ABSTRACT

THAKKAR, VIVEK. Dynamic Page Migration on ccNUMA Platforms Guided by Hardware Tracing. (Under the direction of Associate Professor Dr. Frank Mueller).

Non-uniform memory architectures with cache coherence (ccNUMA) are becoming increasingly common, not just for large-scale high performance platforms but also in the context of multi-cores architectures. Under ccNUMA, data placement may influence overall application performance significantly as references resolved locally to a processor/core impose lower latencies than remote ones.

This work develops a novel hardware-assisted dynamic page migration scheme based on automated tracing of the memory references made by application threads. The developed framework leverages the performance monitoring capabilities of contemporary x86 microprocessors to efficiently extract an approximate trace of memory accesses. This information along with multi-level hop latencies are used to decide *page affinity*, *i.e.*, the node to which a page is bound. After determining affinities, page migration is initiated using Linux kernel mechanisms. All this automation is done in user space and transparent to the main application.

Experiments show that this method, although based on lossy tracing and system configuration limitation on trace hardware, can efficiently and effectively improve local data availability at run time, leading to an average wall-clock execution time saving of over 14% on AMD Opterons with a 1.3x/1.6x access penalty to non-local memory with a very minimal page migration overhead due to the advances in modern memory interconnect technologies. To the best of our knowledge, this is a first experimental study on a popular platform, a combination of x86 processors and Linux operating system.

Dynamic Page Migration on ccNUMA Platforms Guided by Hardware Tracing

by
Vivek Thakkar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, NC

2008

APPROVED BY:

DEDICATION

To my parents, grandmother and dear sister: Without their love, support and guidance, this thesis would not have been possible.

BIOGRAPHY

Vivek Thakkar was born on December 31, 1982. He received his Bachelor of Technology in Computer Science from Maharishi Dayanand University, Rohtak, India in May 2004. With the defense of this thesis he will receive his Master of Science in Computer Science from North Carolina State University in August 2008.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Frank Mueller for his guidance and support towards the completion of this thesis. I learnt a lot from him during our discussions and in particular team meetings. I would also like to acknowledge the support of my thesis committee members, Dr. Xiaosong Ma and Dr. Vincent W. Freeh, for agreeing to be on my committe. Thanks are also due to my professors whose courses I undertook during my graduate studies. The knowledge gained from them are a major input to my thesis. I would also like to thank all my friends, folks at the systems research lab and roommates for their support and encouragement. Lastly, I would also like to express my sincere gratitude towards Dr. Frank Mueller and the Computer Science Department of North Carolina State University for giving me financial assistance and an oppotunity for graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

In the world of computer architecture and system software, the focus is on getting maximum performance from the available resources. On the hardware side, traditionally, this was achieved by developing faster cores and better memory technologies. In the field of computer architecture, the focus was to design the processors to exploit as much parallelism as possible from the applications. However, the trend has changed recently. We are reaching a limit on the number of transistors that can be packed on cores, and varieties of architectural improvements have already been tried. Now, the trend is to have multiple cores to provide more hardware contexts for parallel applications. In the field of high-performance computing, parallel machines have been omnipresent for quite some time. MPP (Massively parallel processors) are cluster-based system where each node has an associated private memory and works only on part of the data. These systems scale well but the programs need to be explicitly modified to share data amongst processors by using the send/receive communication paradigm. In addition, we have SMP (Symmetric Multi-Processors) where multiple processors share a common memory over a shared bus (e.g., Intel's Front Side Bus). In these systems, the bus connecting the processor to the memory controller (called front-side bus in Intel's implementation of the x86 architecture) and the memory become a major bottleneck as they only allow very few memory accesses at the same time. To ameliorate this effect, computer design industry has primarily invested on improving the bus technology by increasing bus segments, increasing bus bandwidth and making caches bigger to reduce traffic on the bus. But in spite of the best efforts, SMP scales only to 8 or

10 processors.

An alternative approach is based on NUMA (Non-Uniform memory access) technology which gives programs a shared linear address space but keeps private memory with each node. Shared linear address space implies that each processor is able to access any memory module using linear addresses but each node has a local physical memory shared by some processors (2-4). The data can be cached by any node and the cache coherence is implemented by the point-to-point memory interconnect technology (e.g., AMD's `hypertransport`). These kinds of systems are popularly called cache coherent Non Uniform Memory Access (ccNUMA) systems. Some large-scale ccNUMA systems are the Stanford DASH , the SGI ORIGIN and the SGI Altix whereas AMD's Opterons and Intel's upcoming `Nehalem` and `Tukwila` processors are medium-scale ccNUMA systems.

The shared linear address space provided by these architectures can be efficiently used for data sharing by the applications written with a shared memory programming model. Designing new shared memory paradigms is an active area of research in itself, but two models are mostly used: `Pthreads` and OpenMP. Pthreads is a POSIX standard thread library that includes calls to create and destroy threads, to synchronize threads using mutexes and to facilitate thread communication via condition variables. OpenMP is a higher-level thread library that also performs the same tasks but further includes work-sharing constructs to auto-distribute work amongst threads. Another major difference is that it is directive-based, meaning that even if programs are written serially, OpenMP provides some useful constructs that can be easily used to parallelize them without much effort.

1.2 Motivation

On a non-uniform latency model like NUMA, the data should be as close as possible to the thread that accesses it more often than the others. To maintain this association, research has primarily focused on policy-driven thread allocations (to processors) and data allocations and also on dynamic techniques like thread migration and page (data) migration. Operating systems implement "CPU affinities", a mechanism where threads can run only on a subset of CPUs defined by the "affinity masks". On the data side, library implementations like "numactl" allow the applications to define policies like "first touch" and "interleaved/round-robin" allocation. First touch allocation means that a physical page frame is allocated on the first write to the virtual page containing the data. Interleaved allocation, in contrast, allocates page frames in a round robin manner across the available memory modules to achieve a better memory balance and to avoid memory pressure on few nodes. In the area of thread migration, threads are moved closer to their data by getting some feedback of memory access patterns. But thread migration has its own associated problem: it removes the cache associations of the migrating thread and thus incurs a significant overhead. Finally, in the field of page migration, the focus is to move data as close as possible to the affine threads. In an environment where threads also migrate, data migration may become futile if the affine thread itself migrates after some time. This may require some modifications to the operating system scheduler to provide information of thread migration to either a user-level or kernel-level page migration engine. In any case, the source of page affinity (to threads) needs to be reliably determined to affect data migration. Past research has focused on various hardware and software techniques to determine this information. On the hardware side, some implementations introduce "hardware monitors" as per page counters for local and remote accesses. These can then be used to determine page affinity. On the software side, approximate memory references are determined by instrumenting the software TLB handler of the OS to count TLB misses. These approaches either have high overhead or require costly hardware implementations.

1.3 Thesis Statement

The objective of this thesis is to develop a user-level dynamic page migration engine for HPC applications on medium scale ccNUMA platforms. In the address space of contemporary x86 architectures and server-based Linux kernel operating environments, we investigate the following issues in this thesis.

- Can modern hardware be assisted by the operating system to provide feedback information to improve page placement on ccNUMA architectures?
- Can we use Linux kernel system calls to realize dynamic page migration in user space in an efficient manner?

We assess the above hypothesis in Chapter 6, which concludes the thesis.

1.4 Contributions

The thesis makes the following contributions:

- The work exploits the PEBS PMU in Intel's P4, Xeon and Core microarchitectures to get feedback information on the memory access profile of applications.
- The Perfmon2 interface [10] in the Linux kernel is leveraged through a user-level library (*libpfm*) to program the PMU for L1 and L2 cache miss events.
- The thesis also evaluates the performance of the two system calls for page migration in the Linux kernel.
- The work introduces novel algorithms to determine affinity of a page to a ccNUMA node based on feedback hints from the the PEBS hardware.
- The work compares the hardware-guided page placement and page migration schemes with policy-based page allocation schemes in Linux. Experiments indicate an average improvement of 8-15% in wall-clock time. We believe that limitations of the trace hardware and software and improvements in memory interconnect technologies currently constrain the experiments and that the approach has even more potential once these limitations are overcome in the future.

1.5 Thesis Layout

This thesis is organized as follows. Chapter 2 details the PEBS support in Intel's Netburst and Core microarchitectures, the Linux kernel support for the same and our base design framework. This is initially used to develop a page placement scheme where the hardware traces collected in a short truncated run are used in the subsequent trace-driven run of the program. Chapter 3 builds on the base framework to develop a user-level page migration engine. Chapter 4 presents some experimental and other evaluation results. Finally, Chapter 5 details the related work and Chapter 6 concludes this thesis.

Chapter 2

Page Placement Exploiting PEBS Traces

2.1 Overview

In the prior work [20], the developed approach was evaluated using Itanium2-specific hardware traces for automated page placement. The work carries forward a similar approach using a completely different hardware tracing mechanism and NUMA platform. The basic idea of exploiting *processor-centric* hardware support for user-level page placement is shown to be portable and widely applicable across multiple platforms (irrespective of interconnect topologies).

Instead of the Itanium architecture, the widely used x86 is used as the target platform. The objective of this work is to perform page placement on a ccNUMA multiprocessor Opteron system available from AMD [16]. In this system, each processor directly accesses (using an on-chip memory controller) a fixed amount of local physical memory. Communication with other processors and their attached physical memories is achieved over the point-to-point *HyperTransport* connection network [16]. Systems exploit a bus-based MOESI coherence protocol instead of the directory-based coherence present in SGI's NUMALINK fabric. Processors can access their local memories faster than memories attached to other processors, and the access penalty increases with the number of hops to reach the remote memory (due to point-to-point interconnect). The experiments will assess the benefits of intelligent page placement on this system. Though the results were obtained on the AMD Opteron, the developed scheme should equally work on future ccNUMA systems from Intel that use the CSI (Common System Interconnect)/QuickPath architecture.

The premise of the developed technique is the ability to obtain hardware-generated traces that efficiently drive the page placement policy. Since current AMD Opterons, prior to the quad-core Barcelona chip, have no published hardware trace capability, the performance monitoring unit built into Intel Pentium4/Xeon/Core2 systems is exploited for this purpose. This hardware is called "Precise Event-Based Sampling" (PEBS). PEBS captures the register state when a specific event, *e.g.*, an L1 cache miss, is detected. By decoding the instruction format and using this register state, the memory address that was accessed can be reconstructed. The following sections describe the PEBS mechanism in two different microarchitectures, Intel NetBurst microarchitecture and Intel Core microarchitecture. We then discuss about the Perfmon2 interface in the Linux kernel 2.6. The interface allows the configuration of the hardware Performance Monitoring Unit(PMU) and handles PMU interrupts. It also provides a user library, *libpfm*, to support easy configuration from user space. We then talk about the design and implementation of our framework to support PEBS.

2.2 PEBS support in NetBurst microarchitecture

Processors based on NetBurst microarchitecture (Intel Pentium 4, Intel Xeon) do not support architectural performance monitoring, i.e., the performance monitoring events and capabilities may change across different processor families. These PMUs have certain model-specific registers (MSR) for performance counting, counter configuration control (CCCR), i.e., to set up an associated performance counter for a specific method of counting, event select configuration MSR (ESCR) for selecting events to monitor and other control MSRs. The following type of events can be counted with these facilities:

- **Non Retirement Events**: These events are not bound to instruction retirement, e.g., events related to Branch Predictor Unit (which happen at the fetch stage of the pipeline).

- **At Retirement Events**: These events are counted at the retirement stage of instruction execution. Modern x86 processors support out-of-order execution and speculations like branch prediction, memory dependence predictions etc. These predictions can go wrong and, therefore, not all of the instructions that pass through the execution stage may eventually retire. With at-retirement counting, μops that encounter performance events during instruction execution are tagged. This allows the events that result in committed architectural state to be distinguished from those that lie on an execution path that eventually gets discarded, e.g., because of branch misprediction or cancellation of some speculative state of execution.

Intel's software manual [11] describes the following usage models for performance monitoring.

- **Event counting**: This is very similar to polling at regular intervals. A performance counter can be configured to count various types of events. The value of the counter is read by the software (usually the operating system as most of the counters require a privilege access) at selected intervals to determine the number of events that have been counted between the intervals. Event counting supports both "Non Retirement" and "At Retirement" events.

- **Non-precise event-based sampling**: Like event counting, a performance counter is configured to count one or more types of events. But instead of reading the counter at regular intervals, the PMU is configured to generate an interrupt when it overflows. The counter overflows after a specific number of events are counted, and this number is defined by presetting the counter to a modulus value, which is the width of the counter minus the sampling rate (in units of number of events). An interrupt service routine handles this interrupt and then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. These RIPs can provide very useful information about the execution profile of the program. This can be tracked by various tools like Perfmon, PAPI, VTune etc. Non-precise counting also supports both "Non Retirement" and "At Retirement" events.

- **Precise event-based sampling (PEBS)**: This type of performance monitoring is similar to non-precise event-based sampling, except that a software-designated memory buffer is used to save a record (called PEBS record) of the architectural state of the processor whenever the counter overflows. The information in the buffer is directly written by the processor without the software's help, and an interrupt is generated after some configured numbers of records have been written. This considerably reduces the overhead of performance monitoring. Architectural state of the processor includes the contents of general-purpose registers, instruction pointer, EFLAGS register etc. These facilities can be used for various purposes, e.g., we use them to decode the instructions for load miss data addresses. Precise event-base sampling can be used to count only a subset of at-retirement events.

2.2.1 Debug Store (DS) Mechanism

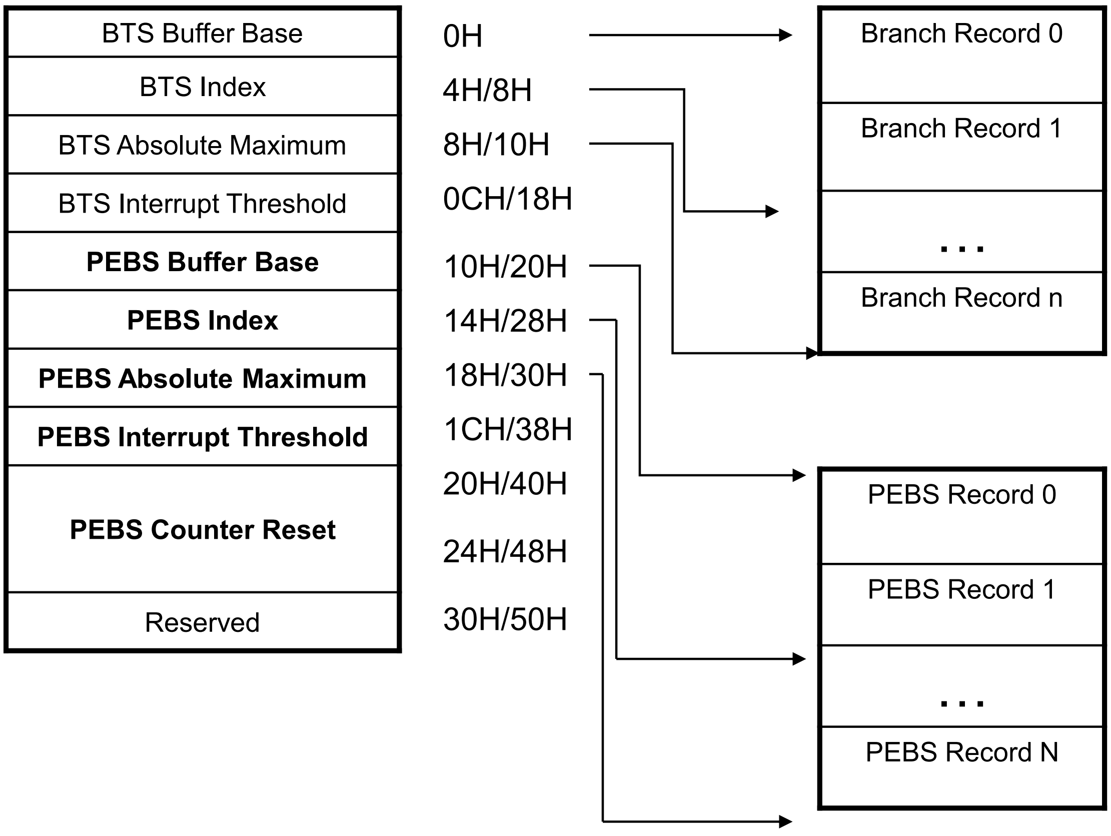


Figure 2.1: DS Save Area

The Debug store (DS) is a non-swappable memory buffer that can be used for storing various information for debugging and tuning programs. Currently, it is used to collect two types of information: branch records and **PEBS sampling records**. Figure 2.1 shows how the DS save area is organized in memory. The DS save area is divided into three parts: the Branch Trace Store (BTS) buffer, the PEBS buffer and the buffer management. Figure 2.1 shows the offset of each entry in the buffer management area for both the NetBurst and Core microarchitectures. Buffers contain the actual information, and the buffer management portion contains the related metadata. As can be seen in Figure 2.1, for the PEBS buffer, the buffer management contains following entries related to PEBS recording:

- The PEBS base pointer: The linear address of the first PEBS record.
- The PEBS index: The linear address of the first byte of the next PEBS record.
- The PEBS interrupt threshold: The linear address (which must be a multiple of PEBS record size), used to generate an interrupt when the counter overflows. It is used to allow the PEBS hardware to safely write PEBS records in the buffer (without causing a buffer overflow) at a time when the interrupt is handled by the software.
- The PEBS counter reset: A value that needs to be written to reset the counter after the PEBS record is written.

As can be seen from Figure 2.2, the PEBS record contains the contents of general-purpose registers, instruction pointer and flags register. For the IA-32, each entry in the PEBS record is 32 bit, and all the register names are prefixed with 'E' , i.e., the registers are named as EFLAGS, EIP, EAX etc. The corresponding registers for the IA-32e architecture are named as RFLAGS, RIP, RAX etc. and are 64 bit wide, which makes each entry of PEBS record 64 bits. The IA-32e has eight additional general-purpose registers (R8..R15). A PEBS record is written in the DS save area by the processor at the next occurence of a PEBS event (after a preconfigured counter overflows).

PEBS uses the "interrupt on overflow" mechanism. The performance counter is initialized with $2^{mcbits} - SI$, where $mcbits$ is the bit-width of the processor. For IA-32, it is 32 and for IA-32e and IA-64 it is 64. SI is the sampling interval. We may not decide to sample each event as it is seen to be lossy and incurs a lot of interrupt overhead. As interrupts can only be triggered on an overflow, the counter should be initialized with a large enough value to enable an interrupt after the configured sampling interval. The operating system kernel then handles this interrupt.

