System Software Utilization of Hardware Performance Monitoring Information

by

Reza Azimi

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

System Software Utilization of Hardware Performance Monitoring Information

Reza Azimi

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2007

Over the past several decades, microprocessors have evolved to assist system software in implementing new functionality or in improving the performance of programs. The relative abundance of available silicon may further motivate introducing new hardware features other than those that are directly required for executing code. The main focus of this dissertation is on how new hardware support can collect accurate performance data so as to enable system software in making more informed decisions in improving the performance of programs.

First, we explore the problem of using Hardware Performance Counters (HPCs) to identify CPU bottlenecks accurately and efficiently. We address the problem of having a limited number of available HPCs by developing fine-grained HPC multiplexing that provides a large set of *logical* HPCs. We develop a simple and useful performance model, called stall breakdown to identify stressed processor components by focusing on cycles where the instruction completion stops. We generate the stall breakdown model by using HPC multiplexing online with negligible overhead.

Secondly, we explore different methods of fine-grained data sampling at the hardware level. Using the continuous data sampling features of the IBM POWER5 processor, we identify a new technique to produce data samples based on their source, and in a case study, we demonstrate how to use source-based data samples to accurately characterize data sharing patterns among concurrent threads to effectively support sharing-aware schedulers.

Finally, we propose novel hardware to track memory accesses at the granularity of virtual pages. Our proposed hardware is simple, efficient, and generic. We show how the proposed page access tracking hardware (PATH) can be used to improve performance in three different areas of memory management. In all three cases, we show that significant performance improvement can be achieved with negligible software overhead.

Dedication

for **Afsaneh**

who has been with me all along this path.

Acknowledgements

I would like to express my deep gratitude to those whose help and support have been instrumental in making the completion of this dissertation possible.

First and foremost, I am indeed grateful to my supervisor, Professor Michael Stumm, who has always provided me with his constant care and support, his guidance which proved to be crucial in many circumstances, and his much needed critical view of my work. Also, I greatly appreciate the help from the members of my PhD committee, Professor Ashvin Goel, Professor David Lie, and Professor Andreas Moshovos, whose technical feedbacks have played an important role in improving the quality of this dissertation. I should also thank the external examiner of this thesis, Professor Dimitris Nikolopoulos who spent much time in reading my thesis in some rough personal circumstances and provided a thoughtful and thorough evaluation of this dissertation. Last but not least, I need to thank Professor Angela Demke Brown whose active and dedicated collaboration has been instrumental in the development of the chapter 4 of this dissertation.

In the past five years, I have enjoyed both the company and the effective assistance of my colleagues at the department of Electrical and Computer Engineering at the University of Toronto. That includes Jonathan Appavoo who was my mentor in the early years, David Tam who has always been selflessly ready to help others, Livio Soares who has provided substantial help in developing and improving chapter 4, Adrian Tam, Raymond Fingas, Thomas Walsh, Alexandre Depoutovitch, and Adam Czajkowski. I should also thank members of the IBM research, Robert Wisniewski, Orran Krieger, and Dilma da Silva both for their intellectual assistance and for the important logistics they provided.

Throughout my PhD years, I depended so much on my wonderful wife, Afsaneh Fazly who, for me, has constantly been a source of unconditional support, energy, and positive inspiration. Without her, I certainly would not have been able to finish this dissertation. I am also grateful to my best friends and mentors, Ramtin Khosravi, Kiarash Bazargan, and Reza Ziaei who were inspiring figures for me to pursue my PhD.

Finally, I am grateful to the financial support I have received from the department of Electrical and Computer Engineering, University of Toronto, and the Government of Ontario.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As operating systems have evolved over the last fifty years, new hardware structures and mechanisms were periodically introduced to assist the operating system in its tasks. Most of these structures and mechanisms have one of the following objectives.

- **To facilitate implementation:** The hardware provides mechanisms that facilitate the implementation of operating system abstractions. Examples include the introduction of atomic instructions for implementing synchronization primitives more easily, the separation of kernel and user protection domain at the hardware level, the automatic virtual-to-physical address translation in the hardware, and memory-mapped I/O mechanisms.

- **To improve performance:** The hardware provides mechanisms that accelerate the execution of some of the most common operations inside the operating system. Examples include the introduction of the Translation Lookaside buffer (TLB) as a cache of page tables to accelerate the process of virtual-to-physical address translation, the introduction of Direct Memory Access (DMA) mechanisms to reduce the overhead of transferring large amount of data from and to peripheral devices, and automatic user-kernel stack switching to remove the burden of frequently copying data back and forth between user and kernel address spaces so as to reduce the cost of context-switching.

- **To provide information:** The hardware provides detailed information on the cur-

rent state of the computer system to assist system software either in implementing new functionality or in improving performance of existing operations. Examples include a special register for the currently executing thread to access the thread's private data more efficiently, a number of bits in the page tables that are automatically updated to indicate whether a page has recently been accessed or modified, and the introduction of hardware performance counters (HPCs) to help system software measure the performance of running applications more accurately.

While much prior work exists, and numerous proposals have been made over the years, we believe that a lot more work can be done on either introducing new ways of providing hardware support or enhancing the existing mechanisms. The relative abundance of available silicon may further motivate introducing new hardware abstractions.

The main focus of this dissertation is on how new hardware support can assist the operating system in collecting accurate performance data so as to enable the operating system in making more informed decisions. Usually, the state of the system represented by the hardware is detailed, low-level, semantically raw, and therefore, voluminous. An option is to offload to hardware much of the processing of such raw information into higher-level performance models so that the hardware provides the system software with more concise and, at the same time, semantically richer information. The problem with this approach, besides making the hardware design complicated, is that the information provided by the hardware will be specific to certain algorithms. Moreover, any further change to the software algorithms will require changes to the hardware.

Another approach is to add minimal hardware support to provide generic information and then have a thin layer of software that efficiently processes hardware-generated information and produces information that can be understood using a high-level performance model. The main advantage of this approach is that the hardware design will be simple and the information generated by the hardware remains generic so that in can be used by a variety of algorithms that may change over time. The key challenge to this approach, however, lies in the tradeoff between (i) functionality assigned to hardware to reduce the runtime overhead and (ii) functionality assigned to software to make the generated information more generic and flexible.

In this dissertation we explore this later approach in three different performance-related cases: (i) analyzing the CPU performance bottlenecks through Hardware Performance Counters (HPCs), (ii) analyzing data access patterns through hardware data sampling, and (iii) fine-grained page access tracking to improve performance of memory management algorithms. In each case, we start with the functionality currently provided by the existing hardware and then build efficient middleware to provide higher-level information that is based on a higher-level performance model. If the current hardware does not provide adequate information (even in raw form), we propose new, but minimal, hardware support. The following subsections briefly describe each of these three cases in more detail.

1.1 CPU Bottleneck Analysis

A Hardware Performance Monitoring Unit (PMU) is an integral part of most microprocessors today. It usually provides a few HPCs that are able to count, in real time, hardware events that occur in the processor. Potentially, the PMU can play an important role in analyzing performance and identifying the root causes of performance problems. However, the PMU is usually difficult to use effectively for a number of reasons. First, there are too few physical HPCs considering that any meaningful performance analysis requires the simultaneous monitoring of many different types of events. Moreover, HPCs primarily count low-level micro-architectural events from which it is difficult to extract high-level insight required for identifying causes of performance problems.

We explore two techniques that help overcome these limitations, allowing the use of HPCs to dynamically optimize both the operating system and user applications. First, fine-grained HPC multiplexing is introduced to make a larger set of *logical* HPCs available for analysis. Secondly, we introduce a performance summary model called *stall breakdown* which speculatively attributes CPU cycles to different hardware components, and as result, demonstrates which hardware structure is most stressed. Such a model can be used to guide automatic optimization both in operating system kernels or in user-level system software.

1.2 Analyzing Data Access Patterns through Hardware Data Sampling

Hardware data sampling is a PMU feature that is provided in some modern microprocessors such as Intel's Itanium and IBM POWER processors family [Inta, IBM06]. It allows for statistical sampling of data addresses that are used by programs under certain conditions such as TLB misses or data cache misses. Data sampling is a potentially powerful mechanism that can be used analyze the data access pattern of programs, the result of which can be used in a number of optimizations. Examples of such optimizations include prefetching data both for memory and CPU cache [LCF+03], superpage allocation and management [CDSW05], and NUMA page placement [THb].

While data sampling has proven to be effective in several cases [LCF+03, BH, THb], we believe there are a number of issues with the way current data sampling schemes are implemented in today's processors. First, the set of conditions under which data is sampled by hardware is not flexible, limiting how data sampling can be used. For instance, it is not possible to sample data based on the specific storage source from which the data is fetched. Secondly, only one selection criterion can be specified at a time and combining multiple selection criteria in either conjunctive or disjunctive forms is not supported. Finally, data sampling is not always *precise* in that the recorded data sample might not be an operand of the instruction that caused the sampling conditions to be fulfilled (e.g., a cache miss). This is mainly due to the high level of Instruction-Level Parallelism (ILP) implemented in todays microprocessors with deep pipelines, superscalar structure, and out-of-order execution.

We explore both hardware and software techniques to address the above-mentioned problems. We show an example where hardware data sampling could be used effectively to produce signatures to dynamically identify sharing among threads that run in a multiprocessor. We describe the data sampling features that are desired, and how we implemented a workaround in an existing microprocessor to indirectly obtain the information we needed. Finally, we provide specific suggestions for new data sampling features.

1.3 Fine-grained Page Access Tracking

To implement memory management algorithms, operating systems traditionally use a coarse approximation of memory accesses, obtained by monitoring page faults or scanning page table entries. The problem with this approach is that any information on the order in which pages are accessed is lost, yet, there are important classes of memory management techniques that require page access order information.

Unfortunately, hardware data sampling cannot be directly used for page-access tracking. The problem with data sampling (or any other statistical sampling technique) is that it favors only hot pages, for which memory management is quite trivial. However, more sophisticated memory management schemes require every single page access to be is recorded which is obviously impractical due to the very large volume of the information generated.

We propose simple, yet powerful, new hardware support for tracking page accesses with substantially higher precision and lower overhead than current software-based strategies can provide. We show how the use of this hardware facilitates the implementation of various algorithms that (i) implement more adaptive page replacement policies, (ii) allocate memory to VMMs, processes or virtual memory regions so as to improve performance or to provide isolation and better process prioritization, and (iii) effectively prefetch pages from virtual memory swap space or memory-mapped files when applications have non-trivial memory access patterns. Our simulation results show that significant performance improvements can be achieved, especially when the system is under memory pressure, while the basic overhead of providing fine-grained information to the operating system remains negligible for most applications.

1.4 Summary of Contributions

This dissertation makes a number of specific contributions in how hardware can provide the operating system with accurate and timely information that can be used for dynamic performance optimization purposes:

- We demonstrate the efficient implementation of fine-grained HPC multiplexing to allow larger number of logical counters with low overhead and reasonable accuracy.
- We develop a simple and useful performance model, called stall breakdown, to analyze CPU bottlenecks. Using facilities in the IBM POWER5 processor, we generate stall breakdown information online with negligible overhead.
- We demonstrate how hardware data sampling can be used in detecting the sharing patterns of concurrent threads on a shared memory multiprocessor with high precision and low overhead.
- We propose a novel hardware support for fine-grained page access tracking with minimal overhead and high precision. We also show how this hardware support can be used in improving memory management in three different areas: (i) adaptive page replacement policies, (ii) process memory allocation, and (iii) virtual memory prefetching.

1.5 Organization of the Dissertation

In Chapter 2, we present our work on how to use HPCs to analyze CPU bottlenecks. In Chapter 3, we demonstrate how we use hardware data sampling to detect sharing patterns among threads in a shared memory multiprocessor. Then, in Chapter 4 a new hardware support for fine-grained page access tracking is presented, along with three use cases that can effectively utilize the new hardware support in improving memory management. We conclude the dissertation by Chapter 5, which provides a summary of our work and presents directions for the future work. Chapters 2, 3, and 4 all have the same following structure. First, an overview of the problem and its existing solutions is presented. Then, we present our techniques to address the problem, followed by the description of our experimental framework and results. Then, a summary of related work is presented. Finally, each chapter ends with concluding remarks containing our conclusions and specific directions for future work.

Chapter 2

CPU Bottleneck Analysis

2.1 Introduction

Hardware Performance Counters (HPCs) are an integral part of modern microprocessor Performance Monitoring Units (PMUs). They can be used to monitor and analyze performance in real time. HPCs allow the counting of detailed micro-architectural events in the processor [Inta, Spr02, IBMb, IBM06, AMD02], enabling new ways to monitor and analyze performance of running software. There has been considerable work that has used HPCs to explore the behavior of applications and identify performance bottlenecks resulting from excessively stressed micro-architecture components [AV02, DCD03, SHC+04, CMDAN06, BH, ANP03]. However, there are a number of challenges that make HPCs difficult to be widely used in identifying CPU bottlenecks. In this section, we first provide a description of some characteristics of HPCs in today's processors that make them challenging to use effectively for online bottleneck analysis, and then, provide an overview of our techniques to deal with some of these problems.

2.1.1 Challenges of Using HPC

Some of the major challenges in using HPCs in today's microprocessors include the limited number of available HPCs, their complex interface, and the potentially high overhead of their use.

Processor	IBM POWER4	IBM POWER5	Intel Pentium 4	Intel	AMD Athlon
# of	and PPC970	(per H/W thread)	and Xeon	Itanium II	and Opteron
HPCs	8	6	9 pairs	4	4

Table 2.1: The number of HPCs available in today's major microprocessors.

Small Number of HPCs

PMUs typically have a small number of HPCs available. Table 2.1 shows the number of available HPCs in some of the more popular processors. Most processors provide up to 8 HPCs. Intel Pentium 4 is an exception with 9 pairs of HPCs. However, due to programming constraints imposed by the hardware implementation, not all of its HPCs can be programmed simultaneously. This is not specific to Intel Pentium 4, as we have observed many, rather restrictive cases of such constraints in the IBM PowerPC processors as well.

The limited number of HPCs implies that only a limited number of hardware events can be counted simultaneously at any given time. This is a serious limitation considering that detecting performance bottlenecks in complex superscalar, and potentially out-of-order, microprocessors often requires detailed and extensive performance knowledge of several processor components. For instance, in order to measure the L1 data cache miss rate on IBM POWER4, one has to use 4 HPCs simultaneously (L1 Loads, L1 Stores, L1 Load Misses, and L1 Store Misses). One way to get around this limitation is to execute several runs of an application, each time with a different set of hardware events being counted. Such an offline approach can be time-consuming (especially for long running applications), and is completely inappropriate for online analysis. Moreover, merging the traces generated from several application runs is not straightforward, because there are asynchronous events (e.g. interrupts and I/O events) in each run that may cause significant timing drifts.

Complex Interface

The events that can be monitored by HPCs are typically low-level and specific to a micro-architecture implementation. As a result, they are hard to interpret correctly without

detailed knowledge of the micro-architecture implementation. In fact, in the processors we have studied, most high-level performance metrics of interest such as *Cycles Per Instruction* (CPI), cache miss ratio, and memory bus contention, can only be measured by carefully combining the occurrence frequency of several hardware events. At best, this makes HPCs hard to use by average application developers, but even for seasoned systems programmers, it is challenging to translate the frequency of particular hardware-level events to their actual impact on end performance due to the complexity of today's micro–architectures.

High Overhead

Because PMU resources are shared among all system processes, they can only be programmed in supervisor mode. Thus, whenever a user process needs to change the set of events being captured, it must call into the operating system. These expensive kernel boundary crossings can happen frequently when a wide range of hardware events need to be captured for a single run of the application.

2.1.2 Our Approach

We have developed two techniques to address some of the problems mentioned above. First, to overcome the limitation in the number of HPCs, we multiplex the existing HPCs in a fine-grained way. This technique allows us to provide a much larger set of *logical* HPCs to the user, making it is possible to count the occurrences of many micro-architectural events during a single application run. The fine multiplexing granularity enables us to capture even short-lived fluctuations in the occurrence rate of hardware events. In Section 2.7 we present our statistical analysis to show that our multiplexing approach provides sufficient accuracy for performance tuning and optimization purposes.

Second, we use our multiplexing approach to concurrently interpret the impact of different hardware events on the applications' end-performance. We present a model called *Statistical Stall Breakdown(SSB)* which is based on the traditional CPI breakdown model that provides insightful and timely information on which micro-architecture components

are most stressed. SSB categorizes the sources of stalls in the microprocessor pipeline, and quantifies how much each hardware component (e.g., the caches, the branch predictor, and individual functional units) contributes to overall stall in a way that is simple and easy to understand for the user. SSB information is collected as the program runs and can be used, for example, by a dynamic optimizer to apply effective optimizations. We also show that the run-time overhead of collecting the SSB information is small.

2.1.3 Organization of the Chapter

In the next section, we provide more detailed background on basic HPC mechanisms in today's microprocessors. Then we present an overview the design of our HPC-based performance monitoring facility and the features it provides. We follow this section, by describing the details of fine-grained HPC multiplexing. Next, we present how the statistical stall breakdown model is defined and generated on a real microprocessor. Then, we provide more details about our implementation and the platform we used for our experiments. Next, we present the result of our experiments. We then discuss the related work, and finally, we present our conclusions and directions for future work.

2.2 Current HPC Capabilities

In most of today's microprocessors, HPCs are implemented as a small set of registers that each can be programmed to count the number of occurrences of a particular hardware event. There are several HPC control registers that define (i) which hardware events each HPC should count, and (ii) how the events are to be counted.

2.2.1 Event Types

The basic types of events that HPCs can count include CPU cycles, instruction completions, storage hierarchy accesses (hits and misses), TLB misses, branch mispredictions, and bus snooping activities. Some processors may also provide counts of more detailed event types that are related to the specific implementation of the micro-architecture, such

as prefetch buffer accesses, instructions that pass a given stage of the system pipeline, flushing of instructions upon certain conditions, and fullness of different queues inside the processor.

The hardware often provides limited capabilities on how the events at the hardware level can be combined or aggregated. Aggregation is usually in the form of summing up the events that occur on multiple instances of a component type (e.g., functional units, or load/store channels). The control registers can be used to define specific conditions under which an event is to be counted or not to be counted. For example, HPC can be programmed to either count while an interrupt service routine (ISR) is running or not. However, such conditions are usually primitive and fixed, i.e., it is not possible to logically combine several hardware supported conditions to define a new, more elaborate hardware event type.

2.2.2 Counting Methods

The value of HPCs can be recorded through either *instrumentation* or *sampling*. Next, we describe a brief background on these two methods.

Instrumentation

To use instrumentation, the source code is augmented, or the binary is patched, with code that configures the control registers and reads the HPCs at particular points in the program. The main advantage of instrumentation is that it is possible to gather information between two specific points in the dynamic execution path of a program. However, using instrumentation also has its drawbacks. First, modifying source or binary code can be time consuming and cumbersome. Second, it introduces perturbations mainly in two forms (i) the increase in the program code size and subsequently in the size of instrumented programs instruction cache footprints, and (ii) the overhead of executing extra code in the common path. This type of overhead is more pronounced in dynamic instrumentation systems [BH00, CSL04, TM94] where *trampoline* code, which usually includes several branch instructions, must be installed at each instrumentation site in

order to keep the program code layout unchanged. Such trampolines have negative impact on the spatial locality of program instructions that directly affects instruction cache performance.

Sampling

With sampling, the values of the HPCs are periodically collected either after a specified time period (i.e., *time-based sampling*) or after counting a specified number of a specific hardware event (i.e., *event-based sampling*). In order to do time-based sampling any timer facility can be used. For event-based sampling, the PMU can be programmed to generate an *overflow exception* after reaching a certain *threshold* on the count of a specific hardware event. To generate overflow exceptions, the control registers must be programmed properly, and the HPCs must be loaded with an initial value that corresponds to the overflow threshold.

Unlike instrumentation, sampling does not require modification of the source or binary of the programs but only requires an appropriate exception handler. Hence, sampling typically incurs lower overhead because no code is executed in the common path, and also because it doe not increase the code size and hence does not increase the program instruction cache footprint. The overflow exception handler, however, has a direct overhead due to its execution, and some indirect overhead due to polluting both the instruction and data caches. The overall sampling overhead, therefore, depends on the sampling frequency.

An alternative to reduce the sampling overhead is to use *polling* in combination with sampling. With this approach is that the operating system reads and records the value of the HPC registers at certain events that invoke the operating system (e.g., context switchings, page-faults, system calls, and other hardware interrupts). The basic idea is to piggyback the process of recording HPC values on already expensive operating system invocations that occur anyway as a result of system activities, and therefore, to avoid incurring extra exceptions (either timer-based or event-based) to record HPCs. If the operating system is not invoked as frequently as the desired rate for recording HPCs, exceptions can be raised.

The key advantage of the polling-based approach is that it reduces both the perturbation and the runtime overhead of sampling the HPC values. However, this approach introduces several challenges. First, it is difficult to explicitly control the sampling rate since operating system invocations may occur with an irregular pattern which directly depends on the activities of running programs. Secondly, with the current architecture of the PMUs, some useful information about the current state of execution is provided by the PMU, only at the time where an HPCs overflow exception occurs. By recording HPCs at arbitrary spots with respect to the function of PMU, such information cannot be captured. Finally, the modifications required to the operating system kernel in this approach is relatively intrusive as potentially many invocation points in the kernel must be modified to include calls to record HPCs.

Sampling and instrumentation methods can be used in a complementary fashion. In attempting to locate performance bottlenecks, it is typically too costly to start with instrumentation because the location of the problem is not known. Sampling can be used to efficiently identify program hot spots or stressed hardware components. Then, if the collected information is not sufficiently precise, instrumentation can be used on specific targets (e.g. the detected hot spots) to gather further detailed data at the instruction level.

2.2.3 Counting Modes

HPCs can be programmed to count events only when the processor is executing in user mode, in kernel mode, or in either of the two modes. With cooperation from the operating system, it is possible to further extend this and virtualize the HPCs by process or thread so that each process or thread is presented with their own set of dedicated HPCs. To implement this, the operating system must save and restore the value of the HPCs as part of the context switch.

2.3 Our Performance Monitoring Facility

We have designed and implemented a performance monitoring facility that can be used both for sampling and instrumentation. Figure 2.1 shows the block diagram of our facility. At the application level, users are provided with a programming interface through a user-level library. Thus, an application can be instrumented by inserting library calls manually or by using dynamic instrumentation tools. Calls from user applications are received by the operating system component which consists of a sampling module and a programming interface module.

The sampling module implements HPC multiplexing, PC and data sampling, and the stall breakdown model which we will discuss in detail later. The programming interface module allows for configuring the sampling engine, or for programming the hardware PMU directly. In the latter case, it receives the specification of a set of hardware events to be counted and automatically configures the hardware PMU. The values of the HPCs can be read directly by the user program, or logged in a per-process *trace buffer* by the sampling engine.

The key to achieving acceptable overhead is to minimize the frequency of crossing the user-kernel protection boundary. In our implementation, the sampling module is fully implemented in the operating system kernel. As a result, except for infrequent control operations (such as initialization or reset), there will be no context-switches between the user code and the performance monitoring module located in the kernel space.

The sampling engine can obtain HPC values either periodically or after a designated number of a hardware event occurrences. In both cases, we use PMU overflow exceptions. For periodic sampling we use one of the HPCs as the *CPU cycle counter*, allowing sampling intervals accurate down to a CPU cycle.

The frequency of sampling is a critical parameter. Sampling too infrequently may result in inaccuracies because changes in system behavior might be missed. On the other hand, too fine-grained sampling may result in unnecessarily high overhead. Our experience shows we can afford to take samples every 200,000 cycles (100 microseconds on a 2GHz CPU) with approximately 2% runtime overhead. This rate is our default

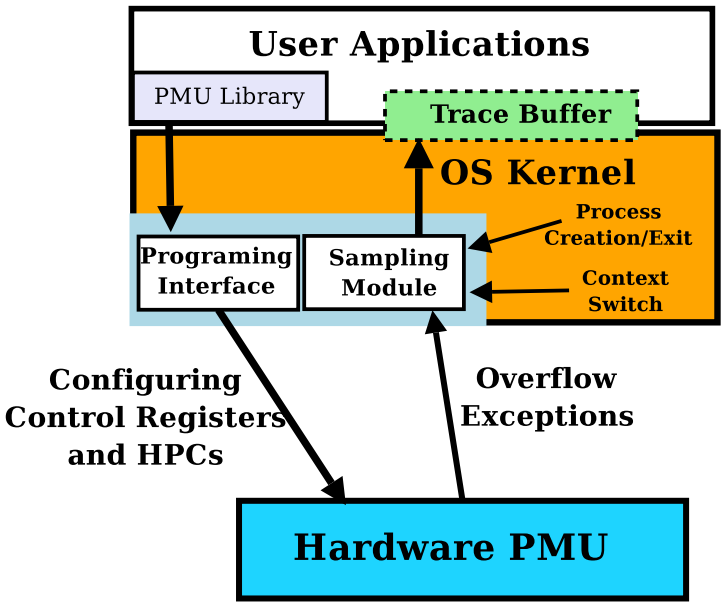


Figure 2.1: The block diagram of our HPC-based performance monitoring facility.

sampling frequency, although it can be overridden by the user.

In order to be able to isolate measurements of individual applications and the operating system, the sampling engine maintains a set of *HPC contexts*. HPC contexts are switched whenever the operating system switches processes. For this, the operating system must notify the sampling engine of all process creations and exits, as well as context switches. Upon each context switch, the current value of the HPCs are saved into the current HPC context and the corresponding HPC values for the next scheduled process is reloaded. There is inherent inaccuracy associated with this operation since each process inherits the residual hardware state manipulated by the previously running processes. To help reduce this inaccuracy, one may increase the size of the scheduling quantum so that the noise of initial *warm-up* period becomes insignificant.

For each process, there are three modes of operations: *kernel only, user only,* and *full system*. In kernel-only mode, hardware events are only counted when the hardware is in supervisor mode. This mode is appropriate if we are interested in monitoring operating system activities incurred by a particular target process. We assume kernel activities that occur in a process time slice are related to the target process. This assumption

may not be valid when context switches between different processes occur frequently or for interrupt handling. This, kernel-only mode is best suitable when a given application runs in isolation for a long time (for instance, on the order of several seconds) with no interruption. In user-only mode, logical HPCs (including the cycle counters) are suspended when the processor switches into the kernel. Finally, in full–system mode, HPCs count all hardware events whether due to kernel or application code. When a context switch occurs, the hardware events occurring both in the kernel and user mode will be counted by the HPCs of the new process.

We use the notion of an address space as the main indicator of a context. Therefore, the sampling engine is capable of reporting performance numbers for individual processes as well as the operating system. At this time, we do not differentiate between the user-level threads that share the same address space. One possible way of addressing this is to send a performance monitoring upcall to the user process when a hardware exception occurs so that a user-defined handler can associate the recorded HPCs with the current user-level context (e.g. user-level thread ID). Such a technique seems to be plausible only if there is a fast (low perturbation) upcall delivery mechanism. We do not currently support such an upcall mechanism.

2.4 Fine-grained HPC Multiplexing

To alleviate the problem of having a limited number of physical HPCs, we dynamically multiplex the set of hardware events counted by the HPCs using fine-grained time slices. The programming interface component takes a large set of events to be counted as the input and assigns them to a number of HPC *groups* such that in each group there are no conflicts due to PMU constraints. The sampling module assigns each group a fraction of g cycles out of a *multiplexing round* R, the time period in which all HPC groups will have a chance to be *scheduled*. At the end of each HPC group's time slice, the sampling engine automatically assigns another HPC group to be counted by the hardware PMU. The value that is read from an HPC after g cycles is scaled up linearly as if that group had counted during the entire R-cycle period. As a result, the user program (e.g. a

run-time optimizer) is presented with N *logical* HPCs on top of n *physical* HPCs where N can be an order of magnitude larger than n.

The system can easily be programmed to favor certain HPC groups by counting them for longer periods of time. This is accomplished by allocating multiple g-cycle time slices to the group. In fact, one can treat a period of g cycle as a unit for the hardware PMU time allocation. This PMU multiplexing scheme is analogous to the time-sharing of a CPU among processes. Figure 2.2 shows an example of four HPC groups, where each is given a time share (one or more time slices) of the multiplexing round. The share size of each group depends on the desired accuracy of the hardware events that are included in the group and on the expected rate of fluctuation of such events. Moreover, the accuracy may differ for different hardware events with the same share size. A default share assignment scheme might be overridden by explicit requests from the user that is interested in closely monitoring a specific hardware event.

Without loss of generality, for the rest of the chapter, we assume all groups are given equal time shares, which is one time slice (g cycles). We call $\frac{R}{g}$ the *Multiplexing Ratio*. Larger multiplexing ratios allow a larger number of logical HPCs. For instance, a multiplexing ratio of 10 can provide roughly 80 logical HPCs on an 8-HPC processor. This has to be traded-off with the fact that sampling accuracy decreases as the multiplexing ratio increases.

