ABSTRACT

BEU, JESSE G. PMPT – Performance Monitoring PEBS Tool. (Under the direction of Dr. Thomas M. Conte.)

For many applications a common source of performance degradation is excessive processor stalling from high memory latencies or poor data placement. Performance degradations from program and memory hierarchy interactions are often difficult for programmers and compilers to correct due to a lack of run-time information or limited knowledge about the underlying problem. By leveraging the Pentium 4 processor's performance monitoring hardware, specific run-time information can be provided, allowing code modifications to reduce or even eliminate problematic code, resulting in reduced execution times.

Furthermore, many tools currently available to aid programmers are program counter centric. These tools point out which area of the code produce slowdowns, but they do not directly show where the problem data structures are. This is a common problem in programs that dynamically allocate memory. By creating a "malloc-centric" tool, we can develop an interesting perspective of the memory behavior of the system, providing better insight into the sources of performance problems.

PMPT – PERFORMANCE MONITORING PEBS TOOL

by
JESSE G. BEU

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

COMPUTER ENGINEERING

Raleigh, North Carolina

Summer 2006

APPROVED BY:

_____	_____
Dr. Gregory T. Byrd	Dr. Suleyman Sair

_____

Dr. Thomas M. Conte,

Chair of Advisory Committee

DEDICATION

I’d like to dedicate this work to my wife Valerie.

Thank you for always being there to celebrate the good times

and keeping me sane throughout this process.

BIOGRAPHY

Jesse G. Beu was born in Monroe, NY on July 21st, 1981 and moved with his family to New Smyrna Beach, FL in 1989. After being home taught for the last two years of high school, he was accepted to the University of North Florida where he obtained a Bachelors of Science in Electrical Engineering in 2004. He then married and moved to North Carolina to begin pursuing his master and doctorial degrees in computer engineering.

Outside of academia, Jesse enjoys playing piano and guitar, surfing, playing board games, cooking and nearly anything related to computers and electronics.

ACKNOWLEDGMENTS

I'd like to thank the Tinker research group and the Center for Embedded Systems Research (CESR) faculty for their support and assistance. I would also like to thank Red Hat for their contributions as well as their financial support for this research. Last I would like to extend a special thanks to my undergraduate advisor at the University of North Florida, Dr. Gerald Merckel, and my graduate advisor at North Carolina State University, Dr. Thomas Conte, both of whom have been instrumental in my development as a researcher.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 - Introduction

As processor cycle times continue to decrease, what is known in computer architecture as the memory wall becomes an ever-increasing concern. Currently, main memory strives for high data storage densities rather than reduced access times [22]. As a result, a request from the processor may take hundreds of processor cycles to be serviced. To complicate matters, as application data requirements grow in size, so will the requirements of main memory, encouraging main memory manufacturers to continue increase density rather than speed. Meanwhile processor speeds continue to advance, exacerbating this problem.

To address this growing concern, computer architects introduced a memory hierarchy to processor design. Intermediate memory levels between the processor and main memory, referred to as caches, allow for a progression from high-speed, low-density memory to low-speed, high-density memory. As a result, a processor can take advantage of high-speed memory when able, but still leverage the higher densities available in main memory with little additional overhead. With an awareness of these structures, programmers can write code in such a way to take full advantage of caches and achieve high performance.

This is often a difficult task due to the many layers of abstraction separating the programmer and the actual processor operation. These abstractions are a necessary evil as programming would be far too complex for even the simplest of applications without them. Operating system intervention, memory virtualization and paging, and processor optimizations further distort the programmer's perception of what underlying memory behaviors are truly taking place. As a result, the feedback available to programmers is very limited.

Some processors provide performance monitoring hardware that can be used to bridge the gap between the programmer and program behaviors. In this work, the focus is primarily on the Intel Pentium 4 processor, specifically the precise event based sampling (PEBS) capability. By configuring the hardware to record the machine state (logical register values) when cache miss events occur, important information can be extracted, analyzed, and provided to the programmer about run-time memory behaviors. What makes this approach particularly attractive is that by using the performance monitoring hardware, a program can be sampled with minimal interference by the sampling mechanism, providing more accurate data. Coupling this data with information collected about memory allocation during run-time can provide the programmer with a clear view of what kinds of memory behaviors a program exhibits and how they can be improved. This project, the Performance Monitoring PEBS tool (PMPT), is an effort to accomplish this.

The remainder of this work is organized as follows. Chapter 2 will briefly discuss several related background topics as well as the details of the Pentium 4 processor's PEBS and perfmon2. Chapter 3 will discuss the implementation, capabilities and limitations of PMPT. Chapter 4 will then present insights discovered using PMPT on various benchmarks, followed by related works in Chapter 5 and a conclusion in Chapter 6.

Chapter 2 - Background

Before beginning discussion of PMPT and its workings, it is necessary to provide background information on a variety of topics. This chapter will be subdivided into short discussions on each of the following subjects to provide the prerequisite knowledge required to understand the design choices made in Chapter 3: cache properties, Intel Pentium 4 Performance Monitoring hardware and perfmon2, x86 Assembly, ELF file format, and glibc Malloc.