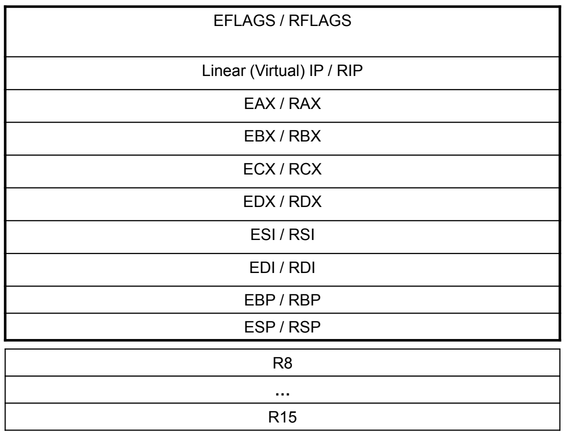


Figure 2.2: PEBS Record Format on P4/Xeon (IA-32) and Core2 Duo (IA-32e)

2.2.2 Event Tagging And Replay

As mentioned before, the NetBurst microarchitecture supports *at-retirement* events. The following terminology is widely used to describe such events:

- **Tagging**: Tagging is a mechanism of labelling a μop that triggers a particular performance event so that it can be counted at retirement. Event counting by itself is not an accurate method because a μop may trigger that event many times during its course of execution, e.g., a speculative load miss is not a true indicator of a cache miss event, only a retired load instruction should increment the "cache miss" counter.

- **Replay**: To fully utilize the various pipeline resources, the Intel NetBurst microarchitecture aggressively schedules μops for execution before all the conditions for correct execution are guaranteed to be satisfied. For example, it may perform load value prediction, which might be used by dependent instructions. Now, if that load misses the cache and its returned value is found to be different from the predicted value, all the prior work must be undone. In other words, μops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of μops is called replay.

The following kinds of tagging mechanisms are described in the Intel's manual [11]:

1. **Front-End Tagging**: This mechanism is used to tag μops that encounter front-end events, i.e., those events that occur during the fetch and decode stages in the

instruction pipeline, e.g., trace-cache and instruction-cache related events.

2. **Execution Tagging**: This mechanism is used to tag μops that encounter execution events (e.g., counting of retirement of special types of instructions).

3. **Replay Tagging**: This mechanism pertains to tagging of μops whose retirement is replayed (e.g., a load μop that causes a cache miss). Events like branch mispredictions are also tagged with this mechanism. *We use this tagging mechanism in our scheme to tag μops for 'L2/L1' load cache miss events.*

4. **No Tags**: This mechanism does not use tags as the events to be counted are directly related to retirement. Instr_retired and μops_ retired events can be used with this method.

Associated with the replay tagging mechanism is an event called replay event. To set up the replay event count, certain metrics are defined. Among them, we need the following two metrics for our work:

a) 1stL_Cache_load_miss_retired: First level cache misses; and

b) 2ndL_Cache_load_miss_retired: L2 cache misses.

On our test systems, we have only two cache levels, L1 and L2. Hence, L2 cache miss events enable us to track loads that miss cache and hit memory. An L1 cache miss is tracked to determine the difference in performance as the L1 cache miss would not necessarily guarantee that address references were memory hits as some data could be cached in L2. Intuitively, L2 cache misses should be a better indicator for tracking page references in the memory, and our results reaffirm this.

2.3 PEBS support in the Core Microarchitecture

Processors based on the Intel Core microarchitecture also support the PEBS mechanism using a DS buffer area. Compared to PEBS support in the NetBurst microarchitecture, the difference is mainly in the way PEBS is setup. The details are depicted in Table 18.16 in Intel's software manual [11]. The other major difference is that the precise state returned by core microarchitecture is the state of execution of the *next* instruction for the instruction that generates the event. This has some minor implications in the design of PEBS framework. The description is mentioned in the design section on PEBS.

The following events in processors based on the Core microarchitecture are useful for us:

1. MEM_LOAD_RETIRED:L1D_MISS; and
2. MEM_LOAD_RETIRED:L2_MISS.

PEBS is extensively detailed in Chapter 18 of Intel's software manual - Volume 3b [11] . The reader is strongly encouraged to refer to that for further details.

2.4 Perfmon2: Linux Kernel support

The Performance Monitoring Unit (PMU) is integrated in the processor cores. A PMU has control and data registers, which are read or written by the software. This requires access at the most privilege level of software, i.e., the operating system kernel. Hence, it is not possible to develop only a user-level library, some kernel support is also required. However, a lot of diversity exists in PMU space. On the hardware side, PMUs can be very different, e.g., some PMUs may be very simple and provide only basic counters whereas others may be very sophisticated and may capture addresses, latencies and branches. On the software side, we have various tools with different requirements and different designs. A tool meant to exploit one PMU may be very difficult to port on a different PMU, not only because architecturally the PMU is different but also because the software design may have limitations. Similarly, some tools may support per-thread monitoring while others support system-wide (all running threads on one processor) monitoring or even monitoring across all processors. Due to this diversity in space, a standard kernel interface is highly desirable.

The Linux kernel, up until now, has had multiple tools and kernel interfaces for various architectures (like the IA-32, x86-64/IA-32e, IA-64, Power-PC etc). Some popular tools and kernel interfaces are:

a) The VTUNE [6] performance analyzer with its own proprietary kernel interface, which is implemented by an open-source device driver;

b) The OProfile [9] interface used by tools such as Prospect [27];

c) The Perfctr [25] interface used by the tools based on the PAPI [28] toolkit.

Having multiple interfaces for one PMU creates issues with code maintenance and coordination between various interfaces, especially when they need shared access to the PMU at the same time. To solve this issue, a standardized Perfmon2 interface has been developed in the kernel. Its goal is to solve the challenge described in the official Perfmon2 manual [10]:

"how to design a generic performance monitoring interface that would provide access to all existing and future PMU implementations and that would also support a variety of monitoring tools? "

The interface provides access to PMU resources but leaves the task of programming it to the users. It also exploits certain key features across all PMU models. These guidelines help Perfmon2 to be a generic kernel interface for a variety of PMUs that is easily portable across various profiling tools.

The following salient features of this interface design are useful for this thesis:

1. **Logical PMU**: The implementation exposes a logical view of the PMU to the monitoring tool (like our framework). Any PMU model would have certain control registers and data registers. Control registers describe what is to be measured and can direct commands to start/stop/mask monitoring. Data registers are used to store the results. These registers are called Performance Monitoring Control (PMC) and Performance Monitoring Data (PMD) registers, respectively. PMC and PMD are logical registers that are mapped to the actual PMU by each implementation. Each register is identified by a simple (index, value) pair. This scheme works across any PMU architecture because any register can be uniquely named with this scheme.

2. **System Call Model**: The interface is intended to be a part of the mainstream Linux kernel in the near future (starting from 2.6.25). But for the Linux kernels used for our research (2.6.17 and 2.6.23), the compiler support was enabled at the time of kernel compilation. In any case, the interface follows the system call model instead of the device driver model. The device driver model is not applicable because the interface supports per-thread monitoring, which requires hooks to the context switching code of the kernel to save and restore the PMU states on thread context switches. For system security reasons, access to kernel context switching code is not given to the device drivers. Therefore, applications make system calls (instead of ioctls used with device driver modules) to access the services provided by the Perfmon2 interface.

3. **Sampling Support**: There are two types of sampling possible with the Perfmon2 interface: time-based and event-based sampling. The sampling period is defined by

a timeout in time-based sampling whereas it is defined as the number of occurrences of a PMU event with event-based sampling. Using the Perfmon2 interface, AMD's Barcelona [2] processors can be programmed for time-based sampling and Intel's PEBS-based PMU can be programmed for event-based sampling.

The interface can be programmed to send a message-based overflow notification on occurrence of a specific event. However, for efficiency reasons, samples are automatically stored into a kernel buffer and notification is only sent to the monitoring tool once the kernel buffer fills up. For PEBS, we use event-based sampling to record architectural state after a configured number of L1/L2 misses (events).

The Perfmon2 interface is described in detail in [10]. The interface is also supported by a user level library (libpfm), which can be used by monitoring tools/applications to program the PMU. We use *libpfm-3.2* to configure PEBS support in Xeon and Core2 duo PMUs. This is detailed in the next section on the design and implementation.

2.5 Design and Implementation of the Page Placement Framework by Exploiting PEBS

Quite often, multi-threaded shared memory programs are not tuned for ccNUMA architectures where locality of access is paramount to achieve good performance from the applications. In most modern operating systems (like Linux), a policy of first touch page placement is used where the physical frame is allocated on the node where the page is first touched by a local processor. This often leads to a suboptimal allocation with OpenMP programs since a lot of communication happens through shared variables and more often than not, some threads access the variables more regularly than others. Thus, one could either tune the application for page placement or make some changes in the run time system or operating system to achieve better page placement. Our approach also attempts to obtain better locality but without modifying any of these subsystems. We leverage the hardware performance monitoring unit of the processors to derive "memory reference hints" by each thread of the application during a truncated run of the program. These hints are then used to derive better affinities of virtual memory pages to the processors. The existing first-touch policy of Linux is used to touch the hinted pages before the second execution of the main program. During this second execution, better locality is achieved and many remote references are avoided.

Section 2.5.1 gives an overview of the design of our page placement framework. Previous work on Itanium processors [20] proved that the "load cache miss" is a better indicator than "DTLB misses" to filter out memory accesses representative of an application's memory profile. For the x86 architecture, the PMU supports PEBS (as described above) in Intel's Netburst and Core microarchitectures. In Appendix B, we detail the PEBS configuration to generate information for L1/L2 cache miss events. The PEBS stores the architectural state of the processor on occurrence of the configured event(s). Appendix C details the design of a parser that decodes the effective linear addresses from the instruction pointers and the contents of the general-purpose registers. To transfer this information into user space, library support from Perfmon2 (*libpfm*) is also needed. In particular, we use the asynchronous notification mechanism of signals to obtain information from the kernel. The generated SIGIO signal triggers a signal handler, which is elaborated in Section 2.5.2.

2.5.1 Design Overview of the Framework

The following section gives a complete overview of our framework which operates in multiple phases. An environment variable "NUMA_PHASE" determines the phase in which the program is being run. The following types of program executions are possible:

- Trace Run: To get hardware traces (NUMA_PHASE = 1).
- Unmodified Run: Not to use our framework (NUMA_PHASE = 0).
- Profile Guided Run: Uses hardware traces (NUMA_PHASE = 2).

We now summarize the overall design. Reader is strongly encouraged to refer to previous work [20] [19] for further details.

1. **Collecting PEBS Hardware Traces**

   Configure PEBS-based PMU for L1/L2 misses using libpfm: This is described in detail in Sections 6 and 6 . If traces are obtained on a Core2 machine, a special handling for a skid of 1 instruction also needs to be done. This is explained in Section 2.5.3 .

   Instrument the benchmarks to start and stop Perfmon2 monitoring: The OpenMP NAS benchmarks and selected SPEC benchmarks have "hot regions" where threads make a lot of memory references, especially reads, in a recurring manner. For example, in time-stepped programs, after initialization, one or two time-steps are usually sufficient to snapshot each thread's memory access pattern. These can be termed as *the stable phase of execution*. Thus, we manually instrument the "hot regions" of the benchmarks to start and stop tracing. This tracing continues up to 5 iterations to get stable traces. Afterwards, the process is prematurely terminated. This is the truncated run of the program.

Get effective addresses and dump IP, Effective Address: During tracing, the PEBS-based PMU writes the architectural records in a kernel buffer. This buffer is exported to user space using 'mmap()' system call. We configure Perfmon to send an asynchronous notification when a certain fraction of the buffer gets filled. Our framework handles this notification. This is explained in Section 2.5.2. A parser module (see Appendix C) decodes the instructions to return effective addresses. Instruction pointers and effective addresses are then dumped in a per-thread trace dump file.

Trace Dynamic Allocation: Calls to malloc, calloc and free are intercepted and logged by the framework. The following information is logged by the threads that make dynamic allocation calls:

- Timestamp: The x86-64 architecture has high-precision timestamp counter, which is used to return time elapsed from a base time.
- Tid: An OpenMP thread identifier for the thread making dynamic allocation calls.
- Seq_id: A global sequence counter for dynamic allocation calls.
- Size: The size of the region requested
- File: The program file from which a call to malloc was made.
- Line: The line number in the program file.
- Address: The address returned by malloc/calloc.

2. **Affinity Decision**

   After obtaining per-thread memory traces, the per-node page affinity is determined, *i.e.*, it is decided on which node a physical memory page should reside for a particular virtual memory page.

   Based on the approximate memory access trace and the dynamic memory allocation information, the affinity decision module currently supports two metrics, one for uniform remote latencies and one for hop-sensitive remote latencies.

   The *uniform latency policy* allocates a page on the node that issues the maximum number of accesses to that page.

$$p_i \rightarrow n_j \Leftrightarrow rw_{i,j} = max_k(rw_{i,k}) \tag{2.1}$$

   This allocation rule requires that page p_i is allocated ($\rightarrow$) in the local memory of node n_j iff the number of read/write references $rw_{i,j}$ within page p_i issued from node n_j is maximal within the read/write references issued by any node for this page, where m is the total number of nodes. Intuitively, the average latency of an access can be reduced when a page is allocated closer to the processor that issues the largest amount of requests.

   The page affinity decision process consists of a number of steps. Initially, accesses are grouped by page address, and the total accesses from all threads to each page are calculated. Here, accesses are grouped by processor to calculate the per-node access count for each page.

   In practice, remote access penalties are not uniform but rather vary with the distance (number of hops over the interconnect) to the target node. The *hop-sensitive latency policy* allocates a page on the node that has the lowest aggregate access cost for references to this page issued from any node.

$$p_i \rightarrow n_j \Leftrightarrow \sum_{l=1..m} rw_{i,l} \times w_{j,l} = min_k(\sum_{l=1..m} rw_{i,l} \times w_{k,l}) \tag{2.2}$$

   This allocation rule requires that page p_i resides on node n_j iff the aggregate number of read/write references $rw_{i,l}$ for this page issued from all nodes, weighted by the hop-sensitive cost $w_{j,l}$ relative to local allocation on this node, is minimal within all aggregate weighted costs of any node allocations of this page. Intuitively, the average latency of an access can be reduced when a page is allocated close to all processors that issue large amounts of requests, *i.e.*, this metric takes references from multiple nodes into account instead of using the winner-takes-all paradigm.

The page affinity decision process consists of grouping accesses by page address and by thread (node). Yet, the page cost for an allocation is then calculated for a hypothetical allocation to each node. This cost is the additive weighted number of accesses issued for any node for this mapping. The weight $w_{i,l}$ denotes the latency (cost) of resolving a reference from node n_l that is locally mapped onto node n_i. Such pair-wise latencies can be experimentally obtained once and for all for a given architecture using the *bplat* microbenchmark [8] (see Section 4.1.1 for details).

The affinity decisions are generated differently for statically defined and dynamically allocated regions of memory. Statically defined memory (*i.e.*, the `bss` segment) contains space for uninitialized global variables. The starting address and extent of the static region is determined at link time. The affinity decision module simply generates a per-node list of page address offsets that have affinity to that node. The first logical processor in a node is responsible for using these page offsets to issue the actual "first-touch" page placements during the final trace-guided program run.

A more sophisticated scheme is required for dynamically allocated regions. Here, the starting address of the allocated region can and does change over multiple runs of the same program. For the benchmarks evaluated, two distinct dynamic memory allocation patterns were observed. Many programs had a small number of calls, each of which allocated a large chunk of contiguous memory. For such cases, we adjust the affinity page offsets relative to the starting address of the region. The affinity offsets will be used to "touch" the pages on the appropriate nodes during the trace-guided run just after the region is allocated.

A second dynamic allocation pattern was observed for programs issuing a large number of calls clustered in time, each allocating a small region of memory (*e.g.*, NAS-2.3 MG). The resulting heap regions are mostly allocated contiguously in space. However, due to the lossiness of memory access traces, many small allocated regions ("silent regions") are not represented by even a single access record in the trace. By inspecting trace records for other small regions allocated close by, these silent regions are allocated in their vicinity (on the same page) since physical memory is allocated on *page* granularity.

3. **Trace-guided Page Placement**

   The affinity decisions in the previous phase are used in trace-guided page placement runs. Linux uses the "first touch" page placement policy by default, i.e., the page is allocated on the node running the thread which writes first on the page. This policy is leveraged in our schemes for static and dynamic allocations. For static allocations, each thread touches the set of its "affine pages", which are read from its static affinity file. All the threads synchronize at a barrier after the touching phase to ensure that no processor accesses a statically defined page before the affinity hint for the page has been applied. This scheme has a minimum overhead because the initialization is only done at startup.

   For heap (dynamic) allocations, the fact that no thread in a legal program accesses any part of allocated address before malloc is done, is utilized to affect first touch based page placement. The scheme works as follows.

   During NUMA_PHASE 1, a wrapper only intercepts dynamic allocation calls like malloc(), calloc() and free() and logs some bookkeeping information as described in Section 1. However, in NUMA_PHASE 2, dynamic affinity hints generated by the offline analysis (see Section 2) are utilized to direct correct page placement. The dynamic affinity hints provide information about which parts of dynamically allocated regions should be allocated on which processor. The hints are given in the form of processor identifiers and the address offsets (from the start of an allocated region). The wrapper first performs the actual allocation and then applies these hints on a fresh allocation (Different virtual address may be returned in different runs of the program). Here, the difference from static allocation is that some pages in the dynamically allocated region may be more affine to another processor, i.e., processor on which this thread is not running. Recall that each thread was bound to a processor using *sched_setaffinity* calls. Again, the CPU set implementation of the above call is used to move threads to the affine processor. Afterwards, that page is "touched" by the thread. For every allocation request for which there are affinity hints for n processors, there are $n+1$ context switches (one switch to every target processor and a final switch back to the original processor). The experimental Section 4.1.2 shows that there is a very high overhead of page placement with benchmarks like MG. The approach for page placement employed for dynamically allocated regions is still subject to one caveat. While the benchmarks of the experimental sections always allocate memory only at the start of the program and do not free it until the end of execution, such behavior is not guaranteed. Some programs may repeatedly allocate and free memory in the stable execution phase. In those cases, the effectiveness of the "first-touch" scheme would be reduced. This occurs because portions of the virtual address space may be "reused" by the library allocation function after they were initially freed, and it is possible that the "reused" virtual page may change its affinity. With our current scheme, the *physical* memory will only be allocated once on the node where the page of virtual memory is first touched. This could potentially give suboptimal allocations because the first touch scheme tends to pin the page only once. Dynamic page migration supported by the kernel can solve this issue if a change in affinity pattern is detected.