An issue that must be addressed is the fact that a sampling period may happen to coincide with loop iterations in the program. If the order of HPC groups within a period is fixed and a sampling period happens to coincide with a loop iteration, then an HPC group might always count the events that occur in the same fixed part of the iteration. To avoid this scenario, we randomize the order of the HPC groups in each sampling period. As a result, each HPC will have an equal chance of being located at any given spot of the iteration.

With HPC multiplexing, time is usually measured in terms of CPU cycles. Therefore, one counter in each HPC group is reserved to count CPU cycles. The use of cycle counters as timers allows us to define arbitrary fine time-slices down to a few thousand cycles. Another metric that can be used to define HPC group share sizes is the number of

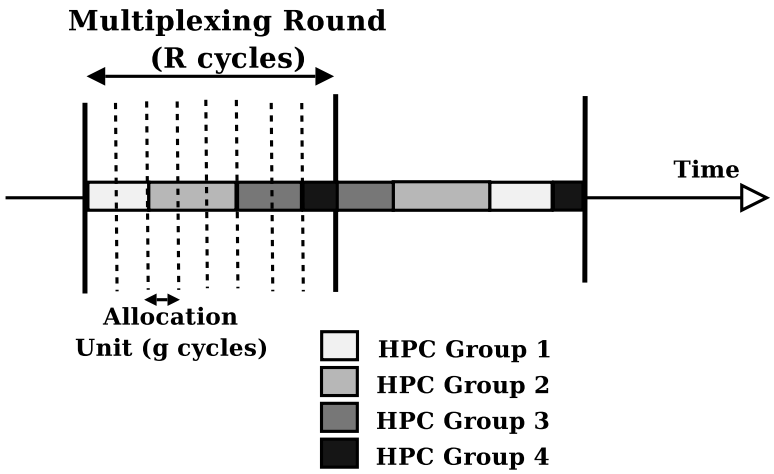


Figure 2.2: Time-Based Multiplexing example: There are four HPC groups in this example. Each HPC group is a collection of events that are counted simultaneously. An HPC group is counted in a number of time slices of g cycles within sampling period of R cycles. The order of the HPC groups is changed randomly in different sampling periods to avoid accidental correlations.

instructions retired. The main advantage of instruction–based multiplexing is that the HPC group share sizes are aligned more closely with the progress of the application. Share sizes, with respect to physical time, depends on the available instruction level parallelism (ILP) and the frequency of the miss events.

A pathological case for the multiplexing engine is the existence of a large number of short-lived bursts of a particular hardware event. If the burst time is shorter than R cycles, then the multiplexed HPC value of that hardware event might be inaccurate because the PMU actually counts the event only during a fraction of R, and thus it may miss short-lived bursts. However, we expect the execution of most applications to go through several phases, each longer than R, in which the occurrence rate of hardware events is stable in the common case. In Section 2.7, we provide experimental results that demonstrate that the statistical distance between the sampled and real rates of hardware events is small in most cases.

2.5 Statistical Stall Breakdown

With HPC multiplexing, a potentially large number of *logical* HPCs becomes available. As a result, a wide range of hardware events can be counted simultaneously. However, it is often difficult to interpret and understand the HPC values without having a proper model for CPU performance. For instance, we do not know whether having a million cache misses in a billion CPU cycles is a significant factor in the performance of the CPU or not, unless we have a model based on which we have an estimate of the penalty each cache miss incurs directly (i.e., by causing latency in the execution of instructions) or indirectly (e.g., by causing other pipeline structures to saturate, or by causing other useful cache lines to be replaced).

A naïve approach is to associate a *fixed* penalty to each event and simply multiply it by the event frequency to determine the actual effect of the event on CPU peformance [WLLB97]. While this approach is simple to understand and easy to implement, it is not accurate due to the fact that in a superscalar CPU with out-of-order execution, multiple latency-incurring events can overlap. Therefore, the naïve approach may result in an overly pessimistic estimate of the effect of each event on the CPU performance.

Another approach is to calculate a full Cycle-per-Instruction (CPI) breakdown where CPU cycles are attributed different hardware components or events so that each hardware component h accounts for CPI_h cycles per instruction out of the real CPI on average [HP03]. CPI breakdown is a simple and powerful model, as it can clearly identify both program and CPU bottlenecks. For instance, if we know that 60% of CPU cycles are spent waiting for cache misses to resolve, we know that the running programs are stressing the system caches and a dynamic optimizer will have to work on reducing the programs' CPU cache footprint, removing potential cache conflicts, or employing runtime prefetching.

The problem with the CPI breakdown model, however, is that it is extremely difficult to compute accurately on a real processor. The main reason is that in a superscalar out-of-order microprocessor many latency-incurring events overlap with each other. In such cases, it is not clear which component the caused latency should be charged to, as

each event alone can cause the latency even without the presence of the other.

A simplifying modification to the CPI breakdown model the *Statistical Stall Breakdown* model which attributes each stall cycles to processor components that are likely to have caused them. We losely define a *stall* cycle to mean a processor cycle in which no instruction completes (retires). Based on this distinction, the CPU cycles are either stall (non-completion) cycles or completion cycles.

The rationale behind focusing only on non-completion stall cycles (as opposed to all cycles) is based on two important observations. First, *most CPU cycles are stalls*. This is despite having large a instruction window and a wide pipeline, and doing sophisticated analysis for extracting Instruction-Level Parallelism (ILP). Table 2.2 shows average real CPI versus No-Stall CPI for sixteen applications from the SPEC CPU2000 benchmark suit, running on an IBM POWER5 processor. Also Figure 2.3 shows real CPI and no-stall CPI for the individual applications. It can be seen that between 60% to 85% of CPU cycles are stall cycles among these applications, with 73% being the average.

The second observation is that when there are no stall, CPU throughput, in terms of IPC, is fairly close to the pipeline width and is more or less application-independent. This is assuming that the design of the micro-architecture is well balanced and there are no obvious bottleneck components [KS04]. This can be seen in Figure 2.3: for most of the SPEC CPU2000 applications the *No-Stall CPI* is very close to the *ideal* CPI, which is around 0.2 on the IBM POWER5 processor (due to having a fetch bandwidth of 5 instructions per cycle)- on average, no-stall CPI is 0.35 among these applications. The real CPI of course, can vary dramatically for different applications and can be as high as 4.25 (e.g., for `mcf`). So, Table 2.2 shows that while the coefficient of variation for no-stall CPI is only 14%, it is as high as 53% for the real CPI for the selected applications.

These two observations suggest that in order to characterize curable performance bottlenecks (i.e, those that are not caused by limited pipeline width), it is sufficient to focus only on the stall cycles as opposed to *all* CPU cycles.

An important advantage of focusing only on stall cycles, is that it is easier to speculatively attribute each stall cycle to a particular hardware event, using the argument that if the particular hardware event had not occurred, the stall would not have oc-

Average Stall Cycles Percentage:	**73**
Average Real CPI:	**1.53**
Coefficient of Variation for CPI (%):	**53**
Average No Stall-CPI:	**0.35**
Coefficient of Variation for NSCPI (%):	**14**

Table 2.2: Summary of stall cycles and CPI for the SPEC CPU 2000 applications on the IBM POWER5 processor.

curred. The key observation is that, in most cases, the time hardware components spend in processing instructions will eventually result in stalls. Therefore, if the CPU resumes completing instructions after receiving the results from a hardware component, the last latency-causing hardware event in that componenet may be a good candidate as the *cause* of the stall. In order to do a stall breakdown, a basic hardware support is required to assign a *cause* to each stall. The IBM POWER5 and PowerPC970 processors both provide such a stall-to-cause assignment, and to the best of our knowledge, they are the only processors with this capability. We have used both these processors in all of our experiments for analysis the stall breakdown.

Such a stall-to-cause assignment is speculative mainly due to the fact that stalls from different causes may overlap and as a result, the latency caused by a component is hidden by the latency caused by another componenet. Hence, in order to identify the *real* causes for latency, an iterative scheme may be needed since removing or substantially reducing one cause of stall either improves performance proportional to the stalls assigned to it, or another cause for stalls to be revealed.

In the next subsection, we provide a more detailed description of our hardware model based on which hardware components and causes for stalls are defined.

2.5.1 Hardware Model

A simple hardware model is required to understand how different type of events that cause latency in the operation of a processor may result in stalls. In this section, we

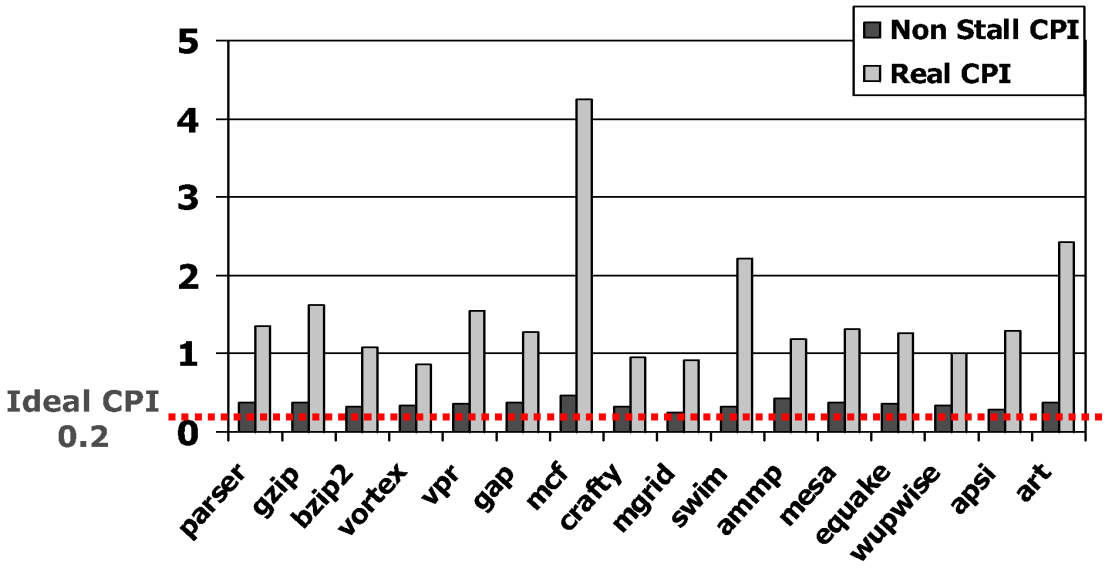


Figure 2.3: No-Stall CPI versus Real CPI for SPEC CPU2000 applications.

provide a high-level model of the functioning of a processor. While, our hardware model is influenced by the architecture of IBM POWER processors, we believe it is sufficiently general to be used for other modern microprocessors with minor modifications.

Figure 2.4 depicts the hardware model used and Figure 2.5 depicts the state-transition diagram for each instruction. Instructions are fed from the Instruction Cache (ICache) to the front-end pipeline in program order. Up to W instructions, at the level of the Instruction Set Architecture (ISA), can be fetched from the ICache in each cycle. These instructions are decoded and possibly translated into μ-instructions. The front-end pipeline generates *bundles* of B μ-instructions, each associated with one or more ISA instructions. In RISC architectures, however, we expect most ISA instructions to be translated into a single μ-instruction, and hence, we assume at most B ISA instructions can co-exist in a bundle. The μ-instructions within a bundle may have dependences between them; for example, the output of one may be used as an input for another.

At most one bundle can be *dispatched* in a single cycle, where each μ-instruction within the bundle is dispatched to its target Functional Unit (FU). The instruction bundles are dispatched in program order. At most one μ-instruction can be dispatched to an FU at

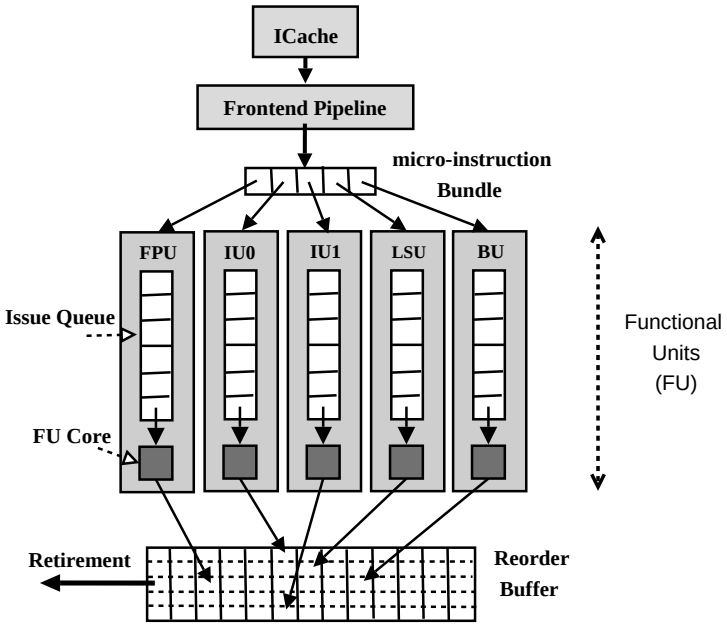


Figure 2.4: The basic hardware model for a super-scalar out-of-order processor. FPU stands for Floating-Point Unit, IU stands for Integer Unit, LSU stands for Load/Store Unit, BU stands for Branch prediction Unit, and FU stands for Functional Unit.

a time, although there may be several FUs of the same type. The total number of FUs may exceed the number of μ-instructions in each bundle, so some FUs may not receive new μ-instructions every cycle.

Before a μ-instruction bundle can be dispatched to the functional units, the following resources must be available for each μ-instruction in the bundle:

1. **Rename Buffer Entries**: Rename buffers are logical registers that are used to eliminate *Write-After-Read* and *Write-After-Write* dependencies.

2. **A Reorder Buffer Entry**: The reorder buffer is a queue that keeps track of the status of the dispatched bundles. Instruction bundles retire from the reorder buffer in the order they were dispatched after all of their μ-instructions have *finished*, and all earlier bundles have retired.

3. **Load/Store Buffer Entries**: Load/Store buffers are used to buffer the values

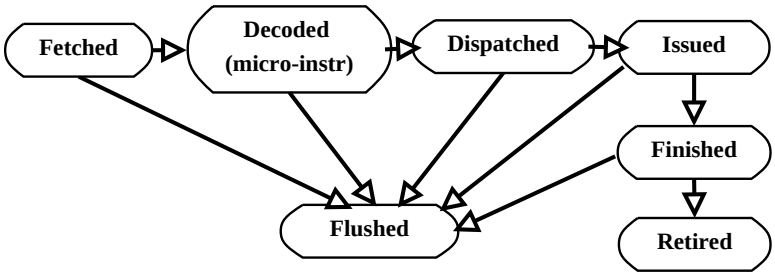


Figure 2.5: The state transition diagram for instruction execution.

read by the load instructions or written by the store instructions.

4. **FUs Issue Queue Entries**: Each FU has a separate issue queue. Each μ-instruction in the bundle needs an entry in the corresponding FU's issue queue.

If any of these resources are not available, the instruction dispatch will be delayed until they become available. Typically, this only occurs when there are long latencies in the FUs so that one of the structures mentioned above becomes full.

Once a μ-instruction bundle is dispatched, each μ–instruction in it will be queued in the corresponding FU issue queue. The instruction remains in the issue queue of the FU until all the data it depends on becomes available, after which it can be *issued*. An issued μ-instruction will be processed by the FU core to produce the result. Once the result is ready, the instruction's state becomes *finished*. The FU core may reject a μ-instruction for a number of reasons, in which case the instruction will be put back in the FU issue queue and will be re-issued later. Instruction issue occurs out-of-order with respect to program order. Once the μ-instruction bundle retires (completes), all resources allocated to it, including the entries in the rename buffers, the reorder buffers, and the load store buffers, are released. An instruction may be *flushed* for different reasons, including branch mispredictions or exceptions. When an instruction is flushed, all resources allocated to the instruction are released and the instruction must be fetched and decoded again later to execute.

A finished μ-instruction may *retire* only if, (i) all other μ-instructions in the instruction's bundle have also finished and (ii) all earlier (with respect to the program order)

Cause	Effect	Comment
ICache Miss	Empty Reorder Buffer	Instructions must be brought into the ICache either from L2 or memory.
Branch Misprediction	Empty Reorder Buffer	All in-flight instructions after the mispredicted branch are flushed.
Data Cache Miss	Retirement Stops	A delay in the LSUs to finish a load or store instruction due to a data cache miss.
Address Translation Misses	Retirement Stops	A miss occurs as the hardware accessed address translation structures (e.g. TLB). The miss either delays the processing of a load/store instruction in the LSU, or results in the temporary rejection of the instruction from the LSU.
LSU Basic Latency	Retirement Stops	A delay in one of the LSUs to finish the execution of an issued instruction.
Rejections	Retirement Stops	Any of the FUs (most likely the LSU) rejects an instruction for any (e.g. hitting a resource limit). The instruction must be reissued after some delay or reordering.
FPU Latency	Retirement Stops	A delay in one of the FPUs to finish the computation for an issued instruction.
IU Latency	Retirement Stops	A delay in one of the IUs to finish the computation for an issued instruction.
Other causes	Retirement Stops	A delay in any other hardware component, usually resulting in a pipeline flush.

Table 2.3: Types of miss events with their potential effect in the microarchitecture function.

bundles in the reorder buffer have already retired. Thus, bundle retirement happens in program order. At most one bundle can retire per cycle. Therefore, the maximum number of ISA instructions that in theory can retire in a cycle is equal to B (which is expected to be close to the fetch bandwidth W in a RISC architecture).

The key idea behind the stall breakdown model is that most bottlenecks can be identified by speculatively attributing a *cause* to each stall, i.e., a cycle in which no bundle from the reorder buffer can retire. There are two major categories of such stalls:

- *Empty Reorder Buffer:* This implies that the front-end has not been able to feed the back-end in time. Assuming the micro-architecture is designed and tuned properly, such situations happen mostly when there is an ICache miss, or when a branch misprediction occurs. We assume the hardware designates the most recent event (an ICache miss, or a branch mispredication) as the cause of the stall.

- *Completion Stops:* The reorder buffer is not empty, but the oldest bundle in the reorder buffer cannot retire. This happens mainly because one or more of its μ–instructions have not yet finished (i.e. they are waiting for an FU to provide the results). We assume in this case that once all μ-instructions of a bundle finish and

retirement resumes, the hardware will designate the cause of the stall as the last FU that finished a μ-instruction so that the instruction retirement could resume.

We call the hardware events that can cause a stall *miss events*. The miss events we consider in this study are listed in Table 2.3 along with the type of stalls they cause and the potential effect they may have.

The association between a stall and a miss event is not necessarily precise because of the dependencies among instructions within the same bundle. For instance, an instruction i may depend on the output of another instruction, j, of the same bundle. In this case, stalls caused by miss events during the execution of j are charged to i because it is the last μ–instruction in the bundle to finish.

Finally, even if a stall is identified as being caused by a particular event, removing that event does not necessarily translate into an elimination of the stall. This is because of the highly concurrent nature of superscalar out-of-order microprocessors and the fact that events may overlap so that removing one of them may not regain all the performance lost because of the stall. This issue is discussed extensively in other work [FBHN03a, FBHN03b, TTC02]. Addressing this issue in the general case is complex, because in today's out-of-order processors, hundreds of instructions may be in-flight simultaneously. To solve the problem in its generality, it is necessary to consider all possible interactions of any subset of concurrently executing instructions, which is beyond the scope of an on-line tool.

By taking all causes of stalls into account, the following formula can be used to speculatively characterize the potential CPU bottlenecks at each phase in the program execution:

$$CPI_{Real} = \sum_{i=0}^{n} Stall_i + CPI_C$$

where, $Stall_i$ is the number of stalls caused by miss event i in the monitoring period, and CPI_C is number of *completion cycles* in which at least one instruction is completed. In fact, CPI_C can be used as an estimate for the CPI that can be achieved by *ideal* hardware in which all miss events are removed and performance is solely determined by the program dependences and the width of the pipeline. Indeed, as we see in Figure 2.3, CPI_C is very

Source	Size	Latency
Local L2	2MB	14 cycles
Local L3	36MB	91 cycles
Local Memory	4GB	280 cycles
Remote L2	2MB	120 cycles
Remote L3	36MB	205 cycles
Remote Memory	4GB	307 cycles

Table 2.4: The size and approximate access latency of different sources in the memory hierarchy in IBM OpenPower 720 Machine

close to the ideal CPI for all applications we examined. The CPI_{Real} term is easily computed by dividing the number of elapsed cycles by the number of ISA instructions retired at any period of time. We also rely on hardware PMU features to provide values for $Stall_i$. As a result, we can accurately show how much gain is potentially achievable by reducing the miss events of a certain type.

2.5.2 Source-based Refinement

An important refinement to the stall breakdown model is to break down the stalls caused by instruction and data cache misses depending on the source from which the cache miss is eventually satisfied. Table 2.4 shows the different sources in the memory hierarchy in the IBM penPower720 machine and their approximate access latencies [VMTO05]. Several optimization techniques can exploit the source-based stall breakdown. For instance, we later show in Section 3.3 that if most of the data cache miss stalls are due to waiting for data are being fetched from caches on other processor chips, then it is likely that active read-write data sharing is occurring among threads of the same process. Another example is that if most of the data cache miss stalls are due to waiting for remote memory modules in a NUMA architecture, smart code and data placement or migration schemes are perhaps required.

Due to the lack of specific hardware support, we use a naïve approach to break down data cache miss stalls based on their sources. That is we define stalls waiting for a

memory or cache module m as follows:

$$MStall_m = Latency_m * AccessFrequency_m$$

where $Latency_m$ is the average latency for accessing module m (which is defined by the hardware characteristics) and $AccessFrequency_m$ is the frequency of accessing the module.

Such a naïve approach might be pessimistic as it does not take any overlap of multiple data cache misses in flight into account. Hence, $MStall_m$ could be much higher than its *real value*. However, as we show in Section 2.7.2, in practice, for many applications most of the long-latency memory instructions do not overlap, and as a result, stalls caused by data cache misses as reported by the hardware PMU (i.e., $Stall_{DataCacheMiss}$) is fairly close to the sum of the calculated stalls for all available memory and cache modules (i.e., $\sum_m MStall_m$). For a more accurate breakdown of data cache miss stalls based on source, additional, albeit minimal hardware support is required which, to the best of our knowledge, is not available in any of todays' mainstream processors.

2.6 Implementation

In this section, we present about our experimental platform as well as more details about the implementation of our performance monitoring facility.

2.6.1 Real Hardware versus Simulation Environment

We decided to evaluate HPC-multiplexing and stall breakdown on a real microprocessor as opposed to using a cycle-accurate machine simulator because of two major advantages a real environment offers. First, instruction execution on a real processor is much faster than in a simulation environment. Depending on the level of details the simulator is modeling, experiments can take several thousand times more than running them on a real processor. Such a vast difference in execution speed allows us to collect data for much longer periods of program execution, making the collected data more representative and the resulting analysis more complete and accurate.

The second reason for not choosing a simulator is that even detailed cycle-accurate simulators may not be able to reflect some of the limitations of the implementation of real microprocessors. For instance, in a simulation environment, virtually any type of events can be monitored assuming there is no cost or complexity for the monitoring. In real environment, however, many factors such as chip space, wire latency, and complexity of implementation determine whether it is feasible to monitor a certain type of event or not.

There are two major drawbacks in using a real microprocessor, however. First, the hardware programming interface is fixed and provides limited information. It is not possible, for instance, to measure the length of time between any two arbitrary events (e.g., two consequtive stall-causing cache misses). Secondly, because of the complexity of a real processor, understanding the semantics of the hardware events is challenging and requires significant internal (and potentially proprietary) knowledge of the processor implementation. Often, information at such level of details is not provided in public documentation.

2.6.2 Hardware

We have implemented and evaluated our HPC-based performance monitoring facility on two IBM processors, PowerPC970 [IBM06], and POWER5 [SKT+]. The PowerPC970 processor is used in the Apple PowerMac G5 workstation and the POWER5 processor is used in a the IBM OpenPower720 Express computer system. In terms of execution core and pipeline structure, the two processors are quite similar. However, there are significant differences in terms of processor interconnection and the structure of the memory hierarchy. Moreover, the POWER5 processor supports simultaneous multithreading (SMT) to allow instructions from several hardware threads to be dispatched and issued to the functional units simultaneously. In this study, however, we have not explored the challenges of HPC-based performance monitoring under the SMT execution model.

The specifications of the two processors are listed in Table 2.5.

The PMU in both processors is capable to count the number of stalls caused by miss events including the ones listed in Table 2.3. When the CPU stops completing

	PowerPC97	**POWER5**
Clock Rate (GHz)	1.8	1.5
L1 ICache (KB)	32	32
L1 DCache (KB)	64	64
TLB Size	1024	1024
L2 (KB)	512	1875 (shared by two cores)
Fetch Bandwidth	5	5
No. of FXUs	2	2
No. of LSUs	2	2
No. of FPUs	2	2
No. of HPCs	8	6

Table 2.5: The specifications of the IBM PowerPC970 and POWER5 processors used for our experiments.

instructions, a counter starts counting the number of stalls. Once the CPU resumes completing instructions, the stall count is charged to the last miss event speculatively, as the cause for the stall period. The assignment of stall to cause is speculative, since (i) several events can happen in a single cycle and the PMU chooses one of them to attribute the just-ended stall period, and (ii) multiple stall causes may overlap, yet the stall length is attributed to just a single cause.

2.6.3 Operating System

We implemented our performance monitoring facility both in K42 and Linux operating systems. K42 is an open-source research operating system designed to scale well on large, cache-coherent, 64-bit multiprocessor systems [IBMa]. It provides compatibility with the Linux API and ABI. The K42 kernel is designed in an object-oriented fashion, a feature that allows for easier prototyping.

The sampling engine in both operating systems is built as a fairly small kernel module (a few hundred lines of C/C++ code). The OS kernel is slightly modified to notify

the sampling engine of all process creations, exits, and context switches. In K42, we exploit the fact that all major process management events along with other the operating system events are recorded in performance monitoring *trace buffers*. Therefore, upon each overflow exception, the sampling engine checks whether a context switch has recently occurred by consulting the trace buffer. Using this scheme, there is a delay in detecting context switches, but because the granularity of context switches is usually around 10 milliseconds, which is two orders of magnitude larger than the multiplexing granularity we typically use, the imprecision added by a small delay in detecting context switches is insignificant.

In order to record the gathered HPC values, in K42 we used the existing performance monitoring infrastructure [WR03]. The infrastructure provides for an efficient, unified and scalable tracing facility that allows for correctness debugging, performance debugging and on-line performance monitoring. Variable-length event records are locklessly logged on a per processor basis in the trace buffer mentioned above. The infrastructure is uniformly accessible to the operating system and user programs. The recorded events are encoded using XML, and thus, much of the implementation of adding and processing new events is automated [WSS+04]. The HPC values gathered by the sampling engine are added to the buffers and thus available to any interested party.

In Linux, we integrated both fine-grained HPC multiplexing and statistical stall breakdown into the publicly available Linux's OProfile toolkit [OPr]. OProfile is a system-wide profiler that uses HPCs for both time-based and event-based PC sampling of both user programs and the operating system kernel. To implement HPC multiplexing, we added a trace buffer similar to that of K42, to Oprofile's kernel module, mainly to be able to record potentially large of logical HPC vectors. Also, Oprofile's overflow exception handler is modified to switch between different HPC groups.

The PMU library provides a number of calls to allow user programs to include a set of HPC groups to be counted in different counting modes. Also, it allows the user to change the multiplexing round as well as the period between each two consequtive recording of the logical HPC vector into the trace buffer. In K42, the PMU library communicates with the sampling engine through a set of system calls while in Linux, the PMU library

uses newly created entries in the `oprofilefs` file system for this purpose.

2.7 Experimental Evaluation

We developed and ran a number of experiments to evaluate our approach. In this section, we describe these experiments and present their results. First, we briefly describe how we validate the basic values we read from the HPCs for different hardware events. We then present the results of our statistical analysis of sampling accuracy and show how accuracy changes as a function of multiplexing granularity and multiplexing ratio. Finally, we analyze the accuracy and usefulness of computing SSB values.

In our experimental analysis, we have used a subset of the SPEC2000 benchmark suite [(SP], SPEC JBB2000 [Sta], VolanoMark [Vol], and MySQL database server [MyS]. Throughout this section we present our results only for a representative set of applications.

