used IPMI which gives access to the current global power consumption on our servers.

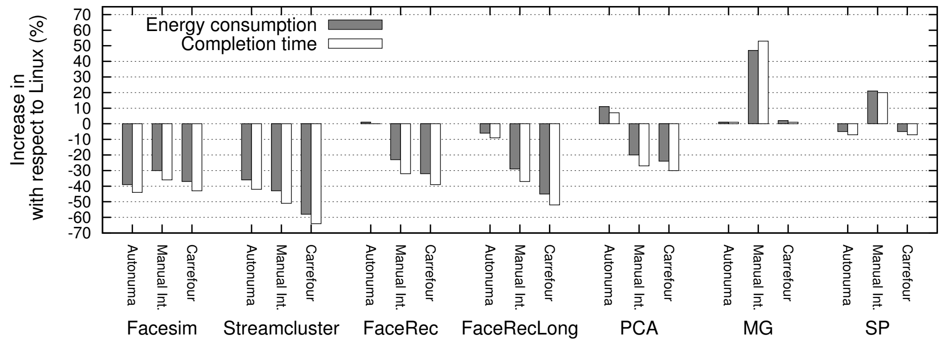


Figure 4.13: Increase in completion time and energy consumption for selected single-application benchmarks with respect to Linux (machine A). Lower is better.

traffic management algorithms.

First, Carrefour would benefit from hardware profiling mechanisms that sample memory accesses with high precision and low overhead. For instance, it would be useful to have a profiling mechanism that accumulates and aggregates page access statistics in an internal buffer before triggering an interrupt. In this regard, the AMD Lightweight Profiling [38] facility seems a promising evolution of profiling hardware[8], but we believe the hardware should go even further, and not only accumulate the samples but be configured to aggregate them according to user needs, to reduce the number of interrupts even further.

Second, Carrefour would benefit from dedicated hardware support for memory replication. We believe that there should be interfaces allowing the operating system to indicate to the processor which pages to replicate. The processor would then be in charge of replicating the pages on the nodes accessing it and maintaining consistency between the various replicas (in the same way as it maintains consistency for cache lines). Given that maintaining consistency between frequently written pages is costly, we believe that such processors should also be able to trigger an interrupt when a page is written too frequently. The OS would then decide to keep the page replicated or to revert the replication decision.

[8]Unfortunately, Lightweight Profiling is only available on recent AMD processors and we were not able to evaluate it in this work.

This hardware support can be made a lot more scalable than cache coherency protocols, because it is controlled by the OS, which, armed with better hardware profiling, will only invoke it for pages that perform very little write sharing. So the actual synchronization protocol would be triggered infrequently.

4.5 Related Work

In this section, we explain how Carrefour relates to different works on multicore systems. First, we review systems aimed at maximizing data locality. Second, we contrast Carrefour with previous contention-aware systems. Third, we consider application-level techniques to mitigate contention on data-sharing applications. Finally, we discuss traffic characterization observations for modern NUMA systems.

NUMA-aware thread and memory management policies were proposed for earlier research systems [24, 28, 58, 98] as well as in commercial OS. Their main difference from our work is that their goal was to optimize *locality*. However, on modern systems, the main performance problems are due to traffic congestion. Our algorithm is the first one that meets the goal of mitigating traffic congestion. Among the above-mentioned works, the one most related to Carrefour is the system by Verghese et al. [98] for early cache-coherent NUMA machines, which leverages page replication and migration mechanisms. Their system relies on assumptions about hardware support that do not hold on currently available machines (e.g., precise per-page access statistics). Thus, Carrefour's logic is more involved, as it is more difficult to amortize the costs of the monitoring and memory page placement mechanisms. The authors noticed that locality-driven optimizations could, as a side effect, reduce the overall contention in the system. However, their system does not systematically address contention issues. For instance, shared written pages are not taken into account, whereas Carrefour uses memory interleaving techniques when there is contention on such pages. Moreover, the load on memory controllers is ignored when making page replication/migration decisions.

Similarly to earlier NUMA-aware policies, Solaris and Linux focus primarily on co-location of threads and data, but to the disadvantage of data-sharing workloads, replication is not supported. Linux provides the option to interleave parts of the address space across all memory nodes, but the decision when to invoke the interleaving is left to the programmer or the administrator. Solaris supports the notion of a home load group, such that the thread's memory is always allocated in its home group and the thread is preferentially scheduled in its home group. This, again, favours locality, but does not necessarily address traffic congestion.

The recent AutoNUMA patches for Linux also implement locality-driven optimizations, using two main heuristics. First, threads migrate toward nodes holding the majority of the pages accessed by these threads. Memory residence is determined by page fault statistics. Second, pages are periodically unmapped from a process address space and, upon the next page fault, migrated to the requesting node. As shown in the evaluation section, this approach yields irregular results. We attribute this limitation to the following sources of overhead: local thread/page migration decisions that do not take data sharing patterns nor access frequencies into account (thus leading to page bouncing or useless migrations) nor memory controller or interconnect load (thus leading to memory load imbalance/congestion), continuous overhead due to the scanning/unmapping of page-table entries and the corresponding soft page faults. In contrast, Carrefour makes global data placement decisions based on precise traffic patterns and adjusts the monitoring overhead based on the observed contention level.

Locality-driven optimizations for data-sharing applications were addressed in a study that dynamically identified data-sharing thread groups and co-located them on the same node [93]. However, that solution was for a system with a centralized (UMA) memory architecture. Thus, it only studied the benefits of thread placement for improved cache locality and did not address the placement of memory pages on multiple memory nodes.

Zhou and Demsky [106] investigated how to distribute memory pages to caches on a many-core Tilera processor, in order to implement an efficient garbage collector. The authors tried various policies but found that maximizing locality was the best approach for their system. This is in contrast to the systems Carrefour is targeting, where reducing congestion is more important than just improving locality. Also, the authors used a very different method for monitoring page access patterns that relies on software-serviced TLB misses, which is not possible on x86.

Several recent studies addressed contention issues in the memory hierarchy.

Some of these works were designed for UMA systems [56, 69, 107] and are inefficient on NUMA systems because they fail to address or even accentuate issues such as remote access latencies and contention on memory controllers and on the interconnect links [23]. Other works have been specifically designed for NUMA systems but only partially address contention issues. The N-Mass thread placement algorithm [63] attempts to achieve good DRAM locality while avoiding cache contention. However, it does not address contention issues at the level of memory controllers and interconnect links. Two studies [9, 64] have shown the importance of taking memory controller congestion into account for data placement decisions, but they did not provide a complete solution to address multi-level resource contention. The most comprehensive work to date on NUMA-aware contention management is the DINO scheduler [23], which spreads memory intensive threads across memory domains and accordingly migrates the corresponding memory pages. However, DINO does not address workloads with data sharing between threads or processes, which require identifying per-page memory access patterns and making the appropriate data placement decisions.

When introducing a new resource-management policy in the OS, it is worth asking whether a similar or better effect could be achieved by restructuring the application. In our context, it is important to consider the so-called *no-sharing* principle of application design. The key idea behind no-sharing is that the data must be partitioned or replicated between memory nodes, and a thread needing to access data in a different domain than its own either migrates to the target domain or asks the thread running in that domain to perform the work on its behalf, instead of fetching the data over the memory channels [16, 27, 47, 70, 74, 83, 89]. While the no-sharing architecture was primarily motivated by the need to avoid locking, it could similarly help reduce the amount of traffic sent across the interconnect, and thus alleviate the traffic congestion problem.

Unfortunately, no-sharing architectures are not a universal remedy. First of all, they trade-off data accesses for messages or thread migration; the trade-off is only worth making if the size of the data used in a single operation is much larger than the size of the message or the state of the migrated thread [27]. Second, adopting a no-sharing architecture often requires very significant changes to the application (and to the OS, if the application is OS-intensive). A good illustration of the potential challenges can be gleaned from two studies that converted a database system to the no-sharing design. The first study took the path of least resistance and simply replicated and/or partitioned the database among domains, adding a message-routing layer on top [83]. While this worked well for small read-mostly workloads, for large workloads replication had very significant memory overhead (unacceptable because of increased paging), and partitioning required *a priori* knowledge of query-to-data mapping, which is not a reasonable assumption in a general case. A solution that overcame these limitations, DORA [74], required a very significant restructuring of the database system, which could easily amount to millions of dollars in development costs for a commercial database.

Our goal was to address scenarios where adopting a no-sharing architecture is not feasible either for technical reasons or for practical considerations. Providing an OS-level, rather than an application-level, solution allows us to address many applications at once. Understanding the limitations of the OS-level solution and determining what optimizations can be done only at the level of an application is an open research question.

A recent study characterized the performance of emerging "scale-out" workloads on modern hardware [44]. The authors observed that there is little inter-thread data sharing, and the utilization of the off-chip memory bandwidth is low. As a result, they argue that memory bandwidth on existing processors is unnecessarily high. Our findings do not agree with this observation. Although it is also true that the workloads we consider perform very little fine-grained data sharing, they still stress the cross-chip interconnect, because they access a large working set, which is spread across the entire NUMA memory space. The authors of the scale-out study reported a very low bandwidth utilization ($<10\%$), even for database workloads. In contrast, our measurements show utilizations higher than 45% in most cases. These differences could be because the authors of [44] used a system with only two chips and 12 cores in their experiments. On larger systems, more threads are making requests to remote memories and so there is greater pressure on bandwidth. Further, we found that low bandwidth utilization is not necessarily a healthy symptom. In our experiments, performance dropped steeply even as bandwidth utilization went from 10% to 30%. The interconnect became the bottleneck even at a fraction of the total available bandwidth! The reason is that memory requests are not spread evenly in time; they are bursty. Burstiness causes requests to clash on the link even if the overall bandwidth is not exceeded. In summary, we conclude that contrary to the suggestion made in [44], it is too early to reduce the bandwidth of cross-chip interconnects on large multicore systems, especially if they are used for running large data-centric workloads.

4.6 Summary

Carrefour is a memory management algorithm for NUMA systems that manages traffic on memory controllers and interconnects. Earlier NUMA-aware memory management policies aimed to mitigate the cost of remote wire delays, which is no longer the main bottleneck on modern systems. Carrefour's design was motivated by the evolution of modern NUMA hardware, where traffic congestion plays a much larger role in performance than wire delays.

System design principles embodied in Carrefour are important not only for today's systems, but also for future hardware. The amount of memory bandwidth per core is projected to decrease in the future [27], so managing traffic congestion will be as crucial as ever.

Smart memory placement, as achieved by Carrefour, is only part of the equation. If there is only one application using all of the cores of a machine, then good memory placement is sufficient for optimizing memory traffic and therefore performance. But, if an application does not fill the whole machine, then thread placement becomes equally important. The strategy described in Chapter 2 addresses this side of the equation. It can be used to decide which NUMA nodes and interconnect links to use while considering all other relevant shared resources. The workload placement algorithm in Chapter 2 complements Carrefour, and when working in tandem they can optimize multi-application workloads and under-utilized systems.

Chapter 5

Large Pages on NUMA Systems[1]

The previous chapter showed that to achieve good performance on NUMA systems we need to place memory such that interconnect contention is minimized and local accesses are maximized. In this chapter we show that large pages sometimes make optimal memory placement impossible, and specifically that ***large pages can hurt performance on NUMA systems***. This observation, the analysis of the causes, and the solution in the form of an extension to the Carrefour algorithm of Chapter 4, constitute the primary contributions of this chapter.

Attribution: I conducted the initial investigation into NUMA and large pages, including the discovery and analysis of the hot page and page-level false sharing problems (reported in Section 5.2 and Section 5.3.1). Fabien Gaud and Baptiste Lepers designed and implemented the Carrefour-LP algorithm.

5.1 Background

Applications with large memory working sets require many virtual-to-physical address translations in page tables and TLBs. This drives up physical RAM consumption, increases TLB miss rate, and hurts performance [15, 19, 73]. According to one report, a large Oracle DBMS installation with 500 concurrent connections consumed 7GB of RAM for page tables alone! [33]. To address this problem, most modern hardware and operating systems introduced support for large pages. On

[1]This work is a modified version of work previously published in [48]

x86 systems large pages are typically 2MB (512 times larger than regularly-sized 4KB pages), and support for 1GB pages is on the way[2]. Using larger pages requires fewer translations to cover the address space and diminishes the pressure on the TLB and physical memory.

While large pages are crucial for performance of large-memory systems, they, unfortunately, also have downsides. Previous work reported and addressed increased memory footprints and physical memory fragmentation [92]. In this chapter, we report on a new problem: ***large pages can exacerbate harmful NUMA effects, such as poor locality and imbalance***. Using large pages makes the unit of memory management (a page) more coarse. As a result, it is more likely that many frequently accessed memory addresses happen to map to the same physical page and overload the memory node hosting it — the *hot-page* effect. The hot-page effect cannot be addressed by page migration and balancing; page splitting must be performed prior to any attempts to rebalance memory. Likewise, large pages lead to more frequent *page-level false sharing* among threads, where threads access *different* data on the *same* page. False sharing leads to poor locality, which cannot be addressed by page migration alone.