2.5.2 SIGIO Handler

Recall from Sections 6 and 6 that only one thread at a time is profiled for L1/L2 miss events on PEBS-based PMUs. Through standard *fcntl* file control mechanism, an asynchronous notification is configured and the thread to be profiled takes the ownership of the Perfmon file descriptor. SIGIO is masked by all the threads except for the thread being profiled. A kernel level sampling buffer is configured that sends a message notification after the configured fraction of the buffer (90%) overflows. Since our framework requests an asynchronous notification on any I/O event for the Perfmon file descriptor, a SIGIO signal is sent to the user space, which is handled by the profiled thread.

A standard *read* library call is used to read the Perfmon message. We then determine the message type returned by the Perfmon implementation. The expected message is the overflow message (PFM_MSG_OVFL). If the message type is unknown or 'PFM_MSG_END', an error is printed out on standard error and the message is discarded. If the message is an overflow message, the generated samples are processed and the 'overflows_received' counter is incremented for that thread.

Processing Samples: The function *process_smpl_buf(thread_rec_t *)* is used to process the generated samples. The DS management area stores the current PEBS index and the PEBS base. These are useful for determining the number of PEBS records written after the last read of the records. A loop goes over each PEBS record which is nothing but the architectural content of the registers. For each record, the parser module (see Appendix C) is called to determine the effective address. Finally, the instruction pointer and the effective address are written out in a trace file identified by the thread id.

2.5.3 Handling skid of 1 instruction on Core2 machine

PEBS-based Core PMUs indicate the register contents for the next instruction as there is always a skid of one instruction due to some microarchitectural implementation. However, it is non-trivial to find the previous instruction pointer (given the IP of next instruction) and its architectural state as the x86 architecture is a CISC architecture with variable length instructions. Fortunately, we are concerned only with instructions causing load miss. We assume that any instruction that would cause a load miss would not change the contents of the registers except for the register that saves the data from the memory. But that register is not used in the calculation of the effective address. So, this is a safe assumption for the calculation of the effective addresses.

But finding the previous IP address requires some special handling. The master thread takes the assembly dump of the process using the *objdump* tool. From the program, the *system()* library call is used to run objdump as follows:

“ objdump –prefix-addresses -d <executable name> > <executable.asm> ”

```
00000000004e3ef8 <free_slotinfo+0x2f8> add    $0x10,%rcx
00000000004e3efc <free_slotinfo+0x2fc> test   %rax,%rax
00000000004e3eff <free_slotinfo+0x2ff> jne    00000000004e3f67
                                             <free_slotinfo+0x367>
00000000004e3f01 <free_slotinfo+0x301> add    $0x1,%rdx
00000000004e3f05 <free_slotinfo+0x305> cmp    %rsi,%rdx
00000000004e3f08 <free_slotinfo+0x308> jne    00000000004e3ef4
                                             <free_slotinfo+0x2f4>
00000000004e3f0a <free_slotinfo+0x30a> callq  00000000004a8fb0
                                             <__cfree>
00000000004e3f0f <free_slotinfo+0x30f> movq   $0x0,(%r14)
00000000004e3f16 <free_slotinfo+0x316> mov    0x0(%rbp),%rdi
00000000004e3f1a <free_slotinfo+0x31a> mov    (%rdi),%rsi
```

```
00000000004e3f1d <free_slotinfo+0x31d> test   %rsi,%rsi
00000000004e3f20 <free_slotinfo+0x320> je     00000000004e3f46
                                            <free_slotinfo+0x346>
00000000004e3f22 <free_slotinfo+0x322> xor    %edx,%edx
```

As can be seen from the above dump, different instructions are of different length, e.g., instruction at IP 0x 00000000004e3f01 is 4 bytes in length, whereas the instruction at IP 0x 00000000004e3f05 is only 3 bytes. We need to get the IP for each instruction when we calculate the effective load miss address. A "hash map" is the chosen data structure for keeping a map of IP and its previous address. The disassembled instructions are read and for each IP, the previous IP is stored in a hash map. This hash map is referenced to obtain the previous IP.

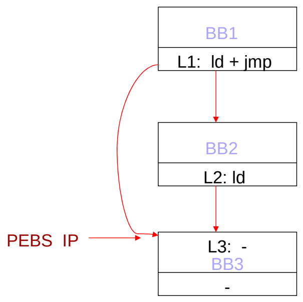


Figure 2.3: A hypothetical control-flow graph

The above mechanism identifies the previous IP from static information whereas load misses are a result of dynamic instruction execution. Theoretically, it could be possible that a load miss instruction may precede the instruction in dynamic execution stream but not in static program layout. For example, consider a hypothetical control-flow graph shown in Figure 2.3. Assume that PEBS tracked the first instruction (L3) in BB3. The previous instruction according to the static layout of the program (L2) is the last instruction of BB2. Also, assume that the last instruction (which includes both load and jump μops) of BB1 (L1) caused a load miss from data memory. In this case, our implementation would wrongly identify L2 as a "load miss" instruction. However, L2 would be discarded if it does not include a load μop. Fortunately, x86 ISA does not have a jump instruction that also references data memory, i.e., L1 is not a valid instruction type. Hence, this problem does not occur in practice.

Chapter 3

Page Migration using PMU Traces

3.1 Overview

On ccNUMA platforms, accesses to local physical memory (on the same node as the requesting processor) result in lower latencies than accesses to remote memory (on a different node). In our previous work (see Chapter 2), we have shown that the hardware performance monitoring unit can be leveraged to generate traces, which can be used as a guide for automatic page placement. The scheme was static as traces were analyzed offline to generate affinity hints for the virtual memory pages, i.e., for each page a better target "NUMA node" was determined. In the next (full) run, the affinity hints were used to derive a better page placement.

We now extend this scheme and make it dynamic, i.e., the affinity decisions are generated at run time, possibly after we get a "confident" estimate of the memory access patterns by each thread. These decisions are then used to steer the migrating pages to suitable nodes. The scheme is based on page migration support provided by the Linux kernel 2.6.18 and above. Linux, even though a NUMA-aware operating system, does not make application-specific policy decisions while allocating pages. However, it does provide the users with mechanisms to enforce their own policies. There are currently two different mechanisms by which page migration can be triggered from user space:

- The mbind system call and
- the move_pages system call.

We introduce both these approaches in Sections 3.2 and 3.3. These are experimentally evaluated in Section 4.1.3. In the course of this work, some optimizations were needed with the mbind system call. These are described in Section 3.2. Page migration incurs some overhead both at the hardware level and at the software level. This is discussed in Section 4.1.3. To lessen the impact of overhead, a heuristic is used to reduce the number of page migrations. This is also described in Section 3.3. For distributed shared memory architectures, locality of processor to referenced memory is essential for performance. Data page migration is certainly not the only means to improve locality. Other approaches like thread/process migration and optimal page placement have also been tried in the past. Linux, by default, uses first-touch page placement, i.e., the physical page is allocated closer to the processor which touches it first. If sufficient memory is available, it is placed on the same node. This first-touch policy can be overwritten using the "interleaved" policy, which can be triggered using the *numactl* tool in Linux.

The experiments in Section 4.1.3 compare the "interleaved" allocation with our hardware trace-driven approach using *hop-sensitive* and *uniform-latency* policies. Also, the filter heuristic over *uniform-latency* policy is compared with the non-heuristic based approaches.

3.2 Design of Dynamic Page Migration on the SGI Altix platform

The SGI Altix is a high-performance computing architecture that uses industry standard Itanium2 processors and runs a commodity Linux kernel. It has a high bandwidth, low latency interconnect called NUMALINK based on the "fat tree" model [26]. Itanium 2 processors also export their PMU details to the users and are supported by the Perfmon2 implementation of the kernel.

Dynamic page migration on this platform uses affinity hints derived from the PMU-generated cache load miss traces. This work of programming the PMU using libpfm was already done and is described in detail in [19].

While processing trace records, effective addresses causing long latency loads (i.e., cache misses) are identified by their page addresses and recorded along with the reference count in a per-thread `map` data structure . Later, at the end of stable phase of execution (which is usually the second iteration of the outermost loop), the master thread aggregates the per-node reference count for all the pages. Other threads synchronize on a barrier. The master thread also performs a hop-sensitive analysis as given by Equation 2.2 (Section 2.5.1) to determine the best node for page migration. Also, the minimum cost of migration is recorded for each page. We also calculate the average of all the costs by assuming page allocation on each of the available nodes. The cost difference is stored for each page, which is used as criteria to restrict the number of page migrations.

Finally, the master thread directs page migration using the mbind system call. Its interface is defined as follows:

*int mbind(void *start, unsigned long len, int policy, unsigned long *nodemask, unsigned long maxnode, unsigned flags).*

mbind() sets the NUMA memory policy for the memory range starting with start and continuing for "len" bytes. The memory of a NUMA machine is divided into multiple nodes. The memory policy defines in which nodes memory is allocated. mbind() only has an effect for new allocations. If the pages inside the range have been already touched before setting the policy, the policy has no effect.

MPOL_PREFERRED can be used as a policy to set the preferred node for allocation.

The latest additions to the flags are the following two flags that support page migration:

- If MPOL_MF_MOVE is passed in flags, then an attempt will be made to move all the pages in the mapping so that they follow the policy. Pages that are shared with other processes are not moved.

- If MPOL_MF_MOVE_ALL is passed in flags, then all pages in the mapping will be moved, even if other processes use the pages. The calling process must be privileged (CAP_SYS_NICE) to use this flag.

We use MPOL_MF_MOVE as our pages are application pages, and our process is not a privileged process.

In our experiments, we also measure the overhead of migration including the overhead of making system calls. To amortize the impact of system call overhead, we provide "chains" of contiguous pages to be moved (so that we can use a length greater than 1), thereby reducing the number of system calls. Also, only the pages with high cost difference are moved since they are more likely to have a greater impact on the overall wallclock execution time.

In contrast to the page placement framework design as described in Chapter 2, this framework does not require any special handling for heap allocated regions as the pages are migrated at run time.

However, in our experiments, a major limitation of the platform was observed: PMU traces on this machine are not stable. This was observed by comparing the sizes of the trace files generated by page placement framework across five runs. The trace file sizes varied a lot. Only one Altix machine gave stable hardware traces but it had an old kernel version which did not support the page migration interface. The unstable traces might be attributed to some Perfmon2 implementation shortcoming in the kernel. In absence of accurate hardware traces, we had to abandon our work on this platform.

3.3 Design of Dynamic Page Migration on the x86 Opteron Platform

We also ported our page migration framework to a contemporary x86-64 Opteron, a platform that is cache coherent and uses a *hypertransport* interconnect. It runs a newer Linux kernel version 2.6.18 with support for the second page migration interface: *move_pages()*. However, this platform unlike the Altix platform does not have a published performance monitoring support in processors. Thus, we had to take the PEBS traces from a single socket Core2 Duo machine. This setup poses inherent limitations since multiple threads run on one core and L2 caches are shared by both the cores. Due to this limitation, there is a great chance of cache thrashing caused by concurrent access to the same cache lines by multiple threads, which potentially pollutes our traces. Currently, we obtain traces and then filter out all the cold misses. Figure 3.1 shows the conceptual design of the dynamic page migration framework.

The trace collection and PEBS configuration is exactly the same as with the PEBS based page placement framework (see Chapter 2). Function `map2page` converts memory addresses to page addresses. After the end of traced loop, the page migration module is activated if a proper environment variable is set (*NUMA_PHASE = 3*).

The master thread reads the page files for threads each of which store tuples of the form < page address, # of references >. A *page distributor module* (see Figure 3.1) creates two pagemaps, one for static allocations and the other for dynamic allocations.

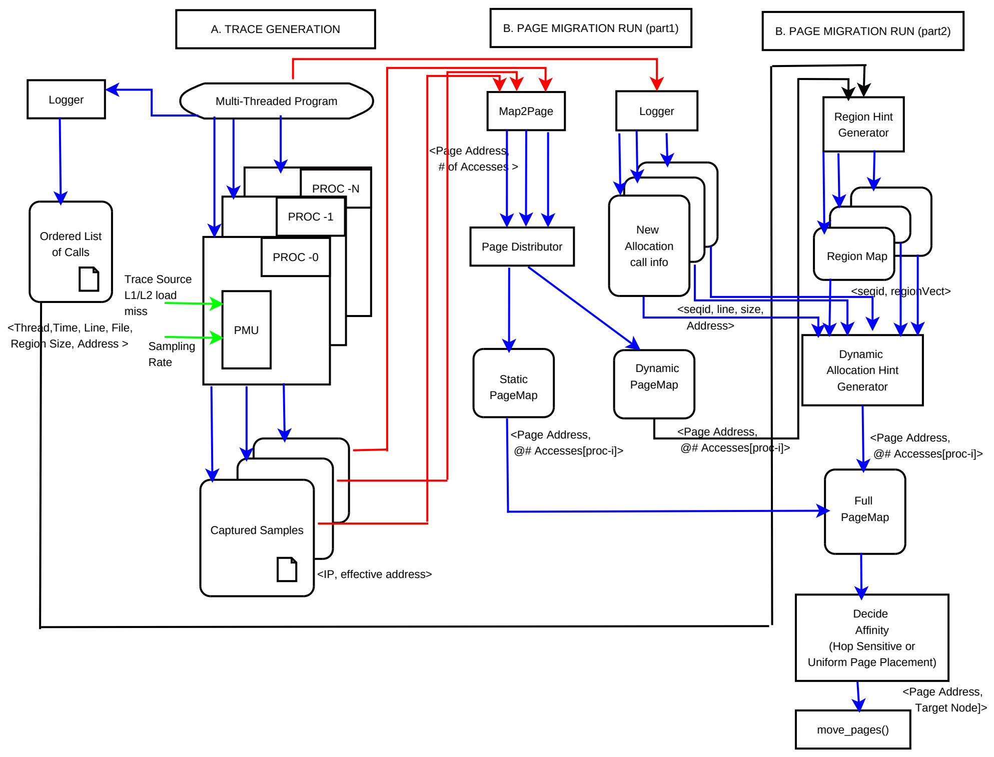

A. TRACE GENERATION
B. PAGE MIGRATION RUN (part1)
B. PAGE MIGRATION RUN (part2)
Logger
Multi-Threaded Program
Ordered List of Calls
PROC -N
PROC -1
PROC -0
PMU
Trace Source L1/L2 load miss
Sampling Rate
<Thread,Time, Line, File, Region Size, Address >
Captured Samples
<IP, effective address>
Map2Page
<Page Address, # of Accesses >
Page Distributor
Static PageMap
<Page Address, @# Accesses[proc-i]>
Logger
New Allocation call info
<seqid, line, size, Address>
Dynamic PageMap
<Page Address, @# Accesses[proc-i]>
Region Hint Generator
Region Map
<seqid, regionVect>
Dynamic Allocation Hint Generator
<Page Address, @# Accesses[proc-i]>
Full PageMap
Decide Affinity (Hop Sensitive or Uniform Page Placement)
<Page Address, Target Node]>
move_pages()

Figure 3.1: Block Diagram of Page Migration Framework

A pagemap is nothing but a hash structure keyed on the basis of page addresses, and it records the number of accesses by each thread. For dynamic allocations, a logger tracks the memory allocation calls and stores (in memory) the newly heap allocated addresses, sequence identifiers, allocation sizes etc. for each calling thread.

Then, the master thread reads the file containing a list of allocation calls. Using the dynamic page map (created by the page distributor), it derives the hints for the regions. A region is defined as the set of page offsets from the start of the allocated address. A region map is created for each thread using heap allocations. A region map is a hash map with sequence id of the allocation calls as a hash key. It returns a *region vector*, which consists of offsets and associated access counters for each thread.

The derived information from the region map is correlated with the current dynamic allocation information as shown in Figure 3.1. As a result, a new dynamic page map is created, which is merged with the static page map created by the page distributor. The combined pagemap is used to decide affinity based on hop-sensitive or uniform-latency policies as shown in Section 2. At the end, a page is deemed more affine to a node. As an optimization over uniform-latency policy, we filter out some of the pages for migration since it may happen that the selected node is not "sufficiently good", e.g., if processors on one node make one or two references (due to cold miss or cache thrashing), the page is not really fit to be moved to that node. To avoid that, we use a simple heuristic that the number of page references for the winner node must be at least two more than the number of references by each node. For statistical measurement, a modified variance is calculated over the maximum references. The decision criteria is

$$variance >= 4 * (n-1)/n \tag{3.1}$$

i.e., the variance for the winner should be greater than 2^2 *(n-1)/n , where 'n' is the number of numa nodes in the system.

It is worth noting that the special handling for heap allocations is only required because the hardware traces are generated on a different machine and then used for analysis on the ccNUMA Opteron. This changes the addressing for heap memory management. Thus, previously used addresses can not be used for tracing. Had the Opteron architecture supported hardware tracing, the page distributor module would have become redundant. In fact, on Itanium architecture (SGI Altix), we did not need this complex analysis.

Finally, page migration is started using *move_pages*[1] interface shown below:

*long move_pages(pid_t pid, unsigned long nr_pages, const void **address, const int *nodes, int *status, int flags);*

It has a user friendly interface to move the pages in the address space of a running process to a different NUMA node. Only the physical mappings to the corresponding virtual addresses change, and the data is copied over to the other node. The call can also be used to determine the nodes to which the pages are currently mapped (if argument *nodes* is NULL). We use this property to filter out the pages that are present on their target node. Argument *address* is an array of addresses of pages subject to being moved. Argument *nodes* is an array of numbers of nodes to move the corresponding pages to. The *flags* argument describes the type of pages that will be moved. The library supports the following flags for page migration:

- MPOL_MF_MOVE: Only pages mapped by the process identified by the 'pid' will be moved;

[1]Appendix A briefs about the kernel implementation of move_pages.

- MPOL_MF_MOVE_ALL: Pages potentially mapped by multiple processes will be moved (mode needs to have sufficient permissions).

Since our framework runs with normal user privileges, we use the former flag for page migration.

Please note that while the master thread does this analysis, other threads synchronize on a barrier and wait for page migration to take affect. Even if other threads do not synchronize, the kernel tends to suspend all the threads while pages are being migrated.

Chapter 4

Experiments And Results

4.1 Overview

The capabilities of PEBS are evaluated with a microbenchmark to assess the degree of lossiness. The microbenchmark strides over a large array with a 12KB stride in order to defeat the hardware stride prefetcher of the Pentium architecture. The *Perfmon2* framework is utilized to access the hardware counters and to collect the PEBS-generated trace [10].