Application	gzip	gcc	perlbmk	crafty	applu	mgrid	art	mesa
Instructions Retired	0.01	0.06	0	0	0	0.05	0	0.12
L1 DCache Loads	0.09	0.04	0	0	0.02	0.07	0.03	0.22
L1 DCache Stores	0.13	0.08	0	0	0	0.03	0	0.10
L1 DCache Misses	1.21	0.05	0	0	0.02	0.07	0.07	0.05
ICache Misses	N/A	0.12	0.02	0.09	0.02	N/A	0.03	N/A
TLB Misses	N/A	0.11	N/A	N/A	0.01	0.15	N/A	N/A
ERAT Misses	0.79	0.13	0.01	0.02	0.07	0.71	0.42	0.21
L2 Cache Misses (Data)	0.24	0.01	0.07	0.02	0.02	0.06	N/A	0.17
Branch Mispredicts	0.36	0.05	0.01	0	0.02	0.18	0	0.13

Table 2.6: KL-distance between probability distribution P, which is obtained by fully counting hardware events, and P', which is obtained through multiplexing. The multiplexing ratio is set to 10 and the multiplexing round R is set to 2 million cycles. N/A implies the event is less frequent than once every 10,000 cycles, on average, and value 0 is used for any value less than 0.01.

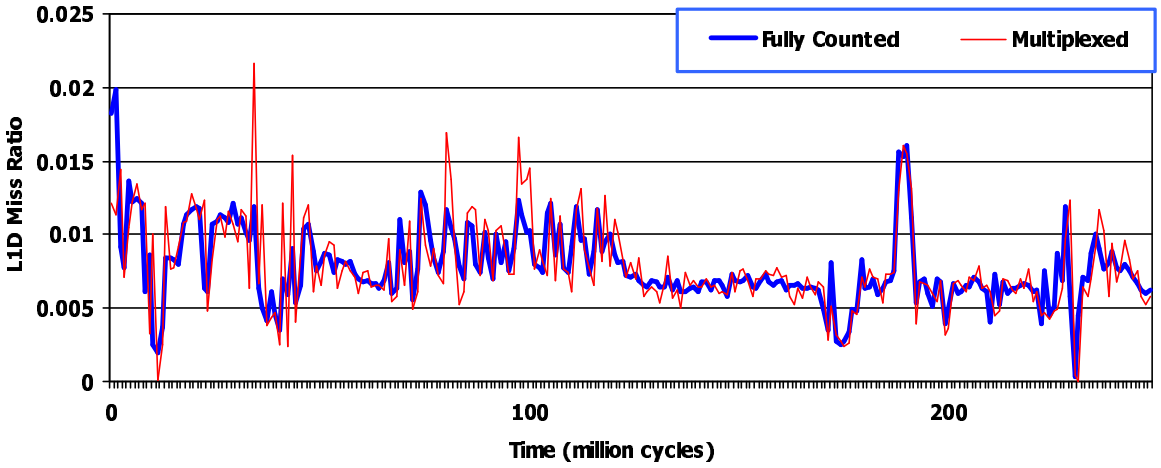


Figure 2.6: Comparing fully counted L1 DCache Miss Ratio with multiplexed (and extrapolated) counts of the same event when running `gcc`. The multiplexing ratio is set to 10 and the multiplexing round is set to 2 million cycles.

2.7.1 Accuracy of Multiplexing

In order to measure the accuracy of multiplexing versus fully counting the hardware events, we use a statistical analysis. When counting events fully, we associate with each hardware event, e, a probability distribution $P_e(R_i)$ representing the probability of event e occurring in the time interval R_i. $P_e(R_i)$ can be simply calculated by dividing the frequency of e re-occurring in the interval R_i by the total number of e events during a monitoring session. That is if N_e is the total number of occurrences of event e, and $N_e(R_i)$ is the number of occurrences of event e in interval R_i, the *probability* of event e occurring within R_i is calculated as $P_e = \frac{N_e(R_i)}{N_e}$ so that $\sum_{i=0}^{N} P_e(R_i) = 1$.

With multiplexing, on the other hand, we count how many times e occurs in a subinterval of R_i, and linearly scale it to the entire interval, which will give us another probability distribution $P'_e(R_i)$. A key question is how the two distributions, P_e and P'_e, corresponding to the actual counts and sampled counts, differ. To answer this question, we use *Kullback Leibler distance* (KL-distance), which is often used to measure similarity (or distance) between two probability distributions[CT03]. KL-distance is defined as:

$$K(P_e, P'_e) = \sum P(x) \log P_e(x)/P'_e(x)$$

and computes the geometric mean over $P_e(x)/P'_e(x)$.

For instance, if $P_e(x)$ is 1.5 times $P'_e(x)$, $\log P_e(x)/P'_e(x)$ is equal to 0.58 (we use log_2 everywhere in our calculations), and if $P_e(x)$ is 8 times $P'_e(x)$, then $\log P_e(x)/P'_e(x)$ would be equal to 3. Therefore, the noise of very short periods of time where $P_e(x)$ is drastically different from $P'_e(x)$ will be reduced.

The reason we use KL-distance (as opposed to, for instance, the mean over $|P_e(x) - P'_e(x)|$) is that in the context of runtime optimization, the absolute values of the hardware event counts are often not really important because there are many short transient states in the hardware. What is more important is whether there is a significant and rather stable shift in the rate of occurrences of a particular hardware event that lasts for a sufficiently long period of time to be worth considering. Therefore, although there may be sampling intervals in which the values of P_e and P'_e differ significantly, if such intervals are limited in number and isolated, they do not distort the distance measure due to the log factor in KL-distance.

In this study, we consider any value of $K(P_e, P'_e)$ below 0.20 to be acceptable. Informally speaking, we consider multiplexing to be adequate if the difference between the values of two probability distributions on average does not exceed 15%.

We measured $K(P_e, P'_e)$ for a large number of hardware events for the selected SPEC2000 applications. Table 2.6 shows the results for several important hardware events and some of the applications. The N/A entries imply that the hardware event was on average less frequent than once per 10,000 cycles, and hence, insignificant. The 0 entries imply the actual value of $K(P_e, P'_e)$ was less than 0.01. The samples are collected over 6-billion cycles (after skipping over the first billion instructions). The multiplexing interval R is 2 million cycles, and the multiplexing ratio is 10. As it can be seen from Table 2.6, the KL-distance value is small for most hardware events in a majority of applications, with a few exceptions we discuss later in this section. In Figure 2.6 we graphically depict the rate of occurrences for L1 DCache Miss Ratio for `gcc` both when the event is fully counted as well as in the multiplexed mode. It can be seen that the multiplexed event rate accurately follows all significant and steady changes in the real occurrence rate of the hardware event even though there are differences over small periods of time.

There are a few cases in Table 2.6 with unacceptably high values. However, we note

that these cases all correspond to fairly infrequent events (one per 100 cycles on average). Although infrequent events are unlikely to cause performance bottlenecks, we explored this issue further by varying the multiplexing granularity and multiplexing ratio for them. We ran several experiments with `gzip` for which at least three hardware events have a relatively large KL-distance: the L1 data cache miss, ERAT (Effective to Real Address Table which is used by IBM POWER processors as a cache for their relatively large TLBs) miss, and branch misprediction. Figure 2.7 shows the results of the experiments. The graph on the left shows how the accuracy changes as a function of the multiplexing granularity. As a general rule, larger granularities have higher accuracy for infrequent events. Therefore, we change the multiplexing granularity from 200,000 to 500,000 cycles. We then wanted to know how sensitive the accuracy is to the multiplexing ratio in this multiplexing granularity. The graph on the right shows the results of our experiments. It appears that none of the three hardware events is highly sensitive to the multiplexing ratio. The general conclusion we draw from these experiments is that it is better to use larger granularities (with a fixed multiplexing ratio) for infrequent hardware events.

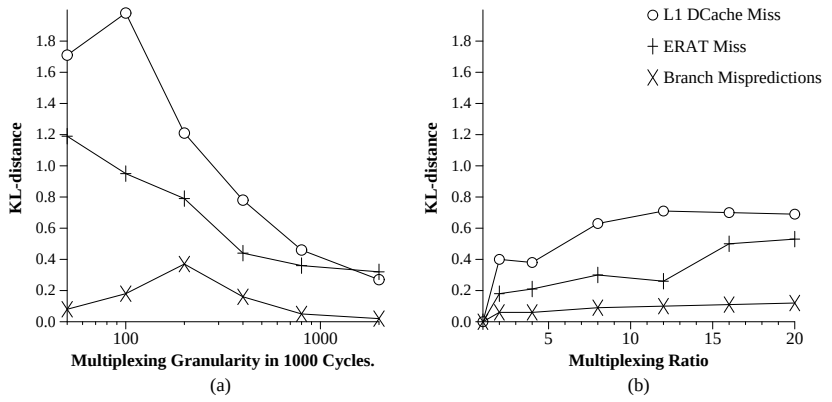


Figure 2.7: Tuning multiplexing ratio and multiplexing granularity for `gzip`: (a) The KL-distance generally decreases as the multiplexing granularity increases. (b) Fixing the granularity to 500,000 cycles, all three hardware events seem to be fairly stable when changing the multiplexing ratio within a realistic range.

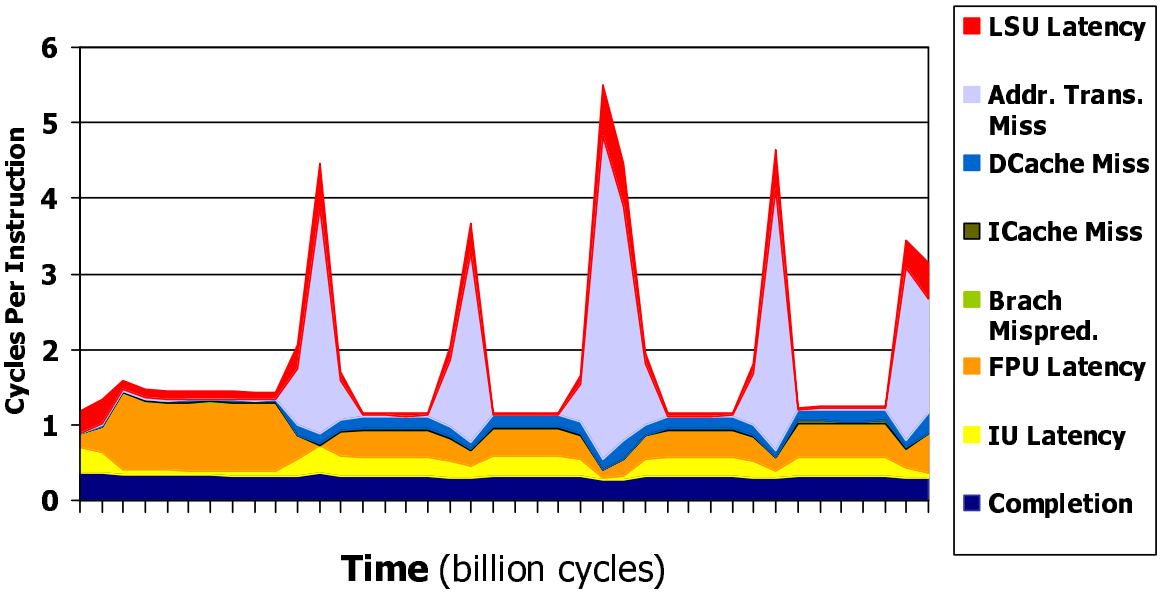


Figure 2.8: Stall breakdown for an instance of `fft` run over a period of 40 billion-cycles on IBM POWER5.

2.7.2 Stall Breakdown

In this subsection, we present an example of how stall breakdown information look like. In Figure 2.8, we show the result of computing stall breakdown for `fft` over a period of 40 billion cycles. There are several observations that can be made from the graph. First, the entire run is divided into several fairly long phases in which either CPI is stable, or CPI changes in a fairly regular fashion. In each phase, it is possible to pinpoint one or more major sources of stalls. Secondly, there is often a large gap between the real, measured CPI and the ideal CPI, most of which can be explained by the stalls. Thirdly, in this particular example, misses in the address translation data structures (i.e., ERAT and TLB) seem to be the a primary source of stalls in certain phases of the program.

The stall breakdown computed by our sampling engine can provide useful and timely hints to a runtime optimizer, allowing it to focus, in this case, on techniques to reduce data cache misses for most of the program and preventing the optimizer from focusing on optimizations that might reduce the computation, branch mispredictions, or ICache misses as they will not have significant effect unless they manage to also reduce data cache misses. Also, the online availability of the stall breakdown information allows the

Applications	stalls (in million cycles in a billion CPU cycles)								
	Local L2	Local L3	Remote L2	Remote L3	Local Memory	Remote Memory	Total Estimated	Total Measured	Error (%)
art	130	470	0	0	0	0	600	363	65
swim	132	80	0	0	255	0	468	346	35
apsi	40	315	0	0	0	0	356	331	7
mcf	43	321	0	0	0	0	365	310	17
specj	78	91	20	33	56	7	287	276	3
volano	197	6	37	0	0	0	241	235	2
vpr	37	75	0	0	0	0	113	97	15
crafty	70	32	0	0	0	0	103	82	25
twolf	49	38	0	0	0	0	88	68	29
ammp	16	106	0	0	0	0	123	53	132
gcc	48	10	0	0	0	0	60	51	17
bzip2	28	16	0	0	0	0	45	37	20
mysql	13	2	1	1	0	0	19	22	-13
gzip	16	0	0	0	0	0	16	14	18

Table 2.7: Source-based L1 data cache miss stall breakdown: stalls of each storage source is estimated by using its access frequency and its average access latency. The total stalls due to the L1 data cache miss is measured by using the IBM POWER5's PMU.

runtime optimizer to monitor the results of the applied optimizations, and measure their benefits and potential negative side effects in a feed-back loop.

Source-based Breakdown

In this subsection, we present the results of our analysis of the accuracy of the naïve approach for breaking down the L1 data cache miss stalls based on their sources which is described in Section 2.5.2. Table 2.7 shows both estimated stalls caused by various sources using their access frequency and average access latency, as well as total number of stalls that are *actually* caused by the L1 data cache misses measured by using the IBM POWER5's PMU.

As expected, in most cases (with `mysql` being the exception), the naïve approach overestimates the stalls caused by different sources, as the sum of all estimated stalls is higher than actual stalls caused by the L1 data cache misses. However, in most cases, especially in memory-bound applications, the overestimation error is not so large

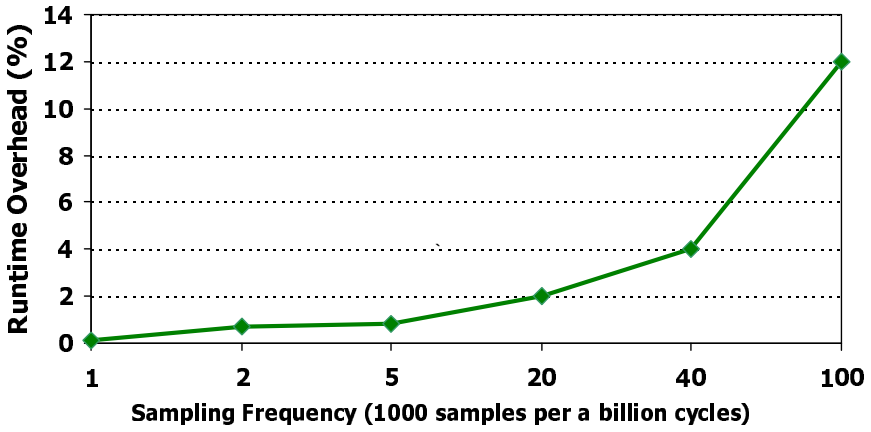


Figure 2.9: The runtime overhead of HPC multiplexing as well as computing and logging SSB (Note that the x-axis is in logarithmic scale).

that makes the source-based breakdown to be misleading. In summary, although the naïve based approach is not a perfect solution, in seems to useful in parctice for many applications.

2.7.3 Runtime Overhead

Figure 4.13 shows the runtime overhead of our performance monitoring facility for different sampling frequencies, which is defined as the number of overflow exceptions generated in a unit of time. At each overflow exception, the HPC values are collected, and depending on the logging period, added to the trace buffer. Also, the next HPC group is selected and program the PMU to count it. The runtime overhead is measured by running several benchmarks to completion and comparing the execution time with and without HPC sampling. We found that the runtime overhead increases linearly with the sampling frequency within the range we examined. Moreover, we found that the runtime overhead is fairly independent of the application that is running among the set of applications we used. In particular, at 20,000 samples per billion cycles (i.e., 20000 overflow exceptions), the runtime overhead is around 2%. We believe that with such low runtime overhead, our sampling engine is suitable for runtime optimization purposes.

2.8 Related Work

Software HPC multiplexing was previously implemented for PAPI [DLM+03], a commonly used performance monitoring library that is available on a wide range of architectures. However, in PAPI multiplexing is implemented at user level using the operating system signal mechanism [May01, MC05]. A fine-grained timer is used as a means for controlling the HPC group switch. The timer will send a signal to the process that has requested a multiplexed set of hardware events. A major limitation of this approach is that due to the large overhead of HPC group switch (the cost of signal delivery plus the cost of kernel/user context switches), the multiplexing granularity must be large, and as a result, the extrapolation error may become high for some applications. Another problem with switching HPC groups in user space is that there is potentially a large latency between the time when the timer expires and the time when the signal is actually delivered and the signal handler (where the current HPC group is read and stored) is called which adds to the multiplexing error. Finally, to the best of our knowledge, there is no quantitative study on the overhead and accuracy of PAPI's multiplexing engine. In theory, one could easily build PAPI's high-level platform-independent interface transparently on top of our low-level and efficient multiplexing scheme.

Intel's VTune [Intb] is one of the most widely used tools to make the PMU facilities available to developers. It provides both sampling and binary instrumentation facilities, and it outputs a graphical display of programs hot spots as well as call graph. There are several other tools built for various hardware platforms with similar sets of features, such as Apple's CHUD [App] and PCL [PCL]. They provide facilities to identify program hot spots and the frequency of important hardware events such as cache misses or branch mispredictions. To the best of our knowledge, none of these tools allows for profiling more events than the number of HPCs at the same time. Also, they often only expose the hardware PMU features directly to the user. It is up to the user to interpret the semantics of the low–level hardware events.

DCPI is another profiling tool that uses fine-grained sampling of the HPCs to identify system-wide hot spots at run-time [ABD+97]. It also attempts to identify pipeline stalls

at the instruction level using event-based sampling. There are some hints that HPC multiplexing is implemented in this system, but no details of the design nor statistical analysis is provided. Moreover, there is a major simplifying assumption made by the authors, namely that the distance between the instructions causing the performance counter to overflow and the actual occurrence of the overflow exception is *fixed*. This assumption is used to attribute stalls to the instructions that are causing them. However, our experience with more modern real processors with deeper and wider pipeline shows that this assumption is fairly unrealistic.

Recent work suggests a performance counter architecture for measuring the CPI components using a simplified model to quantify the negative effect of the *miss events* in the micro-architecture throughput [EEKS06, KS04]. The authors compared the accuracy of their architecture to the one implemented in IBM POWER5 using simulation. Although they improved the accuracy of CPI breakdown information mainly by taking mispredicted paths into account, their model still lacks a comprehensive analysis of potential overlaps of stalls from different causes in the processor back-end. Moreover, due to the difference in the experimental methodology (a simulation environment versus a real and complex processor) the head-to-head comparison with IBM POWER5 may not be meaningful. Nevertheless, such attempts confirm the need to implement features in the processor PMU to analyze the causes of overlapping stalls more accurately.

ProfileMe proposes *instruction sampling* to randomly monitor individual instructions as they pass through the different stages of the system pipeline [DHW$^+$], in order to gather accurate information on what are the major sources of latency. Although instruction sampling can be effective, there is little analysis in the paper that shows the actual runtime overhead of constructing an instruction-level profile. We believe our approach can be complemented by approaches such as ProfileMe to search for bottleneck in a multi-level fashion.

Wassermann et. al presented an analysis of microprocessor performance using a model similar to SSB to characterize the effect of stalls caused by cache and memory latencies [WLLB97]. Estimating the number of stalls caused by a source is done in software by multiplying the number of accesses to the source by its average access latency. Our

approach extends this effort in two directions. First, we exploit hardware support to measure the stalls more accurately. Secondly, while we include all possible sources of stall into our analysis, their approach mainly focused only on cache and memory stalls.

Slack [FBHN03a] and Interaction Costs [FBHN03b] are two models for accurately estimating how much performance gain can be achieved by idealizing latencies of individual instructions. Although these approaches provide accurate information on the potential gain of idealizing individual instructions, they require additional hardware support and extensive postmortem analysis, which make them difficult to use in the context of run-time optimization.

FlashPoint [MOH96] and Lemieux [Lem96] both attempt to integrate monitoring the activities the memory interconnect in a shared memory multiprocessor with the existing cache-coherence hardware. The basic idea is that the cache-coherence hardware automatically activates a software *trigger* on cache coherence activities that are incurred as a result of L2 cache misses. The trigger is able to obtain much information about cache misses including their latency and then builds summary performance information such as histograms. FlashPoint is implemented in the FLASH multiprocessor, and Lemieux approach is implemented in NUMAchine multiprocessor, both presumably with an acceptable runtime overhead (e.g., around 10%). While the features suggested by these approaches are very useful, the ramification of implementing them in today's much faster and more complex microprocessors are not known. Moreover, the focus of both approaches is primarily on off-chip memory traffic. In principle, one can extend these approaches to the case of on-chip communication through shared cache.

2.9 Concluding Remarks

Hardware performance counters (HPCs) are useful for analyzing and understanding the performance of a processor executing code, but there are challenges in using them on line. Too few HPCs are available in most today's microprocessors, and, the definitions of the hardware events that can be counted by HPCs are low-level and complex.

In this chapter, we described two techniques that overcome the limitations of existing

microprocessor HPCs. First, we provide a larger set of *logical* HPCs by dynamically multiplexing physical HPCs using statistical sampling of hardware events. Using real programs, we showed experimentally that counts of hardware events obtained through sampling is statistically similar (i.e. within 15%) to the actual event counts. Secondly, we proposed a simple performance model based on CPI breakdown that focuses on *stall* cycles, which are defined as cycles in which no instructions completes. We show that completion stalls are particularly important, as they contribute to over 73% of all CPU cycles across the SPEC2000 benchmarks. Moreover, removing the completion stalls will result in a CPU throughput which is fairly application independent and is close to the maximum CPU throughput determined by the pipeline width. We exploit IBM PowerPC970 and POWER5 features to speculatively associate each completion stall cycle to the processor component that likely caused the stall. The entire stall breakdown model is computed online by using our HPC multiplexing engine with a run-time overhead of under 2%.

The facility we have implemented is useful for detailed on-line performance analysis of application and system code running at full speed with small overhead. It is also effective in reporting hardware bottlenecks to tools such as a dynamic optimizer that might guide dynamic adaptation actions in a running system. A number of outside groups have started using our sampling-based tool. For example, our tool has been successfully used by several research groups within IBM research over the last few years for the purpose of detailed bottleneck analysis and guiding performance optimization. Their experience indicates that the stall breakdown facility is a powerful model, which is easy to use and understand and that is reasonably accurate even for fairly complex applications.

When we started working on exploiting processor HPCs, we were surprised how difficult it was to use the counters. Their exact semantics are generally not defined in a public way, and we had to spend considerable effort reverse engineering their real meaning. Moreover, we were surprised how different the HPCs were from processor to processor, even within the same processor family.

Throughout this work, we found that to correctly interpret the values the HPCs, one must understand the details of the target processor microarchitecture, something that is

often proprietary and something most software engineers would find difficult. The HPCs, with the events they can count, were clearly designed more for processor architects than for software architects. In many ways, our tool helps map detailed micro-architectural events to higher-level information, understandable by a larger audience.

Looking into the future, we would hope that processor designers increase the number of HPCs available and that an increased number of higher-level events, more useful to software optimizers, be countable. With more countable higher-level events, it becomes possible to standardize HPCs and their interfaces across different processors so that it becomes easier to port tools such as ours to different architectures. Long term, we envision an HPC standard emerging that software can rely on and implemented on all processors (similar to the way the floating point standard is implemented today).

Chapter 3

Hardware Data Sampling to Detect Thread Sharing

3.1 Introduction

Hardware data sampling is a mechanism in the micro-architecture to collect data addresses that are manipulated by programs either periodically or upon occurrence of certain hardware events. Data sampling has been used effectively by a number of researchers for a variety of purposes. For example, it has been used to track access patterns of individual cache lines in order to be able to insert software prefetching hints [LCF+03]. Other uses include algorithms to automatically detect cache working set sizes [BH05], isolation of latency-causing memory regions [BH], or to enhance NUMA page placement [THb] algorithms. Finally, there have been attempts to use data sampling to verify program correctness or enforce security [ZLF+04].

Hardware support for data sampling is present in many modern micro-architectures such as IBM POWER5 [SKT+], Intel Itanium [Inta], Sun's UltraSparc [NZ], and AMD Barcelona [CI06] processors. In most architectures, a special *Data Address Register (DAR)* is dedicated for sampling data addresses. The content of the DAR is automatically updated by the hardware PMU with the operand of a memory instructions (load or store). The PMU can usually be programmed to update the DAR only when certain *selection criteria* are satisfied. Examples of such selection criteria include when a data

cache miss or a TLB miss has occurred during the execution of a memory instruction. Most existing architectures provide only one DAR. Hence, at any point in time, only one selection criterion can be used to filter the data samples by the hardware PMU.

In most cases, however, the underlying hardware support for data sampling is not adequate. This has forced researchers to either roughly approximate the information they need from hardware, or to propose new hardware support specifically for their purpose. Examples of limitations of hardware support for data sampling that we have encountered in our own studies are the following.

Coarse Selection Criteria: In most architectures, the selection criteria supported by the hardware PMU are often not sufficiently specific. As a result, many of the data samples that are collected by the hardware are not relevant to a particular optimization technique. Resolving this issue often requires potentially expensive software filtering techniques. An example of such filter mechanism is presented in Section 3.2.1, where we use a combined hardware-software approach to capture cache misses that are fetched from a specific storage source.

Inflexible Interface: In most cases, the hardware interface for specifying the selection criteria is too inflexible. For instance, only one selection criterion can be specified at a time. Support for combining multiple selection criteria in either conjunctive or disjunctive forms is not provided. We show in Section 3.2.2 how we use the HPC multiplexing facility described in Section 2.4 to implement having multiple selection criteria in a disjunctive form.

Also, due to the inflexible hardware interface, data samples are delivered to software in raw form which is often too voluminous to be stored and processed in their raw form. One has to build build efficient *summary* data structures at the software level to be able to overcome space requirements. A widely used example of such data structures is the histogram. In a case study, we will show how we use a variation of histograms to build sharing signatures for concurrently running threads.

3.1.1 Organization of Chapter

In this chapter, we first present a brief overview of the major methods of hardware data sampling and describe the advantages and shortcomings of each method. We then present our specific mechanisms (i) to sample data according to the source it is fetched from, and (ii) to apply multiple selection criteria simultaneously. We also provide a detailed explanation of how we use these mechanisms to detect sharing patterns among concurrently running threads, and how one can use such sharing patterns to cluster threads in a chip multiprocessor (CMP) architecture to avoid expensive cross-chip data exchange. At the end of the chapter, we discuss some of the problems with current hardware support for data sampling and provide concrete proposals to solve some of these problems.

3.1.2 Data Sampling Methods

In this section we provide a brief description of different methods of hardware data sampling. We also discuss the major advantages and shortcomings of each approach.

Continuous Data Sampling

With continuous data sampling, the DAR is *continuously* updated by the hardware PMU as memory instructions with operands that match the selection criteria arrive in the pipeline. With continuous data sampling, the DAR is constantly overwritten as new instructions are issued. System software can take samples of DAR values by occasionally reading its value, which will refer to the last operand address that has matched the selection criteria.

The main advantage of this approach is that *all* address operands of memory instructions have a fairly equal chance of being captured by system software. That is, it is possible for system software, at least in principle, to record *all* address operands that match the selection criteria (e.g., a cache miss). This is an important property in certain optimization schemes, where it is important to see all data addresses that cause a certain event such as cache miss or TLB miss. Moreover, system software is able to use hardware

performance counters corresponding to the selection criteria to capture exactly one in N address operands that match the selection criteria.

The major limitation with continuous data sampling is that, due to the deep processor pipeline, there is a potentially large distance in the dynamic instruction stream between the memory instruction that has caused the DAR to be updated, and the current program counter (PC). As a result, it is difficult to directly attribute the recorded DAR to a particular instruction. In principle, one can perceive a hardware mechanism to track back each instruction in the pipeline to an instruction address. But to the best of our knowledge, such a mechanism does not exist in any of the todays' processors.

A second issue with continuous data sampling is that it is inherently speculative in the sense that the DAR is updated regardless of whether the issued instruction that caused the DAR to update actually completed or flushed due to branch misprediction. As a result, any analysis of the sampled data addresses must take the noise generated by the mispredicted paths into account.

Instruction Sampling

With *instruction sampling*, an instruction is *tagged* to be monitored by the hardware PMU as it passes through the different stages in the processor pipeline [DHW+, IBM06, Inta, CI06]. The address of the tagged instruction is stored into a dedicated Instruction Address Register (IAR) and the DAR is also updated only when the address operand of the tagged instruction is calculated.