Even though hot pages and false sharing touched only a couple of benchmarks in our set, these effects will become pervasive on systems with much larger pages (e.g., 1GB), which are becoming common. Therefore, we implemented Carrefour-LP which addresses these problems by dynamically splitting large pages as needed. For applications affected by hot pages and false sharing, Carrefour-LP improves performance by 10%-80% relative to Carrefour alone. Carrefour together with Carrefour-LP significantly diminish or completely eliminate the performance degradation introduced by large pages and improve performance of some applications by 2-3$\times$ relative to Linux with large pages.

[2]1GB pages are already supported by the hardware; support by the OS is still nascent, so few applications are able to use them at the time of this writing.

5.2 Large Pages and Adverse NUMA Effects

5.2.1 Experimental Platform

For our experiments, we used two different server-class machines:
Machine C has two 1.7GHz AMD Opteron 6164 HE processors, with 12 cores per processor, and 64GB of RAM. The system is equally divided into four NUMA nodes (i.e., six cores and 12GB of RAM per node).
Machine D has four AMD Opteron 6272 processors, each with 16 cores (64 cores in total), and 512GB of RAM. It has eight NUMA nodes with 8 cores and 64GB of RAM per node.
Both machines have HyperTransport 3.0 interconnect links.

We are running on Linux 3.9 and are using Transparent Huge Pages (THP) for large page allocation[3]. THP works by backing allocations of anonymous memory with 2MB pages whenever possible. Other kinds of memory, such as memory mapped files, are unaffected by THP and use 4KB pages. THP also uses a kernel thread to periodically scan for free memory regions that are at least 2MB in size, which are then used to replace groups of existing 4KB pages.

We used several benchmark suites representing a variety of different workloads: the NAS Parallel Benchmarks suite [11] which is comprised of numeric kernels, MapReduce benchmarks from Metis [65], SSCA v2.2 (a graph processing benchmark) [10] with a problem size of 20, and SPECjbb [2]. From the NAS benchmark suite we picked the benchmarks that ran for at least 15 seconds. The memory usage of the benchmarks ranges from 518MB for EP from the NAS suite to 34,291MB for IS from NAS.

5.2.2 Large Pages on Linux

Figure 5.1 compares the performance of 4KB pages and 2MB pages using THP. We can see that THP increases performance (by up to 109%) for several benchmarks on both machines (e.g. WC, WR, WRMEM, and SSCA), but also significantly de-

[3]Linux also allows using large pages via *libhugetlbfs*, but the latter required recompiling applications and pre-allocating memory for large pages, which was inconvenient, and, moreover, did not perform better than THP in our experiments.

creases performance by as much as 43% in some cases. CG, UA, and SPECjbb are all negatively affected by THP. Therefore, 2MB pages are not universally beneficial and neither are 4KB pages, so there is no “one size fits all.”

To understand this phenomenon, we recorded two metrics that represent the potential benefits of large pages: the number of L2 cache misses caused by page table walks (obtainable from HPEs), and the maximum time spent in the page fault handler by any core. L2 misses due to page table walks is a good indicator for the effect of TLB misses on performance. We expect large pages to increase the TLB coverage and reduce page table sizes. As a result, we expect the number L2 cache misses due to page table walks to drop when we use large pages. Similarly, large pages will reduce the number of page faults for allocations and thus the time spent in the page fault handler.

We also monitored two metrics related to NUMA efficiency: the *local access ratio* (LAR), which is the percentage of accesses to local memory, and the *traffic imbalance* on the memory controllers. Traffic imbalance is defined as the standard deviation of the memory request rate across the controllers, expressed as the percent of the mean. For memory intensive applications, a low LAR and a high imbalance signify a NUMA issue.

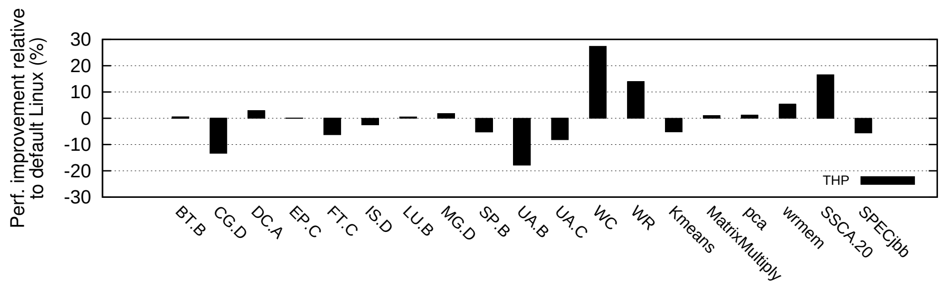
Table 5.1 shows the profiling results for a subset of interesting applications. As expected, applications that benefited from 2MB pages in Figure 5.1 (WC and SSCA) have fewer L2 misses due to page table walks, and for WC significantly less time spent in the page fault handler. The effects can be dramatic. For example, with SSCA on machine C the percentage of L2 misses due to page table walks is decreased from 15% to 2% when using 2MB pages, which results in a 17% performance increase. WC, which experiences a similar decrease in L2 misses but also a large decrease in time spent on page faults, has its performance increased more than two-fold on machine D.

The two other profiled benchmarks, CG and UA, perform much worse with 2MB pages. The profiling reveals that the degradation is caused by NUMA effects. With CG and 4KB pages, the load on the memory controllers is almost perfectly balanced, but with 2MB pages the imbalance is 20% on machine C and 59% on machine D. For UA, the problem is that the LAR decreases when using large pages, from about 88% to around 66%.

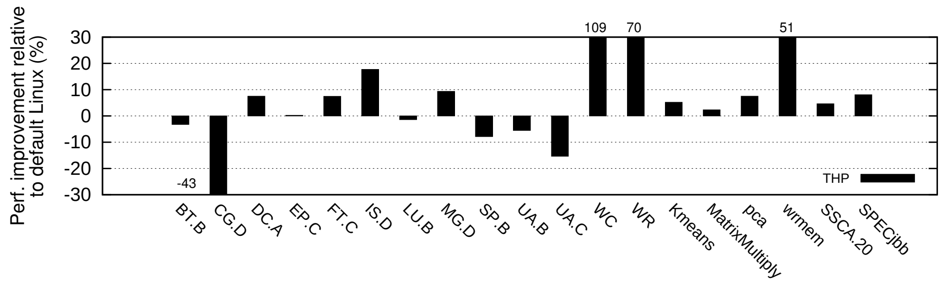
		CG.D (D)	**UA.C (D)**	**WC (D)**
Perf. incr. THP/4k (%)		-43	-15	109
Time in page fault handler (% of total time)	Linux	2182ms (0.1%)	102ms (0.2%)	**8731ms (37.6%)**
	THP	445ms (0%)	53ms (0.1%)	**3682ms (32.3%)**
% L2 misses due to page table walks	Linux	0	0	**10**
	THP	0	0	**1**
Local access ratio (%)	Linux	40	**88**	50
	THP	36	**66**	55
Imbalance (%)	Linux	**1**	14	147
	THP	**59**	12	136

		SSCA.20 (C)	**SPECjbb (C)**
Perf. incr. THP/4k (%)		17	-6
Time in page fault handler (% of total time)	Linux	90ms (0%)	8369ms (2.1%)
	THP	147ms (0.1%)	5905ms (1.5%)
% L2 misses due to page table walks	Linux	**15**	**7**
	THP	**2**	**0**
Local access ratio (%)	Linux	25	12
	THP	26	15
Imbalance (%)	Linux	**8**	**16**
	THP	**52**	**39**

Table 5.1: Detailed analysis of various application on machine C and D. The machine type is indicated in parentheses next to the name of the benchmark.



Machine C



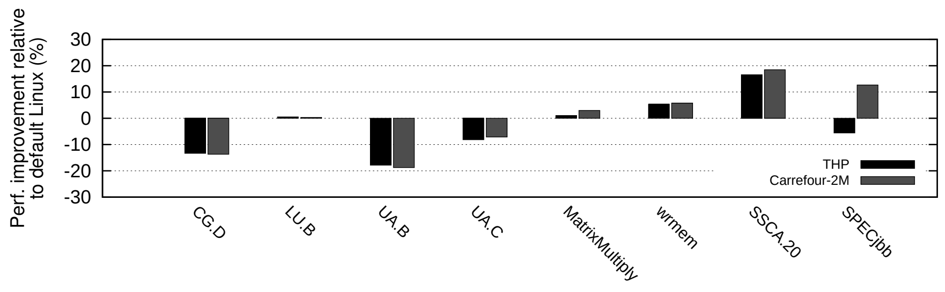
Machine D

Figure 5.1: THP performance improvement over Linux on machine C and machine D. THP sometimes performs better than Linux, sometimes worse.

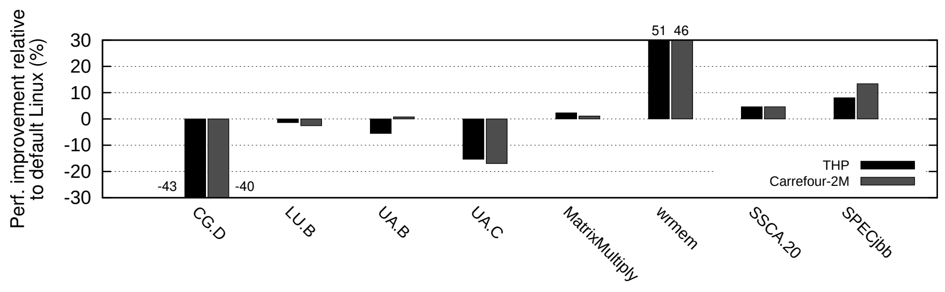
SPECjbb presents an interesting case. While the data in Figure 5.1 suggests that it does not benefit from large pages, profiling reveals that using large pages actually decreases the percent of L2 misses due to page table walks. At the same time, SPECjbb suffers from NUMA issues: the imbalance rises from 16% to 39% with large pages. Therefore, SPECjbb *could benefit* from large pages if NUMA effects were reduced.

5.3 Solutions

The previous section demonstrated that using large pages may introduce NUMA issues, which may either degrade performance relative to small pages (as they did for CG and UA) or leave the performance unchanged but prevent an application from enjoying the benefits of large pages (as they did for SPECjbb). In this section



Machine C



Machine D

Figure 5.2: Performance improvement of Carrefour-2M and THP over Linux on applications whose NUMA metrics are affected by THP (2MB pages). Carrefour-2M is not always able to solve the problems for applications that suffer from THP.

we first demonstrate that using a NUMA-aware page placement algorithm eliminates the NUMA issues for some applications, motivating the use of NUMA-aware page placement with large pages.

We then identify two new problems that a placement algorithm unaware of large pages does not address: the hot-page effect and the page-level false sharing. These effects, while affecting only two applications in our experiments, will become especially important as much larger pages (e.g., 1GB) come into use. To address them, we introduce large-page extensions (LP) to the *Carrefour* algorithm from the previous chapter (all experiments shown in Chapter 4 used 4KB pages).

For clarity of presentation, from now on we will focus on those applications that experience NUMA issues when large pages are used. Specifically, if the LAR or the imbalance is made worse by more than 15% by using large pages as opposed to small ones on either machine, the application is selected for presentation, otherwise it is omitted. The selected applications are: CG.D, LU.B, UA.B, UA.C, matrixmultiply, wrmem, SSCA, SPECjbb. For completeness, and to demonstrate that our solutions do not hurt the applications they cannot help, we do include performance results for the excluded applications at the end of Section 5.4.

5.3.1 Page Balancing is Not Enough

We ran Carrefour in the kernel configured with 2M pages (Carrefour-2M). Figure 5.2 shows the performance of Carrefour-2M compared to Linux with 2M pages (labeled as *THP*) relative to Linux with 4K pages (labeled as *Linux*). We observe that while Carrefour-2M does improve performance for some applications, it fails to solve the problem across the board. For SPECjbb, Carrefour-2M addresses the NUMA issue; as shown in Table 5.2 it restores the balance on memory controllers that was introduced by large pages and improves the LAR.

At the same time, Carrefour-2M fails to improve performance for UA and CG. To understand why, we show profiling data for these applications in Table 5.2. We report five metrics: the percentage of total accesses to the most used page (PAMUP), the number of hot pages (NHP) defined as pages comprising more than 6% of the total accesses[4], the percentage of memory accesses to pages shared by at least two threads (PSP), the percentage of accesses to local memory (LAR), and the traffic imbalance on the memory controllers.

The results for CG reveal that there is a ***hot page problem***. Large pages cause the heavily accessed regions of the address space to be coalesced into a small number of hot pages (the PAMUP significantly increases), and because there are fewer hot pages than NUMA nodes it is impossible to balance them.

UA does not have a hot page issue, but it does have more pages that are shared among threads when large pages are used (the PSP significantly increases). This happens because each page holds more data and is thus more likely to contain data used by multiple threads. Since the threads do not share data, but share the *page*, we refer to this problem as ***page-level false sharing***. Carrefour-2M is then forced

[4]In order to perfectly balance the load on a 8-node NUMA machine, each node must be the target of 12.5% of the total memory accesses. Thus, we consider that if a page represents more than half of this amount, it is likely to create imbalance.

to interleave these pages whereas if there were less sharing the pages could be placed on the nodes where they are most heavily used for maximum locality. As a result, Carrefour-2M delivers a lower LAR than Linux with small pages.