Based on the data size, access pattern and cache parameters, the program is estimated to contain approximately 6 million L1 and L2 load misses. On a different x86 machine, hardware counters reported 6.71 L1 and 6.72 million L2 load misses.

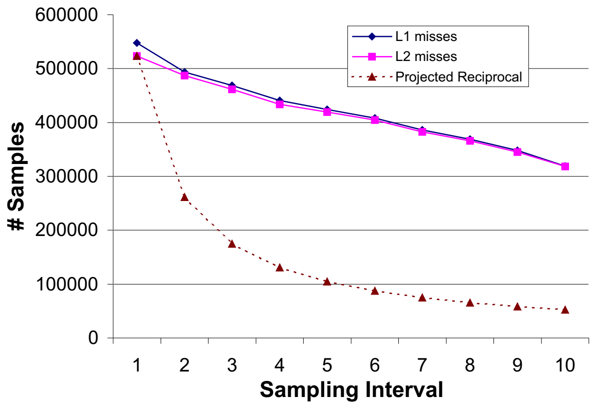


Figure 4.1: Evaluation of load miss tracing by PEBS: Intel Xeon

Figure 4.1 and 4.1 show the number of samples collected for L1 and L2 load misses with increasing sampling intervals on two different microarchitectures, both with

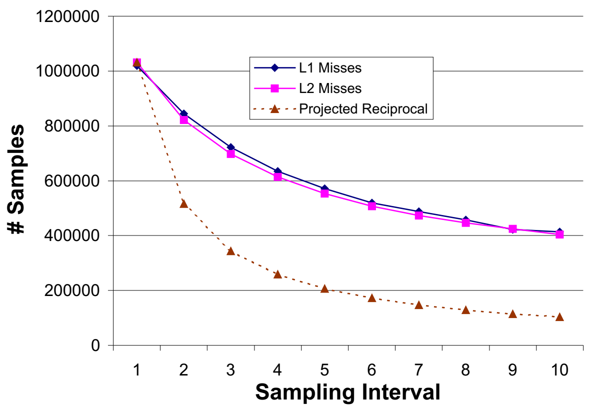


Figure 4.2: Evaluation of load miss tracing by PEBS: Intel Core2 Duo

PEBS support. The numbers are averaged over 10 runs with a standard deviation of less than 1%. The following observations were made:

- Both L1 and L2 traces are quite lossy. At the smallest sampling interval (1), less than 10% of the expected L1 or L2 misses are collected for the Xeon and less than 15% on the Core2 Duo platforms.

- The L1 and L2 curves are very close. This is expected because each L1 miss is almost always an L2 miss in the microbenchmark.

- The number of samples does not decrease in linear proportion to the increase in the sampling interval. This is in contrast to an expected decrease in the number of samples depicted by the "Projected Reciprocal" curve.

- The PEBS PMU on the Core2 machine provides a slightly better representation of memory profile of the microbenchmark. However, the curves are pretty similar for both machines and a significant amount of trace lossiness is observed.

4.1.1 Hop-Sensitive Page Placement

Instead of the uniform latency page placement policy evaluated so far, this section focuses on the implementation of the hop-sensitive page placement policy (Eq. 2.2, Section 2.5.1). As briefly mentioned, remote access penalties are not uniform but vary with the distance to the target node. To measure the load access latency, the *bplat* microbenchmark [8] is utilized with threads and memory bound to different nodes. Two sets of measurements were performed, one on a four-socket Opteron system with one processor core per node and the other on an eight-socket Opteron system with two cores per node.

Table 4.1: Access Latencies 4-Node Opterons [nanosecs]

CPU on Node	Memory on Node			
	0	1	2	3
0	102	138	**172**	140
1	143	107	141	**172**
2	**179**	141	102	141
3	141	**175**	142	108

Table 4.2: Access Latencies 8-Node Opteron system [nanosecs]

CPU on Node	Memory on Node							
	0	1	2	3	4	5	6	7
0	126	123	128	178	148	146	148	**185**
1	125	122	**180**	126	147	141	128	151
2	130	142	93	143	116	112	148	**150**
3	146	125	143	92	117	113	**148**	147
4	**148**	147	116	117	96	148	130	147
5	**151**	145	118	115	146	92	145	127
6	150	131	145	146	130	**180**	125	126
7	**182**	150	147	146	173	127	126	126

Table 4.3: % Pages Changed: Hop-Sensitive vs. Uniform Policies

Benchmark	Trace: L1 Misses	Trace: L2 Misses
BT	3.22	0.70
CG	0.26	0.00
FT	0.31	3.64
IS	1.57	2.37
LU	1.83	7.76
MG	0.10	0.17
SP	1.12	0.88
equake	0.03	0.20

Tables 4.1 and 4.2 show the reported latencies. The values are the average of 10 runs, and the standard deviation was less than 5%. As can be expected, access to node-local memory is always cheaper. But notice that accesses to non-local nodes take differing amounts of time. *E.g.*, consider the access latencies for the CPU on node 1. Normalizing to local node access on the 4-node system, it takes about 30% more time to access memory on nodes 0 and 2, but it takes 60% longer to access memory on node 3. For the 8-node system, additional latencies of up to 100% were observed. This occurs because of multi-hop latencies to remote memory of other nodes over the HyperTransport interconnect with an average of 30-39 nanoseconds per hop depending on the system. For the 4-node system, the

HyperTransport interconnect is laid out in a ring (4-node square) topology for a maximum of two hops. For the 8-node system, the layout is a more complex set of multiple partial squares plus two diagonals in one partial square for a maximum of three hops.

In the hop-sensitive page placement policy, the latencies shown in Tables 4.1 and 4.2 provide the weights $w_{i,j}$ for placement on node n_i and a reference from node n_j (see Eq. 2.2). To implement the cost-based selection of page placement, a histogram of accesses from every node is again constructed for each page by considering each node as the candidate affinity node. The values in the latency table and the histogram values are used to compute the *weighted* score that represents the cost of allocating the page on that node. The candidate node with the *lowest* cost wins, and the page is assigned to that node. This approach is portable because it is *observation-based*, *i.e.*, it uses only the *measured* latencies between different nodes without requiring knowledge of the exact architecture and interconnect topology.

4.1.2 Evaluation of Hardware Trace-Driven Page Placement

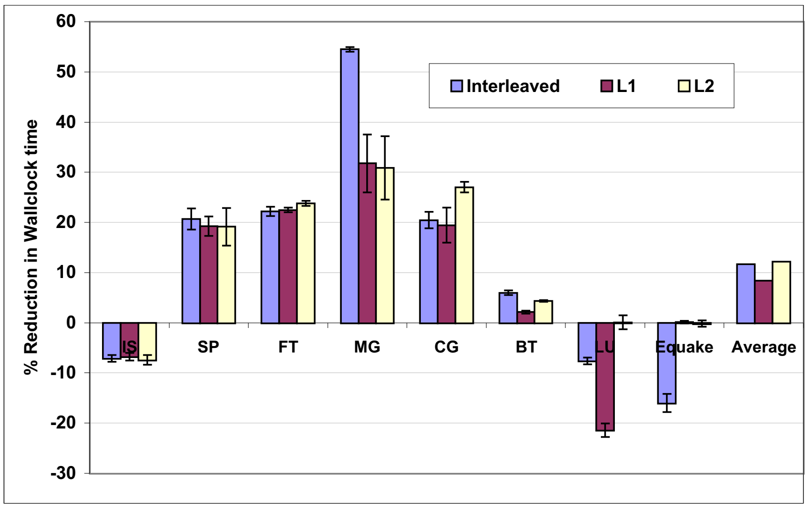


Figure 4.3: Time Savings over Original Program and Arithmetic Mean (hardware traces on Intel Xeon)

We explored the PEBS capabilities on two different architectures, a 32 bit Intel Xeon and a 64 bit Intel Core2 Duo. The corresponding target machines for these two architectures were 4 node and 8 node AMD Opteron machines, respectively.

Benchmarks: Nine OpenMP benchmarks were experimentally evaluated. This includes seven out of the eight NAS-2.3 benchmarks (excluding EP). The NAS benchmarks are C versions of the original NAS-2.3 serial benchmarks [3] provided by the Omni Compiler group [1]. EP is not evaluated since it does not have significant sharing of data [21]. In addition, the 320.equake and 332.ammp benchmarks from the SPEC OMPM2001 benchmark set were assessed in the results. These benchmarks have significant dynamic memory allocation, thereby putting the dynamic touching mechanism to the test. All programs except SP were compiled with gcc at the -O2 optimization level. SP seemed to take a long time to execute when compiled with gcc. This was due to implementation issues with OpenMP run time support inherent to gcc. It was therefore compiled with icc (-O2 optimization level) and linked to its OpenMP library. Due to memory resource limitations on the Xeon, all benchmarks except IS and LU use the smaller Class B data sets while the SPEC benchmarks use the reference data set. However on the Core2 machine, larger Class C data sets are used except for FT and BT which use the Class B data set. All the benchmarks on the Xeon/4 node Opteron run with 4 threads, whereas on Core2/8 node Opteron, they run with 16 threads.

OpenMP thread scheduling was set to static. The Core2 trace hardware platform uses Intel Core 2 processors running at 2.66GHz, with 4 MB shared unified L2 cache, and a 32 KB L1D cache. The target eight node Opteron has Dual Core AMD 8220 processors, each with 1 MB unified non-shared L2 cache and a 128 KB L1D cache.

For each program, markers were inserted to delineate the start and end of the time step. For 332.ammp, the pre-existing round-robin allocation of the "atom" element was disabled for the trace-related runs. However, the benefit metrics (wall clock time, number of remote accesses) are still compared against the original program. For the IS benchmark, a one-time dynamic allocation for the `prv_buff1` array is issued since the program failed to execute with the default stack allocation for this variable. Out of the 9 benchmarks, 4 benchmarks (MG, 332.ammp, 320.equake, IS) utilize dynamic memory allocation. The remaining benchmarks operate with statically declared global arrays.

Each thread was bound to a different core using the *sched_setaffinity()* primitive in Linux. PEBS-based L1 and L2 load misses were obtained for each benchmark for a truncated program run on Xeon and Core2 machines with a sampling interval of ten (For these experiments, the truncated programs ran longer than the Itanium2-based experiments of previous work [19] to allow collection of more trace data). For the first set of experiments, the traces were processed as described earlier and affinity hints were generated using the hop-sensitive affinity decision mechanism. For recent experiments, both the hop-sensitive and uniform latency policies were used.

Performance Analysis (Experiment Set 1)

For each program, the wall clock time was measured with trace-guided page placement and compared to the original program's runtime on the 4-node Opteron system. The system was shared but only lightly loaded. Furthermore, a *round-robin interleaving* of the memory pages across the nodes was evaluated, which was obtained through the *numactl* library interface [17].

Figure 4.3 shows the improvement in wall clock time compared to the original program. The values are an average of 8 runs, and the positive and negative error bars represent one standard deviation each. "L1" and "L2" represent trace-guided page placement with L1 and L2 cache miss traces, respectively. The following observations can be made. The developed trace-guided schemes perform well for 5 benchmarks (SP, FT, MG, CG, BT). Wall clock improvements for the L2 miss trace range from -7% to 30% with an average improvement of 12.2%. Wall clock improvements for the L1 miss trace range are similar, except for LU where a performance loss of 21% is observed. Intuitively, the L2 miss trace filters out loads that hit in L2. Therefore, L2 misses are a better indicator of the true distribution of load requests to a page in memory compared to the L1 miss trace.

The performance improvements obtained with memory interleaving (depicted in Figure 4.3) indicated that simple round-robin interleaving works almost as well as trace-guided page placement on the small-scale ccNUMA system subject to experimentation, but this depends in large on the algorithmic properties of the target applications. With MG, the program runtime is very short ($<$ 30 seconds for original program). Apparently, the developed trace-guided scheme is unable to recoup the overhead of forcing page placement within this short time so that interleaving happens to provide a larger relative improvement. Yet, interleaving performs badly for equake while the trace-guided scheme shows no net impact. Here, the benefits of page placement seem to be as high as the overhead of the page touching mechanism. Overall, long latency misses (L2 in this case) provided a uniformly reliable indicator for page placement decisions while interleaved memory allocation occasionally resulted in a significant performance penalty.

Performance Analysis (Experiment Set 2)

These experiments were conducted on an 8 node dual core Opteron machine. Here, all NAS benchmarks except for FT and BT used the larger input set Class C benchmarks.

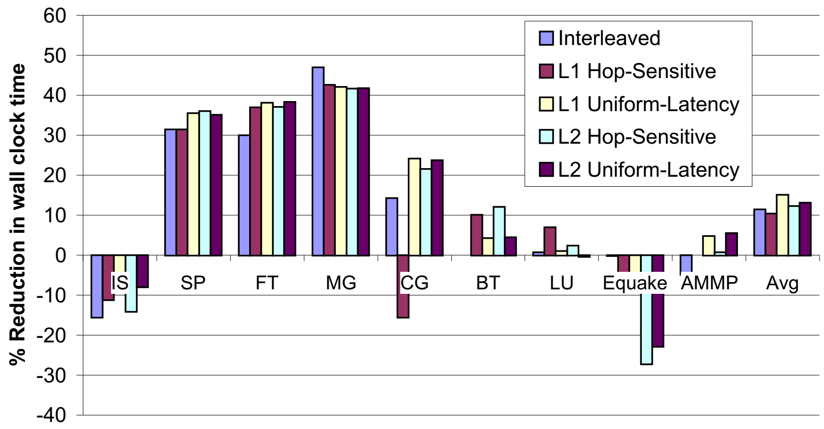


Figure 4.4: Time Savings over Original Program and Arithmetic Mean(traces collected on Core2)

Two of the SPEC benchmarks, equake and ammp, were also evaluated with their reference data set.

Figure 4.4 shows the improvement in wall clock time compared to the original program. The values are an average of 4 runs. The standard deviation observed was very low. Both the uniform latency and hop-sensitive policies were evaluated. The rest of the setup remains the same. The trace-guided scheme (with one variant or the other) performs well for 7 out of 9 benchmarks. The performance improvements for L2 miss traces are in the range of -27% to 41.8% with an average of 12.2% with hop-sensitive and 13.1% with uniform-latency policies, whereas for L1 miss traces, they range from -11% to 42% with an average of 10% for hop-sensitive and 15% for uniform latency policies.

The results in Figure 4.3 show that L2 misses are a better indicator to track long-latency loads that hit the memory. However, the results in Figure 4.4 contradict this result. A counter argument for this could be that since L2 cache is shared on the trace platform (Core2 machine), there is interference by parallel executing threads. This could lead to cache thrashing if multiple threads access the same cache lines. As a result, some loads that hit in cache actually cause a miss, and thus, pollute the trace-profile. This is magnified by running 16 threads (Figure 4.4) as compared to 4 (Figure 4.3) as the probability of them accessing the shared cache lines increases significantly. The results shown in Figure 4.4 exclude the overhead due to explicit touching of virtual memory pages, thereby allocating a physical page on the same node as the requester thread. Touch overheads for L1 miss and L2 miss are indicated in Tables 4.4 and 4.5. The touch overhead is usually less than

Table 4.4: % Touch Overhead in L1 miss Runs

Benchmark	Trace: Hop-Sensitive	Trace: Uniform-Latency
IS	0.11	0.12
SP	0.08	0.08
FT	2.25	2.19
MG	9.99	8.45
CG	0.03	0.12
BT	0.56	0.60
LU	0.11	0.12
EQUAKE	1.17	1.10
AMMP	0.25	0.14
AVERAGE	1.62	1.44

Table 4.5: % Touch Overhead in L2 miss Runs

Benchmark	Trace: Hop Sensitive	Trace: Uniform Latency
IS	1.12	1.14
SP	0.08	0.07
FT	2.26	2.29
MG	9.99	8.45
CG	0.03	0.12
BT	0.54	0.57
LU	0.09	0.09
EQUAKE	1.12	1.08
AMMP	0.25	0.14
AVERAGE	1.72	1.55

1% for most of the benchmarks (except for FT which has a touch overhead of 2.25% and 2.19% for L1 miss and 2.26% and 2.29% for L2 misses and MG for which the corresponding numbers are 9.99% and 8.45% for both L1 and L2 misses). These benchmarks cause many load misses and MG's address space is composed of a large number of pages from heap, which, as mentioned before, has a considerable overhead with our scheme.

Interleaving also performs well with these benchmarks. But the performance of interleaving is tightly coupled with the algorithmic design of the programs. If pages are frequently read by large number of threads, a balanced distribution can improve locality of accesses. We find that interleaving shows an 11% improvement on average. Yet, some benchmarks like IS,BT, LU, EQUAKE and ammp show less significant improvements. Overall, the trace-driven approach shows better performance in most cases despite being disadvantaged by the lossiness of PEBS PMU and suffering a lot of potential cache thrashing effects due to heavily shared caches on the `input processor` where traces are collected.

Uniform vs. Hop-Sensitive Page Placement: In Section 1, the hop-sensitive page affinity policy was introduced as a refinement to the uniform latency policy. Table 4.3 shows the percentage of pages that result in different affinity decisions between these two policies. As can be seen, in most cases, *both policies made the same decision* for both L1 and L2 miss traces as trace inputs. As a consequence, no substantial wall clock differences were observed between the two page placement options.

4.1.3 Evaluation of Dynamic Page Migration using PEBS Traces

Section 3.1 lists two methods of page migration in the Linux kernel. We evaluate both of them for wall clock performance using the same PRBS traces as before.

Evaluation of mbind: The master thread in a microbenchmark allocates pages according to the default (i.e., first touch) policy and the interleaved policy. Interleaved allocation is performed using the *numactl* tool in Linux. To model the wall clock experiments, a similar configuration with 16 OpenMP threads is used where each thread is bound to a different core. The master thread allocates pages by touching elements of a static array. After that, either the master thread or some other thread issues the mbind() call with the MPOL_MF_MOVE flag to migrate pages on a random node. The time to migrate pages is then measured using the *gettimeofday()* call in Linux. Due to the limitation of the mbind interface, policy-based migration can not be affected on more than one node at a time. Figure 4.5 shows the page migration time on varying the number of pages to migrate when a different thread issues migration call. As can be seen, the migration time seems to be a linear function of the number of pages for both allocation schemes. However, as the number of pages increases, interleaved allocations incur more overhead for page migration. This difference is more pronounced if the page migration requests originate from the same node on which the initial memory is allocated with the default allocation scheme. For default allocation, memory was allocated on node 0 and node 7 was chosen as the target node. Table 4.2 shows that the NUMA distance between node 0 and node 7 is the maximum amongst any pair of nodes. This suggests that default allocation should have been a more expensive scheme as migrations have to go through up to 3 hops. However, the "requests to migrate" may underutilize the bandwidth of the NUMA interconnect. This cost is mitigated if the requests are issued to the local memory controller. With interleaved allocation, these requests are distributed over the entire *HyperTransport* interconnect and they travel through links of different latencies. Overall, it is clear that interleaved initial allocation incurs more page migration overhead.