The main advantage of instruction sampling for the purpose of data sampling is that the sampled addresses can be precisely tracked back to the instructions that have accessed them. This has the potential for more complete analysis, as it is possible to characterize computation bottlenecks both in terms of the executing code and the data that is manipulated by the code at the same time. This is significant as identifying the code that is consuming most of the execution time alone may not be sufficient as a single segment of code (e.g., a function) can access many different sets of data addresses (e.g., depending on the input parameters). Similarly, data sampling alone may not be sufficient either, as the data that causes long latency may be accessed by many instructions through

different code paths in the program.

The major limitation of instruction sampling is its low *recall*: of the many instructions that flow through the pipeline, only very few instructions (usually only one) can be sampled. As a result, many relevant data accesses will pass by unnoticed. This problem is aggravated when sampling is conditional to some selection criteria (e.g., cache misses), since instruction tagging occurs independently of the selection criteria in most processors. This is because instruction tagging is usually done at an early stage of the processor pipeline (e.g., the fectch or decode stage) which is too early to evaluate any backend-level selection criteria (e.g., a cache miss). In such cases, a large number instructions that satisfy the selection criteria will not be tagged, and a large number instructions that are tagged do not satisfy the selection criteria.

Hardware Data Breakpoints

With continuous sampling and instruction sampling, it is difficult to *watch* every access to a particular memory address. Such watching mechanisms have been used, for instance, to measure the program cache working set size by measuring the *reuse distance* of a sampled set of cache lines [BH05] or to identify potential bugs or attacks [ZLF+04]. While there are mechanisms to monitor accesses to individual pages at the operating system kernel, (e.g., by reseting and checking page table bits), watching accesses at the granularity of a single cache line is not directly possible without additional hardware support.

An alternative method to monitor specific data addresses is to use the *data breakpoint mechansim* such as the one implemented in the AMD64 architecture [AMD], which is originally designed for debugging purposes. If a cache line-sized data item is selected to be a watch point, every subsequent access to the data item will raise an exception to the operating system. The operating system exception handler can examine the current program context and the watched address.

The breakpoint mechanism is potentially costly to use, since every access to a selected memory item will cause an exception. For instance, Berg *et al.* show that an analysis of cache working set size by using the breakpoint mechanism could result in an average overhead of around 40% [BH04].

Hardware Bus Monitors

Another approach to sample data accessed is to monitor the memory bus transactions and sample the addresses that appear on the bus, rather than sampling the addresses on each processor individually. Real systems such as Sun Microsystem's Fire Link have taken this approach [NZ].

While monitoring the memory bus has the advantage of having the global order of recorded data samples across the entire system, it has several drawbacks. First, addresses that appear on the bus are already filtered by a potentially large on-chip cache. As a result, much of the application data access pattern is not visible to any bus-based analyzer. This problem is particularly aggravated in todays hierarchical multiprocessing architectures (i.e., SMP-CMP-SMT). Second, bus-based data sampling requires special hardware support, which makes it hard to be used for off-the-shelf commodity processors. Finally, only physical addresses appear on the memory bus, and as a result, a software analyzer must map the physical addresses back to their corresponding virtual addresses in an online fashion.

3.1.3 Data Sampling Modes

The DAR can be read usually by both kernel and user-level software. Software may decide to read and record the DAR periodically, i.e., *time-based sampling* or upon a certain number of instances of a particular event, i.e., *event-based sampling*. Time-based sampling is simple and generic, and it captures the frequency distribution of accesses to *all* data items uniformly. On the other hand, event-based sampling is more targeted towards monitoring and sampling addresses that are involved in specific hardware events (e.g., cache misses, TLB misses, or cache invalidations in multiprocessor systems). As a result, event-based sampling may be better suited for certain optimizations, as it automatically ignores all unrelated data accesses.

3.2 Our Sampling Techniques

In this section, we describe two novel data sampling techniques we developed and used for the purpose of detecting data sharing patterns among concurrent threads. These two techniques are fairly generic and can be potentially used for other purposes than the one we used in our work. The first technique is to sample data based on the source it is fetched from, and the second technique is to combine multiple sampling criteria in a disjunctive form.

3.2.1 Source-based Data Sampling

It is often useful to be able to determine the storage *source* from which a sampled data address item is fetched. The storage sources include L1 cache, local or remote L2 caches (in SMP systems), local or remote L3 caches, and local or remote DRAM memory modules (in NUMA systems). For instance, our thread sharing detection scheme is based on the ability to sample data items that are consistently fetched from L2 or L3 caches of remote processor chips. Having source information is, in this case, helpful to be able to conclude that there is consistent data sharing among threads that are running on multiple separate processor chips. Another use of the source information is in a NUMA page placement scheme that dynamically monitors the accesses of threads to both local and remote memory modules and determines the optimal location of a given data page and potentially migrates the pages accordingly.

However, to the best of our knowledge, sampling data according to their sources is not directly available in the PMU features of any of today's microprocessors. As a workaround, we have exploited IBM POWER5's PMU features to conduct source-based sampling *indirectly*. IBM POWER5 supports the continuous data sampling method. The selection criteria, however, is fixed to be only either an L1 data cache miss or a TLB miss or both. As a result, in continuous data sampling, the DAR could hold the address of the last L1 data cache miss. But it is not possible to directly determine the source from which an L1 data cache miss will be eventually fetched. On the other hand, IBM POWER5 PMU can count L1 data cache misses broken down by the sources from which

the cache miss is satisfied. Therefore, it is possible to set the PMU overflow exception to be raised when a threshold on the number of L1 data cache misses from a certain source is reached. Once an overflow exception is raised, the *last* data cache miss is likely to be the data cache miss that caused an overflow exception. Therefore, by reading the DAR only when the cache miss counter of a specific source overflows, we ensure that most of the data samples read are actually fetched from the particular source.

3.2.2 Multiple Sampling Criteria

Most modern CPUs support only one DAR. Hence, at any point in time, only one data sampling criterion can be specified. This is a limitation, as one may need to simultaneously monitor data addresses with different criteria, each with a certain distribution. For instance, in analyzing a data access pattern with the goal of improving page placement in a NUMA architecture, one may need to sample data addresses that are fetched both from local memory modules and from remote memory modules at the same time. A straightforward solution would be to have multiple DARs that can be independently programmed for different criteria. We are not aware of potential challenges in the hardware implementation of having multiple DARs. However, it does not seem that the hardware designers are willing to add more data sampling resources unless researchers show how such resources can be utilized effectively.

Our, rather temporary, solution for this problem is to integrate data sampling with the HPC multiplexing introduced in Section 2.4. With this approach, we sample data for each specified criterion during a time slice of g cycles in a multiplexing round of R cycles. When the time slice is over, another data sampling criterion is specified as the new HPC group is programmed. Specifying a data sampling criterion is typically lightweight, as it requires manipulating the same control registers that are used for switching from one HPC group to the other. Several data samples can be recorded within a time slice g as the HPC of the corresponding event overflows.

The length of a time slice (e.g., g cycles) should be long enough so that the sampling event counter overflows at least once. Otherwise, no samples will be recorded, as the HPCs are reset every time they are scheduled. To cope with this problem, one may

reduce the threshold on which the sampling event is supposed to overflow. However, in the worst case, the sampling event may not occur even once during the time slice. In this case, a possible solution is to treat the sampling event counters differently by saving their value at the end of a time slice and restoring them when they are scheduled in again (as opposed to reseting them). In practice, however, the sampling events of interest are often frequent enough to be captured in a single time slice multiple times. This is because if an event is directly or indirectly causing a performance bottleneck, it must be fairly frequent.

3.3 Detecting Data Sharing

In this section we provide the details of a specific case study where we use the data sampling techniques described in Section 3.2 to improve performance by adding sharing-awareness to the operating system CPU scheduler in a Chip Multiprocessor (CMP) environment. First we describe the motivation behind the work. In Section 3.3.2 we describe the details of our technique in building sharing *signuatures* for running threads. Finally, in Section 3.3.3 we discuss our approach in clustering threads that share data together.

3.3.1 Motivation

As limits in microprocessor technology have slowed improvements in clock frequency, and micro-architecture complexity has limited more radical exploitation of Instruction Level Parallelism (ILP), major microprocessor manufacturers have turned towards providing Thread-Level Parallelism (TLP) as a means to speed up applications. Both CMP and simultaneous multithreading (SMT) technologies were introduced over the last several years even for small-scale computer systems such as laptops, and desktop computers, as well as for large-scale servers. As a result, shared memory multiprocessors have become increasingly prevalent. This trend seems to continue as CPU chips are equipped with an increasing number of processing cores.

A key difference between traditional shared memory multiprocessors (SMPs) and more modern multi-core systems is that the latter have non-uniform data sharing overheads;

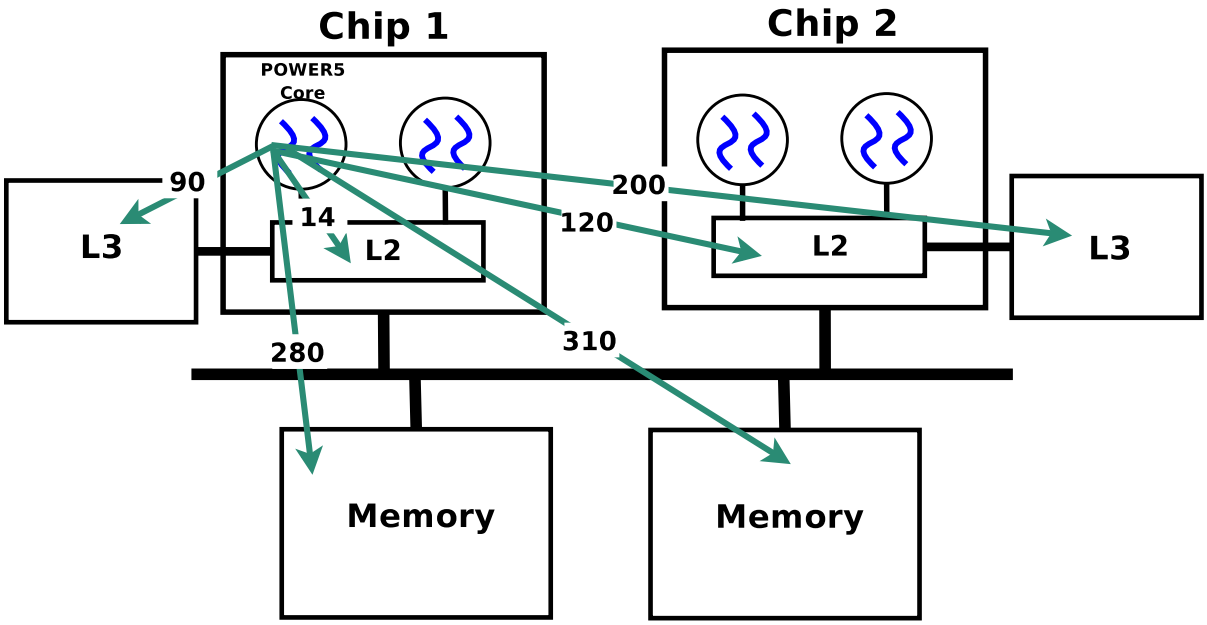
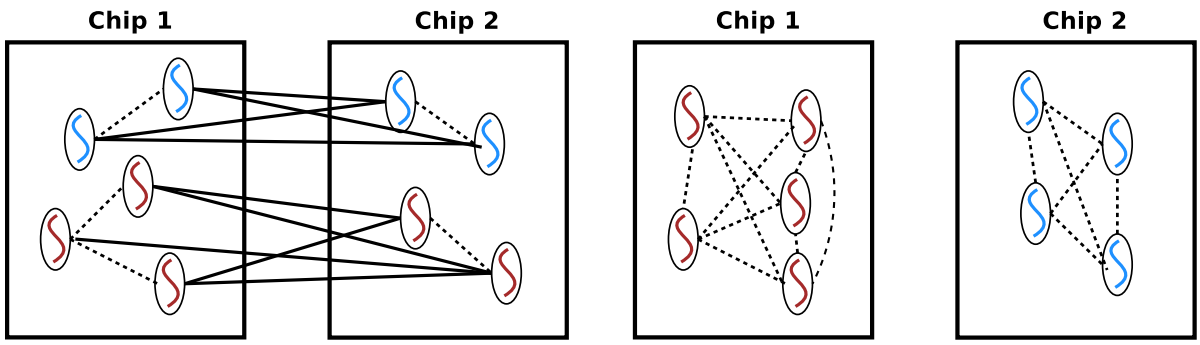


Figure 3.1: The architecture of IBM OpenPower720. The numbers on arrows show the latency of access from a thread to different levels of memory hierarchy. Any cross-chip sharing takes at least 120 CPU cycles.

i.e., the overhead of data sharing between two processing components differs depending on their physical location. For processing units that reside on the same CPU core (i.e., hardware virtual contexts), communication typically occurs through a shared L1 cache, with latency of 1-2 cycles. For processing units that do not reside on the same CPU core but reside on the same chip, communication typically occurs through a shared L2 cache, with latency of 10 to 20 cycles. Processing units that reside on separate chips communicate either through sharing memory or through a cache-coherence protocol, both with an average latency of hundreds of CPU cycles. As a specific example, consider the IBM OpenPower720's latencies depicted in Figure 3.1.

Most research done on cache-aware CPU scheduling has focused on maximizing and exploiting *cache affinity*, both in uniprocessor and multiprocessor systems [TTG95]. However, to the best of our knowledge, current CPU schedulers do not take non-uniform data sharing overheads into account. As a result, threads that actively share data will not necessarily be co-located onto the same chip. Figure 3.2 shows an example of a scenario where two clusters of threads are distributed across the processing units of two chips.



a. default b. clustered

Figure 3.2: Default versus clustered scheduling. The solid lines represent high-latency cross-chip communications, the dashed lines are low-latency intra-chip communications (when sharing occurs within the L1 and L2 caches).

The distribution of threads to processors is usually done as a result of some dynamic load-balancing scheme with no regard for thread sharing. Consequently, when threads within a cluster share data frequently, a typical scheduling algorithm (as shown on the left) is likely to cause threads being assigned to cores of different chips, so that there will be a high degree of high-latency, inter-chip communication (shown with the solid lines). However, if the operating system is able to detect intra-cluster thread sharing patterns and schedule the threads accordingly, then threads that share data heavily could be scheduled to run on the same chip and, as a result, most of the communication (dashed lines) will take place in the form of L1 or L2 cache sharing.

However, automatically detecting sharing patterns among concurrently executing threads is non trivial. A basic approach to this problem is to to use page-level protection and access information provided by hardware in the page tables to track the data each thread is accessing. This approach has been used in the past to implement, for instance, software distributed shared memory (DSM) [ACD+96]. There are two major drawbacks with this approach. First, using page granularity as the unit of sharing is too coarse in many cases resulting in a high degree of falsely detected sharing. Secondly, the information on whether a page is accessed or not is available either through frequently scanning and reseting page table entries, or by protecting pages from access and recording a page access upon a subsequent page fault. Both options are potentially costly, both in

terms of their direct overhead and also in terms of their indirect negative impact on performance through cache pollution and TLB flushing that come as a result of page-table traversal and manipulation.

In the next section, we show how to use hardware data sampling to effectively detect sharing patterns among threads. A major advantage of our approach is that it is able to accurately track data sharing down to a single cache line (which is the unit of hardware cache coherence). Moreover, as we show in Section 3.4, it is possible to achieve low runtime overhead by having a light-weight layer of software that processes the data samples generated.

3.3.2 Detecting Sharing Patterns

Using our source-based data sampling mechanism, we sample data accesses that (i) incur miss in the L1 data cache and (ii) are eventually fetched from caches on a remote processor chip (remote L2 or L3). We then use these samples to construct a summary data structure for each thread, called *shMap*. Finally, compare the threads' *shMaps* with each other to identify the threads that are actively sharing data and cluster them accordingly. Next, we present the details on how we build *shMap*s and use them for thread clustering.

Constructing *shMap*s

Each *shMap* is a small vector (e.g., 256 entries) of 8 bit-wide saturating counters. We partition the application address space into fixed sized *blocks*. Each block is mapped to a counter in the *shMap* vector using a hash function. An *shMap* entry is incremented only when the corresponding thread incurs a remote cache miss on the block. Note that threads that share data but happen to be located on the same chip do not cause their *shMap*s to be updated as they do not incur any remote cache miss.

The block size is an important parameter. The advantage of a large block size is that the total size of the *shMap*'s span over an application's address space increases. However, large block sizes may make the access tracking less precise, which may result in falsely detecting and reporting sharing where in fact, the accesses are in different spots within

the large block. In this study, we set the block size to be equal to the size of an L2 cache line (e.g., 128 bytes), which is the unit of hardware-level data sharing in most hardware cache-coherence protocols. False sharing within a single cache line can still happen, but hardware cache-coherence will not be able to distinguish it from true sharing either, and as a result, it will still incur cache line invalidations and cross-chip communications.

Constructing *shMap*s involves two challenges. First, to record and process every single remote cache miss is prohibitively expensive, especially for applications in which there is a large volume of read/write sharing among threads and subsequently, frequent remote cache misses. Secondly, with a relatively small *shMap*, there will be a lot of collisions in hashing virtual addresses of remote cache misses onto *shMap* entries, as applications virtual address space are much larger than the *shMap* span (e.g., 64Kbytes).

We use two different techniques to deal with the two challenges. To cope with the high volume of data, we use *temporal sampling*, and to reduce the collision rate (actually to eliminate collision altogether) we use *spatial sampling*. Using temporal and spatial sampling of remote cache misses combined instead of capturing them precisely is sufficient for the purpose of detecting sharing among threads, because we are not interested in knowing the absolute volume of sharing and all the addresses that are shared, but rather only need an indication of whether two threads are sharing data or not. Statistical sampling schemes ensures that if a data item is highly shared (i.e., remote cache misses on it occur highly frequently), it will be recorded with high probability.

We now describe the two techniques, temporal sampling and spatial sampling in more detail.

Temporal Sampling: We record and process one in N occurrences of remote cache miss events. In order to avoid undesired coincidental repetitions, we constantly readjust N by a small random value. Moreover, the value of N is further adjusted by the current frequency of remote cache misses which can also be measured by the HPCs. A high rate of remote cache misses allows for larger values for N so as to reduce the runtime overhead and at the same time be able to obtain obtain a representative sample of addresses.

Spatial Sampling: Rather than monitoring the entire virtual address space, we select a fairly small set of *sample* blocks to be monitored for remote cache misses. There

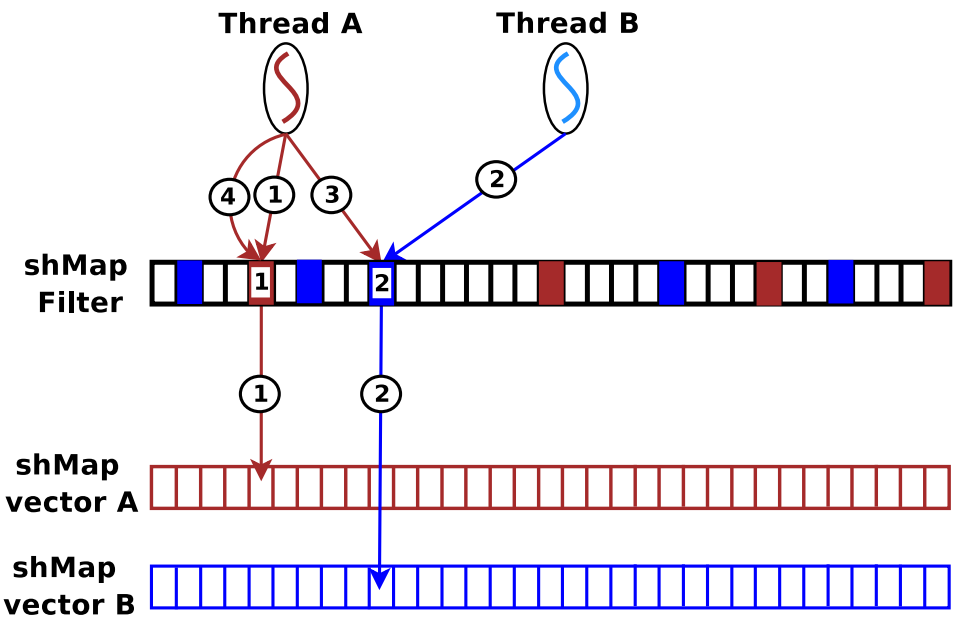


Figure 3.3: Constructing *shMaps*: each remote cache miss by a thread will be hashed to an entry in both the *shMap* filter and the thread's *shMap*. A remote cache miss will be recorded (i.e., *shMap* entry is incremented), *only if* either the *shMap* filter entry is not already allocated, or is previously allocated for the same virtual address. Circled numbers represent the order of access. Remote cache misses "1" and "2" are recorded because their entries in the *shMap* filter are free. Remote cache misses "3" and "4" are discarded because their *shMap* filter entries are already reserved for differnt virtual addresses.

has to be at least one remote cache miss on a block to make it eligible to be selected. The spatial sampling scheme then selects the sample blocks somewhat randomly among the eligible blocks. The justification for spatial sampling is if there is high level of sharing among threads, there will be some hot sharing spots that will likely be captured by the spatial sampling scheme. Also, having several hot spots is a clear indication of high level of sharing among threads.

We implement spatial sampling by using a filter to select remote cache miss addresses. This *shMap Filter* is essentially a vector of addresses with the same number of entries as an *shMap*. All threads of a process use the same *shMap* filter. Figure 3.3 shows the function of *shMap* filter. A sampled remote cache miss address is allowed to pass through the *shMap* filter only if its corresponding entry in the *shMap* filter has the same address

value. Otherwise, the remote cache miss is discarded and not used in the analysis. Each *shMap* filter entry is initialized (in an immutable fashion) by the first remote cache miss that is mapped to the entry. Threads compete for the entries on *shMap* filter.

In an unlikely pathological case, it is possible that some threads starve out others by grabbing the majority of the *shMap* filter entries, thus preventing the remote cache misses of the other threads to be processed. This does not cause a problem with our scheme, as we envision the thread clustering process to be iterative. That is, after detecting sharing among some threads and clustering them, if there is still a high rate of remote cache misses, thread clustering is activated again, and the previously starved threads will obtain another chance of capturing entries on the *shMap* filter.

3.3.3 Clustering Threads

In this subsection, we describe our approach for clustering *shMap* vectors into groups of threads that actively share data. We first describe the similarity metric we use in our clustering scheme. Then we describe the actual clustering algorithm we implemented to form the thread clusters.

shMap Similarity Metric

We define the similarity of two threads' *shMap* vectors as their dot products:

$$similarity(T_1, T_2) = \sum_{i=0}^{N} T_1[i] * T_2[i]$$

The rationale behind choosing this metric for similarity is two fold. First, it automatically takes into account only those entries where both vectors have non-zero values. Note that T_1 and T_2 have non-zero values in the same location only if they have had remote cache misses on the same cache line (i.e., the cache line is being shared actively). We consider very small values (e.g., less than 3) to be zero as they may be incidental or due to cold sharing and may not reflect a real sharing pattern.

Secondly, the metric takes into account the intensity of sharing by multiplying the number of remote misses each of the participating threads incurred on the target cache line. That is, if two vectors have a large number of remote misses on a small number of

cache lines, the similarity value will be large, correctly identifying that the two threads are actively sharing data. Other similarity metrics could be used, but we found this metric to work quite well for the purpose of thread clustering (See Section 3.4).

Forming Clusters

One way to cluster threads based on *shMap* vectors is to use standard machine learning algorithms, such as hierarchical clustering or K-means [JMF99]. Unfortunately, such algorithms are computationally too expensive to be used online in systems with potentially hundreds or thousands of active threads, or they require the maximum number of clusters to be known in advance (for K-means, for instance), which is not a realistic assumption to make in our environment.

To avoid high overhead, we use a simple heuristic for clustering threads based on two assumptions that are simplifying but fairly realistic. First, we assume data is naturally partitioned according to application logic, and threads that work on two separate partitions do not share much except for data that is globally shared (i.e., process-wide) among all threads. In order to remove the effects of globally shared data on clustering, we build a histogram for *shMap* vectors in which each entry shows how many *shMap* vectors have a non-zero value for the entry. We consider a cache line to be globally shared if more than half of the total number of threads have incurred a remote miss on it. We ignore information on globally shared cache line when composing clusters.

The second assumption is that if a subset of threads share data, the sharing is reasonably symmetric. That is, we assume it is likely that *all* of them incur remote misses on similar cache lines, no matter how they are partitioned.

Using the two above assumptions, we define a simple clustering algorithm as follows. Based on the first assumption, if the similarity between *shMap* vectors is greater than a certain threshold, we consider them to belong to the same cluster. Also, according to the second assumption, any *shMap* vector can be considered as a cluster representative since all elements of a cluster share common data equally strongly.

The clustering algorithm, shown in Algorithm 1, scans through all threads in one pass and compares the similarity of each thread with the representatives of previously known

```
1:  NumKnownClusters ⇐ 0 {one pass clustering algorithm}
2:  for t = 0 to NumThreads do
3:     FoundACluster ⇐ false
4:     for c = 0 to NumKnownClusters do
5:        repShMap ⇐ Clusters[c]
6:        if Similarity(shMap_t, repShMap) > SHARING_THRESHOLD then
7:           add t to the cluster c
8:           FoundACluster ⇐ true
9:           break
10:       end if
11:    end for
       {create a new cluster if t is not similar to any of the previously known clusters}
12:    if FoundACluster is false then
13:       Clusters[NumKnownClusters] ⇐ shMap_t
14:       NumKnownClusters++
15:    end if
16: end for
```

Algorithm 1: Clustering $shMap$ vectors for N threads.

clusters. If a thread t is similar to the representative of cluster c (i.e., the similarity metric between the two $shMap$ vectors exceeds a certain threshold), thread t is added to the cluster c. If no such a cluster is found (i.e., $shmap_t$ is not similar to any of the representatives of the previously known clusters), a new cluster is created, and t is assigned to be the representative of the newly formed cluster. The set of known clusters is empty at the beginning.

The computational complexity of this algorithm is $O(T * c)$ where T is the number of threads that are suffering from remote cache misses, and c is the total number of clusters which is usually much smaller than T.

CPU Cores	IBM POWER5, 1.5GHz, SMT
L1 DCache	64KB, 4-way associative
L1 ICache	64KB, 4-way associative
L2	2MB, 10-way associative, shared by the two cores on a chip
L3	36MB, 12-way associative, off-chip, a victim cache for L2
Local Memory	4GB
Remote Memory	4GB
No. of CPU Chips	2

Table 3.1: The Specification of the IBM OpenPower Machine.

3.4 Experimental Evaluation

In this section, we present the results of our experiments to evaluate our hardware data sampling techniques. The basic focus of our evaluation is to exhibit the effectiveness of our techniques for hardware data sampling by showing their uses in a real use case (e.g., thread clustering).

First, we present the details of our experimental platform and the workload we used. Secondly, we demonstrate the runtime overhead of hardware sampling of remote cache misses. Then, we show the accuracy of our sharing detection and thread clustering techniques under the selected real workloads. Finally, we briefly present the performance results of a sharing-aware thread scheduler that uses our sharing detection and thread clustering approach.

3.4.1 Experimantal Platform

The multiprocessor used in our experiments is an IBM OpenPower720 Express computer system. It is an 8-way POWER5 consisting of a 2x2x2 SMPxCMPxSMT configuration, as shown in Figure 3.1. Table 3.1 shows the specification of the hardware components in the system.

While our evaluation platform is sufficiently complete to show the effectiveness and overhead of our basic techniques and mechanisms, in order to fully realize the potentials

and limitations of the thread clustering approach, we will have to evaluate it on machines with a larger number of processors, which is beyond the scope of this thesis.

We used Linux 2.6.15 as the operating system. We modified Linux to add the features we needed for hardware performance monitoring, including the stall breakdown (See Section 2.5) and sampling of remote cache miss addresses. We also modified the Linux CPU scheduler to allow for explicit relocation of threads at runtime, guided by the thread clustering information provided by our approach.

3.4.2 Workloads

For our experiments, we used a synthetic microbenchmark and three commercial server workloads: `VolanoMark` which is a benchmark for Internet chat servers [Vol], `SPEC JBB2000`, which is a Java-based application server workload [Sta], and `RUBiS`, which is an OLTP database workload. For `VolanoMark` and `SPEC JBB2000`, we used IBM J2SE 5.0 as our Java virtual machine. For `RUBiS` [RUB], we used MySQL 5.0.22 as our database server [MyS]. These server workloads are written in a multithreaded, client-server programming style, where there is a thread to handle each client connection for the life time of the connection. We present details of each benchmark below.