2.1 Cache properties

As previously stated, caches were introduced to the memory hierarchy as a means to bridge the gap between fast processors and slow memory. The effectiveness of cache memory relies heavily on the locality of memory references, both temporal and spatial. Temporal locality is valuable because memory references within a program often recur within a short time frame. Similarly, spatial locality is useful because programs often reference multiple neighboring memory locations, such as array accesses or fields within a data structure. [9] As a result, successful cache design and usage should take advantage of these properties.

Memory references that are not present in the cache result in cache misses, which are generally divided into three categories: compulsory, capacity, and conflict misses [11]. Compulsory misses are the result of first-time references to a memory location. These could be mitigated through intelligent data placement; by applying techniques like structure splitting or reorganizing the fields within the object, spatial locality can be exploited to reduce these misses. Capacity misses are due to a program's memory access pattern

exceeding the size of the cache. Programming techniques that partition memory access patterns into smaller pieces, such as tiling or blocking [3], can be effective for reducing capacity misses. Conflict misses are the result of exceeding a cache's associativity. Having a better distribution in the memory references, either by an algorithmic or data placement change, can reduce conflict misses. Additionally prediction mechanisms, like the stride prefetcher present in the Pentium 4, can be leveraged to reduce cache misses as well.

2.2 Intel Pentium 4 Performance Monitoring Hardware and perfmon2

The performance monitoring hardware present in the Intel Pentium 4 allows information to be collected about a running application through a variety of event counters. Perfmon2, a standard Linux interface for programming hardware performance counters, provides the kernel support required to program and handle interrupt requests generated by these counters, enabling runtime information to be collected for future analysis. [5, 6, 7]

The hardware can be programmed via machine specific registers (MSRs), specifically event selection control registers (ESCRs) for selecting events to be monitored and counter configuration control registers (CCCRs) to control event counting behavior. Additionally the Pentium 4 provides precise event-based sampling (PEBS) support for a subset of counter events, where upon counter overflow a PEBS record, consisting of the architectural state of the processor (8 general purpose, EIP and EFLAGS register values), is recorded in the debug storage (DS) area. As this area approaches its storage threshold, an interrupt is generated to notify that records should be processed and the PEBS index reset to avoid overflow of the buffer. [14] The events available for use with PEBS that are of primary concern within this framework are L2 cache load misses retiring.

Perfmon2 provides access to the MSRs by encapsulating the state of the performance monitoring hardware (referred to as the performance monitoring unit or PMU within perfmon2) in perfmon contexts. This abstraction not only enables a generic interface across processors, but also allows for independent monitoring of processes by swapping perfmon contexts as necessary from within the operating system. Through programming of perfmon control (PMC) and perfmon data (PMD) registers, the physical MSRs are manipulated as necessary through underlying logical to MSR mappings. Additionally, custom-sampling formats can be supported through kernel modules, such as those used for Pentium 4 PEBS support. [5, 6, 7]

2.3 x86 Assembly

An important component of this work is x86 disassembly. As explained in Section 2.2, PEBS records consist of the processor state at a miss event. This can be leveraged to determine the address of the miss event by locating the instruction at EIP and calculating the instruction's memory request using the general-purpose register values stored in the PEBS record. Once this is acquired, inferences can be made about the memory behavior of the application.

The x86 ISA supports variable length encoding of instructions as well as multiple addressing modes. This complicates decoding of instructions since the disassembler must first determine the size of the instruction, and then extract operand relevant information accordingly. This is acquired from the operand identifier (ModR/M) byte of the instruction, which must be parsed into mod, reg/opcode and R/M fields to interpret what registers and immediate values are used and how. In some cases further decoding of the addressing mode

specifier (SIB) byte is required to determine how to scale a register value before calculating an address. Below is a diagram from the Intel manual depicting the complexities of the IA-32 instruction format [13]. Clearly there is potential for considerable variation in the position of operand information within the instruction. Additionally, in rare cases when a single instruction can access multiple memory locations, there is the potential for one of these calculated addresses to be incorrectly associated with a miss event.