Evaluation of move_pages: move_pages() gives the flexibility of moving a page to a specified node. Mbind(), in contrast, only allows to migrate a "set of pages" on a node. The same microbenchmark is used, but this time the page migration mechanism is changed to move_pages(). Also, now the target migrations can be done in an interleaved manner. Notice that page migration is started from the second page, i.e., the first page

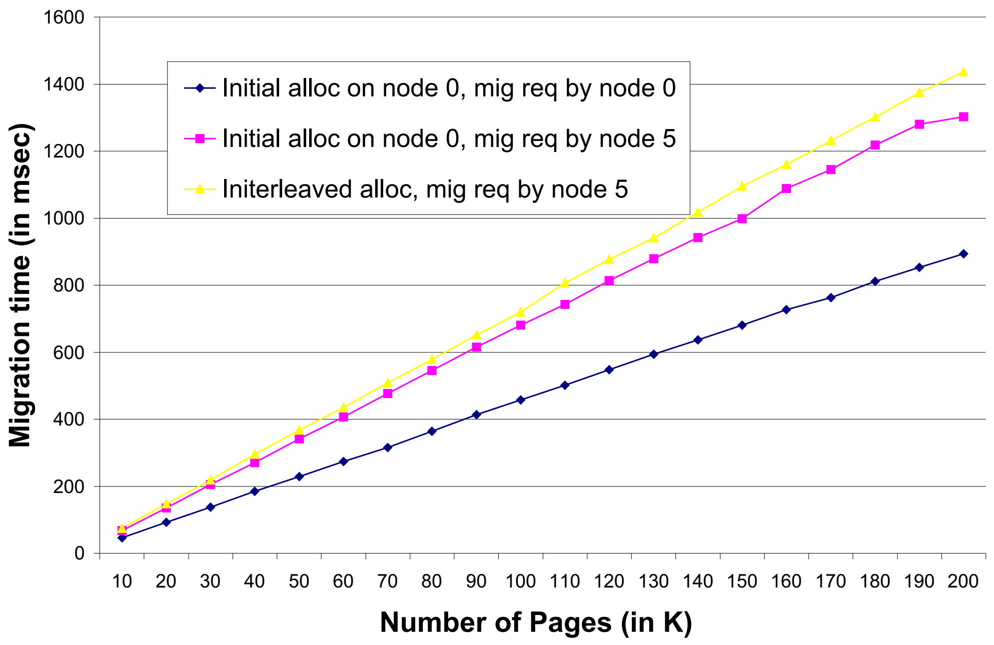


Figure 4.5: Page migration with mbind

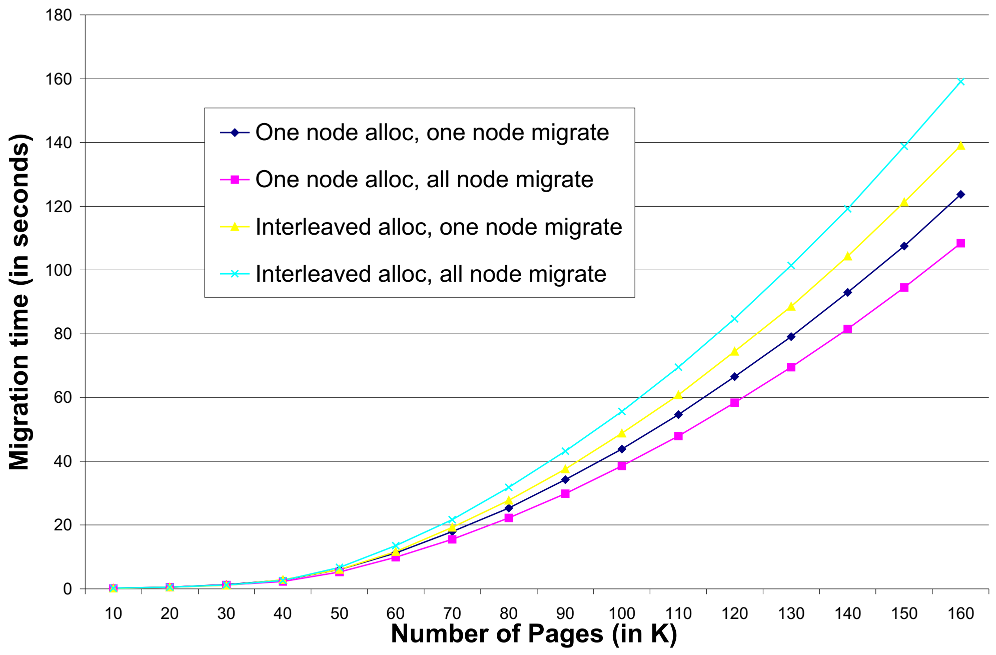


Figure 4.6: Page migration with move_pages

is skipped because the physical page may be shared with data from linked libraries. For initial interleaved allocation, this means that each page is moved to the adjacent node. All the migration requests originate from node 5, an intermediate node in the eight node Opteron system. For default allocation on one node, distributed page migrations cost less as compared to all the migrations on node 7, which, as mentioned above, is the node at a maximum distance from node 0. Initial interleaved allocation on mbind incurs more overhead for page migration (for the same reason). In this scheme, even distributed page migrations cause more overhead because every page needs to be migrated whereas with migration on one target node, some pages may not need to be migrated as interleaved allocation ensures that at least some pages are on correct node. In sum, initial interleaved allocation seems to be an upper bound on the cost of page migration.

Comparison of move_pages and mbind for performance: The migration time for the same number of pages is drastically different for the two schemes. For example, mbind takes only 0.82 seconds to migrate 130K pages with interleaved allocation (for one node migration), whereas move_pages takes around 74 seconds to do the same. Clearly, there is an order of magnitude of difference here. Migration time with `move_pages()` grows super-linearly in contrast to a linear growth for `mbind()`. We believe that mbind is the better mechanism for migration. The analysis is described below:

Our experimental system, the SunFire 6400, has a HyperTransport memory interconnect with a bidirectional bandwidth of 8GB/sec. To migrate 100K pages, i.e., 400 MB of data (each page is 4096 bytes on our system) to an adjacent node should take 100 msec + costs of migration requests (mgreq) + operating system overhead (os). Since, we migrated from node 0 to node 7 at a distance of 3 hops, the total cost should theoretically be 300+mgreq+os. Experiments show that mbind takes around 614 msec and move_pages around 37.5 seconds. Clearly, mbind seems to be a closer approximation of our theoretical analysis. However, in the absence of superuser privileges on the NUMA system, the kernel could not be debugged. Hence, the high overhead of move_pages() is yet to be identified in future work.

In our wall clock experiments, we use move_pages() to migrate pages (because of its amenable interface). However, from the final results, we subtract the cost of page migration using move_pages() and add the corresponding cost with the worst case of mbind(), i.e., assuming initial interleaved allocation.

Page Migration Overhead: Assuming mbind was representative of page migration overhead, we conclude that page migration does not incur a high cost. The only noticeable overhead is for the FT and MG benchmarks (an average overhead of 3.8% and 1.3% respectively). The main causes of overhead due to page migration are as follows:

- Hardware TLB invalidation: The TLB maintains maps of virtual pages to physical frames, and is a part of the processor core. When data is moved, the original TLB mappings in the home node need to be invalidated. This causes invalidation traffic on the interconnect bus.
- Actual OS page migration overhead: This includes the time during which threads are in frozen state and their rescheduling overhead by the operating system. Also included is the overhead due to recreation of virtual-to-physical mappings in the kernel data structures like page tables.
- Application overhead of serialization: After memory traces are captured by each thread, we need to serialize the processing where only the master thread collects information from other threads, determines affinities and initiate page migration.

Wall Clock Performance: The experimental setup for page migration is the same as the setup for page placement (see Section 4.1.2). All the experiments were conducted with the same NAS and SPEC benchmarks on an eight node Opteron system. For FT, MG and BT, class B inputs are used. All other benchmarks run with class C input. For MG, class B is chosen because MG incurs a lot of references to memory and thus results in high overhead for move_pages(). Figure 4.7 shows the percentage reduction in wall clock time over the original program for the interleaved allocation and our hardware trace-driven page migration approach. We use three variants in our approach:

1. Hop Sensitive Page placement: The latency costs of going through multiple hops is also accounted for.
2. Uniform Latency Policy: This assumes that every remote access is equal in terms of latency. The affinity algorithm is a simple competitive algorithm which gives preference to the NUMA nodes with more read references.
3. Uniform Latency with Filtering: This is based on the assumption that not all page migrations are useful. Since page migration incurs an overhead in terms of TLB invalidation traffic and other internal operating system overheads, we filter out some

infrequently accessed pages. A heuristic proposed by our scheme is given by the Equation 3.1 in Section 3.3.

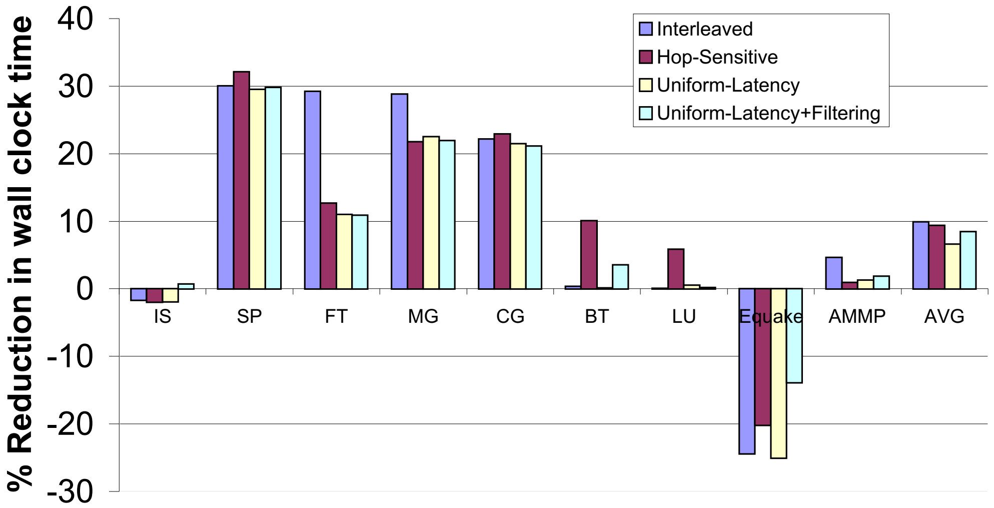


Figure 4.7: Wall Clock Reduction with Dynamic Page Migration and Interleaved Allocation

As can be seen from Figure 4.7, hop-sensitive performs the best for five (SP, FT, CG, BT, LU) of the nine benchmarks. Uniform placement with filtering improves over simple uniform placement in five of the nine benchmarks. On an average, interleaved allocation seems to perform slightly better than our optimizations with an average gain of 9.9% whereas the average improvement of the hop-sensitive scheme is close to 9.4%. The statistics for interleaved allocation look better primarily because of two class B benchmarks, FT and MG. Yet, hop-sensitive page migration performs better with more consistency. Also, cache thrashing could be an artifact of running 16 threads over a shared L2 cache. Thus, it is possible that our trace results are tainted. This is due to hardware limitation. We propose to explore alternative methods of hardware tracing. Infact, as of writing this thesis, we have integrated the IBS (Interrupt Based Sampling) PMU of Barcelona machine (by AMD) in our framework. However, it seems that for IBS, the current support in Linux kernel is still at its infancy. But once we get more accurate hardware traces, our results may improve.

Chapter 5

Related Work

Nikolopoulous *et al.* evaluate various page placement policies for inclusion in the OpenMP run time system and reach the conclusion that the OpenMP run time system should not be changed as the latency difference between remote and local accesses is decreasing in contemporary computer architectures, thus making simple page placements redundant [7]. They also evaluate the performance of the kernel-level page migration engine of IRIX on an SGI Origin 2000. The SGI Origin has set of 11 bit hardware counters for each page and an interrupt mechanism that is triggered when the number of remote hits to a page increases more than the local accesses by a predefined threshold. This interrupt is handled by the Irix kernel, which than initiates page migration if it is enabled through an environment variable. Their evaluation concludes that the kernel-level page migration engine results in modest improvements, if any, in terms of performance. They go on to build a user-level page migration engine [24] based on a combination of compiler support (to identify hot memory regions) and feedback from the operating system (to relay scheduling information to the run time system about thread migration) and dynamic monitoring of memory reference patterns of the program. They propose two page migration algorithms: a predictive algorithm for iterative parallel codes and an aging algorithm for non-iterative codes with hot regions. Both algorithms are based on a competitive criterion and both allow threads to migrate to another node [23]. We contrast our work in following ways: *First*, our framework is based on medium scale contemporary x86 architectures with a maximum of eight nodes as opposed to a large scale, custom-built Origin platform with migration support in the hardware itself. *Second*, we have developed our framework on the *Linux* platform as against *Irix* OS used by them. *Third*, we pin our threads on processors (This can be extended easily to nodes with more than one processor per node using *numactl* interface of Linux). This is a common practice in scientific computing. As a result, we do not need any scheduling feedback from the operating system. *Fourth*, they rely on compiler support for identifying hot memory regions, which are tracked through hardware reference counters for local and remote accesses. We, in contrast, exploit the PEBS hardware performance monitoring unit (PMU) to derive information about the regions causing load misses. The PMU has become an integral part of the hardware as a mechanism to identify performance bottlenecks in hardware and in software. That makes our scheme more appealing to a wider audience. *Finally*, we also use the information of the interconnect topology (inter-node distances) seamlessly as an input to our page migration decisions.

Corbalan *et al.* studied page migration on the SGI Origin and under Irix OS [15]. Their study focuses on a job control environment and correlates the job scheduling policies of the system with page migration support. It concludes that memory migration is not sufficient by itself to improve performance by reducing remote memory accesses but also the scheduling policies and the system load have an impact on overall performance. While we agree with their evaluation, we believe that it is specific to the job control environment on which processor the scheduler may allocate tasks. Our focus of work is high-performance computing where we assume that we can obtain the number of processors requested. We should be able to schedule our threads on selected nodes to obtain better control over application performance.

Tao *et al.* have developed an Adaptive RunTime System (ARS). A user-level engine migrates pages on a simulated environment [14]. It is based on a shared memory architecture built from commodity machines called SMILE (Shared Memory in a LAN like environment) with a standard SCI (Scalable Coherent Interface) interconnect and the HAMSTER programming paradigm. At the time of writing, their hardware monitors were being developed. Hence, they have tested their framework on an SMT simulator. They evaluate the following page migration algorithms: Out-U (a somewhat similar competitive algorithm), Out-W, where the decision to migrate a page to remote node has to be made on the basis of relative remote node references by also counting the weighted references (where weight is based on distance from the page 'P', the target of migration) for neighboring pages on the same node, and In-W, where the decision is made to migrate remote page to the local node. Since the work is based on a simulated environment, we can not directly compare our work with theirs as we have developed our framework on a real hardware and software platform.

Tikir and Hollingsworth also detail the user-level dynamic page migration framework based on memory access traces by sampling the Sun Fire 6800 Fireplane interconnect [22]. They use the *dynist* library to instrument the application code to create additional helper threads for profiling and migration. As a result, they do not require the legacy applications to be recompiled. Their memory traces from Sun Fire Link hardware monitors consist of the physical addresses that are converted to linear addresses through reverse mapping by using `meminfo` system call of Solaris 9 operating system. Pages are migrated using the `madvise` system call in Solaris. In contrast, we use the native hardware PMU support by configuring it with L1/L2 cache miss events. Since our method is *processor-centric*, it is simpler and more general in nature. We do not require any additional support from network interconnects as opposed to them, since their approach is *network-centric*. In their setup, the hardware counters are embedded in the network interconnect and do not distinguish between different processes, i.e., only one application can use them at a time. In contrast, there is no such restriction with our approach. Also, since their method is based on *polling* the network interconnect, it incurs more overhead compared to our scheme, which is interrupt driven, i.e., the PMU raises an interrupt after the configured sampling counter overflows, and the memory buffer is populated with the instruction pointer and contents of other registers. As mentioned before, they need to make a system call to map the physical address obtained through hardware monitors to virtual address, which is a significant overhead in most architectures.

Chandra *et al.* evaluated various scheduling policies along with a page migration policy on the CC-NUMA Stanford DASH machine [5]. They conclude that for sequential applications, affinity scheduling shows good performance if combined with kernel-level page migration based on TLB misses. For parallel applications, the performance improvements shown were modest with affinity scheduling. We also think that affinity of processes to processors are important. We bind our threads to processors to avoid a loss of memory affinity of the threads. However, our scheme is a user-level policy which does not impose any complexity on the kernel by introducing a new scheduling algorithm.

Subsequently, Verghese *et al.* presented an analytical study for page migration [29]. Their study shows that if the pages are shared as read-only across multiple processes (or threads), page replication is a better policy than page migration whereas page migration is useful when the pages are exclusively accessed by one processor. This is primarily because page migration incurs significant overhead of TLB flushing and processor synchronization if the sharing is high. This offsets the benefits of page migration itself. Their study is based on a simulation environment. They describe a kernel-level implementation of dynamic page migration. Our work differs as it is neither simulation nor kernel based and is implemented on contemporary hardware within user space.

Bolosky and Scott propose various page placement policies on two NUMA architectures: one with a global shared memory along with local and remote memories and the other with no global shared memory. Current NUMA processors are based on the latter scheme. They collect traces on a real SMP hardware by single stepping through the processor and handling the trap fault handler. These traces are used offline to analyze different page placement policies based on replication and dynamic programming. This method incurs a significant slow-down of over 200x [4]. These policies were subsequently compared to contemporary kernel-based policies. Our framework, in contrast, is implemented in user space and makes a comparative study between interleaved, first-touch (default policy) and user-space dynamic page migration on the `Linux` operating system. In the Linux domain, it has been decided that the kernel will not enforce any replication or page migration policies . However, it currently does provide with mechanism to migrate pages on a ccNUMA platform.