In summary, Carrefour-2M is only able to address NUMA issues induced by large pages in cases where they are not caused by the hot-page effect and page-level false-sharing.

While these problems affected only two applications in our experiments, they will become pervasive as pages much larger than 2MB come into use. 1GB pages are already supported by the hardware; applications like large DBMS clearly motivate their use [33]. We did not evaluate 1GB pages, because they are poorly supported in Linux. 1GB pages are not compatible with THP, and while in theory it is possible to use them with lighugetlbfs, that has many challenges. First of all, the implementation is unreliable. We were not able to enforce the use of 1GB pages with NAS applications and observed many crashes with the Metis suite (because the latter uses a custom memory allocator). Second, the splitting of large pages, which is crucial to our solution, is not supported by libhugetlbfs and implementing it would require a significant effort.

However, since the use-case for very large pages is definitely there, they will become more common as the OS support improves. Then, the hot-page effect and page-level false sharing will become more common (Section 5.4.4 provides some preliminary data). To address these problems, we propose large-page extensions to Carrefour.

5.3.2 Carrefour-LP

Intuition suggests two basic solutions to the problem: *conservative* — prevent the problem by only creating large pages when necessary, or *reactive* — start with large pages and fix NUMA problems when they are observed. Each approach has potential benefits and drawbacks. The conservative approach can avoid NUMA related performance degradation but can also miss out on the benefits of large pages. On the other hand, the reactive approach will benefit from large pages, but must be able to quickly and accurately detect NUMA issues and must pay the overhead of fixing them.

		Linux	**THP**	**Carrefour 2M**
SPECjbb	PAMUP	2%	6%	6%
	NHP	0	0	0
	PSP	**10%**	**36%**	**36%**
	Imbalance	**16%**	**39%**	**19%**
	LAR	26%	28%	27%
CG.D	PAMUP	**0%**	**8%**	**8%**
	NHP	**0**	**3**	**3**
	PSP	18%	34%	34%
	Imbalance	**0%**	**20%**	**20%**
	LAR	45%	45%	45%
UA.B	PAMUP	6%	6%	6%
	NHP	0	0	0
	PSP	**16%**	**70%**	**70%**
	Imbalance	9%	15%	17%
	LAR	**90%**	**61%**	**58%**

Table 5.2: Proportion of accesses to the most-used page (PAMUP) in %, number of hot pages (NHP), proportion of memory accesses to shared pages (PSP) in %, Imbalance in % and local access ratio (LAR) in % for Linux, THP and Carrefour-2M, on machine C (24 cores).

We found that a good algorithm must be a combination of these approaches. The ***reactive component*** of our algorithm continuously monitors HPEs looking for the presence of NUMA effects under large pages, applies the page balancing techniques of Carrefour and splits the large pages if the latter are ineffective. The ***conservative component*** of the algorithm continuously monitors the virtual memory metrics and re-enables large pages if they are expected to deliver benefit but were previously disabled.

We also found that it is more practical and involves less overhead to enable large pages in the beginning and disable them later if they are deemed harmful. In particular, many applications have intensive memory-allocation phases at the very beginning of the program that suffer from lock contention if small pages are used.

Our full algorithm is presented in Algorithm 5. The algorithm also details the HPEs that are being monitored. Since the monitoring is done continuously, the algorithm caters to phase changes in applications. Below we describe the rationale behind the decisions made in the algorithm.

Reactive Component

The job of the reactive component is to disable large pages when they are harmful to the extent that even Carrefour-2M's page-balancing techniques cannot address the performance degradation. To that end, it estimates the local access ratio (LAR), a vital metric for detecting NUMA issues, with and without Carrefour and large pages.

We use AMD's instruction-based sampling (IBS)[5] to sample memory accesses to pages, and to learn whether the access was made from a local or a remote node. We only consider pages that have at least one sample where the access was serviced from DRAM, so that our decisions are not affected by pages that are easily cached. From the IBS samples, we estimate the LAR that would be obtained if the shared pages were migrated to a random node and if non-shared pages were migrated to the local node (i.e. interleaving and migrating pages with the Carrefour-2M algorithm). We also calculate the LAR that would be obtained if the same technique were used but with all of the 2MB pages split into 4KB pages.

Estimating the LAR for various what-if scenarios (e.g., if a page were migrated or if large pages were split into regular-sized) is trivial with IBS samples. IBS gives us data addresses and the node from which they were accessed. So we can compute the current LAR as well as the LAR that would be obtained if the pages where placed on different nodes. Similarly, we can map the data addresses to 4KB pages and compute the same metrics for the scenario if the large pages were split.

If, based on our estimates, the LAR can be improved by 15% with Carrefour-2M only and without splitting the pages, we simply run Carrefour-2M. Otherwise, if splitting pages would improve the LAR by at least 5%, then all shared 2MB pages are demoted into 4KB pages. Note that we are being cautious here: we try to address NUMA issues by page migration first, and split pages only if absolutely necessary. Splitting pages has overhead and may hurt applications that benefit from large pages. In addition, large pages with more than 6% of the total accesses

[5]Intel systems have a similar facility called PEBS (Precise Event-Based Sampling).

(hot pages, as defined in Section 5.3.1) are split and the constituent 4KB pages are interleaved.

This part of the algorithm relies on two thresholds. The first one is the 15% threshold used to decide whether we can improve the LAR simply by rearranging memory pages, without having to split large pages. That threshold was relatively easy to set across applications: the key is to use a relatively large number, since we want to be rather confident that we can improve performance without having to split pages. The second threshold, the 5% performance gain that we expect from splitting pages, needs to be any non-negligible number that would justify the splitting. Again, that threshold was relatively easy to tune across applications.

In the algorithm, we use the LAR computed per-application. Another option would be to use the LAR computed per-page, however this was difficult to do, because existing hardware monitoring facilities prevent us from obtaining enough samples to accurately compute per-page LAR (and even per-application LAR as explained in the next section). This is why the algorithm splits all 2MB pages when it detects the LAR can be improved.

Conservative Component

The job of the conservative component is to re-enable large pages when they have been disabled but monitoring shows that they would be beneficial again. The conservative component uses two criteria to determine the benefit of large pages: the performance impact of TLB misses (based on the fraction of L2 misses caused by page table walks) and the maximum percentage of time any core spends processing page faults. The reason why we consider the time spent processing page faults is that large pages improve performance by decreasing this time. Indeed, soft page faults not only take CPU time, but also incur costly synchronization [26]. The latter is the reason why we use the *maximum* fraction as opposed to the average: lock contention will be determined by the slowest core that holds page table locks.

The conservative component works as follows. If the impact of TLB misses is estimated to be greater than a threshold of 5%, then 2MB page allocation and 2MB page *promotion*[6] are both enabled via THP. Similarly, if the time spent in the

[6]Page promotion refers to dynamic consolidation of regular-sized pages into large pages. It is supported by the default Linux kernel. We set the frequency for page promotion checks to every

Algorithm 5 Large-page Extensions to Carrefour

```
Enable 2MB page allocation and promotion
while true do
    Gather HPEs and IBS samples for 1 sec
    if L2 misses due to page table walks $> 5\%$ then
        Enable 2MB page allocation
        Enable 2MB page promotion
    else if Max time spent on page faults $> 5\%$ then
        Enable 2MB page allocation
    end if
    if Estimated LAR improvement with only Carrefour $> 15\%$ then
        SPLIT_PAGES = false
    else if Estimated LAR improvement with Carrefour and splitting pages $>$ 5% then
        SPLIT_PAGES = true
    end if
    if SPLIT_PAGES = true or 2MB page allocation is disabled then
        Split all shared 2MB pages into 4KB pages
        Disable 2MB page allocation
    end if
    Split and interleave 2MB hot pages
    Interleave and migrate pages with Carrefour
end while
```

page fault handler was more than a threshold of 5%, then 2MB page allocation is enabled but not 2MB page promotion, since there is little benefit in promoting the pages on which we had already paid the cost of page faults.

In order to estimate the impact of TLB misses on performance, we use *the fraction of L2 cache misses due to page table walks*. This assumes that TLB misses primarily degrade performance when a page table traversal causes an L2-cache miss (in that case, the miss is satisfied either from the L3 cache or from the DRAM, both of which are costly), and that the application's performance is dominated by L2 cache misses. Although this is a coarse approximation, it works well because applications that experience a lot of cache misses due to page table walks are those with large page tables. This implies that they have large memory footprints, and so

10ms.

they are memory-intensive. Therefore, it is safe to assume, for these applications, that variations in performance can be primarily explained by the number of L2 cache misses. Conversely, applications with a very small fraction of L2 cache misses resulting from page table walks are not memory-intensive, so for them the impact of TLB misses is negligible.

5.4 Evaluation

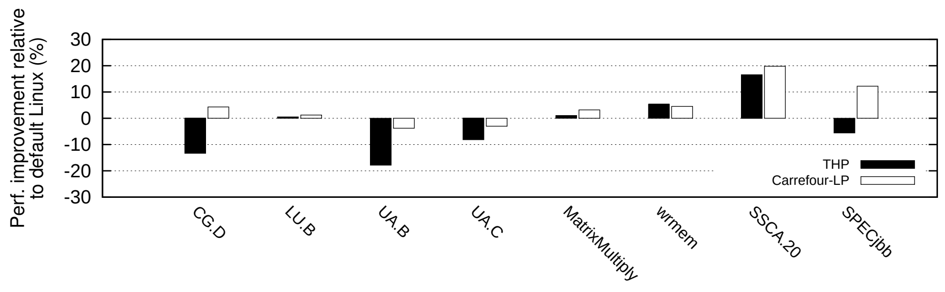
5.4.1 Performance Evaluation

Figure 5.3 shows performance of Carrefour-LP and THP relative to Linux with 4K pages. We continue focusing only on the applications affected by NUMA issues; the remaining applications are presented for completeness in Figure 5.5. Figure 5.3 shows that Carrefour-LP:

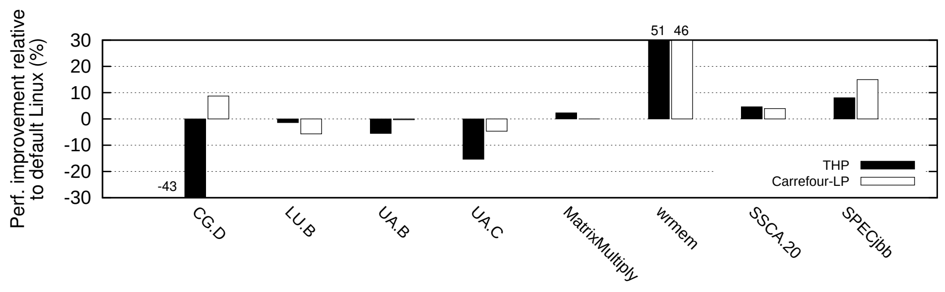
- Restores performance of applications that suffered under large pages and do not stand to benefit from them: CG.D, UA.B, UA.C.
- Improves performance of applications that were expected to benefit from THP but did not (or did not benefit fully): SSCA and SPECjbb, both on machine C.
- Does not significantly hurt performance of the applications where NUMA effects did not cause performance degradation under large pages and where no performance improvements from large pages were expected (the remaining applications).

We next provide the detailed analysis of Carrefour-LP. We analyze the contribution to performance improvements of its three components: Carrefour-2M, conservative and reactive. We demonstrate when and why it is sufficient to just use Carrefour-2M alone and explain how both conservative and reactive components contribute to the solution. The performance breakdown is shown in Figure 5.4.

Workloads other than CG.C, UA.B and UA.C are not affected by the hot-page effect and page-level false sharing, so in these cases Carrefour-LP performs similarly to Carrefour-2M alone. It is able to meet the performance of Carrefour-2M with minimal overhead (at most 3.7% on machine C and 2.1% on machine D).



Machine C



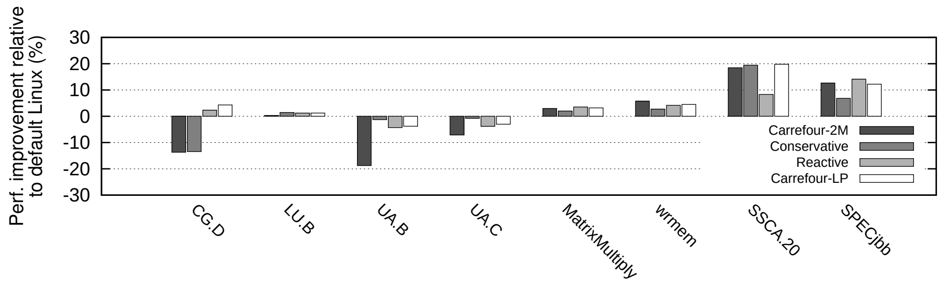
Machine D

Figure 5.3: Performance improvement on a reduced set of applications of THP and Carrefour-LP over Linux, on machine C and machine D.