Synthetic Microbenchmark: The synthetic microbenchmark is a simple multithreaded program in which each worker thread reads and modifies a scoreboard. Each scoreboard is shared by several threads, and there are several scoreboards. All scoreboards are accessed by a fixed number of threads. Each thread has a private chunk of data to work on which is fairly large so that accessing it often causes data cache misses. This is to verify that our technique is able to distinguish remote cache misses that are being caused by accessing the scoreboards from local cache misses that are caused by accessing private data. The clustering algorithm should be able to cluster threads that share a scoreboard.

VolanoMark: VolanoMark is an instant messaging chat server workload. It consists of a Java-based chat server and a Java-based client driver. The number of rooms, number connections per room, and client think times are configurable parameters. The server is written using the traditional, multithreaded, client-server programming model, where

each connection is handled by a designated thread for the life-time of the connection. In actuality, Volanomark uses two designated threads per connection. Given the nature of the computational task, threads belonging to the same room should experience more intense data sharing than threads belonging to different rooms. In our experiments, we used four rooms with 8 clients per room as our test case.

SPEC JBB2000 SPEC JBB2000 is a self-contained Java-based benchmark that consists of multiple threads accessing designated *warehouses*. Each warehouse is approximately 25 MB in size and stored internally as a B-tree variant. Each thread accesses a fixed warehouse for the life-time of the experiment. Given the nature of the computational task, threads belonging to the same warehouse should experience more intense data sharing than threads belonging to different warehouses. In our experiments, we modified the default configuration of SPEC JBB2000 so that multiple threads can access a common warehouse.

RUBiS RUBiS is an online transaction processing (OLTP) server workload that represents an online auction site workload in a multi-tiered environment. The client driver is a Java-based web client that accesses an online auction web server. The front-end web server uses PHP to connect to a back-end database. We focus on the performance of the database server. We made a minor modification to the PHP client module so that it uses persistent connections to the data base, allowing for multiple SQL requests to be made within a connection. While this modification improves performance by reducing the rate of TCP/IP connection (and thread) creation on the database server, it also enables our algorithm to monitor the sharing pattern of individual threads in the long term.

In our workload configuration, we used two separate *database instances* within a single MySQL process. This configuration may represent, for instance, two separate auction sites run by a single large media company. We expect that threads that belong to the same database instance to experience more intense sharing with each other than with other threads in the MySQL process.

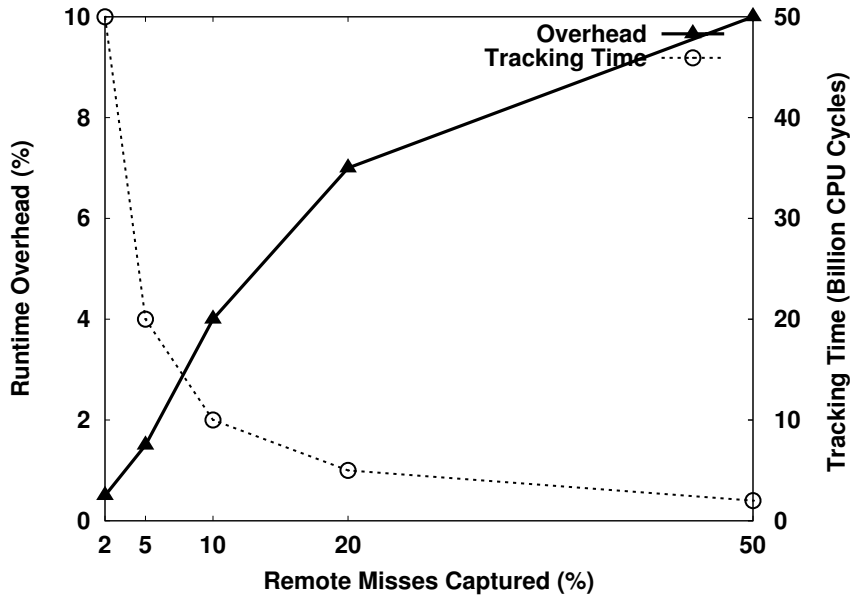


Figure 3.4: Runtime overhead of the sharing detection phase for SPEC JBB2000 as a function of the temporal sampling rate, and the time (in billion CPU cycles) that is required to collect a million remote miss samples given the temporal sampling rate.

3.4.3 Runtime Sampling Overhead

Figure 3.4 shows the runtime overhead of hardware data sampling as a function of the rate we used for temporal sampling in terms of the percentage of the remote misses that are actually examined for SPEC JBB2000. As a higher percentage of the remote cache misses are captured, the overhead naturally increases. However, the time to collect a sufficient number of remote cache miss samples becomes shorter. In our experiments, we have found we need roughly a million samples to accurately detect sharing patterns. Therefore, the right Y-axis of Figure 3.4 represents how long (in billion CPU cycles) we need to stay in the detection phase to collect a million samples of remote cache misses. The higher the sampling rate, the higher is the run-time overhead, but the shorter the sharing detection phase will last.

According to Figure 3.4, it appears that a temporal sampling rate of 10 (capturing one in every 10 remote cache misses) is a good balance point in the trade-off between runtime overhead and the length of sample collection period as it results in a runtime overhead of around 2% for a period of 10 billion cycles (roughly 7 seconds of execution

on a 1.5GHz IBM POWER5).

3.4.4 Thread Clustering Accuracy

Figure 3.5 shows a visual representation of *shMap* vectors after our clustering scheme is applied for the four workloads. Each application is represented by a gray scale matrix of pixels in which each row represents an *shMap* vector for a thread. The gray scale represents the frequency of remote cache misses that are recorded for entries in the shMap vector. Darker pixels represent higher frequencies.

According to the scheme used to construct *shMap* vectors, two *shMap*s having non-zero values on the same entry is a sign of active read-write sharing. In the visual representation in Figure 3.5, this effect is demonstrated as vertical dark lines (which are formed by the dark pixels on identical columns for different rows). In order to simplify the picture, we have removed the dark pixels that are shared by almost all threads (globally-shared data).

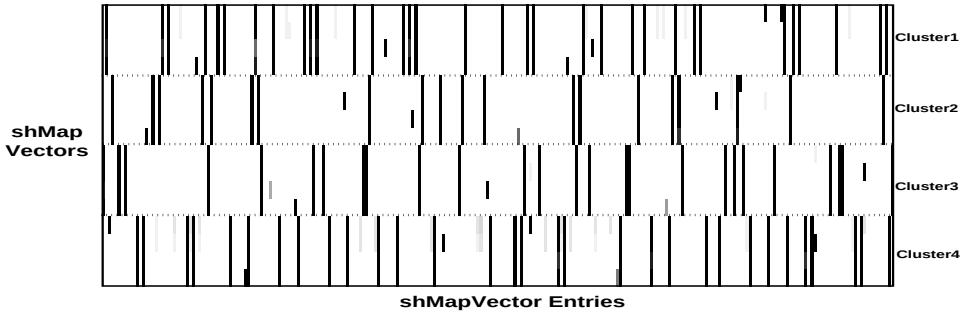
From Figure 3.5 it is clear that the *shMap*'s are effective in detecting sharing and clustering threads for three applications out of four (microbenchmark, `SPEC JBB2000`, and `RUBiS`). In the three cases, the automatically detected clusters are identical to clusters that would have been identified manually, with specific knowledge of applications logic (i.e., a cluster for each scoreboard for the microbenchmark, for each warehouse in `SPEC JBB2000`, and for each database instance in MySQL).

For `VolanoMark` however, the detected clusters do not conform to our perception of the way data is paritioned in the server (i.e., there is one data partition per chat room). It turns out that the read-write sharing patterns among the threads in `VolanoMark` is fairly complicated. Due to unavailability of the workload source code, we were unable to explore the exact behaviour of the application's threads. However, our performance results (described in the next Section) shows that significant performance improvement can be gained by a thread scheduler that takes even such a seemingly imperfect thread clustering information into account.

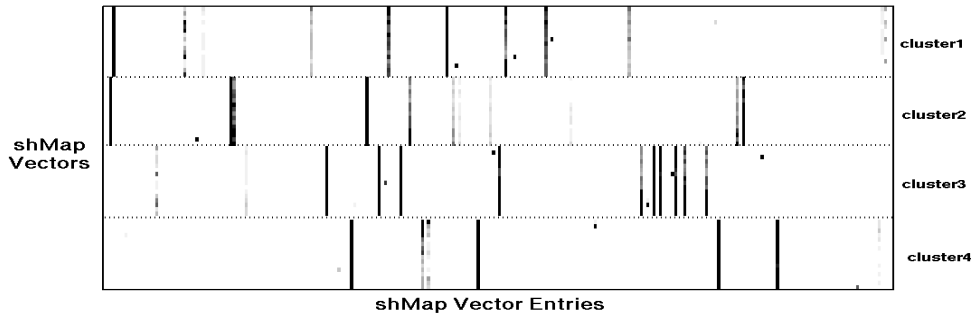
3.4.5 Performance Impact of Thread Clustering

In this subsection, we briefly describe performance results of our sharing-aware thread scheduler, developed by my colleague David Tam. The details of the experiments and their performance analysis can be found in Tam et. al [TAS07]. To summarize, our experiments with a sharing-aware thread scheduler that uses our thread clustering scheme demonstrates that most (up to 70%) expensive, cross-chip read-write sharing can be eliminated across the set of workloads we studied, compared to the default thread scheduler that is used in the Linux kernel.

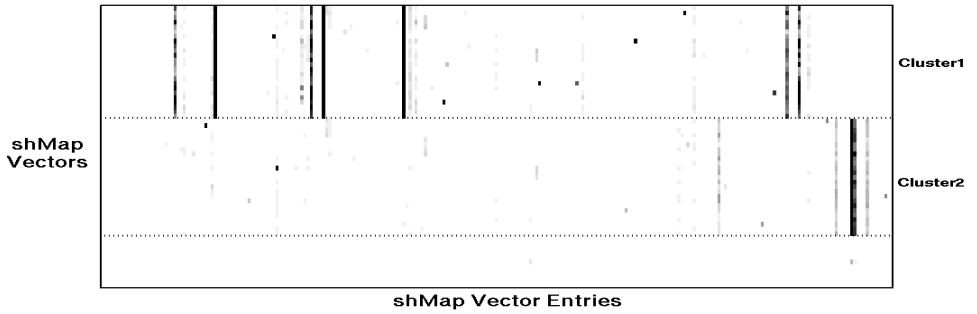
Also, our experiments on our IBM OpenPower720 machine show that the sharing-aware thread scheduler is able to improve end performance by up to 7% compared to the default Linux thread scheduler. Early results of running similar experiments on a larger multiprocessor (with 8 IBM POWER5 chips, instead of two) shows that the potential end-performance improvement can be substantially higher (e.g., up to 20%).



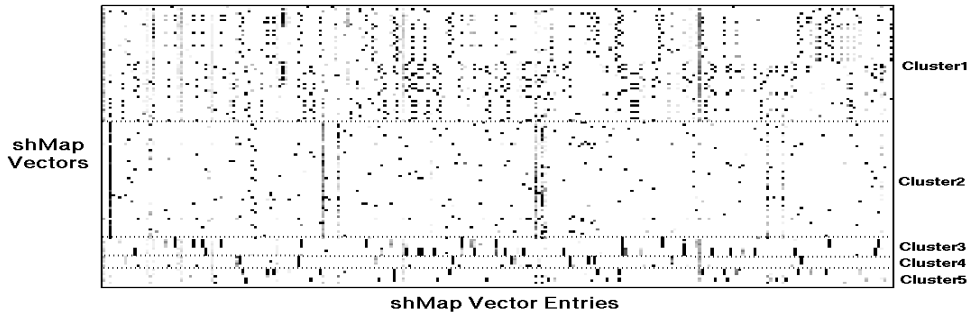
a. Microbenchmark



b. SPECJBB



c. RUBiS



d. VolanoMark

Figure 3.5: Visual representation of *shMap* vectors. Each *shMap* entry is represented with a gray scale pixel. A row of pixels in each picture represents a single thread's *shMap* vector. The more frequent remote misses on the entry, the darker the point.

3.5 Related Work

Most of the existing hardware data sampling techniques are already discussed in Section 3.1.2 where advantages and disadvantages of each of them are described. In this section, we describe some of the research work that use alternative approaches of monitoring data at the hardware level.

Intel Itanium 2 supports a latency-based filtering approach to control hardware data sampling [Inta]. In this scheme, users can specify a lower bound, in terms of number of cycles, on the latency of a data cache miss [MMdS05]. In theory, this scheme can be used to implement source-based sampling, given there is significant difference in the access latency for different memory sources. A major problem with this approach, however, is that due to a potentially large variation in the latency of accessing single source, it is difficult to find a right lower bound for latency that guarantees the capture of most data accesses to a source. Moreover, an *aliasing effect* may occur for different sources that have similar average access latency. Nonetheless, this technique is used by Buck and Hollingsworth [THa] and also Lu *et al.* [LCF+03] to isolate data addresses that frequently cause long-latency cache misses without further exploring the source of the cache misses. However, our approach for source-based sampling of data cache misses is more robust than the latency-based approach used in Itanium 2, as it does not rely on potentially fluctuating access latencies in order to identify the source of the data.

In order to address some of the inherent limitations of data sampling, some researchers have suggested alternative techniques mainly by introducing semantically richer data monitors at the hardware level. For instance, Qureshi and Patt suggest hardware *Utility Monitors* to monitor every L1 cache miss and build a summary histogram at the hardware level based on the reuse distance of cache misses [QP06]. Also, the authors of the *iWatcher* framework suggest a specific hardware data monitoring support to constantly monitor accesses to certain designated memory region, so that whenever an access to a specified region occurs, a user-defined function runs automatically by hardware without generating a trap to the operating system [ZQLT04]. While these approaches appear to be effective, they serve only specific purposes. Ideally, in the new generation of hardware

data monitoring architecture, there should be a fairly small set of mechanisms that are sufficiently flexible to be used by a wide range of optimization or debugging purposes.

3.6 Concluding Remarks

Hardware data sampling is a potentially powerful mechanism as many researchers have been able to build effective optimization schemes using data samples generated by hardware. However, in our attempt to do the same, we encountered some limitations in the existing hardware data sampling mechanisms embedded in today's processors.

First, supported hardware data sampling selection criteria are often too coarse-grained and inflexible. In particular, it is is not possible to sample data cache misses specifically based on the memory or cache source from which cache misses are served. As a result, extra software filtering is required in order to select cache miss samples of a certain source from a potentially large set of data cache misses. In this chapter, we described a technique based on features provided by the IBM POWER5 processor to solve this problem efficiently. We showed how such a filtering scheme can be used to generate sample addresses of remote cache misses.

A second limitation in today's microprocessors is that only one sampling selection criterion can be set at a time. In particular, it is not possible to combine multiple selection criteria conjunctively or disjunctively at the hardware level. In this chapter, we described how to use fine-grained HPC multiplexing to be able to use multiple sampling criteria disjunctively at fine granularity.

As a case study, we described how to use source-based data sampling to address the problem of automatically detecting sharing among concurrently running threads. We showed how to efficiently build small sharing signatures for each thread out of the hardware data samples generated for remote cache misses. Furthermore, we showed that a simple thread clustering algorithm can be used to cluster threads into groups of threads that actively share data. Our experimental analysis shows that both our source-based data sampling and our thread clustering algorithm are reasonably accurate for real commercial server workloads.

In the study, we used a technique to indirectly sample the address of data cache misses based on their sources. Although our approach works reasonably well, it is specific to IBM POWER5 processor, as it uses specific features of this processor (i.e., continuous data sampling, and the ability to count data cache misses by their sources). For more general and reliable source-based data sampling, specific hardware support would be required. Such extra hardware support would be modest, as current PMUs are already able to distinguish the source from which cache misses are satisfied. It only requires another level of filtering at the hardware level so as to update the DAR only if the data cache miss is handled by a source specified by software.

Another characteristic of the continuous data sampling feature we used in our study is that the DAR is constantly updated whether the corresponding instruction completes (retires) or not. As a result, the DAR may be updated while the CPU is executing a code path speculatively, which may turn out to be a mispredicted path and must be flushed later. The DAR, in this case, will be the operand address of an instruction that never *executed* to completion. This is a serious problem considering that approximately one in five instructions is a branch, and at any point in time, there are potentially several branches predicted in a nested fashion. Hence, it will be difficult to analyze whether the recorded DAR corresponds to a valid path or not. We believe that in order to completely resolve this issue additional hardware support is required either to *invalidate* the content of the DAR or restore its previous value in the case of a mispredicted path flush.

Finally, we believe our hardware data sampling techniques can be used for other purposes than sharing detection. For instance, one can explore the use of source-based sampling in adaptive function and data placement algorithms in a NUMA environment. Another idea is to use source-based data sampling to identify highly contended locks.

Chapter 4

Page Access Tracking to Improve Memory Management

4.1 Introduction

Computer system physical memory sizes have increased consistently over the years, yet optimizing the allocation and management of memory continues to be important. A popular perception is that memory is abundant and inexpensive. While the former may be true, the latter is certainly not. Figure 4.1 shows how the price of three medium-scaled multiprocessor systems changes as physical memory size is increased. The base price is for a setup in which each system is equipped with its maximum processing power. All prices are taken from the corresponding companies list-prices. From the figure, it is clear that memory price is the dominant factor in the cost of computer systems as they are equipped with more memory than in their standard setups.

Moreover, numerous applications exist that can exhaust any amount of physical memory available. For instance, many applications from computational biology may approach a terabyte in terms of memory requirements [ZAKB+05, BRS05]. With the re-emergence of Virtual Memory Monitors (VMMs), as a key technology for server consolidation, the number of applications simultaneously running on the same hardware increases significantly with an attendant increase in memory pressure. Worse, extending available memory through demand paging continues to grow more unattractive as disk access times,

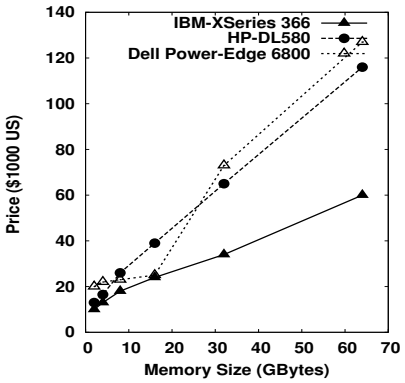


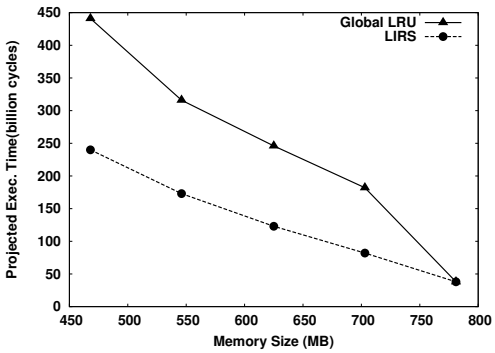
Figure 4.1: The price of medium-sized computer systems as a function of physical memory size.

dominated by positioning delays, fall farther behind relative to CPU and memory speeds.

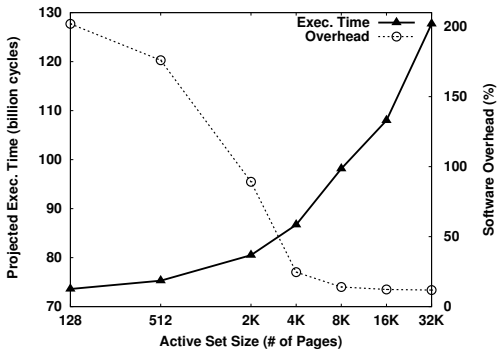
To utilize memory effectively, accurate information about the memory access pattern of applications is needed. Traditionally, operating systems track application memory accesses either by monitoring page faults or by periodically scanning page table entries for specific bits set by hardware. These approaches provide only a coarse approximation of the true order of page accesses for use in memory management algorithms, limiting the ability to implement sophisticated strategies.

An alternative approach available in systems with software-managed TLBs is to record and process page accesses upon each TLB miss. While this approach can provide significantly more fine-grained page accesses information, it adds prohibitively large overhead to a software TLB miss handler, which is already a performance-critical component.

An entirely software-based alternative has been suggested by recent work [ZPS+04, YBKM06], where virtual pages are divided into an *active set* and an *inactive set*. Pages in the inactive set are protected by manipulating page-table bits, so that every access to them will generate an exception and hence the operating system will be notified. Pages in the active set are not protected, and as a result, accesses to these are not directly tracked. Once a page in the inactive set is accessed, it is moved to the active set. A simple replacement algorithm such as CLOCK [CH81] is used to move stale pages out of the active set and into the inactive set. While the active set is much smaller than the inactive set, it is meant to absorb the majority of page accesses, which results in much reduced software overhead compared to raising an exception on every page access.



(a) Performance of LIRS vs. LRU



(b) The effect of active set size

Figure 4.2: Graph (a) shows how LIRS outperforms LRU when executing `fft`, for different memory sizes. Graph (b) shows for a fixed memory size (703Mbytes), how LIRS' performance change as the active set size increases, while the runtime overhead of maintaining the active set decreases (the projected execution time does not include the runtime overhead).

While this software-approach is shown to be effective with certain types of applications, its overhead for many memory-intensive applications is unacceptably high. An approach to reduce the overhead is to increase the size of the active set adaptively [YBKM06]. However, the bigger the active set, the less *accurate* the sequence of page accesses will be, since more accesses are absorbed by the active set. As a result, the utility of having the sequence of page accesses for a particular memory management algorithm will diminish. An example of such a case is shown in Figure 4.2 for `FFT` taken from the Splash-2 suite [WOT+95]. On the left, the performance of LIRS [JZ02], a well-known memory management algorithm is compared against LRU. LIRS takes into account not only recency of page accesses, but also reuse distance when considering a page for replacement. A more detailed description of LIRS is presented in Section 4.3.2. The measurement is done under the assumption that the overhead of collecting page access information is zero and the active set is 128 entries. The graph on the right shows how LIRS performance degrades as the active set size increases, while the overhead of recording page accesses naturally decreases. As a result, to achieve LIRS' potential in improving performance, a high runtime overhead (100% or more) must be paid, otherwise, most of the advantage of LIRS over LRU disappear.

To cope with this potentially large overhead, custom hardware is suggested by Zhou et al. [ZPS+04]. While their approach is effective in tracking physical memory *Miss Ratio Curves*, it does not provide raw page access information to the operating system, and thus cannot be used for memory management algorithms other than the one which is intended for. Moreover, the hardware required by this approach is substantial and grows with the size of physical memory.

In this chapter, we propose a novel Page Access Tracking Hardware (PATH) to be added to the processor micro-architecture for the purpose of monitoring application memory access patterns at fine granularity and with low overhead. Similar to the software approach, PATH is designed based on two observations. First, a relatively small set of *hot* pages is responsible for a large fraction of the total page accesses. Second, the exact order of page accesses within the hot set is unimportant since these pages should always be in memory. By ignoring accesses to hot pages, we can vastly reduce the number of accesses that must be tracked, while focusing on the set of pages that are interesting candidates for memory management optimizations.

The key innovation with our PATH design lies in the tradeoff between functionality assigned to hardware and functionality assigned to software. The hardware we propose is (i) small and simple, (ii) scalable, in that it is independent of system memory size, and (iii) introduces little overhead, imposing no delays on the common execution path of the micro-architecture. We delegate to software (specifically, an exception handler) the online maintenance of data structures to be used by the memory manager when making policy decisions.

We show that the operating system can benefit from our approximate information by considering three uses for the memory manager: (i) implementing more adaptive page replacement policies (ii) allocating memory to VMMs, processes or virtual memory regions so as to provide better isolation and to better support process priorities, and (iii) prefetching pages from virtual memory swap space or memory-mapped files when applications have non-trivial memory access patterns. We briefly describe these use cases in more details.

Adaptive Page Replacement: There is a large body of prior art in page and

buffer cache replacement policies [MM03, SKW03, JS94, BM04, GC97, JZ02, JCZ05, GBH04, ZvBB05, CNMC00, KMC02]. Many of the algorithms proposed are based on an approximation of LRU with various extensions to adapt to sequential and looping patterns for which LRU behaves poorly. In most cases, the effectiveness of these algorithms has only been shown in the context of file system caching, where precise information on the timing and order of accesses is available. With finer-grained virtual memory access information, adaptive page replacement algorithms, such as the one we present in Section 4.3, can lead to significant performance improvements.

Memory Allocation: Most existing operating systems allocate memory pages to applications on-demand and from a global pool. This strategy can lead to unfair prioritization effects, whereby a low-priority process with a high page fault rate is allocated a large number of physical pages to the detriment of higher priority processes. System throughput may also suffer since extra pages may be allocated to applications that do not benefit from them. Working set models, as implemented on current architectures, do not provide an accurate estimate of memory requirements since they only take into account whether a page is accessed or not over a period of time. The number of distinct pages accessed before a given page is reused (i.e., the *reuse distance* [ZPS+04]) gives a better indication of memory needs, but operating systems often do not have access to sufficiently detailed page access information to estimate of reuse distance. In Section 4.4, we show how our PATH provides accurate reuse distance information that can be used to improve memory allocation.

Virtual Memory Prefetching: I/O bandwidth has increased dramatically over the years, which allows for more aggressive and speculative prefetching of memory pages. The danger with an aggressive prefetching scheme, however, is that pages could be replaced that would still be of use (to the same or other applications). Hence it is important to prefetch wisely. Conventional operating system prefetching schemes based on spatial locality assume that whenever a page is accessed, it is likely that neighboring pages will also soon be accessed. While simple and effective for many applications, this can perform poorly for applications with little spatial locality in their access patterns. The availability of finer-grained page access information allows for alternative prefetching schemes; we

describe one strategy based on temporal locality in Section 4.5.

Underlying all these techniques for improved memory management is our PATH support for tracking page accesses at relatively fine granularity, which we describe in detail in Section 4.2.

Our simulation results show that substantial performance improvements (up to 500% in some cases) can be achieved, especially when the system is under memory pressure. While the algorithms based on PATH have different time and space overhead tradeoffs, the basic overhead of providing fine-grained page-access information to the operating system is less than 6% across all the applications we examined (less than 3% in all but two applications) which is at least an order of magnitude less than the overhead of existing software approaches.

4.2 Tracking Page Accesses

Memory management algorithms are often first described assuming the complete page access sequence is available and later implemented using a coarse approximation of this sequence. For example, the well-known Least-Recently-Used (LRU) page replacement algorithm requires the complete access sequence to implement exactly, but is commonly approximated by the CLOCK algorithm, which coarsely groups pages into *recently-used*, *somewhat recently-used*, and *not recently-used* categories. Optimizations to the basic LRU algorithm, and other sophisticated memory management strategies, require more detailed page access information than systems currently provide. Tracking *all* page accesses, however, is prohibitively expensive and generates too much information for online processing. The key question, then, is how to reduce the volume of information to a manageable level, while retaining sufficient detail on the order of page accesses.

Our approach is based on two observations: (i) a relatively small set of *hot* pages are responsible for a large fraction of the total page accesses, and (ii) the exact order of page accesses within the set of hot pages is unimportant since these pages should always be in memory. By ignoring accesses to hot pages, we can vastly reduce the number of accesses that must be tracked, while focusing on the set of pages that are interesting candidates

for memory managmenet optimizations.

4.2.1 Design Options

Current memory management hardware already contains an effective filter to catch accesses to the hottest N pages: the Translation Lookaside Buffer (TLB). Thus, one possibility for tracking page accesses is to augment existing hardware or software TLB miss handlers to record a trace of all TLB misses. Aside from the overhead that this would add to the critical path of address translation, the primary problem with this strategy is that TLBs are too small on today's processors to capture the set of hot pages, which in turn leads to traces that are still too large for online use. Simply increasing the TLB size is not a viable option, since the size is limited by fast access requirements. We note, however, that the TLB provides more functionality than is needed to simply track page accesses. Thus, we propose the addition of a new hardware structure that essentially functions as a significantly larger TLB for the purpose of filtering out accesses to hot pages, while recording a trace of accesses to other pages. We call this structure *Page Access Tracking Hardware (PATH)*. Although software TLB miss handlers could collect the same information (for architectures that provide them), extra work would be required on the performance-critical miss handling path.

The existing TLB can continue to filter out accesses to the hottest pages, while the new PATH maintains a superset of the pages handled by the TLB, and is only needed when a TLB miss occurs. Further, PATH is not required for address translation, and can be accessed asynchronously with respect to TLB miss handling. Since the speed of access is not critical, the components in PATH can be sized independently, constrained only by the resources available on chip and the desired precision of tracking. In the following subsections, we present the details of our PATH design and show how the access traces it collects can be used to build various software data structures used for memory management.