Chapter 6

Conclusion

Cache coherent NUMA architectures have been studied in detail in the literature. They have come into prominence as SMP systems have hit a wall in performance due to their common shared bus to memory. A greater number of cores exacerbates the situation as there is more contention for shared memory. On top of it, the cache coherence traffic also slows down the executions. NUMA architectures, in contrast, only share memory with the processing cores on a single socket, thereby reducing bus contention and, at the same time, keeping the software compatible by providing a shared virtual address space. However, on these architectures, it is essential to have good page placement as remote memory accesses are costlier by a factor of 1.3 to 2. Consequently, many approaches have been proposed in literature to obtain better page placement, either through policy-based page allocations or through dynamic page migrations. For the latter, studies have been performed in the operating system domain as well as in the user space domain using feedback from custom hardware techniques (e.g. per page hardware counters) or by instrumenting privileged software. Some of these studies have been performed in simulated environments, while others have used large-scale proprietary architectures and custom operating system solutions. Yet, we have developed our dynamic page migration engine entirely in user space using information from hardware performance monitoring units. To the best of our knowledge, this thesis provides the first ever user-space dynamic page migration framework for x86 ccNUMA platforms (AMD opterons) under the Linux operating system.

We measure our work against the hypothesis given in Section 1.3.

- We exploit the PEBS PMU of the Intel Xeon and Core microarchitectures to trace memory references by various threads of OpenMP programs. Since Intel has not yet shipped any NUMA-based machines (though their upcoming Quad Core `Nehalem` is based on NUMA technology), we use these traces on another popular x86 architecture (AMD opterons), to analyze trace-driven page placement policies. We have also evaluated the Precise Event Based Sampling(PEBS) PMU support in Intel Xeon/Core2 micro-architectures. Our experiments show that the hardware traces are lossy at lower sampling intervals. However, we do get precise instructions causing load misses. The results have mostly been modest with an average wall-clock improvements of 10%-15% for various schemes. Some benchmarks show as good a performance as 40% over default page placement, but at least one benchmark (equake) shows a slowdown of 30%. However, this can be attributed to lossy traces and hardware constraints, i.e., parallel threads potentially interfere with each other while sharing limited L2 cache. Also, our setup has a limited number of node count (only 8). The wallclock improvements may increase with a large node count.

- Our page migration framework uses the move_pages() interface to dynamically steer the pages on algorithmically determined NUMA node. The competitive page migration algorithm also takes the NUMA interconnect configuration into account by associating weights to hops, i.e., we experimentally determine the distance between any two pairs of nodes in terms of latency and use this information to derive better page placements. We have also evaluated the Linux kernel support for page migration and found that there could be some software issues with one of the interface implementations (move_pages()) due to which it takes exponentially longer to migrate same number of pages. A theoretical analysis, based on the memory interconnect bandwidth of modern processors, supports the above assertion, which is also backed up by another interface implementations (mbind()) in the Linux kernel. In our experimental results, we have replaced the migration time by move_pages() with an estimated time by mbind() system call.

In sum, may we conclude that user-driven dynamic page migration using hardware PMU traces has limited potential in current processors. But given a good kernel implementation, modern architectures have sufficient NUMA bandwidth and low latency to allow a larger number of page migrations at a reasonable cost.

Bibliography

[1] C versions of nas-2.3 serial programs. http://phase.hpcc.jp/Omni/benchmarks/NPB, 2003.

[2] AMD. *BIOS and Kernel Developer's Guide(BKDG) For AMD Family 10h Processors*. AMD, 2007.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[4] William J. Bolosky and Michael L. Scott. Evaluation of multiprocessor memory systems using off-line optimal behavior. *J. Parallel Distrib. Comput.*, 15(4):382–398, 1992.

[5] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, 1994.

[6] Intel Corp. *The VTune performance analyzer*. Intel. http://www.intel.com/software/products/vtune/.

[7] Constantine D. Polychronopoulos Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Jesús Labarta, and Eduard Ayguadé. A Transparent Runtime Data Distribution Engine for Openmp . volume 8, pages 143–162. IOS Press, August 2000.

[8] Anton Ertl and Brend Paysan. Bplat: A memory latency benchmark. http://www.complang.tuwien.ac.at/anton/bplat/, 2004.

[9] John Levon et al. Oprofile. http://oprofile.sf.net/.

[10] Hewlett-Packard. *Perfmon project.*

[11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide.* Intel, 2007.

[12] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M.* Intel, 02 2008.

[13] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z.* Intel, 02 2008.

[14] Martin Schulz Jie Tao and Wolfgang Karl. Improving data locality using dynamic page migration based on memory access histograms. In *Proceedings of the International Conference on Computational Science-Part II*, pages 933–942. Springer-Verlag, 2002.

[15] Jesus Labarta Julita Corbalan, Xavier Martorell. Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000 . In *International Conference on Supercomputing ,Proceedings of the 17th annual international conference on Supercomputing*, pages 121–129, San Francisco, CA, USA, 2003. ACM.

[16] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, 2003.

[17] Andreas Kleen. *A NUMA API for Linux.*

[18] Christoph Lameter. Page migration. Linux kernel documentation at linux/documentation/vm/page_migration.

[19] Jaydeep Marathe. *Trace Based Performance Characterization and Optimization.* PhD thesis, North Carolina State University, 2007.

[20] Jaydeep Marathe and Frank Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 90–99, New York, NY, USA, 2006. ACM Press.

[21] Jaydeep Marathe, Frank Mueller, and Bronis R. de Supinski. A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks. June 2005.

[22] Jeffrey Hollingsworth Mustafa M. Tikir. Using hardware counters to automatically improve memory performance. In *Supercomputing*, 2004.

[23] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguade. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *International Conference on Parallel Programming*, pages 95–103, August 2000.

[24] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade. UPMLIB: A runtime system for tuning the memory performance of openmp programs on scalable shared-memory multiprocessors. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 85–99, 2000.

[25] Mikael Pettersson. The perfctr interface. http://user.it.uu.se/ mikpe/linux/perfctr/.

[26] SGI. Sgi altix 4700 servers and supercomputers. http://www.sgi.com/products/servers/altix/4000/.

[27] Alex Tsariounov. The prospect monitoring tool. http://prospect.sf.net/.

[28] Knoxville University of Tenessee. Performance application programming interface(papi)project. http://icl.cs.utk.edu/papi/.

[29] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on ccNUMA compute servers. In *Proceedings of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, 1996.

Appendices

Appendix A

Page Migration Implementation in Linux Kernel 2.6.18+

This appendix discusses the implementation of move_pages(), a kernel interface for migrating pages to NUMA nodes. It is descibed in detail in [18] .

1. The list of pages to be migrated are taken off the Least Recently used (LRU) list. This ensures that the swapper daemon can not unmap that page while the page migration is in progress. This is implemented by increasing the reference count of the page.
2. Before migration starts, it is ensured that pending writebacks to the page are complete.
3. A new page is allocated on the target node and all the settings are copied from the old page. The page is locked and its state is set as not "up to date". As a result, all accesses to this page are blocked.
4. Page table references to the page are converted to migration entries or dropped. This decreases the counter that maintains a map count of the page. A page is only migrated if the number of references is zero.
5. Radix tree spinlocks are obtained which block all the processes trying to access this page.
6. The radix tree now points to the new page.
7. A reference to the new page is created and the reference to the old page is dropped.

8. The radix tree lock is removed which makes accesses(to the migrated physical page) through virtual memory possible.
9. The actual content of the old page is copied to the new page.
10. The old page table flags are cleared.
11. The migration PTE is replaced with real PTE. This activates the other processes that are not waiting for page lock.
12. The page locks are released.
13. The new page is moved to the LRU list and can be swapped out of memory if necessary.

Appendix B

PEBS Configuration using libpfm

B.1 Configuring PEBS-based PMUs on the Intel Xeon Architecture using libpfm

The Intel Xeon and Pentium 4 have similar support for PEBS. In this work, we got our hardware traces on a 32 bit Intel Xeon machine.

1. **Creating Perfmon context**: The PMU state includes the values of the PMC and PMD registers and possibly other related registers. A Perfmon context can be defined as the state of the PMU hardware and the associated software component. The libpfm library provides an interface to allow users to create and modify this context. But at the same time, the internal structure of the context is specific to the PMU and the operating system, and is never directly exposed to an application.

   It is created by the following function:

   *int pfm_create_context(pfarg_ctx_t *ctx, void *smpl_arg, size_t smpl_size);*

   pfarg_ctx_t structure is defined as:

   typedef struct {

   unsigned char ctx_smpl_buf_id[16]; /* which buffer format to use */

   uint32_t ctx_flags; /* noblock/block/syswide */

   int32_t ctx_fd; /* ret arg: fd for context */

   uint64_t ctx_smpl_buf_size; /* ret arg: actual buffer sz */

uint64_t ctx_reserved3[12]; /* for future use */

} pfarg_ctx_t;

Here, Perfmon keeps compatibility with various buffer formats used by the existing tools. These formats enable various existing tools to represent the information they need in a more efficient manner. For instance, some tools want to keep the samples sequentially ordered while others may need to aggregate identical samples. Furthermore, some tools, for ease of their internal debugging purpose or as a part of an enhancement may also want to record additional information that is not coming from the PMU itself, such as the amount of free memory or the number of active processes, the current thread identifier and so on. To handle a variety of formats, a classic operating system principle, the "separation of mechanism from policy", is used. Basically, tools are given freedom to implement their own formats but at the same time a default sequential sampling format is also provided by the interface. The design provides hooks for the tools to create their own call-backs, which are invoked by Perfmon on specific events. In particular, there is a mandatory call-back handler for counter overflow. The handler can record whatever information it needs into whatever format it wants. PEBS information is also recorded in a particular format that is uniquely identified by a format key. We pass this information in the *ctx_smpl_buf_id[]* string in the above structure.

ctx_flags describes the properties of the context. A context can support per-thread monitoring (default) or system-wide monitoring, i.e., each core is independently monitored by newly spawned threads for monitoring purposes. System-wide support (See Figure B.1) requires that these monitoring threads be pinned to the CPU core and that each thread perform self monitoring, i.e., notifications sent due to events generated by a thread's execution should go to the same thread. Since we run OpenMP parallel benchmarks in our work and we explicitly pin our threads to the CPU cores, we, in essence, perform system wide monitoring. However, we do not need to pass any flag for this purpose. Other possible flags are PFM_FL_NOTIFY_BLOCK and PFM_FL_OVFL_NO_MSG. PFM_FL_NOTIFY_BLOCK indicates that the thread being monitored should be blocked during an overflow notification. This flag is only valid for a non self-monitoring per-thread session. The default behavior is to let the monitored thread run while the overflow notification is processed. Since we perform self monitoring, we do not use this flag as this can lead to a deadlock where the thread receiving the asynchronous notification can itself get blocked, thereby preventing the execution of the signal handler itself. PFM_FL_OVFL_NO_MSG indicates that the application is not interested in receiving overflow notification messages. By default, one message is generated for every notification. This flag is also not set in our implementation.

ctx_fd is the descriptor that identifies the context. It is set to a positive integer if the call to *pfm_create_context()* is successful.

ctx_smpl_buf_size is valid only on successful return, i.e., when the selected sampling buffer format exists and uses the buffer re-mapping service, this field contains the actual size in bytes of the buffer.

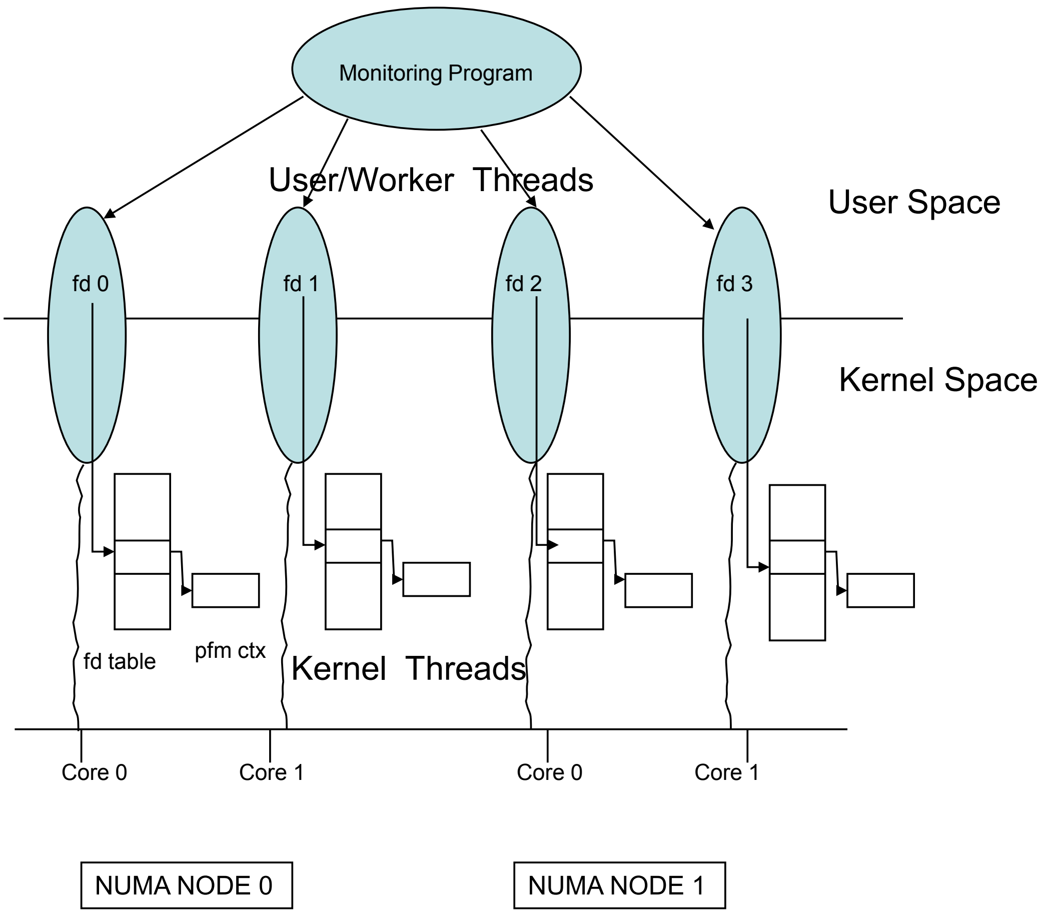


Figure B.1: Monitoring across multiple cpu cores

Sample format-specific parameters (e.g., PEBS format specific) are passed by 'smpl_arg' parameter. The P4/Xeon specific format is defined as:

```
typedef struct {
size_t buf_size; /* size of the buffer in bytes */
size_t intr_thres; /* index of interrupt threshold entry */
uint32_t flags; /* buffer specific flags */
uint64_t cnt_reset; /* counter reset value */
uint32_t res1; /* for future use */
uint64_t reserved[2]; /* for future use */
} pfm_p4_pebs_smpl_arg_t;
```

PEBS writes the records in a buffer. Its size can be configured by 'buf_size' member of the above structure. An asynchronous notification may be generated once a certain fraction of this buffer is filled. This fraction is given by the 'intr_thres' member. We configure it as 90% of the total number of PEBS records (40 bytes) that can be placed in this buffer.

2. **Remapping the buffer to user space**: The kernel-level sampling buffer is remapped in the virtual address space of the user-level application. This is accomplished using the 'mmap()' system call. Without the mmap() call, the buffer exists and samples are stored into it but they remain totally inaccessible from the user-level. The sequence of calls is as follows:

```
pfm_create_context(&ctx, &buf_arg, sizeof(buf_arg)) ;
pfm_fd=ctx.ctx_fd;
. . .
(smpl_hdr_t *)mmap(NULL, ctx.ctx_smpl_buf_size, PROT_READ, MAP_PRIVATE,
pfm_fd, 0);
```

The buffer is of fixed size given by *ctx.ctx_smpl_buf_size* and is always exported as read-only. As such, only PROT_READ is valid. Any write access is disallowed by the virtual memory subsystem and may result in the termination of the controlling process. The effect of mmap is shown in Figure B.2.

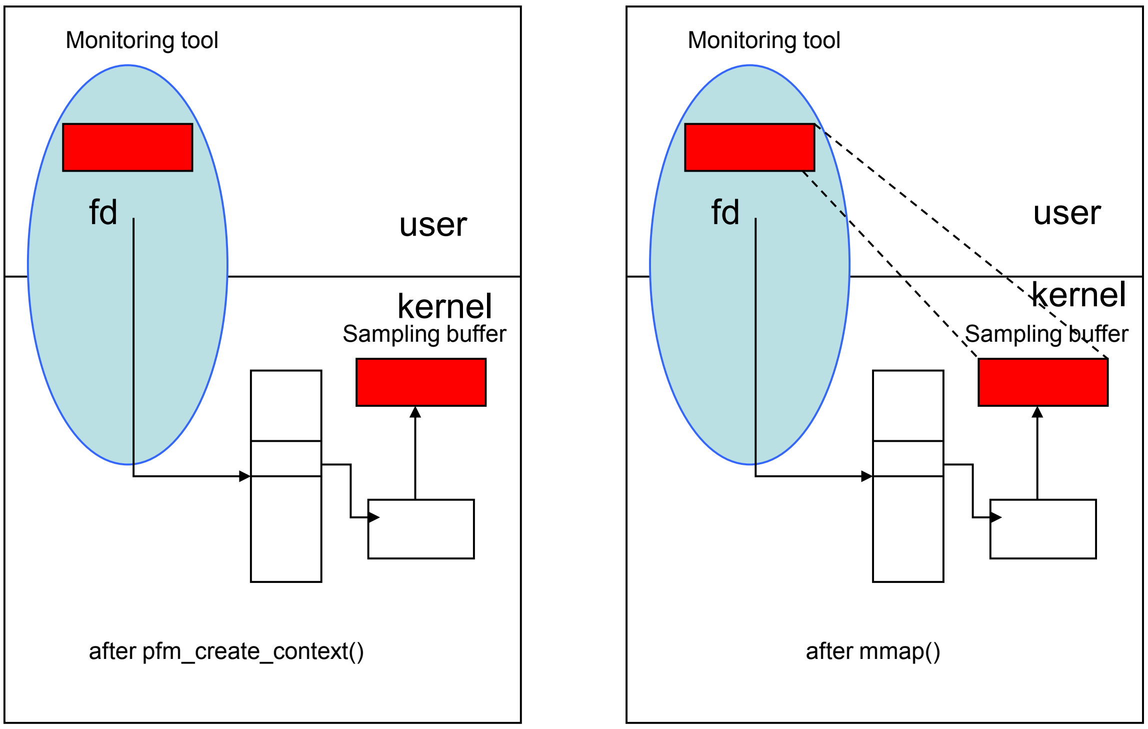


Figure B.2: Memory mapping the sampling buffer

The PEBS format for the P4/Xeon exports a header that starts at the beginning of the sampling buffer returned to the user. The header is declared as follows:

typedef struct {

uint64_t hdr_overflows; /* # overflows for buffer */

size_t hdr_buf_size; /* bytes in the buffer */

size_t hdr_start_offs; /* actual buffer start offset */

uint32_t hdr_version; /* smpl format version */

uint64_t hdr_res[3]; /* for future use */

pfm_p4_ds_area_t hdr_ds; /* DS management Area */

} pfm_p4_pebs_smpl_hdr_t;

Because of PEBS alignment constraints, the actual PEBS buffer area does not necessarily begin right after the header. The *hdr_start_offs* is used to compute the first byte of the buffer. The offset is defined as the number of bytes between the end of the header and the beginning of the buffer. The following expression determines the start of the buffer:

actual_buffer = (unsigned long)(hdr+1)+hdr->hdr_start_offs

3. **Setting up PMC for L1/L2 misses**: Perfmon2 exposes the logical PMU to the users. These logical registers are then mapped to the actual hardware registers by the Perfmon2 implementation. We now describe the physical PMU configuration needed to setup the Performance Monitoring Control (PMC) registers and how this is actually accomplished using *libpfm* support.