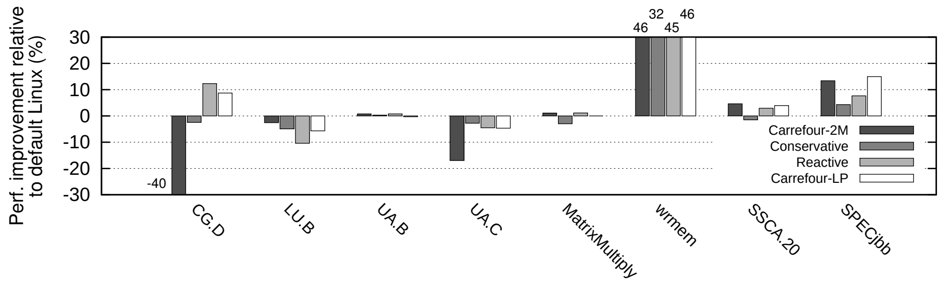
Table 5.3 demonstrates that Carrefour-LP eliminates the hot-page effect and page-level false sharing and improves NUMA metrics where Carrefour-2M fails. For UA, the LAR drops from about 90% to roughly 60% under THP and remains at that low level under Carrefour-2M. Carrefour-LP is able to restore it almost to the previous level by dynamically splitting pages.

For CG.D, enabling large pages disturbs the perfect memory-controller balance enjoyed under small pages. Carrefour-2M is unable to restore it, while Carrefour-LP restores it almost entirely.

We now analyze the importance of the two components in Carrefour-LP. Figure 5.4 presents the performance obtained when running Carrefour-2M alone (labeled as *Carrefour-2M*), Carrefour-2M with the reactive component designed for Carrefour-LP (labeled as *Reactive*), the original Carrefour runtime (working on 4kB pages) together with the conservative component (labeled as *Conservative*), and Carrefour-LP (labeled as *Carrefour-LP*). Figure 5.4 shows that in all cases, en-



Machine C



Machine D

Figure 5.4: Performance improvement on a reduced set of applications of Carrefour-2M, the conservative component, the reactive component and Carrefour-LP over Linux with THP, on machine C and machine D.

	Local Access ratio (%)				**Imbalance (%)**			
	Default Linux	*THP*	*Carr. 2M*	*Carr. LP*	*Default Linux*	*THP*	*Carr. 2M*	*Carr. LP*
CG.D (B)	40	36	38	39	**1**	**59**	**69**	**3**
UA.B (A)	**90**	**61**	**58**	**85**	9	15	17	10
UA.C (B)	**88**	**66**	**68**	**82**	14	12	9	14

Table 5.3: NUMA metrics for CG.D on machine D, UA.B on machine C, and UA.C on machine D.

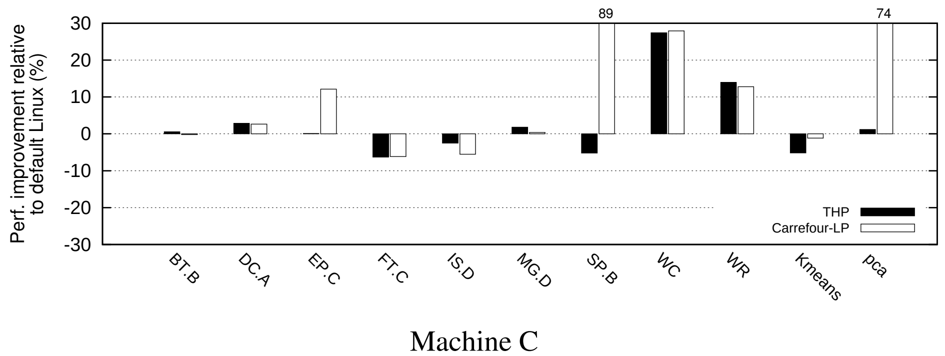
abling the two components (as done in *Carrefour-LP*) is always the best choice (or close to the best). The conservative component alone does not solve the problem, because it begins with 4K pages. For SPECjbb, for example, it does not detect the need for large pages soon enough, so the performance is not as good as it could be. We similarly observed that using the conservative component alone hurts performance of many applications that were not included in this analysis (but shown in Figure 5.5) for the same reason: large pages were not enabled soon enough. These applications have an intense memory allocation phase at startup, which can benefit greatly from large pages due to fewer page faults, but the conservative component does not enable large pages soon enough.

Using the reactive component alone works well on some applications. For CG.D, it is able to detect the hot page and split it. Similarly, it is also able to split the falsely shared pages for UA.B and UA.C. However, on some applications, it fails to bring the maximum performance improvement that can be achieved with 2M pages (e.g. SSCA on machine C and SPECjbb on machine D). The reason is that the LAR is sometimes misestimated, and this results in 2M pages being split in applications that do not suffer from NUMA issues. For instance, on SSCA, the algorithm predicts a LAR of 59% if large pages were all split into 4k pages, whereas the actual LAR obtained after splitting is equal to 25%.

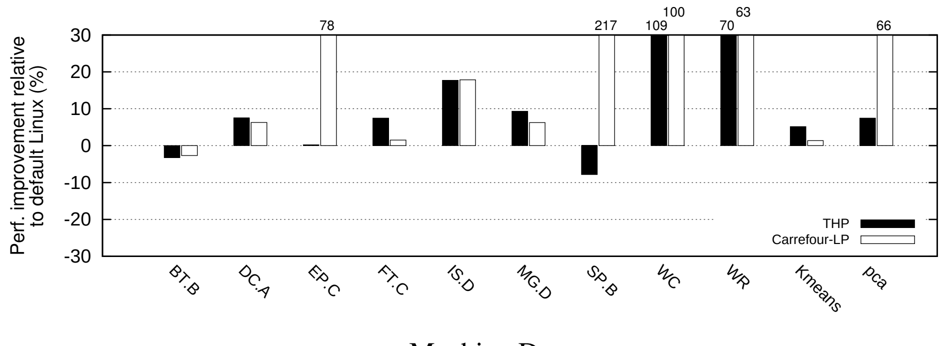
The problem is, in order to estimate the LAR under regular-sized pages given the data samples collected under large pages, we need to have enough samples on the constituent sub-pages. Unfortunately, we found it to be very difficult to gather enough samples; increasing the sampling rate results in unacceptably high overhead. A promising solution would be to use Lightweight Profiling (LWP). LWP is an extension of AMD processors that aims at providing the same level of details as IBS with less overhead. To reduce the overhead, LWP stores samples in a ring buffer and only interrupts the processor when the buffer is full. Unfortunately, on available AMD processors, LWP is only partially implemented: LWP samples only contain the instruction pointer of the sampled instruction and a timestamp. This information is not sufficient to predict LAR. Another potential solution is the Shim profiler [102], but only if the application leaves some cores unused for the profiler.

Because of these deficiencies in hardware profiling, the reactive component may make mistakes in deciding when to split large pages. This is where the conservative component comes to the rescue and re-creates the large pages when they are expected to help.

We conclude this section by explaining some performance results in Figure 5.5, which contains applications where THP did not create any NUMA issues. The



Machine C



Machine D

Figure 5.5: Performance improvement of THP and Carrefour-LP over Linux on applications whose NUMA metrics are **not** affected by THP, on machine C and machine D.

key observation is that the overhead of Carrefour-LP does not significantly hurt these applications. Moreover, EP.C, SP.B and pca enjoy better (sometimes much better) performance with Carrefour-LP than with THP. That is because they had NUMA issues to begin with (which were not exacerbated by large pages), and so the Carrefour-2M component of the algorithm helped to address them.

5.4.2 Overhead Assessment

Overhead in Carrefour-LP comes from collecting and storing IBS samples, computing the metrics based on these samples, migrating and splitting pages. Overall, the overhead of Carrefour-LP compared to the reactive approach is negligible: between 1% and 2% on all applications (on all machines) except CG (3.2%) and IS

(2.1%) on machine D. Even on these two applications, the overhead is still within the standard deviation.

Compared to Carrefour-2M, the overhead is also small. The maximum overhead observed is 3.7% on machine C (SP.B) and 3.2% on machine D (LU.B), but on average it is below 2%.

Compared to Linux with 4k pages, Carrefour-LP has an overhead of less than 3%, except on FT, IS (machine C) and LU (machine D). This overhead is not specific to Carrefour-LP but is rather caused by Carrefour-2M, which spends too much time migrating large pages. Since our solution is built on top of Carrefour-2M, it also suffers from the same overhead.

5.4.3 Discussion

The solution could be much improved if we had a more accurate way of estimating the LAR. Currently, with inaccurate estimates, the solution may split and migrate pages when there is no benefit to be gained, which is why Carrefour-LP degrades performance of LU by 3.5% compared to Carrefour-2M. We believe that the LAR could be predicted more accurately if we could collect more data samples without additional overhead. A complete implementation of LWP (i.e., if LWP provided the same kind of samples as IBS) would solve this problem.

Our earlier implementation had scalability issues on the system with 64 cores. The reason was that the centralized data structure where we stored IBS samples had to be accessed and locked from multiple nodes. We addressed this problem by maintaining a data structure per node. The per-node structures are still accessed by multiple cores, so we may need to revisit this scaling issue on larger machines. Overall, the algorithm is likely to scale well because all work generated by an interrupt is performed independently on each node, so the number of nodes can grow without creating scalability bottlenecks.

Splitting pages did not create too much overhead, but the use of the page table lock for THP operations is clearly a scalability concern. Linux developers are working on finer grain locks at the time of this writing, so we hope that this problem will be avoided.

We did not observe many oscillations, where we go back and forth between splitting and enabling large pages. Overall, Carrefour-LP seems to be more robust than the conservative and the reactive components used independently, because it naturally supports transient states and phase changes by continuously re-examining its decisions.

5.4.4 Very Large Pages

Although accessing the very large 1GB pages via libhugetlbfs proved challenging for most applications, we were able to enable them in SSCA and in streamcluster (an application from PARSEC)[7]. We immediately observed the hot-page and page-level false-sharing problems. With 1GB pages, lots of hot small pages were coalesced on a single NUMA node, and the performance dropped dramatically. For SSCA it degraded by 34%; for streamcluster by a factor of 4. Neither of these applications suffered performance degradation when 2M pages were used. Although preliminary, these data suggest a much more pervasive presence of NUMA issues when very large pages are used, and so Carrefour-LP will become even more important in the future.

5.5 Related Work

5.5.1 Large Pages and TLB Performance

Several studies have characterized the effect of TLB misses and large pages [19, 50, 73, 99, 104]. Battacharjee and Martonosi [19] specifically looked at the effect of TLB misses on multicore systems with multithreaded workloads. They found that some applications, such as Canneal from the PARSEC benchmark suite, spend up to 0.7 cycles per instruction on servicing D-TLB misses. Another study [73] showed performance improvements of up to 25% in the NAS benchmark suite due to using large pages. For large-scale HPC applications, Zhang et al. [104] found that large pages improve communication performance significantly.

Weisberg and Wiseman [99] used the SPEC CPU2000 benchmarks to evaluate the relationship between page size and the number of TLB misses. They argue that

[7]The PARSEC suite was not included in our study, because its applications did not experience performance differences under THP with 2M pages.

a 4KB page size is much too small for most applications, and conclude that a page size of 256KB and a 64-entry TLB is sufficient to drastically reduce the number of TLB misses.

Sudan et al. [91] motivate the need for small pages. They show that using 1KB pages allows optimizing the usage of the DRAM row-buffer, yielding substantial energy savings and decreasing the average latency of memory accesses.

All these works motivate the use of different page sizes, but none of them highlight or quantify the impact of NUMA on the performance obtained when using different page sizes.

5.5.2 Large Page Support and Optimization

Many software systems have been designed that make large pages easier to use or more effective.

Navarro et al. [72] described an algorithm for operating system support of large pages that reduces fragmentation and does not require memory copies to create large pages. Using their algorithm, a page fault reserves a physical memory region of the size of a large page, but it initially only allocates and maps a small page. Subsequent page faults use the reserved space until it has been completely allocated, at which point the region is promoted to a large page. The algorithm does not attempt to optimize the placement of large pages.

Cascaval et al. [29] developed a model to predict the benefit of using large pages on individual data structures of applications, based on the predicted number of TLB misses and page faults. The predictions are computed using HPEs throughout multiple runs of the application. The data structures that are predicted to benefit the most from large pages are backed by large pages. A similar method is described in [78], with the major difference being that large page promotions are performed at runtime.

Magee and Qasem [62] also devised a system for restricting the usage of large pages to applications that benefit the most from them. At compile-time, the working-set size is estimated through static analysis. If the estimated working-set size is greater than the coverage of the target CPU's TLB, then large pages are used.

A different approach is explored by Basu et al. [15]. Instead of managing the use of large pages at the OS level, they propose a hardware extension that allows applications to directly map memory segments. Addresses within directly mapped segments bypass the TLB and so translation is nearly free. The segments are conceptually similar to very large pages and provide similar benefits, but the authors do not analyze the potential NUMA effects which would be exacerbated by the large size of the segments.