Figure 4.3 depicts the three major components of our PATH design. The *Page Access Buffer* (PAB) and the *Associative Filter* work together to remove accesses to hot pages from the trace; other accesses are recorded in the *Page Access Log* (PAL), which raises

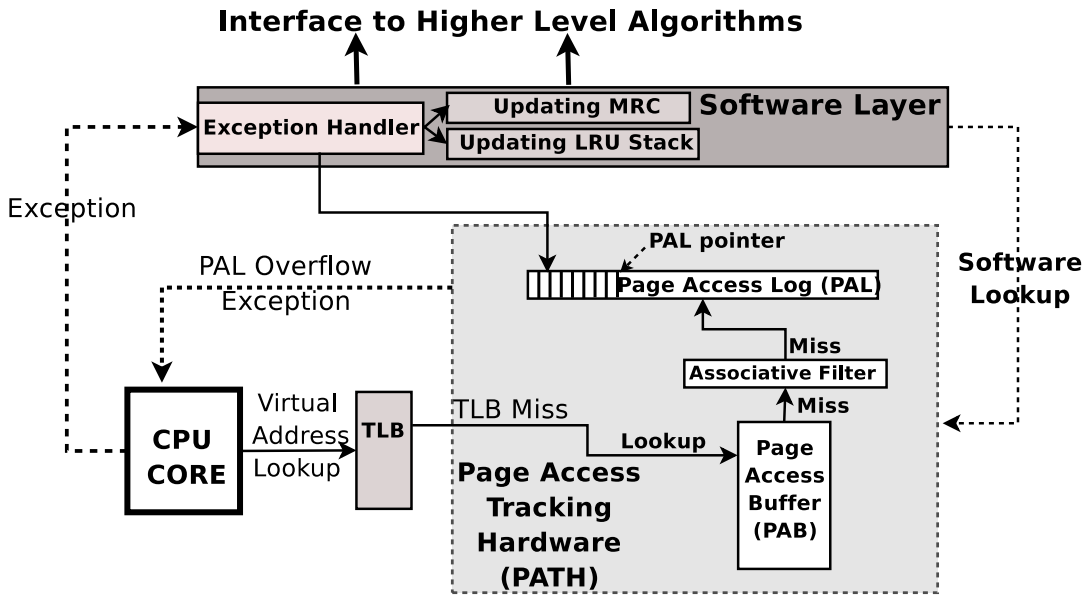


Figure 4.3: The Architecture of Page Access Tracking Hardware (PATH).

an exception when it becomes full, in order to allow for software processing.

The Page Access Buffer (PAB) contains the set of recently accessed virtual pages, augmented with an address space identifier to distinguish between pages from different processes. The PAB is structurally similar to a TLB except that (i) it is updated only on a TLB miss, (ii) it need not contain the physical addresses of the pages it holds, and (iii) it is significantly larger than a typical TLB. As the PAB size increases, more pages are considered *hot* and more accesses are filtered out of the trace, thus reducing both processing overhead and accuracy. Our experiments show that a PAB with 2048 entries is a good point in this tradeoff. In Section 4.6.6, we examine, in detail, the tradeoff between overhead and usefulness of the traces with varying PAB sizes. Moreover, with a 2K-entry PAB, PATH will have a very small chip footprint. Finally, some existing architectures such as IBM POWER and AMD Opteron already have a fairly large (e.g. 512 to 1024 entry) second-level TLB[1]. One can envision integrating PATH with a slightly larger version of such a second-level TLB. We show in Section 4.6.7 that using the same

[1]IBM POWER's first level address translation cache is 128 entries and is called the Effective-to-Real Address Table (ERAT).

2K size for the active set in the software approach will result in an unacceptably high overhead.

A page access is considered for recording only if it misses in the PAB. However, because of the limited associativity of the PAB, it can be susceptible to repeated conflict misses from the same small set of (hot) pages. To deal with this problem, PATH includes an Associative Filter that filters page access information further. The associative filter is a small (e.g., 64 entries), fully-associative table with an LRU replacement policy that is updated on every PAB miss. It effectively filters out the recording of accesses to hot pages due to short term conflict misses in the PAB.

Finally, misses in the associative filter are recorded in the Page Access Log (PAL) which is a small (e.g., 128 entry) buffer. When the log becomes full, an exception is raised, causing an operating system exception handler to read the contents of the PAL and reset the PAL pointer. In the following subsection, we show how system software can use the information recorded in the PAL to construct a variety of data structures used in memory management.

PATH must also provide an interface to allow software to control it and perform lookup operations. This interface allows the operating system to empty the PAL whenever the CPU becomes idle, say during I/O, to reduce the overhead of PAL servicing. The operating system can also dynamically turn off PATH when the system is not under memory pressure, thereby reducing power consumption.

Given this architecture, PATH provides a fine-grained approximation of the sequence of pages that are accessed. Sequential or looping access patterns over an area larger than what is covered by the PAB (e.g., 64MB) are very likely to be completely recorded by PATH in their proper order. Moreover, if a page is not hot so that it does not permanently reside in the PAB, its reuse distance can also be accurately captured by PATH due to the subsequent PAB misses it causes.

4.2.2 Low-level Software Structures

The benefits of having LRU stacks and/or Miss Rate Curves available are well recognized and, correspondingly, hardware support to generate these data structures has been

previously proposed [ZPS+04]. In this section, we argue that these data structures can be constructed efficiently in software from the information obtained by PATH described above. Specifically, we show how both LRU stacks and Miss Rate Curves can be maintained on-line by the PAL overflow exception handling code. Both structures can, in turn, be used by memory management software to make informed decisions. By delegating the maintenance of these data structures to software, our design provides greater flexibility and customizability than previously proposed hardware support.

LRU Stack

The LRU stack maintains a recency order among the pages within an address range. The top of the stack is the most recently accessed page, while the bottom of the stack is the least recently accessed page. In our scheme, each page accessed (as recorded by the PAL) is moved from its current location in the stack to the top of the stack. The LRU stack is updated for every page access recorded in the PAL.

To enable fast page lookup and efficient update in the LRU stack, we suggest using a structure typically used to maintain page tables, such as a traditional multi-level page table or a hash table. Each element in this structure represents a virtual page and contains two references: one to the previous page in the LRU stack and one to the next page in the LRU stack. Conceptually, the LRU stack is a doubly-linked list, and elements are repositioned within the stack by adjusting references to neighboring elements. Thus, a virtual page can be looked up with a few (usually 2 or 3) linear indexing operations, and moving a page to the top of the LRU stack involves updating at most 6 reference fields in the stack: 2 references associated with the page being moved, 2 of its previous neighbors, 1 at the previous head of the list, and the head of the list itself.

The LRU stack has an element for each page that was ever accessed (not just the pages currently in memory). Assuming 4 KB virtual pages, 32-bit page references can be used for address ranges up to 16 TB, resulting in a space overhead of 8 bytes per virtual page used. To save on physical memory usage, LRU stack pages can be swapped out to disk if the elements they contain represent pages that are not currently being accessed. The working set size of the LRU stack is roughly proportional to the working set size of

the address range. Hence, a working set size of several GB implies that several MB will be consumed by the LRU stack.

Miss Rate Curve

A Miss Rate Curve (MRC) depicts the page miss rate for different memory sizes, given a page replacement strategy. More formally, MRC is a function, $\lambda_{r,p}(M)$, defined for address range r and page replacement policy p. $\lambda_{r,p}(M)$ identifies the number of page misses the process will incur on r over a certain time period if M physical pages are available. Often, the slope of λ at a given memory size is of more interest than its actual value. If the slope is flat then making additional pages available will not significantly reduce the miss rate, but if the slope is steep then even a few additional pages can significantly reduce the page miss rate.

We use a definition of MRC that is slightly different from the one used by Zhou *et al.* [ZPS+04] to make it more suitable for our proposed hardware support. Our variant of MRC identifies the absolute number of misses that occur over a period of time and not the miss ratio that is normalized by dividing the number of misses by the total number of accesses. Not requiring the total number of memory accesses significantly simplifies the hardware support required.

Our method of maintaining λ on-line is based on Mattson's stack algorithm [MGST70] and Kim *et al.*'s algorithm [KHW91] used for off-line analysis. We augment the elements of the LRU stack described above with a *rank* field used to record the distance of the element from top of the stack (i.e., the reuse distance). Each λ is maintained as a histogram. Conceptually, whenever a page is accessed, the histogram values corresponding to memory sizes smaller than the rank of the accessed page are incremented by one. In addition, the page is moved to the top of the stack, while setting its rank field to zero and decrementing the rank field of every element between the original position of the page and the previous top of stack by one.

Time is divided into a series of *epochs* (e.g., a few seconds). At the end of each epoch, the value of λ (i.e., the histogram) is saved and reset. Each process may store a history of values of λ for several epochs in order to be able to make more accurate decisions.

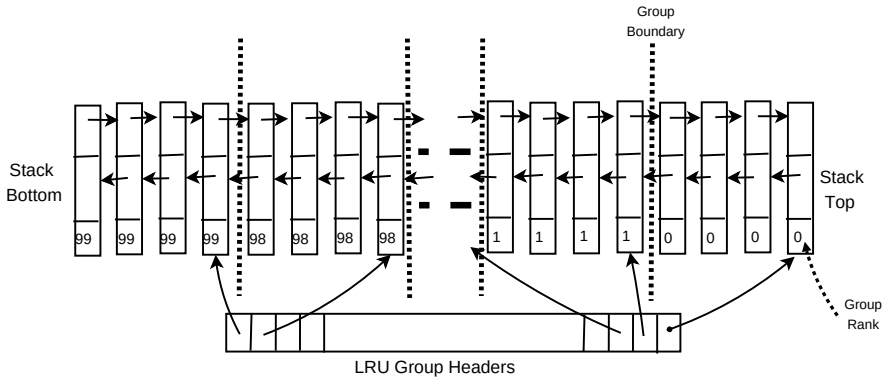


Figure 4.4: The LRU stack with group headers that are used for updating the LRU-ranks of pages efficiently.

In order to reduce overhead, page groups of size g can be defined and the rank field can be redefined to record the distance to the top of the stack in terms of number of page groups. Adding an array of references to the head of each page group reduces the overhead of updating the rank fields by a factor of g. Figure 4.4 shows how the group header array is used to find the group boundaries, since only the elements at the group boundaries need to be updated.

Algorithm 2 shows the basic steps that must be taken for every page that appears in the PAL to maintain λ histograms for the LRU page replacement policy. Note that the group size g is defined by software and can change according to the desired level of precision for λ.

A further optimization is possible based on the observation that at any instance in time, we are only interested in λ at the point corresponding to the amount of physical memory allocated to the virtual address range under study and the slope of λ around that point. Hence, the LRU stack can be divided into only 4 groups as shown in Figure 4.5: the top $M - g$ pages, where M is the current physical memory allocated to the address range, two groups of g pages on both sides of M, and all the remaining pages at the bottom of the LRU stack. With this optimization, only four entries need to be updated on each page access to maintain λ.

In the next three sections, we present algorithms to improve memory management performance in three different areas: adpative page replacement, process memory allo-

Require: $Vaddr \geq RegionStart \wedge Vaddr \leq (RegionStart + RegionSize)$

1: $lruRank \Leftarrow Stack[Vaddr].rank$
2: move $Vaddr$ element to the top of the LRU stack
3: $Stack[Vaddr].rank = 0$
{*update group headers and page ranks for groups lower than lruRank*}
4: **for** $i = 0$ to $lruRank$ **do**
5: $GroupHeaders[i] \Leftarrow Stack[GroupHeaders[i]].prev$
6: $Stack[GroupHeaders[i]].rank++$
7: **end for**
{*update MRC for LRU*}
8: ***for*** $j = 0$ *to* $lruRank$ ***do***
9: $\lambda_{LRU}[j]++$
10: ***end for***

Algorithm 2: Update λ_{LRU} and the LRU stack on each recorded page V_{addr}.

cation, and virtual memory prefetchinag. We describe how these algorithm utilize the information generated by PATH either in the raw form, or in the form of LRU stack or MRC.

4.3 Adaptive Replacement Policies

Using information from PATH, we have implemented two adaptive page replacement algorithms. The first one, *Region-Specific Replacement*, attempts to automatically apply the appropriate replacement policy on a per-region basis for different regions defined in the application's virtual address space. The second one is a recently proposed adaptive policy called *Low Inter–Reference Set (LIRS)* [JZ02]. The reason for choosing LIRS is that it is fairly simple and, for file system caching, has proven to be competitive with the best algorithms.

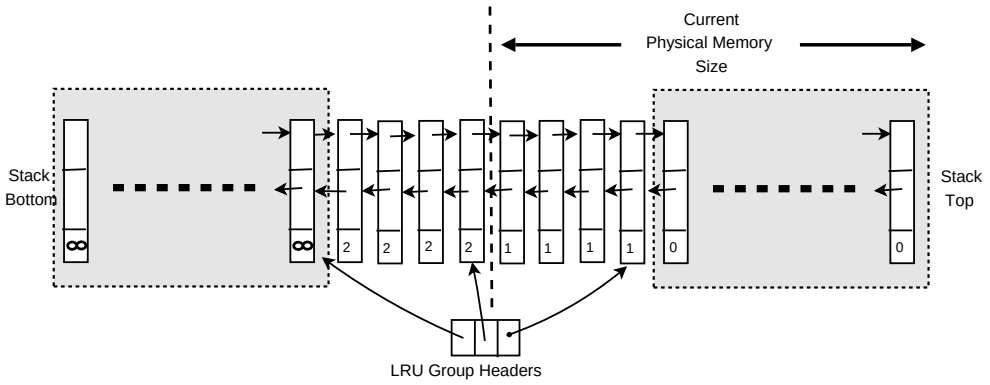


Figure 4.5: The optimized structure for the LRU group headers, considering in most cases it is important to know the slope of λ only around the current physical memory size. Only a fixed number of page groups (4 in this figure) are considered to be updated at each page access.

4.3.1 Region-Specific Replacement

The rationale behind region-specific page replacement is the desire to be able to react individually to the specific access patterns of each large data structure within a single application. Studies in the context of file system caching [CNMC00] have shown that by analyzing the accesses to individual files separately, one can model the access pattern of the applications more accurately. Also, Harty and Cheriton [HC92] presented a framework for application-controlled page caching in which each application can employ caching policies that fit its needs most. We argue that memory-consuming data structures (e.g., multidimensional arrays, hash-tables, graphs) usually have stable access patterns, and by detecting these patterns, one can optimize the caching scheme for each of these data structures individually.

Selecting Regions

Most large data structures reside in contiguous regions in the virtual address space. The contiguity of data structure memory is not an essential factor but significantly simplifies the implementation of region-specific replacement. For large data structures that do not reside in contiguous regions, one can use custom allocators that allocate correlated data from a pre-allocated large chunk of virtual memory. Lattner and Adve [LA05] show how

to cluster individually allocated, but correlated, memory items in an automated fashion. As a result, large data structures (e.g., a graph of millions of nodes) are more likely to be located in a large contiguous region of address space. In our simulation studies, we have assigned a region for each large static data structure as well as any large `mmap`ped areas.

Choosing Replacement Policy

We separately but simultaneously compute λ for each region for both the LRU and MRU policies, and we pick the policy that would result in a lower miss rate. To compute λ_{MRU}, we use the same scheme shown in Figure 4.5 and Algorithm 2, but with pages ranked in reverse order. Hence, for each page, we maintain two ranks, one for LRU and the other for MRU. Given that the rank value is at most 4, the rank can be represented by two bits, so the corresponding space overhead is negligible.

Switching Replacement Policy

We switch to a new policy only if it is consistently better than the current policy. The default policy is LRU. If a region is being accessed in a looping pattern, it will have lower values for λ_{MRU}, but if the region is being accessed in temporal clusters, λ_{LRU} will have lower value.

The algorithm for switching page replacement policy is activated only if a certain threshold in the number of capacity misses is reached in an epoch. Otherwise, we assume the current replacement policy is working well.

However, switching is an expensive operation and should not be done lightly. To avoid over-reacting to short-lived fluctuations, we use a saturating counter that is incremented when one policy is better than the other in an epoch, and decremented otherwise. The policy switch is triggered whenever the counter reaches one of two extreme points. Also, to reduce switching overheads, we do not evict the current pages from physical memory when a policy switch is made. We have observed in our experiments that for many real applications policy switching is indeed a rare event.

Allocating Memory to Regions

With region-specific page replacement, it is necessary to decide how many physical pages to allocate for each region. At the end of each epoch, we use the precomputed λ values to calculate how much memory each region actually needs. We define *benefit* and *penalty* functions for each region as follows:

$$benefit_r(g) = \lambda_{r,p}(M - g) - \lambda_{r,p}(M)$$
$$penalty_r(g) = \lambda_{r,p}(M) - \lambda_{r,p}(M + g)$$

We balance memory among regions within a single process address space by taking away memory from regions with low penalty and awarding them to the regions with higher benefit. The number of regions in an application is typically small (e.g., usually less than 10). Thus, balancing memory within a single application at the end of each epoch is not a costly operation.

4.3.2 LIRS

The key idea behind LIRS is to consider not only recency, but also reuse distance when considering a page for replacement. The LIRS algorithm divides pages into two sets: *High Inter-reference Recency* (HIR) and *Low Inter-reference Recency* (LIR) sets. The pages in the LIR set are always kept resident in memory even if they have not been recently accessed. Candidate pages for replacement are always chosen from the HIR set even if they have been recently accessed. Once a page in the HIR set is accessed with a reuse distance shorter than that of some pages in the LIR set, it is moved to the LIR set. If a page stays in LIR for a long time without being accessed again, it is purged from the LIR set. Only a small fraction of physical memory is allocated to pages in the HIR set. A more detailed description of the algorithm can be found in the LIRS paper [JZ02].

LIRS effectively eliminates LRU's poor handling of sequential and looping patterns in file system caching. However, to apply LIRS to virtual memory, one must be able to measure the distance between two consecutive references to a page fairly accurately, which is challenging with traditional operating system page access monitoring techniques.

In fact, a follow-up attempt to implement LIRS for virtual memory by the LIRS designers using conventional operating system techniques resulted in a complicated algorithm to approximate the reuse distance information and limited success [JCZ05]. By using information from PATH, we were able to implement the original LIRS algorithm in a straightforward way. In fact, we adopted the LIRS designers' original algorithm in our simulation environment with minor modifications.

4.4 Memory Allocation

In multi-programmed environments, how the operating system decides to allocate physical memory to each process is of great importance when the system is under memory pressure. In this section, we show how the availability of fine-grained page access information can help improve memory allocation among processes.

In most general-purpose operating systems today, memory is allocated to a process from a global pool of pages, on-demand, when the process incurs a fault. All pages are equal candidates for replacement, irrespective of which process they belong to. The actual amount of memory allocated to each process is a direct function of the page replacement policy in use and the page fault rate of the process. Processes that access more pages than others over a period of time will be allocated a larger number of pages, since they fault on more pages and keep their own pages recent.

Global memory allocation has two major advantages. First, it is simple and easy to implement with little overhead. Secondly, for workloads with similar access patterns, global memory allocation naturally tends to minimize the total number of page-faults. Despite its wide adoption, global memory allocation has two significant shortcomings: (i) sub-optimal system throughput for workloads with different access patterns, and (ii) lack of process isolation and unfair prioritization effects. We briefly discuss these two shortcomings in more detail.

Sub-optimal System Throughput: Global memory allocation makes the assumption that each application benefits the same when given an extra page. In reality, however, one application's throughput may rise sharply as it is given more pages, whereas others

may experience no performance gains. If the goal is to maximize overall system throughput, memory should be taken away from processes that are not benefitting much from them and be given to processes that benefit the most. The rationale is that while the throughput of the victim processes are not seriously affected, a large boost in the throughput of the processes that are assigned more memory can be obtained. However, a major challenge is how to accurately measure the utility of pages for different applications.

Lack of Isolation and Prioritization: In a system under memory pressure, process prioritization done only through CPU scheduling can be ineffective. Chapin has illustrated the prioritization problem due to lack of memory isolation in operating systems, and motivated the concept of *memory prioritization* [Cha97]. As a simple example, consider two processes A and B, with A's working set size larger than system memory size and B's working set size considerably smaller. Moreover, B is slow to touch the pages in its working set (e.g., due to a high amount of computation). Even if the user gives much higher priority to B than A, a system with global memory management will not isolate B from A. A's page-fault rate will be much higher than B's, despite the fact that B has more CPU time. As a consequence, pages from process B will be victimized in order to accommodate page faults from process A. Prioritization is especially important in large servers used for server consolidation where each user runs its application within a virtual machine. In order to maintain a certain level of service for each user, the operating system must be able to protect processes (i.e., virtual machine instances) from being deprived of memory by other memory-consuming applications.

To address the two shortcomings of global memory allocation discussed above, we employ a local memory allocation scheme, where each process is given a pool of private pages that can then be governed by its independent page replacement policy. Memory pools are dynamically sized as new processes are launched, existing processes' memory demand changes, or processes exit. A major challenge in local memory allocation is to detect how much memory an application needs at any given point in time. Simple sampling schemes, such as the one suggested by Waldspurger et al., have been shown to be effective in measuring the working set of an application [Wal02]. The problem with the working set model is that it does not give an indication of how the performance

of an application will change if it is given less memory than the measured working set size, which becomes an issue in systems under memory pressure. We use the MRC model both for maximizing throughput and enforcing effective isolation and prioritization among processes.

Maximizing Throughput

Our approach to optimizing throughput is similar to the greedy algorithm used by Zhou *et al.* [ZPS+04] with a different level of hardware integration. In this approach, each process is initially allocated an equal amount of physical memory. At each memory allocation step, given λ is calculated for all processes, $penalty_P$ and $benfit_P$ for process P are calculated as follows:

$$benefit_P(g) = \lambda_p(M) - \lambda_p(M+g)$$
$$penalty_P(g) = \lambda_p(M-g) - \lambda_p(M)$$

The greedy algorithm takes g pages away from the process with the least value for $penalty_P(g)$, and assigns them to the process with the highest value for $benefit_P(g)$.

We calculate λ for each process by treating the entire process address space as a single region. If a process uses region-specific page replacement, as described in Section 4.3.1, we can measure the penalty of reducing process memory by using λ functions already calculated for each region, and define $benefit_P$ and $penalty_P$ functions for the process as:

$$benefit_P(g) = \lambda_{r_{max},p}(M_{r_{max}}) - \lambda_{r_{max},p}(M_{r_{max}} + g)$$
$$penalty_P(g) = \lambda_{r_{min},p}(M_{r_{min}} - g) - \lambda_{r_{min},p}(M_{r_{min}})$$

where r_{min} is the region with minimum *penalty*, and $M_{r_{min}}$ is the number of pages currently allocated to region r_{min}. Similarly r_{max} is the region with maximum *benefit*, and $M_{r_{max}}$ is the number of physical pages allocated to region r_{max}

Enforcing Priorities

To better support process priorities, we have implemented a simple policy to try to balance application miss rates among applications with the same priority. Figure 4.6 shows an abstracted example for two processes. At any point in time, the available

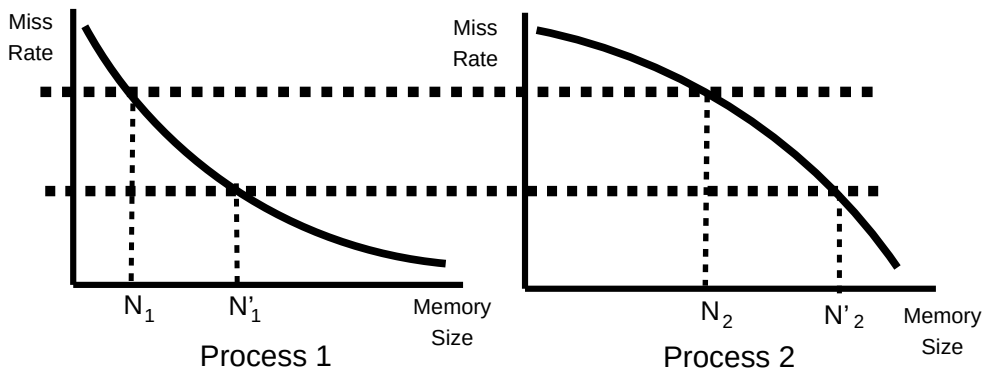


Figure 4.6: Enforcing priority through balancing page miss rate. At each point in time the policy is to ensure same page fault rate for both applications. As available memory changes, different page fault rate is set for both applications.

physical memory is dynamically partitioned between the two processes so that the two processes suffer the same page miss rate. In the example, N_1 pages are allocated to $Process_1$ and N_2 pages are allocated to $Process_2$, where $N_1 + N_2$ is equal to the amount of available memory and both processes suffer from the same miss rate. If the amount of available physical memory changes (as other processes launch or exit, for instance), the balance line will be moved to a new level to accommodate the change. In this case, N_1' pages are allocated to $Process_1$ and N_2' pages to $Process_2$, such that $N_1' + N_2'$ is equal to the amount of available memory and both processes suffer from the same miss rate.

Another policy might dynamically partition memory according to λ values so that each process runs with a performance that is within a small margin of the performance level required by a *Service-Level Agreement* (SLA). In this approach, physical memory allocated to a process changes freely as long as the miss rate stays within the acceptable miss rate range that is specified by the SLA. The λ function is used to predict the miss rate for any given physical memory size. We are continuing to explore fairness and process isolation using the fine-grained memory access information provided by PATH.

4.5 Prefetching

Another common technique to close the latency gap between disk and memory access is prefetching by predicting which pages an application will use in the near future, and start fetching these pages to memory before they are actually used. Prefetching is particularly effective for applications with working set sizes so large that even an optimal page replacement policy still results in a high page fault rate.

Given the rapid growth of disk I/O bandwidth in recent years, one can aggressively employ speculative prefetching techniques that trade potentially wasted I/O bandwidth for additional improvement in latency. The problem with speculative prefetching is that it may result in still-needed pages of either the same or other applications being replaced. In order to avoid this problem, speculation precision must be high, meaning a page that is replaced by a prefetched page should not be accessed earlier than the prefetched page again.

There are several policies for predicting which pages to prefetch. A simple approach is based on *spatial locality*: pages that are adjacent to a faulted page in the virtual address space are candidates for prefetching on the assumption that they will also be accessed soon. More precisely, whenever a page-fault occurs, the next w next pages in the address space would be prefetched, where the value of w could be either fixed or dynamically adjusted based on how accurately the prefetching policy has been performing. This scheme is effective for many cases, since many large memory-consuming applications access pages in contiguous chunks that are much larger than a virtual page size. However, there are important classes of applications that access memory with different types of regularity than spatial locality.

Another prefetching approach is based on automatically analyzing application logic in order to identify regular access patterns. With this approach, a compiler inspects program source code and inserts code into the executable to provide hints to the operating system on which pages should be prefetched soon. The main advantage of this approach is that it automatically exploits high-level information on programs page accesses and hence can identify regularities that are hard to identify by just monitoring the sequence of

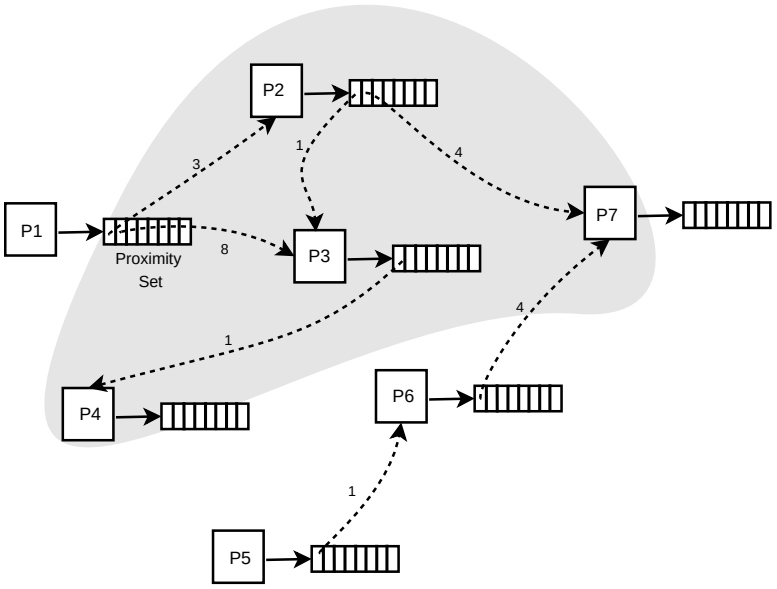


Figure 4.7: Page Proximity Graph (PPG). Each node represents a virtual page. Each node has a fixed maximum number of edges to other nodes. An edge represents the fact that there is temporal proximity between the adjacent nodes. The weight on each edge represent the number of times such temporal proximity is observed between the two nodes.

accessed virtual pages. The major drawbacks of compiler hint-based prefetching are two fold. First, it is applicable only to applications whose access pattern can be analyzed by a static compiler analysis. In principle, one can extend this approach to a run-time environment (e.g., Java virtual machine) where more information regarding program data structures and execution path is available. To the best of our knowledge, this approach has not been explored yet. The second drawback of compiler hint-based prefetching is that it is specific to particular programming environments that have a compiler modified for generating prefetching hints.