   Selecting Replay Event: As mentioned in Section 2.2 , we use the replay tagging mechanism for tracking L1/L2 misses. There are a total of 45 Event Select Control Register (ESCR) MSRs in the PEBS PMU of the Intel Xeon that can be written to by the software to count specific events. Associated with each ESCR is a pair of performance counters and each performance counter can count events selected by multiple ESCRs. The format of ESCR MSR is shown in Figure B.3.

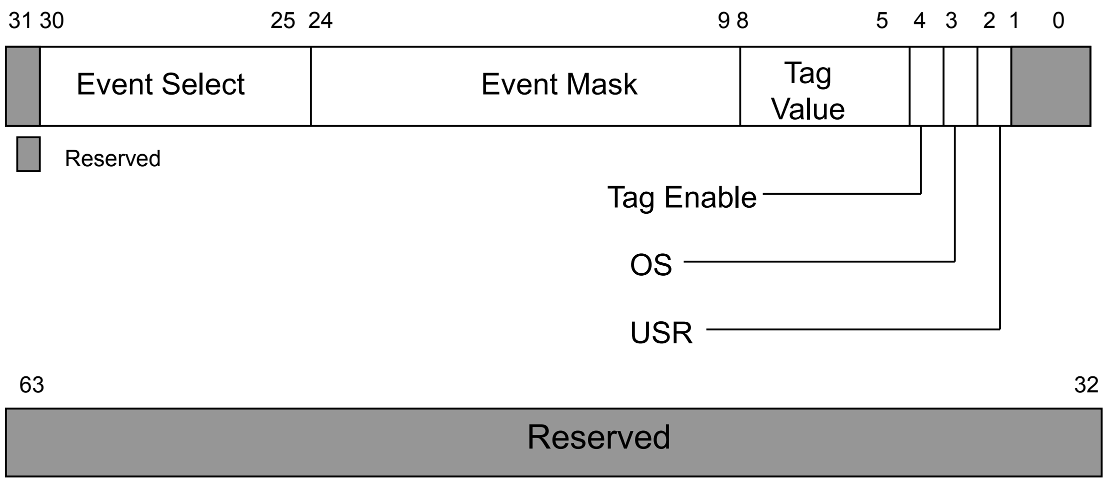


Figure B.3: Event Select and Control Registers for Pentium4 and Intel Xeon Processors

The following flags and fields are relevant to us:

- USR flag: If set, the events are counted when processor is running at the current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code. This needs to be set.

- Event mask field, bits 9 through 24: Selects specific events from an event class that is specified by the event select field. We set a mask of 1 which signifies that the marked micro-ops are “Non Bogus” (NBOGUS).
- Event Select field, bits 25 through 30: Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field. For the replay event, a value of 0x9 needs to be assigned to it.

libpfm configuration

The MSR_CRU_ESCR2 register is mapped to logical register 21 (See lib/pentium4_events.h). Through libpfm, it is configured as:

npmcs = 0 ;

pc[npmcs].reg_num = 21;

To set the values in the above fields/flags, we do the following operation:

pc[npmcs].reg_value = (9ULL << 25) | (1ULL << 9) | (1ULL << 2);

npmcs++ ;

Counter Configuration Control Register (CCCR) Setup: Each performance counter has an associated control register called CCCR. These registers can enable or disable counting and also associate events with the counters. The following flags and fields are relevant for the current work:

- ESCR select field, bits 13 - 15: Selects events to be counted by identifying the ESCR associated with the CCCR. It gets the value 0x5 (refer to Table A-7 of [11]).
- Enable flag, bit 12: We need to set this flag to enable counting L1/L2 cache misses.
- Reserved bits 16,17: These must be set to 11_b as shown in the x86 manual [11].

libpfm configuration

For PEBS, the MSR_IQ_CCC4 register is used. This is mapped to the PMC register number 31. Here is how we configure CCCR using libpfm:

pc[npmcs].reg_num = 31;

pc[npmcs].reg_flags = PFM_REGFL_NO_EMUL64;

pc[npmcs].reg_value = (5ULL<<13) | (1ULL<<12) | (3ULL<<16);

npmcs++ ;

With PFM_REGFL_NO_EMUL64 flags, 64-bit emulation is disabled by the kernel on the associated counter. Otherwise, spurious interrupts get generated for every counter overflow.

Enabling PEBS for L1/L2: Additional MSRs are required for replay tagging for L1/L2 cache miss (refer Table A11 in [11]). In MSR_PEBS_MATRIXD_VERT, bit 0 needs to be set for both L1 and L2 cache load miss metrics. This is configured as follows:

pc[npmcs].reg_num = 63;

pc[npmcs].reg_value = 1;

npmcs++ ;

In MSR_IA32_PEBS_ENABLE, for L1 load miss, bits 0 (PEBS enable bit), 24 and 25 need to be set, whereas for L2 load miss, bits 1, 24 and 25 need to be set. This is done as follows:

```
pc[npmcs].reg_num   = 64;

if(prop_pebs_source == pebs_source_l2)
    {
       pc[npmcs].reg_value = (1ULL<<25)|(1ULL<<24)| (1ULL << 1) ;

    }
    else
    {
       pc[npmcs].reg_value = (1ULL<<25)|(1ULL<<24)| 1ULL ;
   }
```

npmcs++ ;

To program the registers, the following library function needs to be called:

*int pfm_write_pmcs(int fd, pfarg_pmc_t *pmcs, int count)*

Fd is the context descriptor returned after a call to *pfm_create_context()*, control registers and their values are passed through the 'pmcs' data structure, and count is the number of PMC registers to be programmed.

4. **Setting up PMD registers**:

   The logical PMU also exposes Performance Monitoring data (PMD) registers that contain the actual information or counter values useful for the monitoring tools. We now describe various PMD registers needed by our framework.

   Event Notification:

   Perfmon manages a first-in, first-out (FIFO) queue of messages that can be used to notify an application when a counter overflows. A message with the following structure is appended to the end of the queue when a counter overflows or a monitored thread terminates:

```
typedef union {
        uint32_t        type;
        pfm_ovfl_msg_t  pfm_ovfl_msg;
} pfm_msg_t;
```

   The message can be read by using the "read" I/O system call on Perfmon's file descriptor. Only one message queue is maintained per context.

   The following types of messages are defined:

   - PFM_MSG_OVFL: overflow notification message. It is associated with the pfm_ovfl_msg_t structure.
   - PFM_MSG_END: termination message. It is not associated with any particular message type.

   We need to pass a flag 'PFM_REGFL_OVFL_NOTIFY' to get an overflow notification (message of type PFM_MSG_OVFL). This is done as follows:

   pd[0].reg_num = 8; /* Selecting IQ_CTR4 register */

   pd[0].reg_flags = PFM_REGFL_OVFL_NOTIFY;

Sampling Period:

For event-based sampling, a sampling period is expressed as a number of occurrences of an event. The PMU needs to be informed of this sampling period. As it turns out, most PMUs can not detect a user defined threshold for recording number of events. For example, it does not work to program a value of 1000 to configure a PMU to record a sample after every 1000 events. The way it works with modern PMUs is, that a counter needs to be set with a value equal to maximum counter value minus the sampling interval since the counter logic would only increment the counter after every occurrence of the specified event. After the counter reaches a maximum value, an overflow would occur that can be detected by the hardware (as an overflow implicitly causes setting of a flag that can be checked easily by the hardware) and an interrupt is raised. The kernel catches the interrupt and notifies the monitoring tool. Using this mechanism, a sampling period is expressed as an offset from the maximum value of the counter. In our implementation, we read the sampling period from a configuration file. From the point of view of a monitoring tool, counters are always 64 bits. Therefore a sampling period 'p' is always expressed as:

pmd value = 2^{64} - p = ~0 - p - 1 = -p

The interface provides three sampling periods per counter. They are defined as follows:

- *The current period* is specified in the *reg_value field.* It represents the current sampling period. When the context is detached, this is actually the initial sampling period to be used.
- *The short period* is specified in the *reg_short_reset* field. It represents the sampling period to reload into the PMD register after an overflow that does not trigger a user-level notification.
- *The long period* is specified in the *reg_long_reset* field. It represents the sampling period to reload into the PMD register during a call to restart monitoring (via pfm_restart()).

The following 'C' statement assigns the sampling period:

pd[0].reg_value = -prop_pebs_period;

It was found that a sampling interval of 10 is a good compromise. A sampling interval of '1' is too lossy, whereas higher sampling intervals can miss a lot of events.

pd[0].reg_long_reset = -prop_pebs_period;

pd[0].reg_short_reset = -prop_pebs_period;

To program the pmd registers following function is called:

*int pfm_write_pmds(int fd, pfarg_pmd_t *pmds, int count) ;*

5. **Getting ownership of the descriptor and setting up asynchronous notification**:

   Our framework is intended to provide library support for OpenMP parallel applications. While tracing, the application is instrumented to configure, start and stop the monitoring. It is usually a good choice to be informed of the available records written by the PMU rather than polling the memory buffer. This allows the application to keep doing its main job for most of the time and thus reduces the overhead of monitoring. The Perfmon2 implementation facilitates this by allowing us to request a SIGIO signal when a message is appended to the message queue. The setup follows the regular procedure to request asynchronous notification on a file descriptor. On Linux, for instance, the thread to be traced uses the fcntl() system call to:

   - request that the file be put in asynchronous notification mode using the O_ASYNC flag and to
   - request ownership of the descriptor using the F_SETOWN command.

6. **Loading the Perfmon context**: Perfmon context can be loaded by the following function:

   *int pfm_load_context(int fd, pfarg_load_t *load);*

   This function returns error if the context could not be loaded, else it returns 0. The command applies to the context identified by 'fd'. The descriptor must identify a valid context. The command takes one argument of type 'pfarg_load_t' pointed to by the argument 'load'. For our purpose, only one field in 'load' is important, namely 'load_pid', which is set to the thread-id of the controlling thread. On Linux, this is the value returned by *gettid()* on NPTL-enabled (Native Posix Thread Library) systems.

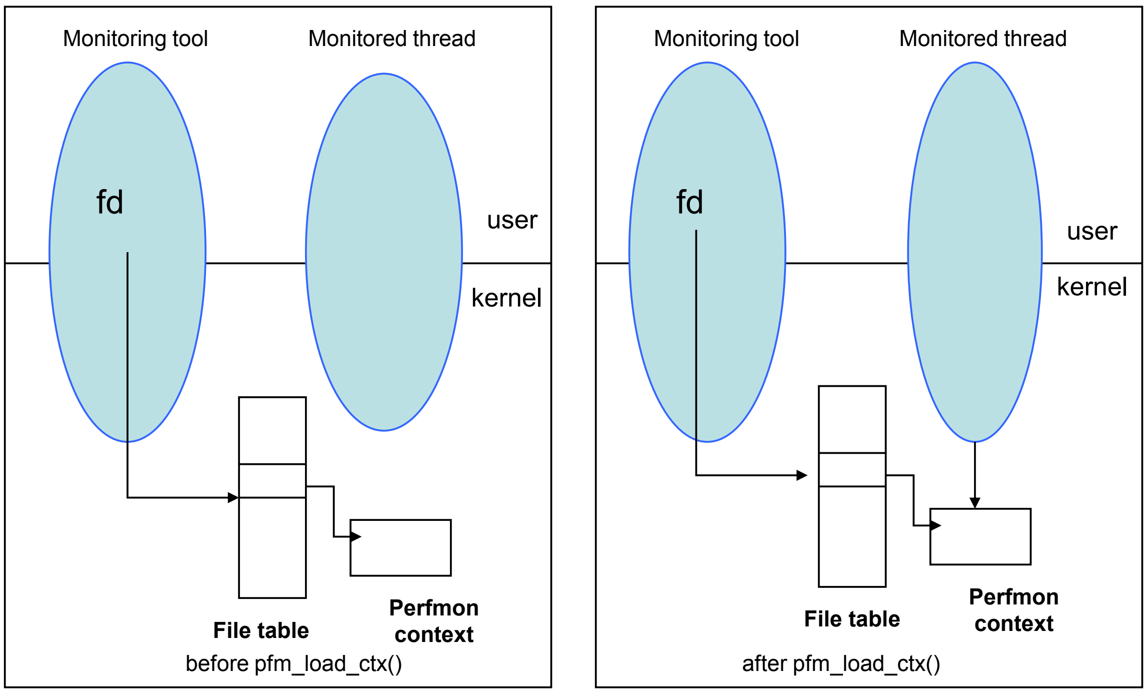


Figure B.4: Effect of pfm_load_context

Loading the Perfmon context is necessary to monitor a specific thread since it loads the software state of the designated event set onto the actual PMC and PMD registers. The thread to which the context is attached is called *monitored thread* and the thread that does this attachment is called *controlling thread.* For *self-monitoring threads*, the controlling thread is the monitored thread (see Figure B.4).

With our framework, each thread is explicitly pinned on a CPU core, and each thread performs self monitoring. For PEBS, we profile one thread at a time, i.e., we run the trace execution run 'n' number of times, where 'n' is the number of threads of OpenMP parallel benchmarks. This is because the traces were obtained on a machine with fewer cores than threads, and it was observed that overflow notification signal meant that a thread was interrupted by another thread sharing the processor. This defeats the purpose of self-monitoring, which is experimentally found to be more stable. Also, the truncated runs ran longer compared to the runs on Itanium2 processors (in previous work) as PEBS traces on the Xeon took more runs before becoming representative of memory accesses. However, this is not a limitation of our design. If we had full hardware resources (number of cores >= number of threads and stable PEBS traces), we could have performed tracing along the lines of the trace design on Itanium2 machines [20].

B.2 Configuring PEBS-based PMU on Core2 Machine using libpfm

For Core2 Duo machines, a later version of *libpfm, libpfm-3.2-071017* is used. Configuration steps are enumerated below.

1. **Initialize Perfmon library**: A call to pfm_initialize() is made. It detects the PMU support for the machine and reports error if the internal sanity tests fail.

2. **Create Perfmon context**: This step is same as in Section B.1.

3. **Search for an L1/L2 miss event**: The latest libpfm provides a very useful function:

   *int pfm_find_full_event(const char *str, pfmlib_event_t *ev) ;*

   An event name is passed in one of the fields of the 'ev' structure. This function searches for the corresponding event descriptor and event mask (if needed). These fields along with the number of masks are populated in the fields of 'ev' as a result of this function call.

   For L2 and L1 cache misses, the following strings are respectively searched:

   (a) MEM_LOAD_RETIRED:L2_MISS and

   (b) MEM_LOAD_RETIRED:L1D_MISS.

4. **Remapping the buffer to user space**: This is same as in Section B.1.

5. **Get the PMC and PMD register values from libpfm**:

   This library makes it easy for the users to get the values of PMC and PMD registers for the 'pfmlib_event_t' returned by the 'pfm_find_full_event' data structure. This is accomplished through the following function:

   *int pfm_dispatch_events(pfmlib_input_param_t *p, void *mod_in, pfmlib_output_param_t *q,void *mod_out);*

The input arguments are divided into two categories: the generic arguments in *p* and the optional PMU model specific arguments in *mod_in*. The same applies for the output arguments: *q* contains the generic output arguments and *mod_out* the optional PMU model specific arguments.

In *p*, we pass the event count (1 because we measure either L1 or L2 miss event in test run) and the privilege level of the code to be traced. Since we are interested in tracing user-level application code, we pass PFM_PLM3 (least privilege level 3) in the *pfp_dfl_plm* flag of *p*. Through *mod_in*, we inform libpfm that we are using PEBS support of the Core2 PMU.

The generic output parameters contains the register index and values for the PMC and PMD registers to obtain the measurement.

6. **Program PMC and PMD registers**: This step is same as in Section B.1.
7. **Getting ownership of the descriptor and setting up asynchronous notification**.
8. **Load Perfmon Context**: This is also same as in Section B.1.

Appendix C

The Instruction Decoder (Parser)

This is the most important module in our framework. It takes the instruction pointer (of cache load miss instructions), and the contents of other architectural registers as input and gives the *effective address* as the output after decoding the instructions. The major components of its design are as follows:

1. **Parser Initialization**: The function parser_init() initializes the parser. This function reads the "/proc/<process id>/maps" file for code segments. An example of a map file is shown below:

```
00400000-00555000 r-xp 00000000 00:15 787656     /home/b
zcmuell/NAS_RUNS/mg.C
00755000-00763000 rw-p 00155000 00:15 787656     /home/b
zcmuell/NAS_RUNS/mg.C
00763000-0077e000 rw-p 00763000 00:00 0
123c1000-3fff4000 rw-p 123c1000 00:00 0
40000000-40001000 ---p 40000000 00:00 0
40001000-40a01000 rw-p 40001000 00:00 0
40a01000-40a02000 ---p 40a01000 00:00 0
40a02000-41402000 rw-p 40a02000 00:00 0
41402000-41403000 ---p 41402000 00:00 0
41403000-41e03000 rw-p 41403000 00:00 0
41e03000-41e04000 ---p 41e03000 00:00 0
41e04000-42804000 rw-p 41e04000 00:00 0
```

```
42804000-42805000 ---p 42804000 00:00 0
42805000-43205000 rw-p 42805000 00:00 0
43205000-43206000 ---p 43205000 00:00 0
43206000-43c06000 rw-p 43206000 00:00 0
```

...