In summary, previous works mostly focused on the limited availability of large pages and on reducing memory fragmentation. Several systems have been designed to ensure that applications that benefit from large pages actually use them, but no existing work has revealed and addressed the NUMA issues raised by large pages.

5.6 Summary

We demonstrated that using large pages can create or exacerbate NUMA issues like reduced locality or imbalance. We showed that these problems can be in some cases addressed by using a NUMA-aware page placement algorithm, but the latter stumbles upon two problems: the hot-page effect and page-level false sharing, which cannot be addressed via page migration. To address these problems, we implemented Carrefour-LP: large-page extensions to the NUMA-aware page placement algorithm described in Chapter 4. Our results show that Carrefour-LP restores the performance when it was lost due to large pages and makes their benefits accessible to applications.

Solutions like Carrefour-LP will be even more important in the future, when very large pages (1GB in size) will be in widespread use.

Chapter 6

Conclusion

A reoccurring theme throughout Chapters 2–5 is the continual increase in hardware complexity and the burden it places on system software to get the most out of it. Due to physical limitations, hardware architects must improve chip performance by increasing core counts and adding features to improve utilization rather than increasing frequency as has been done in the past. This leads to a multitude of shared resources such as core functional units when using SMT, caches at various levels, interconnects, and memory controllers. As we have seen, this resource sharing can cause significant performance effects, and reducing the contention for the shared resources is essential for optimizing performance.

And yet modern operating systems rely on simple and usually insufficient mechanisms to address shared resource contention. For example, Linux's default NUMA memory allocation policy is first-touch, meaning that memory is allocated on the NUMA node it is first accessed from. This can lead to extensive contention for memory controllers and interconnect bandwidth, as shown in Chapter 4. Another example is the Linux thread scheduler which generally tries to spread threads apart in an effort to give the threads as much resources as possible. Again, this relatively simple strategy does not provide good performance for every application as seen in Section 2.5. In general, the simple solutions employed by operating systems are not optimal for all applications but only a subset of them.

The first step towards a comprehensive solution is insight into the problem itself, specifically how applications interact with the hardware and the effects on per-

Concern	Score	Resources	Cost?	Inverse Perf?
L2/SMT	Number of L2 caches in use	L2 cache, L1 cache, instruction front-end, and functional units	Y	Y
L3	Number of L3 caches in use	L3 cache	Y	Y
Mem. Controller	Number of memory controller pairs in use	Memory controllers, DRAM bandwidth, I/O	Y	Y

Table 6.1: Scheduling concerns for an AMD Zen system.

formance. The insights are important research contributions on their own. Chapter 2 showed the predictive power of performance measurements and relative lack of predictive power of HPEs, Chapter 3 demonstrated the importance of the instruction mix with respect to SMT preference, Chapter 4 proved the importance of managing NUMA traffic congestion over focusing solely on locality, and Chapter 5 identified the hot page and page-level false sharing issues with NUMA and large pages.

From these observations we built a complete and practical solution to the problem of NUMA and multicore resource contention. The first step is workload placement, described in Chapter 2. Our approach relies on an offline training phase that runs a training set of benchmarks and builds a machine learning model to predict the performance of applications at different thread placement configurations. At runtime, a new, previously unseen application requires only two performance measurements in two different placements as inputs into the machine learning model, and then accurate predictions for other placements can be made. In our experiments the average prediction error was 6.6% and 4.4% on our two test systems, and in Section 2.5 we showed how this translates into more efficient machine packing while maintaining performance targets.

An important aspect of the solution to workload placement in Chapter 2 is that it is flexible and easy to adapt to future architectures. As long as new hardware architectures conform to the basic structure of having a hierarchy of shared resources, scheduling concerns can be developed for them easily. To demonstrate this, Table 6.1 shows the scheduling concerns that would be used for AMD's recently released Zen architecture [8, 32]. The Zen architecture is significantly different from the Bulldozer architecture used in Chapter 2 (Table 2.1). Many more resources are shared at the SMT level on the Zen architecture, including all the functional units rather than just the floating point units, and the L1 cache is shared as well. In addition it has another level of hierarchy because the L3 cache is shared at a lower level than the memory controllers. Despite these differences it is still simple to derive the scheduling concerns in Table 6.1 (current Zen platforms have symmetric interconnects, but it would also be simple to add an asymmetric interconnect concern if it is required in the future).

The second step after thread placement is memory placement. The Carrefour-LP algorithm (Chapter 5) places memory pages on NUMA nodes so that interconnect congestion is minimized and additionally it manages the size of memory pages so that TLB and page fault effects are weighed against the needs of balancing interconnect traffic. By measuring HPEs and utilizing instruction-based sampling, Carrefour-LP monitors NUMA statistics and per-page access patterns which it uses to make its decisions. This is done at runtime with low overhead and without requiring changes to the application. In extreme cases Carrefour-LP can improve performance by 3$\times$ compared to default Linux and it consistently improves performance over other standard techniques in other cases.

The practicality of our approach is what sets this work apart from other recent research in the field, such as Pandia [49]. Current research in the field tends to require burdensome offline profiling for new applications, or extensive expert knowledge to adapt the techniques to new hardware architectures. Neither is the case for our solution.

Still, there is much work to be done in the area. The workload placement algorithm of Chapter 2 assumes that the level of parallelism of the workload (i.e. the number of vCPUs the workload uses) is known beforehand. While this is a reasonable assumption in many use-cases such as clients of a cloud environment, it would still be quite helpful to be able to predict the effect of increasing or decreasing the level of parallelism. The results reported in Chapter 3 are a start to this problem but they only apply SMT systems, so more work could be done to extend the technique to a broader context. A second area of future work is studying the precise relationship between thread placement and memory placement. Our solution places threads first and then places memory, but it is possible that if thread and memory placement is considered at the same time then further performance gains could be obtained. Lastly, our workload placement method makes some assumptions. For example, it assumes that workloads will not interfere with each other which can lead to cores being left idle in some cases. Another example is that it only considers balanced placements. If these assumptions can be loosened or removed through further research then the method would be able to increase efficiency to an even greater extent. One avenue of complementary research would be to leverage application-specific data from the compiler or run-time system to improve predictions or help remove the assumptions and limits of the solution. For example compiler and run-time information can be used to determine how the load is balanced among workers (and the likelihood of stranglers), which could be useful for removing the assumption of balanced placements.

Bibliography

[1] 2011. Apache DayTrader Benchmark. (2011). Retrieved May 30th 2017 from http://geronimo.apache.org/GMOxDOC22/daytrader-a-more-complex-application.html. → pages 51

[2] 2013. SPECjbb2005. (2013). Retrieved May 30th 2017 from http://www.spec.org/jbb2005/. → pages 52, 104

[3] 2016. Dell EMC Enterprise Infrastructure Planning Tool. (2016). Retrieved May 30th 2017 from http://dell-ui-eipt.azurewebsites.net. → pages 1

[4] 2016. The WiredTiger key-value store. (2016). Retrieved May 30th 2017 from http://source.wiredtiger.com. → pages 7, 22

[5] 2017. Amazon EC2 Instance Types. (2017). Retrieved May 30th 2017 from https://aws.amazon.com/ec2/instance-types. → pages 13

[6] 2017. Selecting the number of clusters with silhouette analysis on KMeans clustering. http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html. (2017). → pages 22

[7] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic local alignment search tool. *Journal of molecular biology* 215, 3 (1990), 403–410. https://doi.org/10.1016/S0022-2836(05)80360-2 → pages 22

[8] AMD. 2017. *Processor Programming Reference PPR for AMD Family 17h Model 01h, Revision B1 Processors*. Sunnyvale, CA, USA. → pages 126

[9] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. 2010. Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation*

Techniques (PACT '10). ACM, New York, NY, USA, 319–330. https://doi.org/10.1145/1854273.1854314 → pages 99

[10] David A. Bader, John Feo, John Gilbert, Jeremy Kepner, David Koester, Eugene Loh, Kamesh Madduri, Bill Mann, and Theresa Meuse. 2007. *HPCS Scalable Synthetic Compact Applications #2 Graph Analysis*. → pages 51, 104

[11] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925 → pages 22, 51, 84, 104

[12] Mohammad Banikazemi, Dan Poff, and Bulent Abali. 2008. PAM: A Novel Performance/Power Aware Meta-scheduler for Multi-core Systems. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 39, 12 pages. http://dl.acm.org/citation.cfm?id=1413370.1413410 → pages 9

[13] Scott Barielle. 2011. Calculating TCO for energy. *IBM Systems Magazine: Power* (2011), 38–40. → pages 1

[14] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. 2008. A Regression-based Approach to Scalability Prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08)*. ACM, New York, NY, USA, 368–377. https://doi.org/10.1145/1375527.1375580 → pages 66

[15] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943 → pages 102, 123

[16] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579 → pages 99

[17] Tom Bawden. 2016. Global warming: Data centres to consume three times as much energy in next decade, experts warn. (2016). Retrieved September 16th 2017 from https://www.independent.co.uk/environment/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html. → pages 1

[18] J. Ross Beveridge, David Bolme, Bruce A. Draper, and Marcio Teixeira. 2005. The CSU Face Identification Evaluation System. *Machine Vision and Applications* 16, 2 (2005), 128–138. https://doi.org/10.1007/s00138-004-0144-7 → pages 84

[19] Abhishek Bhattacharjee and Margaret Martonosi. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE, Washington, DC, USA, 29–40. https://doi.org/10.1109/PACT.2009.26 → pages 102, 121

[20] Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*. → pages 22, 51, 84

[21] Sergey Blagodurov and Alexandra Fedorova. 2011. In Search for Contention-descriptive Metrics in HPC Cluster Environment. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE '11)*. ACM, New York, NY, USA, 457–462. https://doi.org/10.1145/1958746.1958815 → pages 7

[22] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. 2015. Multi-objective Job Placement in Clusters. In *Proceedings of the 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 66, 12 pages. https://doi.org/10.1145/2807591.2807636 → pages

[23] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC '11)*. USENIX Association, Berkeley, CA, USA. → pages 7, 99

[24] William Bolosky, Robert Fitzgerald, and Michael Scott. 1989. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of*

the 12th ACM Symposium on Operating Systems Principles (SOSP '89). ACM, New York, NY, USA, 19–31. https://doi.org/10.1145/74850.74854 → pages 97

[25] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 43–57. → pages 69, 72

[26] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. USENIX Association, Berkeley, CA, USA, 1–16. → pages 113

[27] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2011. *A Software Approach to Unifying Multicore Caches*. Technical Report MIT-CSAIL-TR-2011-032. MIT. → pages 99, 101

[28] Timothy Brecht. 1993. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *USENIX Experiences with Distributed and Multiprocessor Systems (SEDMS '93)*, Vol. 4. USENIX Association, Berkeley, CA, USA, 1–18. → pages 97

[29] Calin Cascaval, Evelyn Duesterwald, Peter F Sweeney, and Robert W Wisniewski. 2005. Multiple page size modeling and optimization. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*. IEEE, Washington, DC, USA, 339–349. https://doi.org/10.1109/PACT.2005.32 → pages 122

[30] Sheng Chen, SA Billings, and PM Grant. 1990. Non-linear system identification using neural networks. *International journal of control* 51, 6 (1990), 1191–1214. → pages 11

[31] Sheng Chen, Stephen A Billings, and Wan Luo. 1989. Orthogonal least squares methods and their application to non-linear system identification. *International Journal of control* 50, 5 (1989), 1873–1896. → pages 11

[32] Michael Clark. 2016. A New, High Performance x86 Core Design from AMD. Video. In *Proceedings of the 28th Symposium on High Performance*

Chips (HC '16). Retrieved May 30th 2017 from https://www.hotchips.org/archives/2010s/hc28/ → pages 38, 126

[33] Kevin Closson. 2009. Quantifying Hugepages Memory Savings with Oracle Database 11g. (July 2009). Retrieved May 30th 2017 from http://kevinclosson.wordpress.com/2009/07/28/quantifying-hugepages-memory-savings-with-oracle-database-11g/. → pages 102, 110

[34] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. *LWN* (March 2012). → pages 83

[35] Bill Dally. 2011. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *Proceedings of the 25th International Parallel & Distributed Processing Symposium (IPDPS '11)*. IEEE, Washington, DC, USA, 878–. https://doi.org/10.1109/IPDPS.2011.420 → pages 95

[36] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157 → pages 4, 7, 10, 68

[37] Arash Deshmeh, Jacob Machina, and Angela Sodan. 2010. ADEPT scalability predictor in support of adaptive resource allocation. In *Proceedings of the 24th International Parallel & Distributed Processing Symposium (IPDPS '10)*. IEEE, Washington, DC, USA, 1–12. https://doi.org/10.1109/IPDPS.2010.5470430 → pages 66

[38] Advanced Micro Devices. 2010. *AMD64 Technology Lightweight Profiling Specification*. Sunnyvale, CA, USA. → pages 96

[39] Norman Richard Draper and H Smith. 1966. *Applied regression analysis*. John Wiley & Sons. → pages 27

[40] P.J Drongowski and B.D. Center. 2007. *Instruction-based sampling: A new performance analysis technique for AMD family 10h processors*. Sunnyvale, CA, USA. → pages 76