As an alternative, we have developed a prediction model, similar to a Markov predictor [JG99], that incorporates the temporal locality of accesses to pages into the prefetching strategy. By temporal locality we refer to the fact that a set of pages are accessed within a short period of time (e.g., time to access a few pages).

Similar to recency-based prediction models, such as the one proposed by Saulsbury el. al [SDS00], we use the LRU stack to find temporal locality among pages. Note, however, that for this purpose the LRU stack must be precisely maintained. As we showed in

Section 4.2.2, the LRU stack is accurately maintained by using the PATH-generated information.

We propose a new strategy based on *temporal locality*, which assumes that if a set of pages are accessed repeatedly, they are likely to be accessed again together within a short period of time.

To detect pages that are accessed with temporal locality, we build a weighted directed graph, called *Page Proximity Graph* (PPG). Each virtual page is a node in the graph. An edge (p_1, p_2) indicates that page p_2 was accessed shortly after p_1. Each edge has a weight, w, that indicates how many times the two pages were accessed within a short period of time. To save space, PPG's degree D is limited to a small number (e.g., 10). For each page p, we maintain a *Proximity Set*, X_p, where $|X_p|$ is at most D. Figure 4.7 shows a simple example of PPG where D is equal to 8.

The PPG is updated on each page fault as follows. A window of W_{scan} pages in the LRU stack is considered, starting from the current location of the faulted page p towards the top of the stack. If any page q in the scan window is already in X_p, the weight on (p, q) is incremented by one. Otherwise, q is considered as a candidate to be added to X_p. The weight to all other nodes in X_p that do not appear in the scan window is decremented in order to decay obsolete proximity information. If the weight on any edge (p, s) reaches zero, s is removed from X_p.

Prefetching is initiated whenever a page fault occurs on a page, such as p. To generate the set of pages to be prefetched, the PPG is traversed starting from p in a breadth-first fashion, and all pages encountered are added to the prefetch set. In Figure 4.7, the prefetched set starting from P_1 is shown in gray, when traversing to a depth of 2. The deeper the breadth-first traversal, the more speculative prefetching will be. One can dynamically adjust the depth of the traversal according to the current prefetching effectiveness and available I/O bandwidth. If a page in the prefetch set is already resident in memory, it will be artificially touched to prevent the page replacement algorithm from evicting it under the assumption that the page will likely be accessed soon.

We evaluate prefetching effectiveness using two metrics, *recall* and *precision*, where recall is measured as the number of page-faults that are prevented from occurring by

Application	Suite	Description	Footprint (MB)
MrBayes	N/A	Bayesian inference of phylogeny	600
MMCubing	Illimine	Data Cubing by factorizing lattice space	480
SPECJbb2000	N/A	Commercial Server Workload	850
FFT	Splash 2	Fast-Fourier Transform	770
Ocean(contiguous partitions)	Splash 2	Large-scale Ocean Movement Simulator	889
Ocean(non-contiguous partitions)	Splash 2	Large-scale Ocean Movement Simulator	903
LU(contiguous partitions)	Splash 2	Simulated CFD using SSOR	760
LU(non-contiguous partitions)	Splash 2	Simulated CFD using SSOR	800
FMM	Splash 2	N-body problem, Fast Multipole Method method	480
CG	NPB	Conjugate Gradient Method	476
BT	NPB	Block Approximate Factorization	691
MG	NPB	Mult-Grid Kernel	430
SP	NPB	Solving a system of Pentadiagonal equations	724

Table 4.1: Selected Memory Intensive Applications

prefetching, and precision is measured by measuring the extra I/O bandwidth that is imposed by prefetching. The more precise prefetching is, the lower the required I/O bandwidth will be.

In order to limit the potential negative effect of prefetching in evicting still-needed pages, we limit the number of pages that are prefetched but not yet accessed by the application. Once this limit is reached, the prefetching algorithm stops until some of the prefetched pages are actually used. As a result, a prefetching scheme that is mispredicting will not be able to pollute the cache of pages more than a certain amount. The limit can be set as a proportion of the size of the cache of pages.

4.6 Experimental Evaluation

4.6.1 Experimental Framework

The goal of our evaluation is to show that the information generated by PATH is indeed useful for the memory management algorithms discussed in this chapter. Towards this goal, we used a trace-based simulation approach for two reasons. First, the information generated by PATH is not directly available in any of today's processors. One solution would be to implement PATH functionality in a cycle-accurate simulator. The problem

with this approach is that cycle-accurate simulation is extremely slow, especially for the type of memory-intensive applications we are considering.

Secondly, fully implementing all of the algorithms discussed in this chapter in a real operating system would require substantial changes to the operating system kernel. Moreover, many implementation-specific issues that are not necessarily related to memory management may interfere. For instance, prefetching from swap space cannot be effective unless the layout of the swap space is dynamically re-organized in order to minimize the number of disk head seeks. Otherwise, no matter how accurate prefetching is, performance will be completely determined by the disk I/O subsystem. Our investigation of the swap space implementation in the Linux kernel showed that swap space becomes quickly fragmented under most workloads we examined. As a result, only a very small fraction of available I/O bandwidth can be utilized for prefetching. Dealing with all such issues is simply beyond the scope of this evaluation, which is to simply show that PATH-generated information is useful.

Therefore, to measure the execution time of applications, we ran all workloads individually on a real system with an AMD Athlon 1.5GHz processor, and timed their execution with their entire working set size fitting in memory so that no page faults occur. We estimate *projected execution time* given the page fault rate determined by our simulation experiments. We use Bochs [Boc], a widely used full-system functional simulator for the IA-32 architecture, to run the applications and record their memory accesses. The memory trace generated by the machine simulator is fed into a memory manager simulator that simulates the memory-management algorithms in a multi-programmed environment.

The projected execution time is calculated using the following formula:

$$Projected\_Exec\_Time = Exec\_Time_0 + Wait_{PF}$$
$$Wait_{PF} = Average\_Latency_{Page\_Fault} * Total\_Page\_Faults$$

where $Exec\_Time_0$ is the execution time measured when no page fault occurs. We assume that once a process faults on a page, it will be blocked for $Average\_Latency_{Page\_Fault}$ cycles; we use a fixed value of one million CPU cycles for $Average\_Latency_{Page\_Fault}$. This value conservatively underestimates the cost of page faults as the average disk access

latency of even fast disks is in the order of a few milliseconds.

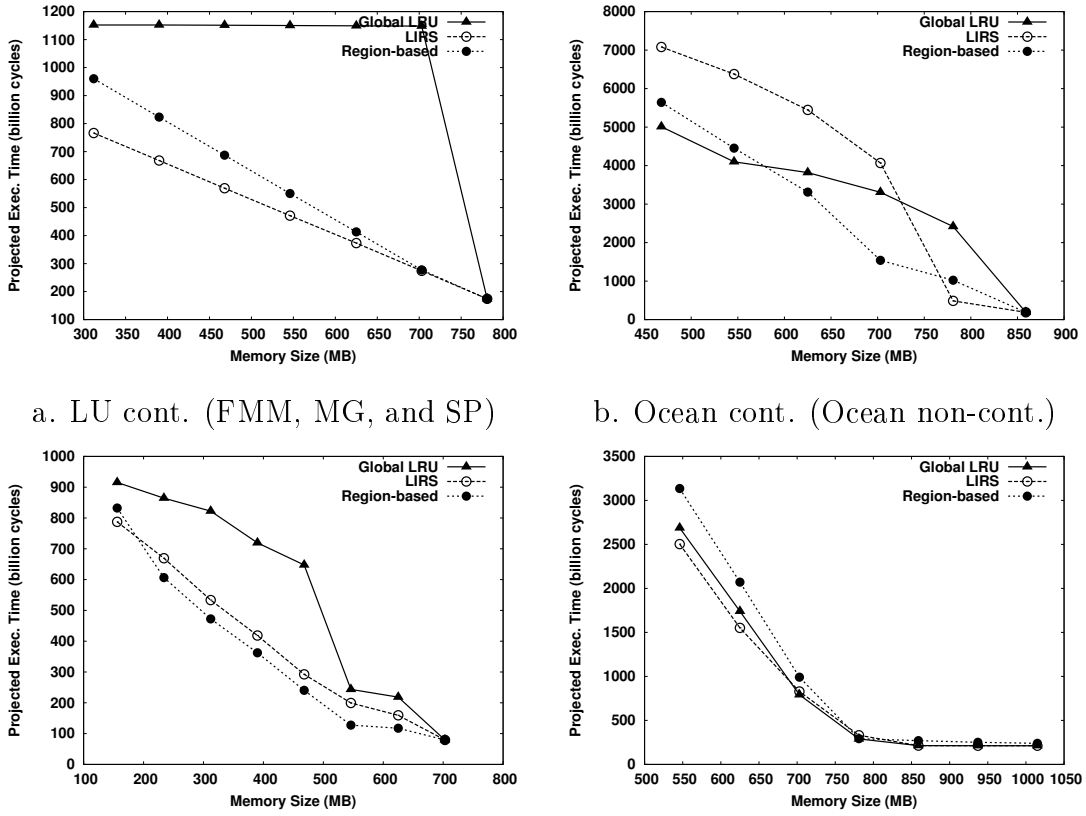
Moreover, we optimistically assume that I/O bandwidth is not a bottleneck; i.e., we assume that saturation of I/O channel capacity will not delay execution. However, we measured the potential impact of the algorithms on required I/O bandwidth.

We added a TLB simulator to Bochs so it could gather TLB misses generated by applications. In our experiments we set the TLB size to 128 entries and its associativity to 16. Although Bochs simulates the entire software stack (i.e., user programs as well as the operating system kernel), we record only user-level TLB misses. A memory trace is essentially a series of page accesses that are time–stamped by the number of instructions completed by an application since the last TLB miss. In order to record modification of pages by the applications, a memory write instruction that hits on a non-dirty TLB entry is considered to be a TLB *write miss*, and is also recorded into the trace. We slightly modified the Linux kernel version 2.6.10 to inform the machine simulator of any process `fork`, `exit`, context-switch, or page-fault events. Moreover, all `mmap` related system calls are relayed to the simulator. Having this information enables us to isolate the exact sequence of virtual addresses each process has accessed or modified throughout its execution.

4.6.2 Applications

Table 4.1 shows the set of memory-consuming applications we use from various benchmark suites: six applications from the Splash-2 suite [WOT+95], four from the NAS Parallel Benchmark (NPB) suite [NAS], SPECjbb2000 [Sta], MMCubing from the Illimine data mining suite [Ill], and MrBayes, a Bayesian inference engine for phylogeny [MrB]. We did not include SPEC CPU benchmarks, as they have fairly small memory footprints. Also, we did not include database benchmarks, primarily because database servers usually exploit their complete knowledge of accessed pages to optimize the replacement policies more effectively inside the server.

We ran the applications with large problem sizes within the practical limits of the simulation environment. However, all of these applications could consume tens of gigabytes of memory for large but still realistic problem sizes. For our experiments, we



a. LU cont. (FMM, MG, and SP)

b. Ocean cont. (Ocean non-cont.)

c. BT (FFT and MrBayes)

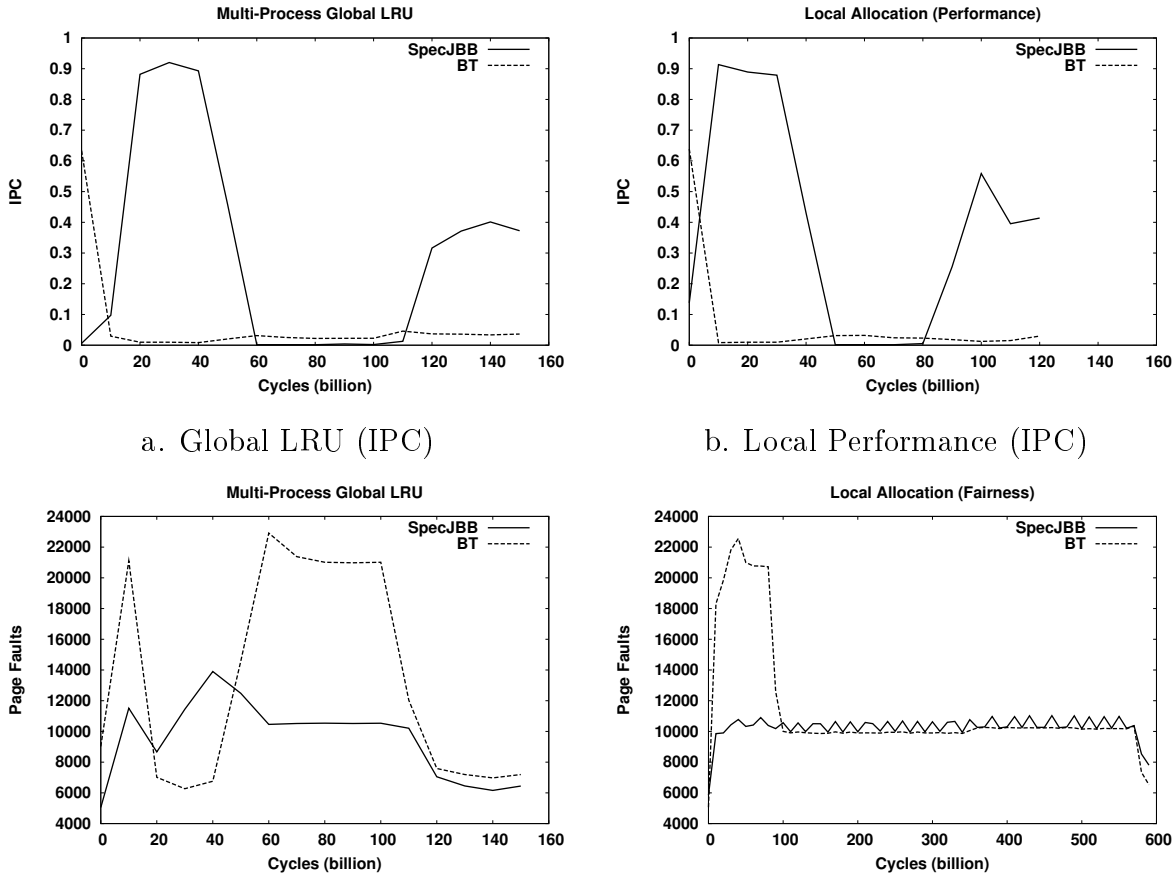
d. SPEC JBB (LU non-cont. and CG)

Figure 4.8: Projected execution time of selected applications with different replacement policies. The applications listed within parentheses have similar behavior.

collected memory traces that cover the execution of a few hundred billion instructions for each application. A *warm up* time is considered at the beginning of the simulation in which no measurement is done. The length of the warm up time is observed by each application's initialization time. Note, however, that we did not execute applications to completion.

4.6.3 Analysis of Adaptive Replacement Policies

Figure 4.8 shows the effect of using different replacement policies on application execution time as memory size is varied. The figure shows the results for a set of four applications with representative behavior. For most of the applications, using one of the adaptive policies (i.e., LIRS or region-based) resulted in a significant improvement in the projected



a. Global LRU (IPC)

b. Local Performance (IPC)

c. Global LRU (Faults)

d. Local Fairness (Faults)

Figure 4.9: Global and Local Allocation policy in multi-programmed scenario: SpecJBB and BT. In (a) and (b), the performance of the two algorithms is shown, while the goal is to maximize overall system throughput (in terms of IPC). In (c) and (d), the fairness of the two algorithms is shown, while the goal is to reach the same page-fault rate for both applications.

execution time (e.g., around 500% for `LU cont.`). Comparing region-specific and LIRS policies, in some cases one performs slightly better than the other and vice versa, but generally their difference is not significant. There are also rare cases in which one of the adaptive policies performs slightly worse than the basic LRU algorithm (e.g., `Ocean` for LIRS and `SPECJbb` for region-specific).

4.6.4 Analysis of Local Memory Allocation

To demonstrate the benefits of fine-grained memory access pattern information for local (per-process) memory allocation schemes, we have designed two experiments. In the first experiment, we show that total system throughput (in terms of Instructions Per Cycle) can be improved over a traditional global allocation scheme. The second experiment demonstrates that in a system under memory pressure, it is possible to obtain fairness, in terms of page-fault rate, through memory isolation.

In all setups, two applications are running simultaneously: `SPECJbb` and `BT`. Without loss of generality, in order to make the experiment more clear, we assumed that the IPC of both applications is 1 when running in isolation. Also, as mentioned earlier, each page fault is considered to have fixed latency of one million cycles. We used a warm-up time of 30 billion instructions and a running time of 60 billion instructions combined.

Figure 4.9 (a) shows the average IPC for both applications when run with global allocation mode; graph (b) shows the average IPC when the applications run with local memory allocation, set to maximize throughput. The trend in IPC is similar for both setups; however, our local allocation policy achieves higher overall IPC in that the number of cycles needed to execute 60 billion instructions with local memory allocation is about 18% lower than that is required with global memory allocation (145 vs. 178 billion cycles). This is mainly because `SPECJbb` has a higher benefit from getting extra pages than `BT` while a global memory allocation scheme considers the utility of each page the same for both applications.

Graphs (c) and (d) of Figure 4.9 show the page-fault rate of the same two applications running with global and local allocation policies, respectively. For the local allocation policy, however, we have configured the policy to maintain page-fault fairness, as described in Section 4.4. Although the local allocation policy configured for fairness takes much longer to complete, it is quite visible that the page-fault rate each application is suffering is similar, therefore successfully reaching its objective.

4.6.5 Analysis of Prefetching

In a set of experiments, we have compared the temporal and spatial locality-based prefetching algorithms. Figure 4.10 shows their effect on both page-fault rate and required I/O bandwidth for a set of selected applications. The rest of the applications we examined perform similarly to one of the applications shown in the figure, and are listed in parenthesis in the figure. For each application two graphs are shown. The graphs on the left show how the page-fault rate is affected as a result of prefetching. The graphs on the right side show the impact of prefetching on I/O bandwidth both for page-in and page-out operations.

For the spatial locality-based policy, we set the initial prefetching window w to 64 which can dynamically grow depending on achieved precision. For the temporal locality-based policy, we set the size of the proximity set for each page to 10 and the scan window size W_{scan} to 64 pages. The depth of the breadth–first traversal in the PPG graph was limited to 3. Finally, for both algorithms we set the size of the pool of pages that are prefetched but not accessed yet to be at most 10% of the physical memory.

In our experiments, we assumed unlimited I/O bandwidth and that the only source of stall is I/O latency. This means that once a set of pages are designated to be prefetched (at most 64 pages), they are assumed to be available in memory within a constant delay time. Furthermore, we have not taken the effect of disk positional delays into account.

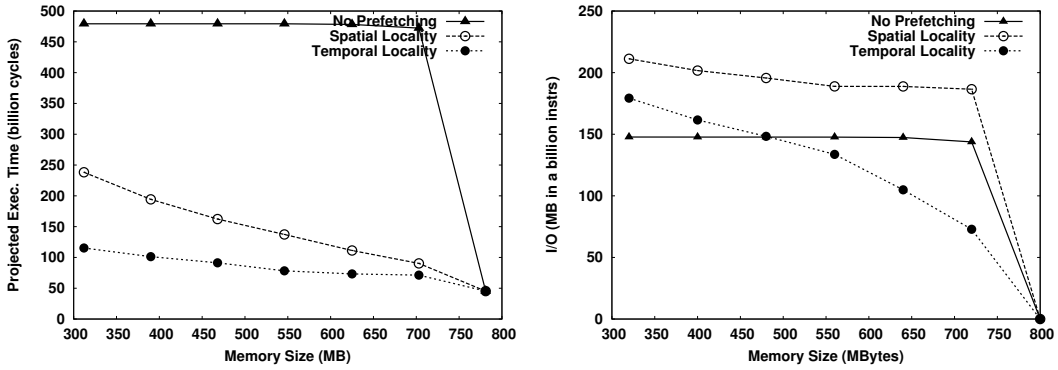
For many applications, such as `MG` and `FFT`, the spatial locality-based policy is quite effective both in terms of recall and precision. Our temporal locality-based algorithm that monitors the sequence of the accessed pages is also able to detect regularity in the access pattern with similar effectiveness. There are applications, such as `LU non-contiguous`, for which the temporal locality-based algorithm significantly outperforms the spatial locality-based algorithm, both in terms of reducing the page-faults and precision. The effect of prefetching on I/O bandwidth for `LU non-contiguous` is remarkable in the sense that prefetching manages to prevent pages in the prefetched set from being replaced by artificially touching them. As a result, the required I/O bandwidth with prefetching is lower than that required without prefetching for some memory sizes. Finally, for some

applications, such as `SPECJbb`, neither of the prefetching algorithms is effective. This can indicate that more application-level information is required to predict next accesses. For instance, Demke-Brown *et al.* shows effective use of compiler analysis to generate accurate prefetching hints automatically [BMK01].

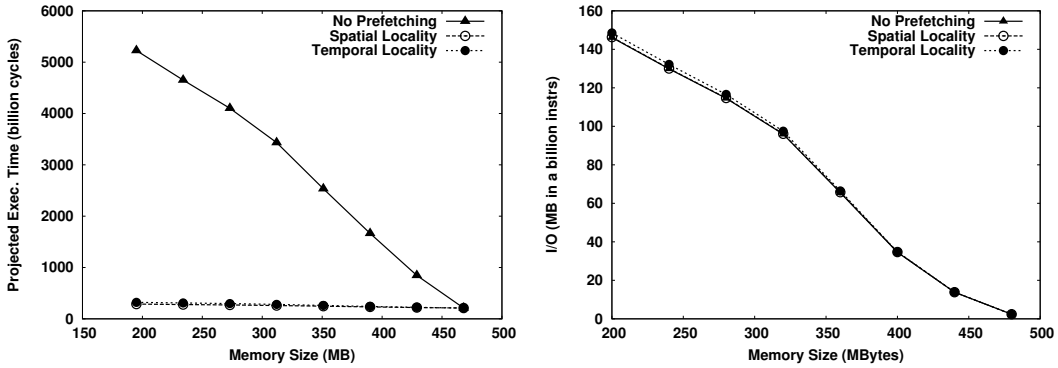
4.6.6 Effect of PAB Size

Figures 4.11 and 4.12 show the effect of different PAB sizes on the projected execution time and runtime overhead for both the page replacement and prefetching algorithms for some of the applications that benefit from fine-grained page access information. Recall that the PAB absorbs the accesses to hot pages and prevents them from appearing in the page access trace. In these experiments, we vary the PAB size from 128 to 32K entries, So that the PAB will span from 512KB to 128MB respectively. As the PAB size increases, we expect that an increased number of page accesses to be filtered by PATH and thus the page access information generated becomes less accurate. At the same time, we expect processing overhead to decrease as fewer page accesses are recorded.

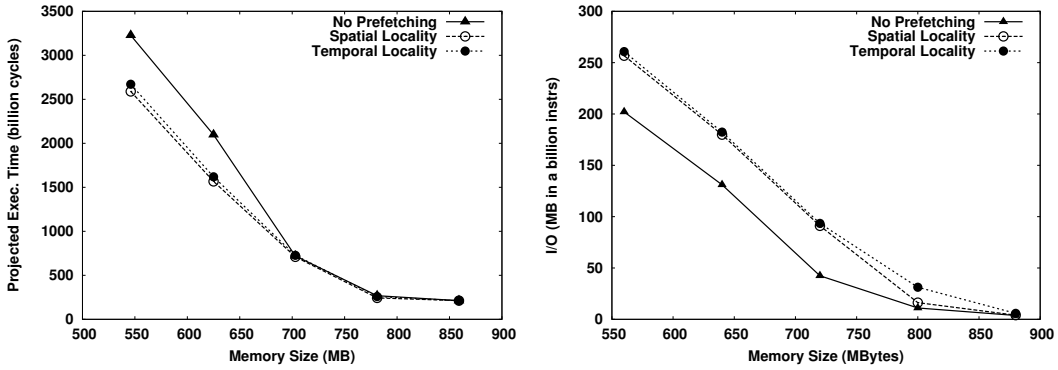
As we see in the graphs, runtime overhead drops significantly as PAB size increases. At the same time, the projected execution time does not seem to be varying much as the PAB size is increased from 128 to 2K entries. One exception is `FFT` with LIRS (shown in Figure 4.2). Overall, it appears that a 2K-entry PAB represents a good tradeoff between overhead and accuracy.



a. LU Non-contiguous Partitions (MMCubing, and MrBayes)

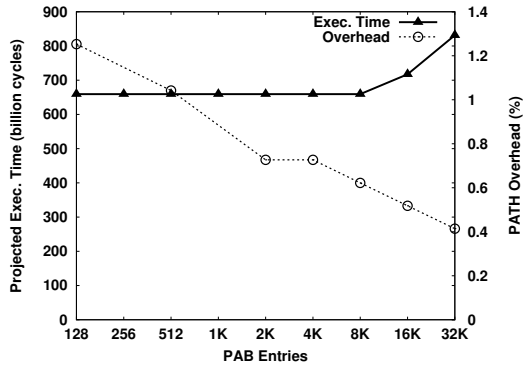


b. MG (SP, LU cont., BT, FFT, and Ocean)

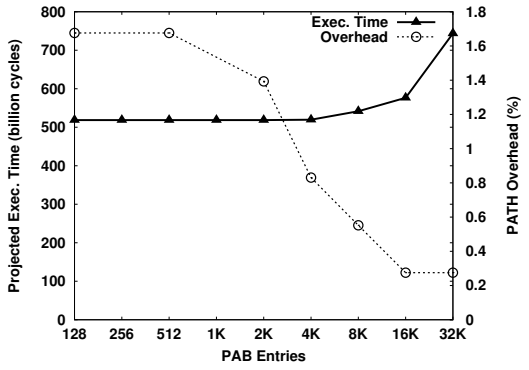


c. SPEC JBB (FMM)

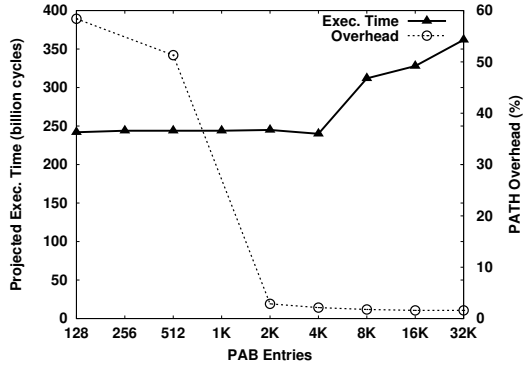
Figure 4.10: The effect of prefetching both on page-fault rate and on required I/O bandwidth. In parenthesis are applications that exhibit similar behaviour.



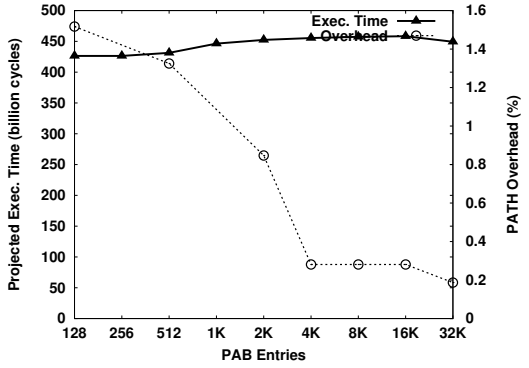
(a) LU Contiguous LIRS (575 MB)



(b) Ocean Contiguous LIRS (780 MB)

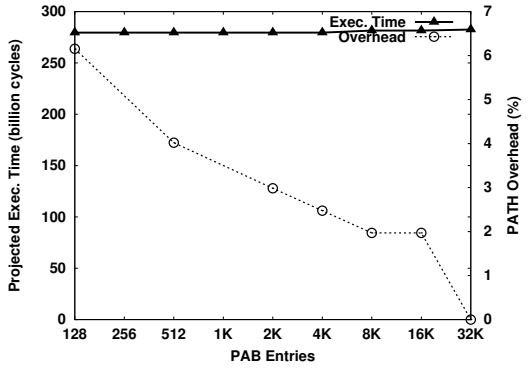


(c) FFT Region-Specific Repl. (576 MB)

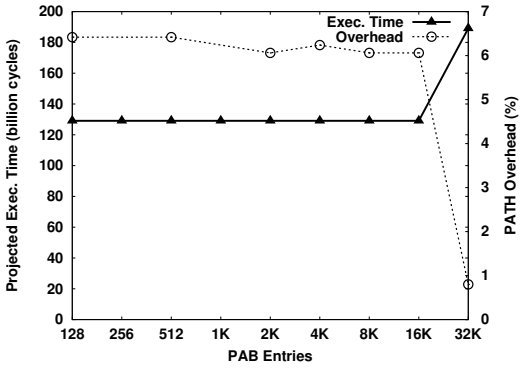


(d) BT Region-Specific Repl. (515 MB)

Figure 4.11: The effect of PAB size on the projected execution time and runtime overhead for page replacement algorithms.