In Linux, the code segment is usually mapped to the lower addresses (upper view in the maps file). We scan at most first 50 lines to get the lower and upper value of the address ranges and store it in an array of a range structure. Since these ranges are arranged in increasing order, we coalesce two or more ranges if the upper value of one range is the same as the lower value of the next range. This is a small optimization for fast lookup of instruction pointers.

2. **Validating the Instructions**

   The instruction pointers from the Perfmon traces are validated for correctness. Either of these failure cases are possible:

   (a) The instruction pointer (IP) is not in a valid range as verified by comparing it in the address ranges calculated above. An error code of 1 is returned for this case.

   (b) The opcode of the instruction is not understood by our parser. The x86 architecture is a complex instruction set architecture (CISC) that has multiple instructions referring to memory. It is challenging to decode each and every instruction, mostly due to the variable-length instruction format. Each test benchmark is run to determine the load miss instructions, and the corresponding opcode is supported in the parser. However, some opcodes which are not that frequently seen (by the instructions causing load miss) are neglected. An error code of 2 is returned for this case, and these instructions with their opcodes are printed out on standard output.

   (c) Due to some PMU bugs, some invalid instructions that may not refer to memory (do not involve a load μop) may be traced. For these, we evaluate 0 as the effective address and return an error code of 3.

   Internal counters are incremented for all the above cases. Invalid instructions are filtered out after this step.

3. **Parsing Valid Instructions**: This component is very specific to the x86 ISA. In the current code base, support for the x86-64 (IA-32e) is also provided in addition to the support for x86 (IA-32) Instruction Set Architecture (ISA). Manuals [12] and [13] can be consulted to decode the instructions.

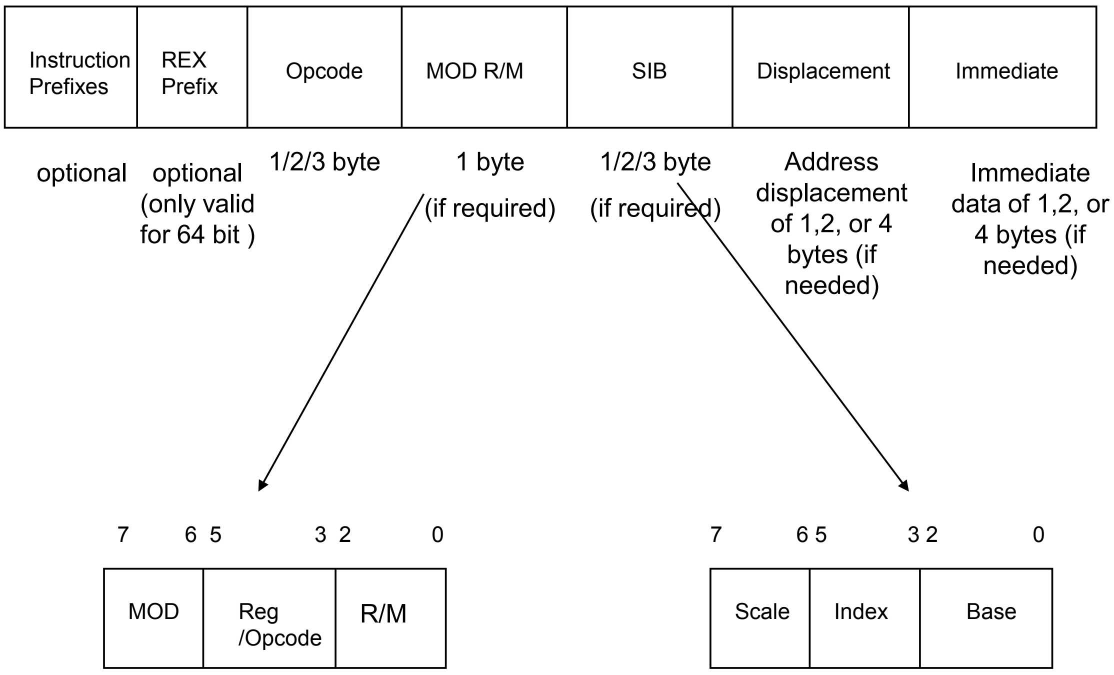


Figure C.5: Instruction format for Intel's EMT-64 and IA-32 architectures

Figure C.5 shows the instruction format for a 32bit/64 bit architecture. The REX prefix is not present in 32 bit processors. The following steps are needed to determine the effective address of a memory access from an instruction:

(a) **Remove Instruction Prefixes**: There are four groups of instruction prefixes. At most one prefix from each group may be used by an instruction. The following opcodes are checked to verify if instruction prefixes are present:

- Group1: Lock and Repeat prefixes: 0xf0, 0xf2, 0xf3
- Group2: Segment override prefixes: 0x2e, 0x36, 0x3e, 0x26, 0x64, 0x65
- Group3: Operand size override prefixes: 0x66, 0x0f
- Group4: Address size override prefixes: 0x67

A detailed explanation can be found in [13]. Function *legacy_prefix_check(unsigned char **ip_ptr)* takes an IP and advances the IP beyond the instruction prefix part of the instruction (if instruction prefix is present) .

(b) **Filtering REX prefix if present**:

The IA-32e (e.g., Core2 machine is 64 bit or IA-32e) mode has two sub-modes. These are:

- Compatibility Mode: This is the mode used by a 64-bit operating system to run legacy software written for a 32 bit architecture without any modifications.
- 64-Bit Mode: This mode is used by a 64-bit operating system to run native applications written to access a 64-bit address space.

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- specify GPRs and SSE registers;
- specify 64-bit operand size; and
- specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. REX prefixes are opcodes in the range of 40H-4FH. We filter out REX prefix bytes by using the following function:

int rex_present(unsigned char rex_byte, uint8_t rex_val) ;*

This function returns a boolean value 1 if the REX prefix is present, otherwise 0. If present, it returns the prefix in argument 'rex_val' . We increment the instruction pointer by 1 byte. Sometimes, the compiler may add a legacy prefix (instruction prefixes of group 1,2,3 or 4) after the REX prefix, which we may need to filter out by calling *legacy_prefix_present()*.

The same opcodes for the REX prefix are available as INC/DEC instructions in the 32 bit mode (IA-32) and there is no REX prefix there. But even in that case, this assumption of the REX prefix range from 40H-4FH holds because INC/DEC instructions do not have any memory operands and, thus, would never be traced by the PMU configured for L1/L2 load miss events.

(c) **Identifying 'load' memory instructions**: By experimental runs of our target benchmarks, some instructions (and their opcodes) have been identified that reference memory quite often. These instruction opcodes are enumerated in our framework as follows:

```
typedef enum \{
    FLDL=1,
    FLDCW,
    MOV_A1,
    MOV_8B,
...
    CMP_XCHNG_B1, // Compare and exchange
   // instruction also references memory.
    XADD_C1
} Linst;
```

After all the prefixes have been filtered out, the IP now points to the opcode of the instruction. The function *get_load_ins()* compares the instruction opcode against the predefined opcodes. If a match is found, the appropriate enumerated opcode (as shown above) of the instruction is returned, else 0 is returned, which implies that the opcode was not found. The actual IP and unsupported opcode is printed out on standard output for further analysis.

(d) **Filling 'Instr' structure**: The following structure defines a valid instruction for us:

```
typedef struct
{
    unsigned int opcode;
    short int mod_rorm ; // ModR/M byte... one byte extra here
    short int sib ; //scaled index... one byte extra
    unsigned int displ; //Displacement
    unsigned int imm; // immediate
} Instr ;
```

These fields are described below:

- *opcode*: This field stores the primary opcode of the instruction. It can be 1, 2 or 3 bytes. If it is 2 or 3 bytes, it contains some legacy prefix that is filtered out in previous steps. We only store the last byte of the opcode in this field.
- *mod_rorm byte and sib byte*: The x86 architecture supports various addressing modes for instructions referring to an operand in the memory. It contains the following three fields of information (see Table C.5).
  - The 'mod' field identifies types of addressing modes. Combined with 'r/m' field, this identifies an exact addressing mode. For example, if mod is 01_b, the effective address would be of type [<reg_value>] + one byte displacement. But if mod is 10_b, the effective address is of type [<reg value>] + four byte displacement. Here, 'reg value' is the base address stored in the general purpose register defined by 'r/m' field of ModR/M byte.
  - The 'reg/opcode' field specifies either a register number or three more bits of opcode information. It is not used in the calculation of the effective address even though the purpose of the reg/opcode field is specified in the primary opcode.
  - If mod is 11_b, the value of r/m specifies a register as an operand, else it helps to identify a register that contains the base address for a specific addressing mode.

Table C.1: Example of use of MODR/M byte

MOD	R/M	Effective Address (EA)
11	001	ECX
10	001	[ECX]+disp32
01	001	[ECX+disp8]
00	001	[ECX]

Table C.1 depicts the use of the MODR/M byte in evaluating the effective address. Note that in the first row, the ECX register is the operand and there is no effective address. In second row, a 32 bit displacement is added to the contents of the ECX register to calculate the effective address (EA) whereas 8 bit displacement is added to the contents of the ECX register in the third row to calculate the effective address. In the fourth row, the contents of the ECX register indicates the effective address. This is how the MODR/M generates various addressing modes.

A second addressing byte (SIB byte) may be needed with certain encodings of ModR/M byte. The SIB byte includes the following fields:

- The scale field (SS) specifies the scale factor, i.e., the multiplication factor that is used to calculate the scaled index.
- The index field (Index) specifies the register number of the index register.
- The base field specifies the register number of the base register.

Figure C.5 depicts the bit positions in the SIB byte for the above fields. The following example shows how the 'scaled index' and 'base' fields are used in calculating effective address:

Table C.2: Example of use of SIB byte

MOD	SS	Index	Base	Effective Address (EA)
00	00	101	000	[EAX] + [EBP]
10	01	101	000	[EAX] + [EBP]*2 + disp8
11	10	101	000	[EAX] + [EBP]*4 + disp32

The multiplication factor of two and four in the second and third column is due to the scale factor(SS). [EAX] is the base address because the base field is 000_b.

These instruction encodings are explained in detail in [12] .

For our implementation, the enumerated opcode received from the previous step helps to identify if the instruction has the ModR/M byte present. Most of the instructions apart from 'MOV_A1' have the ModR/M byte present.

The SIB byte is determined by calling a function

check_sib(Instr lmiss, unsigned char* ip).*

This function checks if R/M bits of the MODR/M byte have only bit 2 set, i.e.,

if ((sib & 0x7) == 0x4) { lmiss-> sib = *ip }

- *Displacement And Immediate Bytes*: Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes. If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes. However, they do not contribute to the effective address calculation. Hence, we ignore them.

In our implementation, we always read the next 4 bytes, and depending on the type of instruction, the extra bytes may be discarded. Also, a *Little Endian* architecture is assumed where the least significant byte of a word is stored at the lowest address. An implication of this assumption is that we need to left shift the higher bytes to get a correct displacement value.

(e) **Calculate the Effective Address**: After filling the 'Instr' structure, the effective address is computed. The prototype of the function for this operation is:

uint64_t calc_effective_add(Instr instr, smpl_entry_t* ent,uint8_t rex_val)*

It takes instruction 'instr', the contents of general purpose and other registers returned by the PEBS record format in 'ent' and the REX prefix in 'rex_val'. For instructions that do not have any REX prefix, a value of 0 is passed in 'rex_val'.

Effective Address Calculation for 32 bit instructions (No REX Prefix)

As mentioned before, the two most significant bits in the ModR/M byte represent MOD, which identifies the basic addressing mode used by the instruction. A binary MOD value of '11' is not relevant for us as it represents a register addressing mode. Load miss instructions inherently refer memory. Fields MOD and R/M are extracted from the MODR/M byte using bitwise operations. An internal table (implemented through a switch-case construct) is first indexed by MOD and then by R/M to calculate the effective address based on Table 2.2 shown in Intel's software manual [12] .

We may need to decode the SIB byte if R/M is 100_b. The following function is called to decode SIB byte.

unsigned long decode_sib(int mod,int sib,smpl_entry_t ent,unsigned int disp, int *is_displ_added,uint8_t rex_val) ;*

MOD is passed in the 'mod' argument. The architectural register state is stored in the 'ent' argument. At most 4 bytes of displacement are given by disp. If this function adds displacement, a flag (is_displ_added) is set to 1. For 64 bit addressing modes, a 'rex_val' argument is also passed (as explained in next Section).

The base field in the SIB byte is extracted using bitwise operations. This identifies the base register. Then SS and index fields are also extracted. These fields are used to identify the scaled index (using switch-case block). The scaled index is added to the contents of the base register identified above. However, there is an exception. If base field has a value 0x5H, the addressing modes are given by Table C.3.

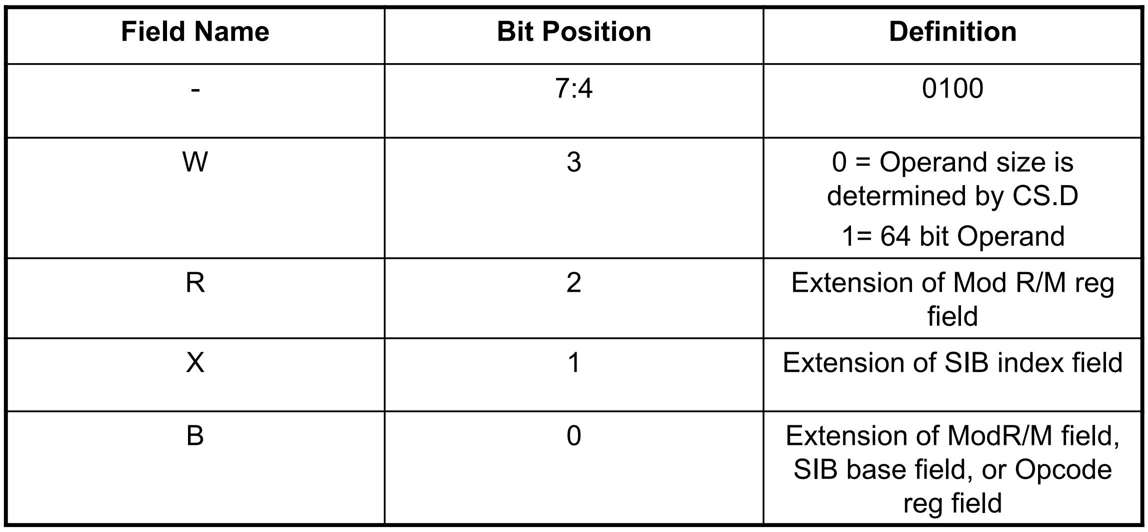
Table C.3: Decoding SIB byte if base = 0x5

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

In these cases, the above function calculates and returns the effective address and sets the is_displ_added flag to 1 (true). This flag then identifies if further addition of displacement values is required to calculate the effective address.

Effective Address Calculation for 64 bit addresses

Recall from 3b that the REX prefix helps in the identification of extra registers (R8-R15). The format of the REX prefix is shown in Figure C.6.



Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size is determined by CS.D 1= 64 bit Operand
R	2	Extension of Mod R/M reg field
X	1	Extension of SIB index field
B	0	Extension of ModR/M field, SIB base field, or Opcode reg field

Figure C.6: REX prefix fields

Bits 'X' and 'B' are used for extending registers. If SIB is not present (Figure

C.7), bit 'X' is not used. Bit 'B' is then used to extend the R/M flag of the MODR/M byte. So, R/M now becomes 4 bits wide, which means that it can now identify 16 unique combinations of effective addresses for an addressing mode identified by MOD bits (remember that MOD=11_b/ specifies the register addressing mode that is not represented by a load miss instruction).

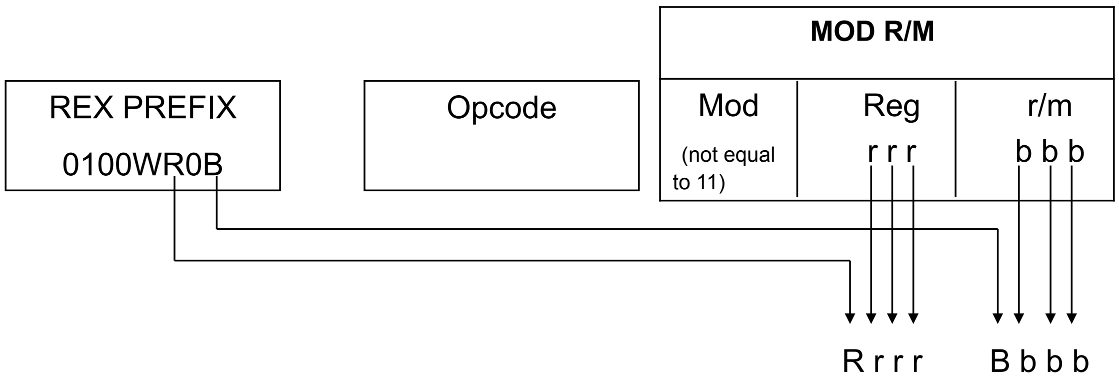


Figure C.7: Memory Addressing when SIB byte is not present

If the SIB byte is present (Figure C.8), bit 'B' is not prefixed to the R/M flag. Rather, it is prefixed to the base field of the SIB byte, thus identifying more base registers (R8-R15). The 'X' bit is now prefixed to the index field of the SIB byte, thus identifying 8 more scaled index values.

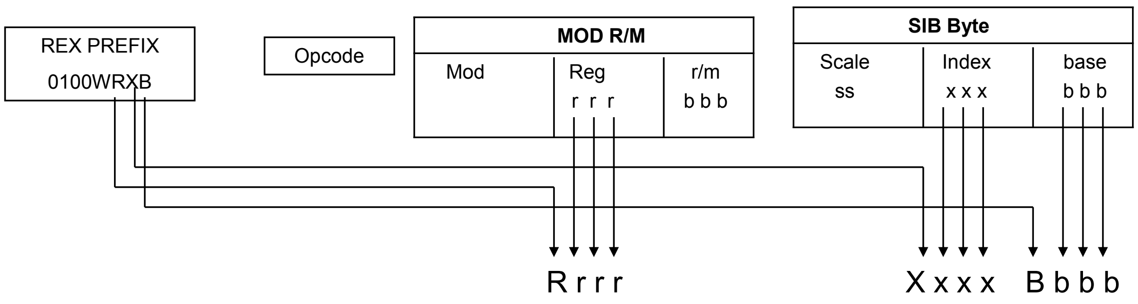


Figure C.8: Memory Addressing when SIB byte is present