[41] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. 2012. A Practical Method for Estimating Performance Degradation on Multicore Processors, and Its Application to HPC

Workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE, Los Alamitos, CA, USA, Article 83, 11 pages. https://doi.org/10.1109/SC.2012.11 → pages 7, 11, 26

[42] Stijn Eyerman and Lieven Eeckhout. 2010. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/1736020.1736033 → pages 66

[43] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. 2007. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE, Washington, DC, USA, 25–38. https://doi.org/10.1109/PACT.2007.40 → pages 9, 10

[44] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982 → pages 100, 101

[45] Justin Funston, Maxime Lorrillere, David Vengerov, Baptiste Lepers Jean-Pierre Lozi, Vivien Quema, and Alexandra Fedorova. 2018. A Practical Model for Placement of Workloads on Multicore NUMA Systems. In *Submitted to the 13th European Conference on Computer Systems (EuroSys '18)*. → pages 6

[46] Justin R. Funston, Kaoutar El Maghraoui, Joefon Jann, Pratap Pattnaik, and Alexandra Fedorova. 2012. An SMT-Selection Metric to Improve Multithreaded Applications' Performance. In *Proceedings of the 26th International Parallel & Distributed Processing Symposium (IPDPS '12)*. IEEE, Washington, DC, USA, 1388–1399. https://doi.org/10.1109/IPDPS.2012.125 → pages 7, 9, 39

[47] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 1999. Tornado: Maximizing Locality and Concurrency in a Shared Memory

Multiprocessor Operating System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Berkeley, CA, USA, 87–100. → pages 99

[48] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*. USENIX Association, Berkeley, CA, USA, 231–242. → pages 7, 102

[49] Daniel Goodman, Georgios Varisteas, and Tim Harris. 2017. Pandia: Comprehensive Contention-sensitive Thread Placement. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 254–269. https://doi.org/10.1145/3064176.3064177 → pages 3, 7, 12, 13, 126

[50] Mel Gorman and Patrick Healy. 2010. Performance Characteristics of Explicit Superpage Support. In *Proceedings of the 6th Annual Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA '10)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-24322-6_24 → pages 121

[51] Wei Huang, Jiang Lin, Zhao Zhang, and J. Morris Chang. 2005. Performance Characterization of Java Applications on SMT Processors. In *Proceedings of the 2005 International Symposium on Performance Analysis of Systems and Software (ISPASS '05)*. 102–111. https://doi.org/10.1109/ISPASS.2005.1430565 → pages 40

[52] Intel. 2011. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Santa Clara, CA, USA. → pages 49

[53] Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Portable Performance on Asymmetric Multicore Processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/2854038.2854047 → pages 66

[54] George H John, Ron Kohavi, and Karl Pfleger. 1994. Irrelevant features and the subset selection problem. In *Machine learning: proceedings of the eleventh international conference*. 121–129. → pages 27

[55] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. 2010. POWER7: IBM's Next-Generation Server Processor. *IEEE Micro* 30, 2 (March 2010), 7–15. https://doi.org/10.1109/MM.2010.38 → pages 43, 47

[56] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. 2008. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro* 28, 3 (May 2008), 54–66. https://doi.org/10.1109/MM.2008.48 → pages 7, 9, 10, 99

[57] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*. USENIX Association, Berkeley, CA, USA, 53–64. → pages 7, 80

[58] R. P. LaRowe, Carla S. Ellis, and Mark A. Holliday. 1992. Evaluation of NUMA memory management through modeling and measurements. *IEEE Transactions on Parallel and Distributed Systems* 3, 6 (Nov 1992), 686–701. https://doi.org/10.1109/71.180624 → pages 97

[59] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC '15)*. USENIX Association, Berkeley, CA, USA, 277–289. → pages 3, 7, 9, 10, 34, 37

[60] Lennart Ljung. 1998. System identification. In *Signal analysis and prediction*. Springer, 163–173. → pages 11

[61] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326 → pages 11

[62] Joshua Magee and Apan Qasem. 2009. A Case for Compiler-driven Superpage Allocation. In *Proceedings of the 47th Annual Southeast Regional Conference (ACM-SE 47)*. ACM, New York, NY, USA, Article 82, 4 pages. https://doi.org/10.1145/1566445.1566553 → pages 122

[63] Zoltan Majo and Thomas R. Gross. 2011. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *Proceedings of the 2011 International Symposium on Memory*

Management (ISMM '11). ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1993478.1993481 → pages 99

[64] Zoltan Majo and Thomas R. Gross. 2011. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/1987816.1987832 → pages 99

[65] Yandong Mao, Robert Morris, and Frans Kaashoek. 2010. *Optimizing MapReduce for multicore architectures*. Technical Report. MIT. → pages 22, 84, 104

[66] Harry M. Mathis, Alex E. Mericas, John D. McCalpin, Richard J. Eickemeyer, and Steven R. Kunkel. 2005. Characterization of simultaneous multithreading (SMT) efficiency in POWER5. *IBM Journal Research and Development* 49 (July 2005), 555–564. Issue 4/5. → pages 40, 65

[67] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (December 1995), 19–25. → pages 17, 51

[68] Celso L. Mendes, Jhy chun Wang, and Daniel A. Reed. 1995. Automatic Performance Prediction and Scalability Analysis for Data Parallel Programs. In *In Proceedings of the CRPC Workshop on Data Layout and Performance Prediction*. 45–51. → pages 66

[69] Andreas Merkel, Jan Stoess, and Frank Bellosa. 2010. Resource-conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 153–166. https://doi.org/10.1145/1755913.1755930 → pages 7, 9, 99

[70] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. CPHASH: A Cache-partitioned Hash Table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 319–320. https://doi.org/10.1145/2145816.2145874 → pages 99

[71] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *Proceedings of the 44th International*

Conference on Parallel Processing (ICPP '15). IEEE, 739–748. https://doi.org/10.1109/ICPP.2015.83 → pages 38

[72] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. USENIX Association, Berkeley, CA, USA, 89–104. → pages 122

[73] Ranjit Noronha and Dhabaleswar K Panda. 2007. Improving scalability of OpenMP applications on multi-core systems using large page support. In *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS '07)*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/IPDPS.2007.370683 → pages 102, 121

[74] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959 → pages 99, 100

[75] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, New York, NY, USA, 335–348. https://doi.org/10.1145/1755913.1755947 → pages 80

[76] Petar Radojkovic, Paul M. Carpenter, Miquel Moretó, Vladimir Cakarevic, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. 2016. Thread Assignment in Multicore/Multithreaded Processors: A Statistical Approach. *IEEE Trans. Computers* 65, 1 (2016), 256–269. https://doi.org/10.1109/TC.2015.2417533 → pages 13

[77] Lior Rokach and Oded Maimon. 2005. Top-down induction of decision trees classifiers - a survey. *IEEE Transactions* 35, 4 (Nov 2005), 476–487. https://doi.org/10.1109/TSMCC.2004.843247 → pages 61

[78] Theodore H Romer, Wayne H Ohlrich, Anna R Karlin, and Brian N Bershad. 1995. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. IEEE, Washington, DC, USA, 176–187. https://doi.org/10.1145/223982.224419 → pages 122

[79] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65. → pages 22

[80] Yaoping Ruan, Vivek S. Pai, Erich Nahum, and John M. Tracey. 2005. Evaluating the Impact of Simultaneous Multithreading on Network Servers Using Real Hardware. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/1064212.1064254 → pages 65

[81] Satish Kumar Sadasivam and Prathiba Kumar. 2011. SPECfp2006 CPI Stack and SMT Benefits Analysis on POWER7 Systems. In *Proceedings of the 17th Meeting of the IBM HPC Systems Scientific Computing User Group (ScicomP '11)*. Paris, France. → pages 40

[82] Hideki Saito, Greg Gaertner, Wesley Jones, Rudolf Eigenmann, Hidetoshi Iwashita, Ron Lieberman, Matthijs Waveren, and Brian Whitney. 2002. Large System Performance of SPEC OMP2001 Benchmarks. In *Proceedings of the 4th International Symposium on High Performance Computing (ISHPC '02)*. Springer-Verlag, Berlin, Heidelberg, 370–379. https://doi.org/10.1007/3-540-47847-7_34 → pages 51

[83] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. 2011. Database Engines on Multicores, Why Parallelize when You Can Distribute?. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1966445.1966448 → pages 99, 100

[84] Alex Settle, Joshua Kihm, Andrew Janiszewski, and Dan Connors. 2004. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE, Washington, DC, USA, 63–73. https://doi.org/10.1109/PACT.2004.7 → pages 65

[85] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. 2016. *United States data center energy usage report*. Technical Report LBNL-1005775. Lawrence Berkeley National Laboratory. → pages 1

[86] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2009. *Operating System Concepts, Eighth Edition*. Wiley, Hoboken, NJ, USA. → pages 12

[87] Allan Snavely and Dean M. Tullsen. 2000. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*. ACM, New York, NY, USA, 234–244. https://doi.org/10.1145/378993.379244 → pages 7, 9, 65

[88] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. 2002. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. ACM, New York, NY, USA, 66–76. https://doi.org/10.1145/511334.511343 → pages 65

[89] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. 2011. A Case for Scaling Applications to Many-core with OS Clustering. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*. ACM, New York, NY, USA, 61–76. https://doi.org/10.1145/1966445.1966452 → pages 99

[90] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 169–180. https://doi.org/10.1145/2594291.2594292 → pages 13

[91] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. 2010. Micro-pages: Increasing DRAM Efficiency with Locality-aware Data Placement. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. ACM, New York, NY, USA, 219–230. https://doi.org/10.1145/1736020.1736045 → pages 122

[92] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. ACM, New York, NY, USA, 415–424. https://doi.org/10.1145/139669.140406 → pages 103

[93] David Tam, Reza Azimi, and Michael Stumm. 2007. Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd ACM SIGOPS European Conference on Computer*

Systems (EuroSys '07). ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/1272996.1273004 → pages 3, 10, 66, 98

[94] Michael E. Thomadakis. 2011. *The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms*. Technical Report. Texas A&M University. → pages 48

[95] Transaction Processing Performance Council (TPC). 2010. *TPC Benchmark C Standard Specification*. San Francisco, CA, USA. → pages 22

[96] Transaction Processing Performance Council (TPC). 2014. *TPC Benchmark H Standard Specification*. San Francisco, CA, USA. → pages 22

[97] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1998. Simultaneous multithreading: maximizing on-chip parallelism. In *25 Years of the International Symposia on Computer Architecture (selected papers) (ISCA '98)*. ACM, New York, NY, USA, 533–544. → pages 39

[98] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*. ACM, New York, NY, USA, 279–289. https://doi.org/10.1145/237090.237205 → pages 69, 97

[99] Pinchas Weisberg and Yair Wiseman. 2009. Using 4KB page size for virtual memory is obsolete. In *2009 IEEE International Conference on Information Reuse & Integration (IRI '09)*. IEEE, Washington, DC, USA, 262–265. https://doi.org/10.1109/IRI.2009.5211562 → pages 121

[100] Chee Siang Wong, Ian K. T. Tan, Rosalind Deena Kumari, and Fun Wey. 2008. Towards achieving fairness in the Linux scheduler. *Operating Systems Review* 42, 5 (2008), 34–43. https://doi.org/10.1145/1400097.1400102 → pages 12

[101] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2485922.2485974 → pages 3, 7, 9, 10

[102] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer Performance Microscopy with Shim. *SIGARCH Comput. Archit. News* 43, 3 (June 2015), 170–184. https://doi.org/10.1145/2872887.2750401 → pages 118

[103] Gerd Zellweger, Denny Lin, and Timothy Roscoe. 2016. So Many Performance Events, So Little Time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '16)*. ACM, New York, NY, USA, Article 14, 9 pages. https://doi.org/10.1145/2967360.2967375 → pages 26

[104] Panyong Zhang, Bo Li, Zhigang Huo, and Dan Meng. 2009. Evaluating the Effect of Huge Page on Large Scale Applications. In *Proceedings of the 2009 International Conference on Networking, Architecture, and Storage (NAS '09)*. IEEE, Washington, DC, USA, 74–81. https://doi.org/10.1109/NAS.2009.18 → pages 121

[105] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO '14)*. IEEE, Washington, DC, USA, 406–418. https://doi.org/10.1109/MICRO.2014.53 → pages 3, 7, 10

[106] Jin Zhou and Brian Demsky. 2012. Memory Management for Many-core Processors with Software Configurable Locality Policies. In *Proceedings of the 2012 International Symposium on Memory Management (ISMM '12)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2258996.2259000 → pages 98

[107] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. ACM, New York, NY, USA, 129–142. https://doi.org/10.1145/1736020.1736036 → pages 3, 7, 9, 10, 99

[108] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Comput. Surv.* 45, 1, Article 4 (Dec. 2012), 28 pages. https://doi.org/10.1145/2379776.2379780 → pages 7

Appendix A

Supporting Materials

A.1 Non-Interference of Workloads on Separate NUMA Nodes

One of the assumptions of our workload placement solution stated in Section 2.2.2 is that workloads placed on separate NUMA nodes and not sharing interconnect links will not interfere with one another. Figures A.1–A.3 show the results of experiments (conducted on the AMD test system shown in Figure 2.2). The experiments run two workloads at a time: one potentially interfering workload and one workload where the performance change relative to running alone is reported. The slowdown observed is always less than 5% and usually less than 2%, confirming our assumption that co-scheduled workloads placed on separate NUMA nodes and not sharing interconnect links will not significantly interfere with one another.