(a) MMCubing Prefetching (234 MB)



(b) LU Non Contiguous Prefetching (312 MB)

Figure 4.12: The effect of PAB size on the projected execution time and runtime overhead for prefetching algorithms.

4.6.7 Analysis of Overhead

In this section, we compare PATH's runtime overhead to a software-only approach. To measure PATH's basic overhead, we *emulated* exceptions generated by PATH in a real environment using an 1.5GHz AMD Athlon processor. For each application, we collected a trace of PAL overflow exceptions along with the content of the PAL at the time of exception. Each overflow event is time-stamped using the number of instructions retired since the start of the application. We then *replayed* these traces by artificially generating exceptions at the same rate as in the trace by using hardware performance counter overflow exceptions. At each exception, we read the contents of the PAL from the trace and updated the LRU stack and MRC data structures. To calculate the overhead, we measure the total number of CPU cycles needed to execute a certain number of application instructions (e.g. a few tens of billions), with and without PATH exceptions.

The software-only approach is implemented in Linux-2.6.15. We measure only the cost of maintaining the active set which includes the cost of extra page protection faults, page table walks to set the protection bits, flushing the corresponding TLB entries, and occasionally trimming the active set using CLOCK.

Figure 4.13 shows the runtime overhead of both PATH and the software-only approach across the selected set of applications, as a function of active set size (PAB size in PATH). There are a number of observations. First, the overhead of the software-only approach is quite high (up to more than 200% of the base execution time) for a number of applications (e.g., `FFT, LU-nonc., MMCubing` and `SPECJbb`) even with a fairly large active set size. Second, the runtime overhead of PATH is very small in all applications if a large PAB (e.g., 32K) is used. For the target 2K PAB size, the overhead of PATH remains less than 3% in all but two applications (`LU-nonc., and SPECJbb` for both of which the overhead is less than 6%). The relatively small overhead is easily offset by the substantial performance improvement achieved by the PATH-generated information when the system is under memory pressure. Note that the OS can turn PATH off when the system is not under memory pressure, and as a result there will not be any unwanted runtime overhead.

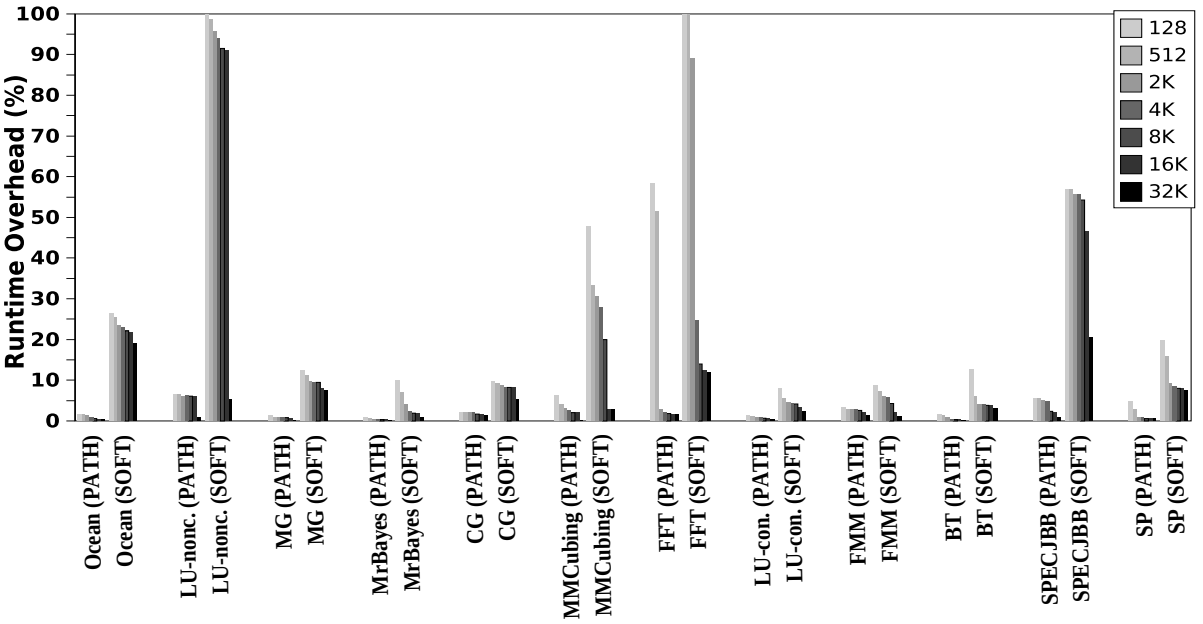


Figure 4.13: Runtime overhead of PATH-generated information compared to the software-only approach (SOFT). To help visualize the comparison, all runtime overhead numbers larger than 100% are truncated.

It is important to note that our approach for measuring the overhead is pessimistic as we ensure the programs' working sets fit in memory and no page faults occur during the course of our measurement. In practice, however, much of the processing of PATH-generated information can be overlapped with potentially long I/O operations caused by page faults.

4.7 Related Work

Zhou *et al.* suggest the use of a custom-designed hardware monitor on the memory bus to efficiently calculate MRC online [ZPS+04]. In their approach, much of the overhead of computing MRC can be avoided by offloading to hardware almost completely. In contrast, we argue in favor of having a simpler hardware that provides lower-level, but more generic, information about page accesses that can be used to solve many problems including the memory allocation problem. We have shown that with the use of fine-grained page access information, the operating system can make better decisions on at least three different

problems. In terms of hardware resources required, the data structures in PATH are simpler and smaller, and unlike the MRC monitor in Zhou *et al.*'s approach, do not grow proportionally with the size of system physical memory.

Cooperative Robust Automatic Memory Management(CRAMM) collects detailed memory reference information to be used to adjust the heap size of a Java virtual machine dynamically in order to prevent a severe performance drop during garbage collection due to paging [YBKM06]. The authors have used the software-only approach to track MRC in order to predict memory usage and adjust the JVM heap size accordingly. To reduce overhead, CRAMM dynamically adjusts the size of the active set by monitoring runtime overhead. Such an approach is presumably effective in tracking MRC for JVM's heap size. However, our results show that for many memory intensive applications, increasing the size of the active set will result in significant performance degradation of memory management algorithms.

Tracking memory accesses at the hardware level has been suggested by other researchers, although to address different problems. For instance, Qureshi et al. suggested the use of hardware *utility monitors* to monitor memory accesses solely to compute MRC at the granularity of individual CPU cache lines [QP06]. Their hardware uses the computed curves to dynamically partition shared L2 caches to improve performance or enforce prioritization.

4.8 Concluding Remarks

Traditionally, operating systems track application memory accesses either by monitoring page faults or by periodically scanning page table entries. With this approach, important information on the reuse distance and temporal proximity of virtual page accesses that can be used for improving memory management algorithms is lost. Previous work has suggested the use of a purely software-based approach that uses virtual page protection to track page accesses more accurately. While this software-based approach is effective for some applications, for many applications it incurs unacceptably high overhead.

In this chapter, we proposed novel Page Access Tracking Hardware (PATH) that

records page access sequences in a relatively accurate, yet efficient way. In terms of structure and function, PATH is simple and easy to implement. In terms of hardware resources required, PATH's structures are fairly small (e.g., around 10KB in size in total) and, unlike previously proposed hardware mechanisms for page access tracking, they do not grow proportionally with the size of physical memory.

We explored several algorithms in the operating system that can exploit the information provided by PATH to improve memory management in three different areas: (i) to implement more adaptive adaptive page replacement policies, (ii) to make smart decision in allocating memory to concurrently running processes, and (iii) to guide the prefetching of pages from virtual memory swap space. Our experimental analysis showed that with PATH, significant performance improvements (e.g., as high as 500% in some cases) can be achieved for applications, especially when systems are under memory pressure. Unlike software-only approaches for tracking fine-grained page access information, the runtime overhead of PATH remains small (i.e., in the 3%-6% range) across a wide range of memory-intensive applications.

Further work is still required in evaluating the effectiveness of information generated by PATH with a more diverse set of applications. Moreover, to ensure scalability of PATH for very large memory setups, more experiments with larger application problem sizes must be conducted.

Another important extension is to explore the use of PATH in a multiprocessor setup. There are important open issues, such as how to collectively use PATH traces of parallel applications that are generated on multiple processors. Similarly, work needs to be done in perfecting PATH support for multithreaded applications. Currently, the PATH trace generated for an application running on a CPU is processed into a single LRU stack or the Page Proximity Graph. If the application is multithreaded, this approach results in intermingling traces of several threads into a single aggregate data structure. As a result, important information about both reuse distance and temporal proximity of page accesses on a per thread basis is lost. To solve this problem, simple extensions can be made to the software layer to keep track of multiple LRU stacks on a per thread basis.

We believe that additional uses of information provided by PATH will become appar-

ent over time, as we experiment with a wider variety of memory intensive applications. Two possible ideas are super page management and page placement in a NUMA architecture.

Finally, we have observed that steps have already been taken by the hardware performance monitoring community to facilitate integration of PATH into real hardware. For instance, the idea of adding a generic trace buffer to the PMU of next generation CPUs seems to have attracted attention [Mer06, Cal06]. One can easily envision adding modest-sized filters, such as those in PATH (or to use a second level TLB for this purpose), to the existing hardware substrate to support accurate capture of the page accesses sequences, as proposed in this chapter.

Chapter 5

Concluding Remarks

Over the past several decades, microprocessor architectures have evolved to increasingly provide system software with information for implementing new functionality or for improving the performance of application and operating system code. This evolution is partially accelerated by the increasing abundance of silicon in modern microprocessors, which enables embedding new hardware features other than those that are directly required for executing code.

In this dissertation, we explored hardware performance monitoring features of today's microprocessors and we explored software techniques for exploiting these features at the operating system level to improve software performance.

At a high level, our approach, has been to try to utilize, as much as possible, existing microprocessor performance monitoring features for the purpose of performance analysis and optimization. If the information required for specific performance optimization techniques was not provided through existing hardware performance monitoring features, or was too costly to obtain, we proposed minimal extra hardware support.

We based our research and experimentation primarily on existing hardware, and default to simulation only when we explore newly propose hardware support. This approach has several advantages. First, it allows us to observe hardware-software interaction scenarios in a real environment, taking into account all complexities of real systems. Secondly, using real hardware allows us to run long-running experiments at real-time speed which is several orders of magnitude faster than a detailed system simulator. Finally,

having explored existing hardware in great detail provides us with the insight to propose new hardware support more realistically, and to the minimal extent needed.

To conclude this thesis, we first provide a brief summary of our work and the major contributions of this thesis. We then provide directions for future research in improving the effectiveness and utility of hardware performance monitoring and how the operating system can benefit from such improvements.

5.1 Summary

We first present a summary of our research effort on different areas of hardware performance monitoring. We then enumerate specific research contribution our research has made.

5.1.1 CPU Bottleneck Analysis

We explored the problem of accurately and efficiently identifying CPU bottlenecks by using Hardware Performance Counters (HPCs). Towards achieving this goal we faced two challenges. First, too few HPCs are available in microprocessors today. Secondly, there has to be a simple and efficient performance model with which CPU bottlenecks can be defined and quantified. We addressed the first challenge by applying low-level HPC multiplexing to make a large set of *logical* HPCs available. We addressed the second challenge by characterizing a simple, but powerful, performance model, called stall breakdown, to identify those processor components that are stressed most. Our model focuses on cycles where the instruction completion stops. We show that such cycles are responsible for most of the difference between the ideal and real throughput of today's CPU. To generate stall breakdown online, we used IBM POWER5 and PowerPC970 hardware performance monitoring features to speculatively associate stalls to the CPU components that are likely to have caused them. By using our HPC multiplexing engine, we build the stall breakdown model online with negligible runtime overhead.

5.1.2 Hardware Data Sampling

We explored different methods of fine-grained data sampling at the hardware level, having recognized that precise information on the data access patterns of applications is required for implementing many performance optimizations. Accurately analyzing application data access patterns is particularly important because of the widening gap between CPU and memory speed causing most CPU cycles to be spent waiting for long-latency memory modules to provide data.

We found existing hardware data sampling techniques to have major limitations, making them only partially useful. For instance, the source from which data is fetched is not directly identified by any of the existing microprocessor performance monitoring units. However, using IBM POWER5's continuous data sampling features, we were able to implement a technique to sample data based on source indirectly. Moreover, we were able to sample data based on multiple selection criteria simultaneously by using our HPC multiplexing engine. In a case study, we showed how to use source-based data sampling to accurately characterize data sharing patterns among concurrent threads in a multiprocessor environment. We further showed how to use this characterization of sharing among threads to cluster them into groups of threads that actively share data.

5.1.3 Page Access Tracking Hardware

To improve the performance of memory management, we proposed simple hardware capable of tracking memory accesses at the granularity. Our proposal was based on the observation that the existing data sampling methods have inherent limitations. First, it is difficult to find the reuse distance of a particular memory address, and secondly, it is not possible to precisely identify sets of pages that are accessed together.

Our proposed hardware is simple and scalable, and it is generic in that it produces a raw trace of memory accesses from which the most frequently accesses pages are automatically removed by the hardware. We used our proposed page access tracking hardware (PATH) in efficiently constructing precise LRU stack and Miss Rate Curves (MRCs) for virtual pages. We further showed the use of these data structures in implement-

ing algorithms for three different areas of memory management. In all three cases we showed, through simulation, that significant performance improvement can be achieved with negligible software overhead.

5.1.4 Summary of Contributions

Our research has resulted in the following specific contributions:

- Our techniques and in particular the proposed architecture for HPC multiplexing with the sampling engine based in the operating system kernel, allows for sampling at a finer granularity and more efficiently than previously possible. Moreover, through the use of fine-grained HPC multiplexing we were able to make a larger set of logical HPCs available.

- We developed the Stall Breakdown model that assists in identifying the most stressed components of the microprocessor. The key insight in developing this simple model was focusing on non-completion CPU cycles, as opposed to focusing on individual stages in the processor pipeline. Using IBM POWER5 facilities, we were able to generate stall breakdown information online with negligible overhead.

- We identified a novel technique to sample data cache misses based on the source from which they are served. WE demonstrated the value of this type of data sampling by efficiently constructing sharing signatures for concurrent threads to support sharing-aware schedulers.

- We proposed a novel page access tracking hardware (PATH) that has negligible overhead and high precision, and we showed how to use this hardware support to improve memory management in three different areas: (i) adaptive page replacement policies, (ii) process memory allocation, and (iii) virtual memory prefetching.

5.2 Future Directions

The architecture of the Performance Monitoring Unit (PMU) has dramatically evolved over the last decade. Processor architects have started to devote additional resources to provide more precise and diverse functionality in the microprocessor PMU. Today, in almost every major microprocessor, a large set of different hardware events can be monitored. Furthermore, there have been major enhancements in techniques of closely monitoring individual instructions as they flow through the pipeline to allow pinpointing exact root causes of performance problems. Finally, several processor PMUs have introduced new data structures, such as trace buffers. These data structures greatly increase the power of PMUs, which traditionally have been composed of only a set of counters.

Despite the fact that PMUs can be found in most today's microprocessors, their features are not widely exploited by software developers and thus, PMUs have remained "second class citizens" [Cal06]. On the one hand, the software community often finds PMU features inadequate or complex to use. On the other hand, the hardware architecture community is not willing to adopt new PMU features unless their utility is clearly demonstrated. We believe that, in order to further motivate the evolution of PMUs, the software community will need to provide more concrete cases of real performance improvements (or reduction in energy consumption) that are only made possible by using accurate PMU-generated information. Moreover, we believe our approach of using the existing PMU as much as possible and proposing only minimal extra hardware, whenever necessary encourages further enhancements in the architecture of next generation PMUs.

A key reason why PMU features are not widely used for software-level optimizations may be due to the fact that specific PMU features required are available only on a particular architecture. Even when the required features are available across several architectures, it is often a non-trivial task to exploit these features because of substantial differences in the user-interface, terminology, and semantics of hardware events across different processor PMUs. We believe that, in order to resolve this issue, PMU features should, at least partially, be standardized. The process of standardization may involve making a clear distinction between (i) hardware implementation-specific features intended

primarily for processor architects to aid in the debugging and performance tuning of several revisions of a processor family, and (ii) higher-level, implementation-independent features to aid software designers in understanding and improving the performance of their software.

Towards standardization of PMU features to aid software optimization, we believe an extended stall-breakdown model, which focuses on precisely measuring the penalty of *miss events* is a useful feature that should be implemented with a similar interface across all processor families with similar semantics. The key underlying feature is the ability to attribute CPU cycles wasted as a result of a miss event to processor components, program instructions, and affected data addresses that are involved in the miss event.

Additionally, the hardware performance monitoring infrastructure should be extended to *all* components in of the computer system, not only the CPU. Components of interest include the memory bus (or any other type of memory interconnect in a NUMA system), the processor interconnect, the I/O interconnect, and the individual I/O devices such as network interface, graphical processing unit (GPU), and hard disks. Having accurate information on the performance of all these components will enable the system software to have a more complete view of system performance and its potential bottlenecks. For instance, Antonopoulos *et al.* demonstrate the concrete possibility of memory bus bandwidth limitation to become a performance bottleneck for highly optimized parallel applications, and how a *bandwidth conscious* CPU-scheduler can utilize memory bus bandwidth information to avoid this bottleneck [ANP03]. There are also clear indications that the computer hardware industry has acknowledged the importance of the system-wide hardware performance monitoring and is taking meaningful steps towards it [Kei06, Kag06, NZ].

Finally, with the widespread revival of virtualization technology, we believe an important future challenge for hardware performance monitoring facilities is to provide proper support for virtualized environments. As virtual machines have become increasingly popular, they introduce new questions on how the hardware can monitor the system to find performance bottlenecks of a virtual machine running on a physical machine shared by many other virtual machine instances. When a virtual machine is scheduled to run, it

inherits the *residual state* of virtual machines running previously on the same CPU. This effect introduces additional noise to the performance measurements done through the PMU. Also, concurrently running virtual machines interfere with each other on shared resources such shared on-chip caches, memory bus, and I/O interconnect fabric.

Bibliography

[ABD+97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium of Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997.

[ACD+96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[AMD] AMD. AMD64 architecture programmer's manual volume 2: System programming rev 3.11.

[AMD02] AMD. Athlon Processor X86 code optimization guide. *AMD Inc.*, pages 235–243, 2002.

[ANP03] Christos Antonopoulos, Dimitrios S. Nikolopoulos, and Theodore S. Papatheodorou. Schedulding algorithms with bus bandwidth considerations for smps. In *International Conference on Parallel Processing (ICPP)*, Taiwan, October 2003.

[App] Apple Computer Inc. Computer Hardware Understanding Development (CHUD) tools. *http://developer.apple.com/tools/performance/*.

[AV02] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of Conference on Supercomputing*, Baltimore, MD, USA, November 2002.

[BH] Bryan Buck and Jeffrey Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Proc. of the International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[BH04] Erik Berg and Erik Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software ISPASS*, Austin, TX, USA, March 2004.

[BH05] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2005.

[BM04] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, March 2004.

[BMK01] Angela Demke Brown, Todd C. Mowry, and Orran Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, 19(2), 2001.

[Boc] Bochs. The open source IA-32 emulation project. *http://bochs.sourceforge.net/*.

[BRS05] David A. Bader, Usman Roshan, and Alexandros Stamatakis. Computational grand challenges in assembling the tree of life: Problems and solutions. *Proceedings of ACM/IEEE conference on Supercomputing (SC), tutorial session*, November 2005.

[Cal06] Jim Callister. The future of hardware performance monitors. In *Presentation at the 2nd Workshop on Functionality of Hardware Performance Monitors, held at MICRO-39*, Orlando, FL, USA, December 2006.

[CDSW05] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Saint Louis, MS, USA, September 2005.

[CH81] Richard W. Carr and John L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles, (SOSP)*, Pacific Grove, CA, USA, 1981.

[Cha97] John Chapin. A fresh look at memory hierarchy management. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 130, 1997.

[CI06] David Christie and Anoop Iyer. Performance monitoring features in AMD Barcelona. In *Presentation at the Workshop on Functionality of Hardware Performance Monitors, held at MICRO-39*, Orlando, FL, USA, December 2006.

[CMDAN06] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using event-based prediction. In *Proceedings of the 20th ACM International Conference on Supercomputing (ICS)*, pages 157–166, Queensland, Australia, June 2006.

[CNMC00] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference, General Track*, Boston, MA, USA, June 2004.

[CT03] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., 2003.

[DCD03] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of 12th International Conference on Parallel Architecture and Compiliation Techniques (PACT)*, New Orleans, LA, USA, December 2003.

[DHW+] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, Research Triangle Park, NC, USA, December.

[DLM+03] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *Proceedings of Workshop Parallel and Distributed Systems: Testing and Debugging (PATDAT), joint with the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, Niece, France, April 2003.

[EEKS06] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, USA, October 2006.

[FBHN03a] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Slack: Maximizing performance under technological constraints. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, San Diego, CA, USA, June 2003.

[FBHN03b] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, San Diego, CA, USA, December 2003.

[GBH04] Ghris Gniady, Ali R. Butt, and Y. Charlie Hu. Program-counter-base pattern classification in buffer caching. In *Proceedings of the 6th Symposium on Operating System Design and Implementation(OSDI)*, San Francisco, CA, USA, December 2004.

[GC97] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, USA, June 1997.

[HC92] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, USA, October 1992.

[HP03] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Los Altos, CA, 2003.

[IBMa] IBM Corporation. K42 research Operating System. *http://www.research.ibm.com/k42.*

[IBMb] IBM Corporation. The POWER4 processor introduction and tuning guide. *http://www.redbooks.ibm.com/pubs/pdfs/redbooks /sg247041.pdf.*

[IBM06] IBM Corporation. IBM PowerPC 970FX risc microprocessor user's manual. *http://www-3.ibm.com/chips/techlib/techlib.nsf/producs/PowerPC_970_and_970FX_Microprocesso* 2006.

[Ill] Illimine. An open–source data mining toolset. *http://illimine.cs.uiuc.edu/.*

[Inta] Intel Corporation. Intel Itanium 2 reference manual for software development and optimization. *http://www.intel.com/design/itanium2/ manuals/251110.htm*.

[Intb] Intel Corporation. VTune performance analyzers. *http://www.intel.com/software/products/vtune*.

[JCZ05] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the clock replacement. In *Proceedings of the Usenix Technical Conference (USENIX'05)*, Anaheim, CA, USA, April 2005.

[JG99] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[JS94] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB)*, Santiago, Chile, September 1994.

[JZ02] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Performance Evaluation Review*, 30(1), 2002.

[Kag06] Michael Kagan. Infiniband hardware performance monitoring future and visions. In *Presentation at the 2nd Workshop on Functionality of Hardware Performance Monitors, held at MICRO-39*, Orlando, FL, December 2006.

[Kei06] Jeffery Keil. GPU performance analysis: A developer's perspective. In *Presentation at the 2nd Workshop on Functionality of Hardware Performance Monitors, held at MICRO-39*, Orlando, FL, December 2006.

[KHW91] Yul H. Kim, Mark D. Hill, and David A. Wood. Implementing stack simulation for highly-associative memories. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, San Diego, CA, USA, May 1991.

[KMC02] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM)*, Berlin, Germany, June 2002.

[KS04] Tejas Karkhanis and James E. Smith. Modeling superscalar processors. In *Proceedings of the 31th International Symposium on Computer Architecture (ISCA)*, Munchen, Germany, June 2004.

[LA05] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, June 2005.

[LCF+03] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, 2003.

[Lem96] G. Lemieux. Hardware performance monitoring in multiprocessors. Master's thesis, University of Toronto, 1996.

[May01] John M. May. MPX: Software for multiplexing hardware performance counters in multithreaded systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, USA, April 2001.

[MC05] Wiplove Mathur and Jeanine Cook. Improved estimation for software multiplexing of performance counters. In *Proceedings of the 13th Interna-*

tional Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Atlanta, GA, USA, September 2005.

[Mer06] Alex Mericas. IBM Cell hardware performance monitoring and what's hard about multi-threading. In *Presentation at the Workshop on Functionality of Hardware Performance Monitors, held at MICRO-39*, Orlando, FL, December 2006.

[MGST70] R. L. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques and storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, March 2003.

[MMdS05] Jaydeep Marathe, Frank Mueller, and Bronis de Supinski. A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks. In *Proceedings of the 19th International Conference on Supercomputing (ICS'05)*, Cambridge, MA, USA, June 2005.

[MOH96] Margaret Martonosi, David Ofelt, and Mark Heinrich. Integrating performance monitoring and communication in parallel computers. In *In Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1996.

[MrB] MrBayes. Bayesian inference of phylogeny. *http://mrbayes.csit.fsu.edu.*

[MyS] MySQL. Open source database. *http://www.mysql.com.*

[NAS] NASA Advanced Supercomputing. NAS Parallel Benchmarks. *http://www.nas.nasa.gov/Software/NPB/.*

[NZ] Lisa Noordergraaf and Robert Zak. SMP system interconnect instrumentation for performance analysis.

[OPr] OProfile. A system profiler for Linux. *http://oprofile.sourceforge.net/*.

[PCL] PCL. The Performance Counter Library: A common interface to access hardware performance counters on microprocessors. *http://www.fz-juelich.de/zam/PCL/ doc/pcl/pcl.html*.

[QP06] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–432, Washington, DC, USA, 2006.

[RUB] RUBiS. Objectweb open source middleware. *http://rubis.objectweb.org*.

[SDS00] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenstrom. Recency-based TLB preloading. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, 2000.

[SHC+04] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of 3rd Virtual Machine Research and Technology Symposium (VM)*, May 2004.

[SKT+] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner.

[SKW03] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, 2003.

[(SP] Standard Performance Evaluation Corporation (SPEC). Spec cpu2000. *http://www.spec.org/cpu2000*.

[Spr02] Brinkley Sprunt. Pentium 4 performance monitoring features. *IEEE Micro*, 22(4):72–82, July/August 2002.

[Sta] Standard Performance Evaluation Corporation (SPEC). SPECjbb2000. *http://www.spec.org/jbb2000*.

[TAS07] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the second EuroSys Conference (EuroSys'07)*, Lisbon, Portugal, March 2007.

[THa] Mustafa M. Tikir and Jeffrey Hollingsworth. Data centric cache measurement on the Intel Itanium 2 processor.

[THb] Mustafa M. Tikir and Jeffrey Hollingsworth. Using hardware counters to automatically improve memory performance.

[TM94] A. Tamches and B. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Symposium on Programming Languages Design and Implemenation (PLDI)*, Orlando, FL, USA, June 1994.

[TTC02] E.S. Tune, D. M. Tullsen, and B. Calder. Quantifying instruction criticality. In *Proceedings of the 11th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Charlottesvill, VA, USA, September 2002.

[TTG95] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[VMTO05] D. Villa, M. Meswani, P. Teller, and B. Olszewski. Profiling memory subsystem performance in an advanced POWER virtualization environment. In *Proceedings of the Workshop on Operating System Interference on High Performance Applications*, Saint Louis, MS, USA, September 2005.

[Vol] VolanoMark. Volano LLC, San Francisco. *http://www.volano.com/benchmarks.html*.

[Wal02] C. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, December 2002.

[WLLB97] Harvey J. Wassermann, Olaf M. Lubeck, Yong Luo, and Federico Bassetti. Performance evaluation of the SGI Origin2000: a memory-centric characterization of lanl asci applications. In *Proceedings of ACM/IEEE Conference on Supercomputing (SC)*, San Jose, CA, USA, November 1997.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. *SIGARCH Computer Architecture News*, 23(2):24–36, 1995.

[WR03] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proc. of the Supercomputing Conference (SC)*, Phoenix, AZ, USA, November 2003.

[WSS+04] R. W. Wisniewski, P. F. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and environment monitoring for whole system characterization and optimization. In *Proc. of the 2nd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (PAC)*, Yorktown Heights, NY, USA, October 2004.

[YBKM06] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, USA, November 2006.

[ZAKB+05] Yun Zhang, Faisal N. Abu-Khzam, Nicole E. Baldwin, Elissa J. Chesler, Michael A. Langston, and Nagiza F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In

Proceedings of the ACM/IEEE conference on Supercomputing (SC), Washington, WA, USA, November 2005.

[ZLF+04] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proeceedings of the 37th International Symposium on Microarchitecture (MICRO)*, Portland, OG, USA, December 2004.

[ZPS+04] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, USA, November 2004.

[ZQLT04] Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. iWatcher: Efficient architecture support for software debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, Munchen, Germany, June 2004.

[ZvBB05] Feng Zhou, Rob von Behren, and Eric Brewer. AMP: Program context specific buffer caching. In *Proceedings of the USENIX Technical Conference (USENIX'05)*, Anaheim, CA, USA, April 2005.