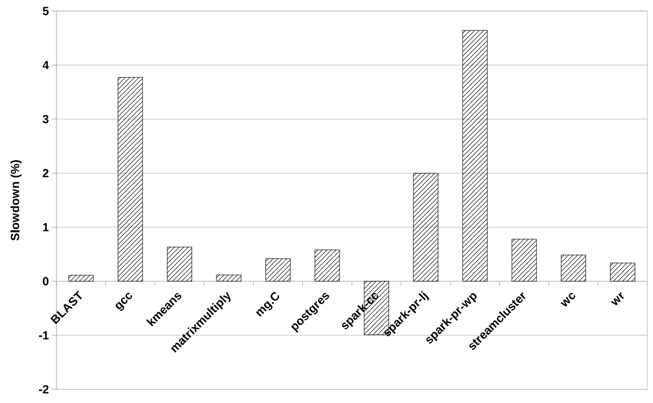


Figure A.1: Performance slowdown with cg.C as the interfering workload

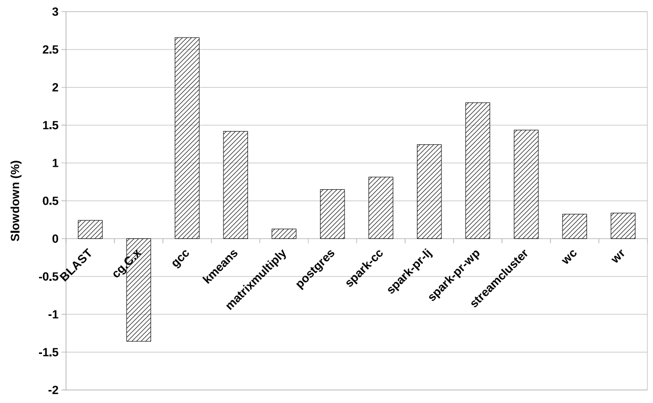


Figure A.2: Performance slowdown with mg.C as the interfering workload

A.2 Performance Prediction Results for the ML Model

This section contains the full performance prediction results for the ML model described in Section 2.4.2. Figures A.4 and A.5 show the actual and predicted performance for each workload for important placements on the AMD and Intel systems. The x-axis shows the IDs of the important placements, numbered 0–11 on the AMD system and 0–6 on the Intel system. The y-axis shows the performance in the placements relative to the baseline. Placement #2 was chosen as the baseline for the AMD system, and placement #1 for the Intel system.

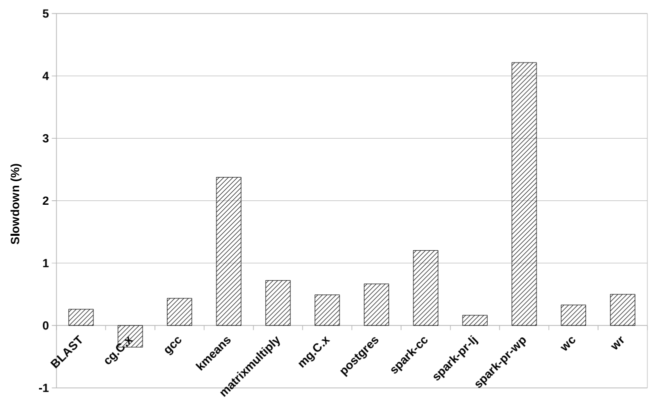


Figure A.3: Performance slowdown with streamcluster as the interfering workload

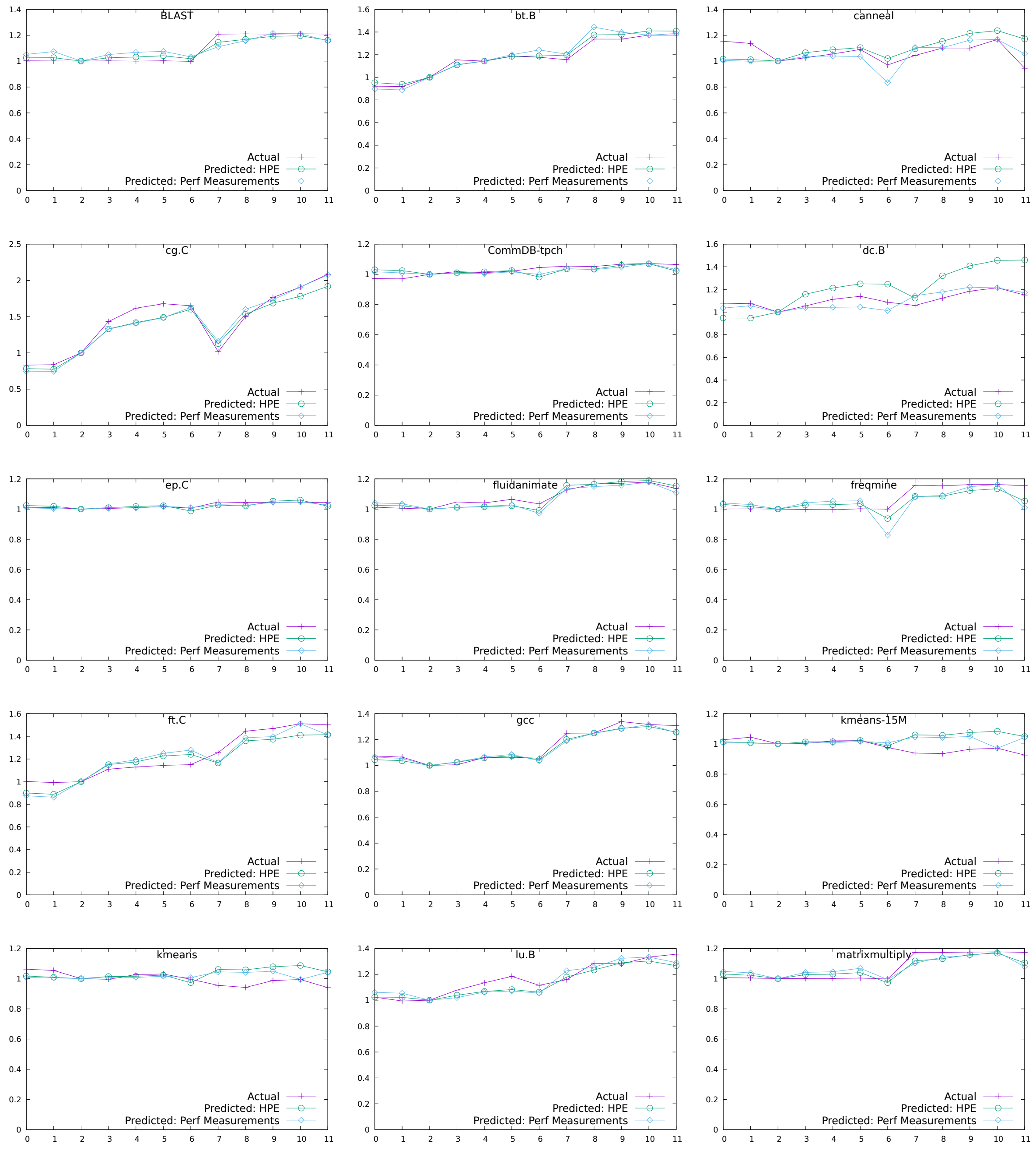

BLAST
bt.B
canneal
cg.C
CommDB-tpch
dc.B
ep.C
fluidanimate
freqmine
ft.C
gcc
kmeans-15M
kmeans
lu.B
matrixmultiply
Actual
Predicted: HPE
Predicted: Perf Measurements

Figure A.4

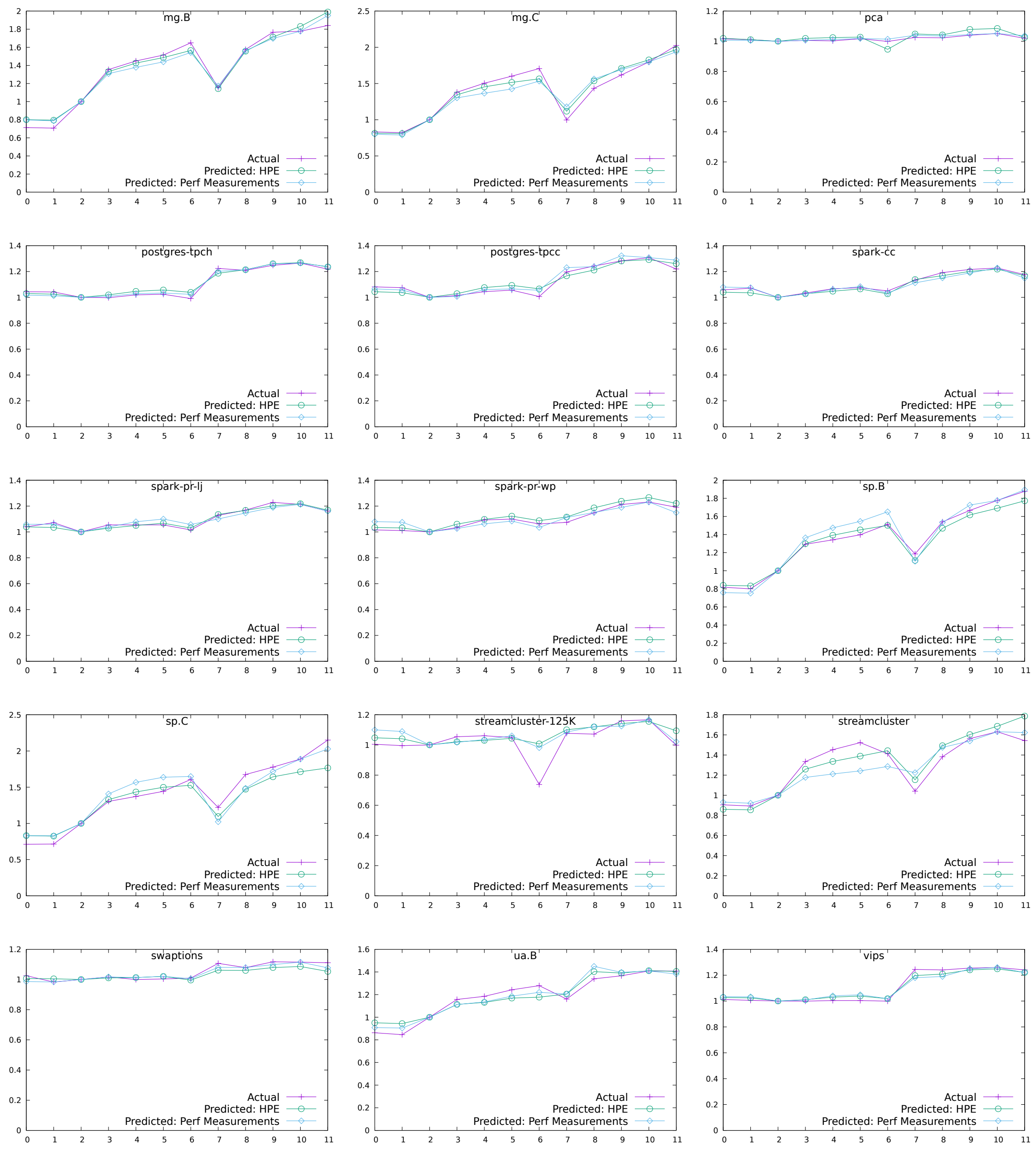

mg.B
mg.C
pca
postgres-tpch
postgres-tpcc
spark-cc
spark-pr-lj
spark-pr-wp
sp.B
sp.C
streamcluster-125K
streamcluster
swaptions
ua.B
vips
Actual
Predicted: HPE
Predicted: Perf Measurements

Figure A.4

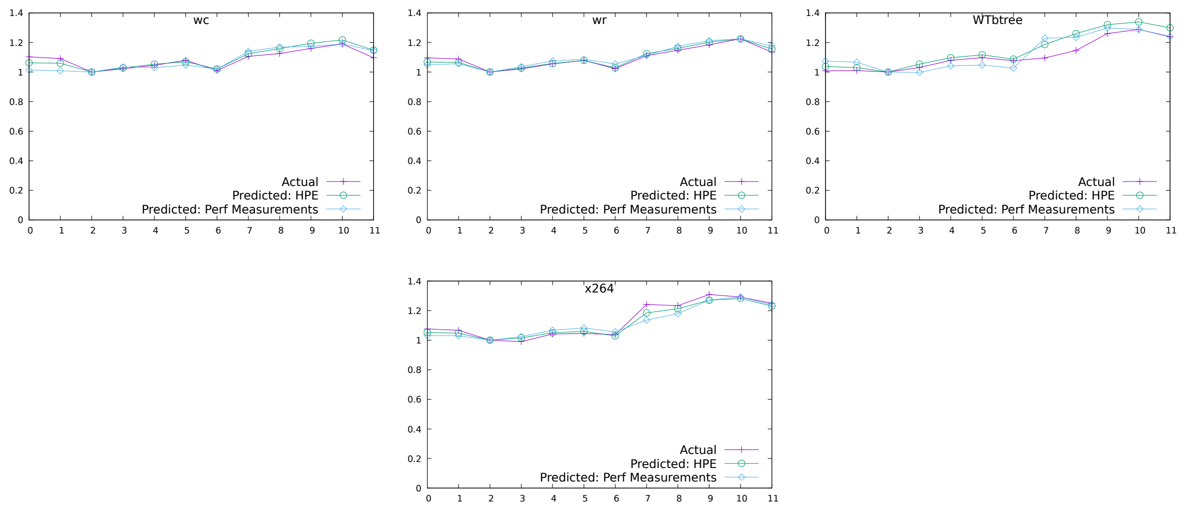


Figure A.4: Accuracy of predictions on the AMD system.

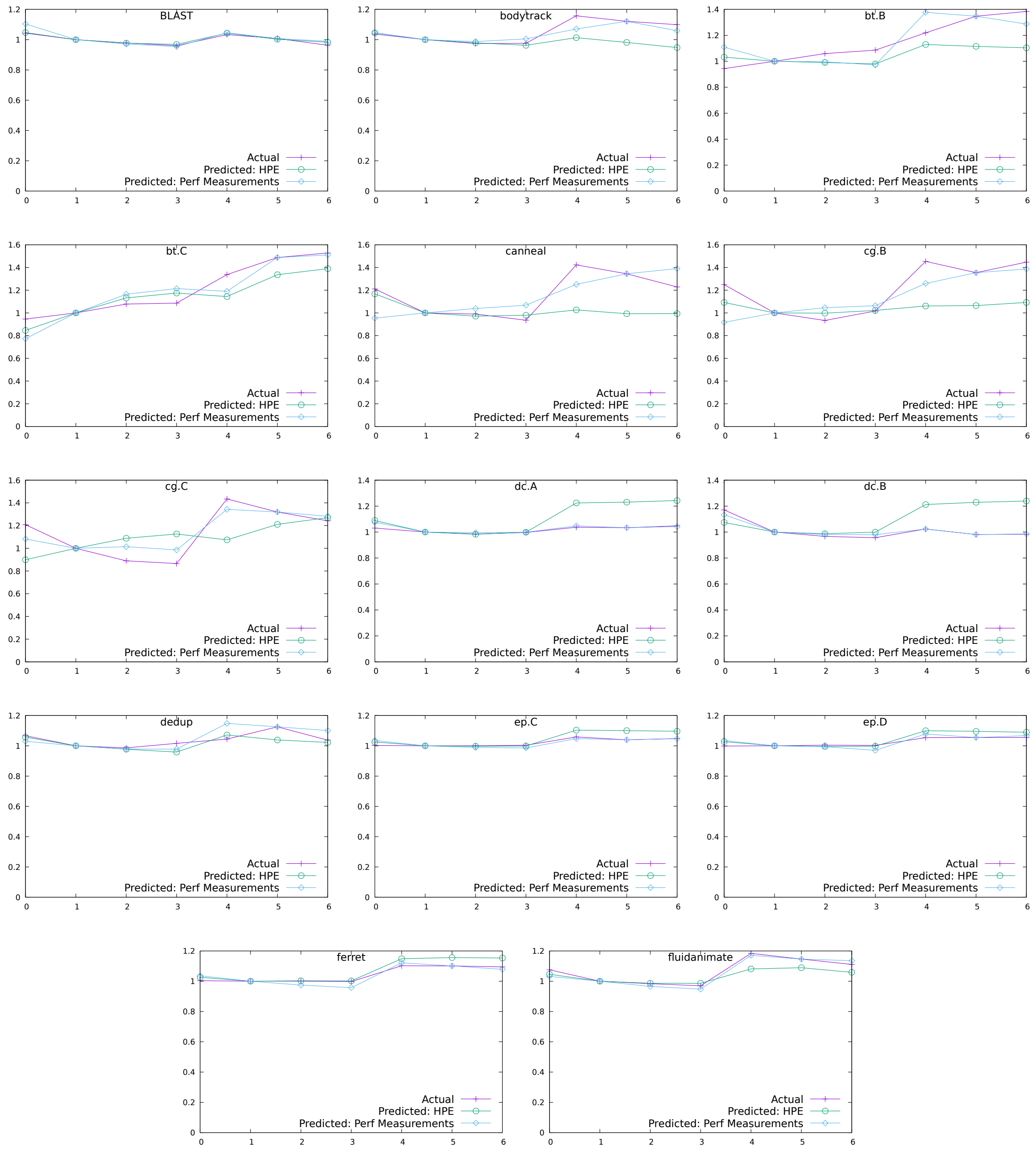

BLAST
bodytrack
bt.B
bt.C
canneal
cg.B
cg.C
dc.A
dc.B
dedup
ep.C
ep.D
ferret
fluidanimate
Actual
Predicted: HPE
Predicted: Perf Measurements

Figure A.5

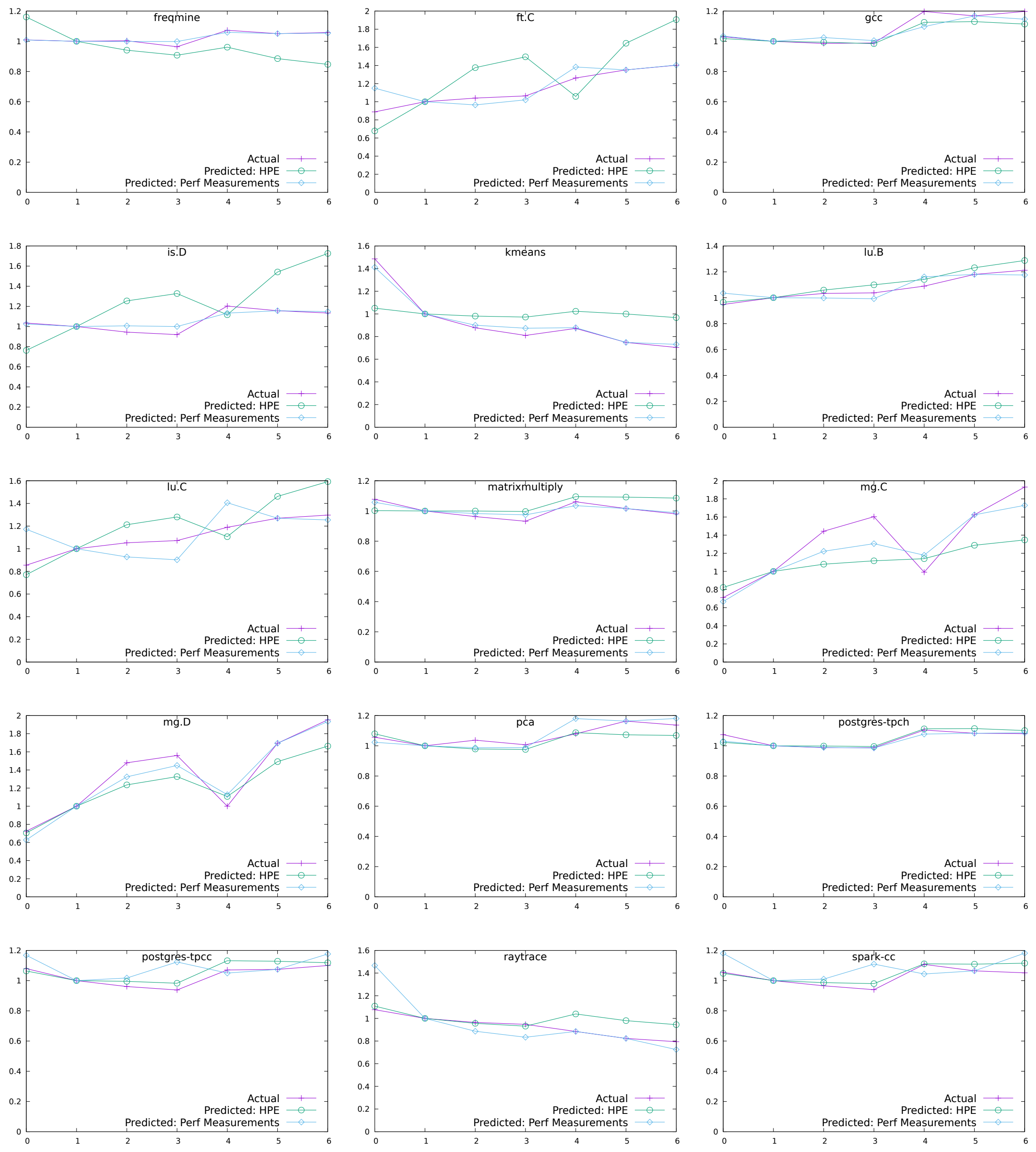

freqmine
ft.C
gcc
is.D
kmeans
lu.B
lu.C
matrixmultiply
mg.C
mg.D
pca
postgres-tpch
postgres-tpcc
raytrace
spark-cc
Actual
Predicted: HPE
Predicted: Perf Measurements

Figure A.5

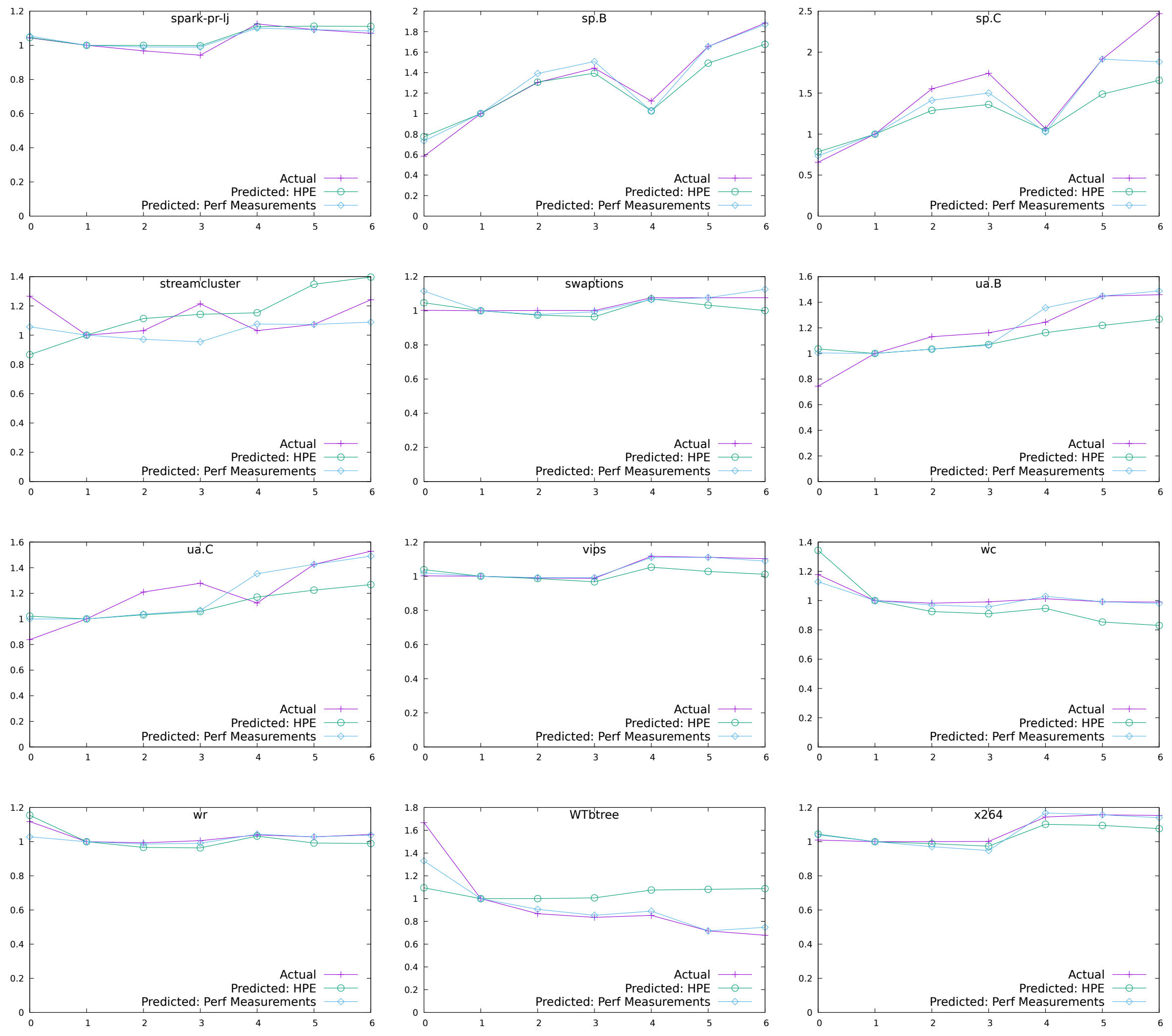


Figure A.5: Accuracy of predictions on the Intel system.