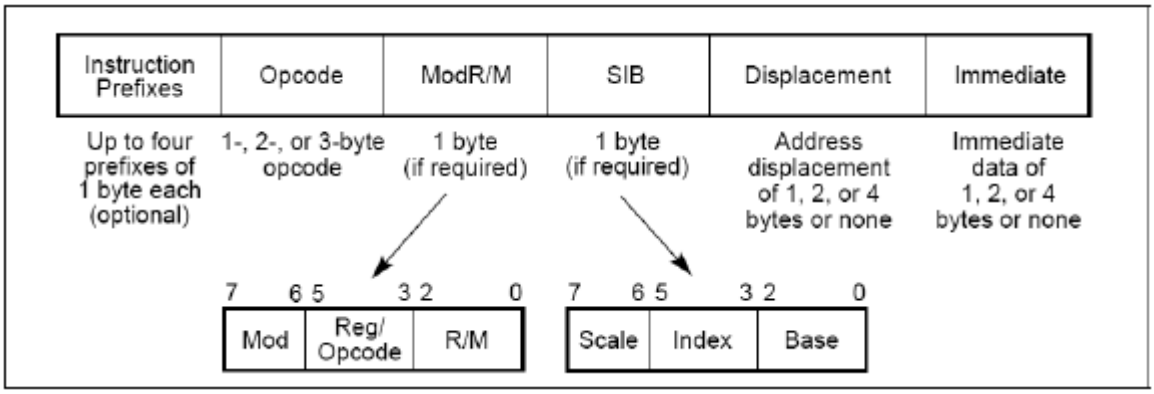

Figure 1 – IA-32 Instruction Format

One simplification can be made to this process: only instructions with operands that can manipulate memory need to be taken into consideration. After disassembly with the assistance of libudis86 [17], each operand of the instruction is processed to check whether a memory reference could potentially be made. If so, the address is calculated using the register and immediate values (when applicable) determined by the ModR/M and SIB bytes and stored for later analysis. If no memory access is possible for a given operand, it is simply ignored.

2.4 ELF file format

In order to acquire the instruction pointed to by the EIP register, it is necessary to locate the instruction within the executable. The executable and linking format (ELF) provides the means to do this.

ELF is a file format intended to create a standard interface for binary interpretation by storing the layout of program memory in an ELF header [4]. The ELF file header, always the first section within the object file, contains many essential pieces of information about the file including the section header table. This is effectively a map of the file, an array of structures that holds information about the file sections, such as the name, file offset, image starting address, and section size.

In this work the section header table can be scanned to locate the text section of the file, which holds the executable instructions. It also supplies the logical offset of the instructions in memory. Using simple arithmetic, this can be used in conjunction with a PEBS entry's EIP value to calculate the appropriate offset into the file required to retrieve the instruction for disassembly. This instruction can then be used to calculate the miss address for the PEBS entry as described in 2.3.

2.5 Glibc Malloc

For PMPT to provide more useful information to the programmer about data structures, malloc information is collected during execution. This is required to map problematic addresses extracted from PEBS to the underlying data structures causing the problems. This can be leveraged to not only provide specific information about what fields within a data

structure may be responsible for high miss activity, but also provide information about when the data structure was allocated, which can have had an impact on its placement in memory.

Malloc data collection is implemented via built in malloc hook functions. By introducing a layer of indirection, malloc calls are intercepted by specialized functions that in turn call the requested malloc function and record relevant information before returning control to the application. This information is stored in a shared memory buffer that PMPT can access. Since these specialized functions require setting glibc malloc function pointers to the hook function addresses, they must be built into the application by including object files at compile time.

Unfortunately the dependence on glibc's malloc currently limits the tool from working with alternative malloc implementations that do not support the malloc hook routines.

Chapter 3 – Performance Monitoring PEBS TOOL

This chapter will outline the development of the Performance Monitoring PEBS tool, PMPT. First the design goals of the project will be reviewed, followed by how these influenced the implementation. This is followed by discussion of the limitations of the tool. Finally considerations for the future of the project are outlined before proceeding to Chapter 4, where PMPT is used to gain insight into application performance.

3.1 Design Goals and Concerns

The primary goal of this tool is to provide useful information about an application with minimal execution overheads and interference with the original program's memory behaviors. This can be a challenging task as reducing execution times and minimizing the interference are often conflicting goals.

Considering the number of performance monitoring applications available, it is important to set this work apart by providing a unique perspective of an application's performance problems. Many tools can provide information about where the majority of execution is spent, but often times the problem is not the instructions themselves but rather stalling from excessive cache misses [9]. In this scenario collecting information about the missing data structures is much more useful than simply pointing out what code is problematic.

To create a complete view of the memory behaviors, several components are needed. For any analysis, the address of the miss is required as well as information about its miss frequency in order to separate truly delinquent addresses from those that are exhibiting tolerable miss behaviors. Once this has been determined, the miss address must be tied to its

corresponding data structure. Determining the offset into the allocated memory is also useful since it provides programmers with a better view of the problem.

Other statistics are also important for the interpretation of these results to enable optimizations. Information about malloc request sizes vs. returned sizes can provide insights into how to reduce the memory footprint. Additionally, a depiction of where and how much memory is being used can expose memory leaks or help developers locate areas that need improving. This would be particularly useful if the tool determines that capacity is the primary cause of cache misses. If conflict misses are found to be problematic, determining what lines are exhibiting high associativity stress may reveal trends in the access pattern and allow for algorithmic changes or data rearrangement. Discovering address groups that frequently miss together may also reveal ways to improve performance.

When designing the tool static, finite sized data structures should be used rather than dynamic techniques in order to reduce interaction with the monitored application. One potential problem, however, is that as execution continues, the storage and analysis of such a large amount of information can become unruly. For this reason, the sample data should be analyzed piecemeal rather than in bulk at the end of execution. This not only allows for smaller storage structures, but also provides the programmer with a chronological view of the memory behaviors. This view may enable a developer to monitor the evolution of a problem and thus more effectively correct it.

Analysis plays an important role since frequent interruption of the application by the tool can both distort the monitored application's memory footprint and slow its execution. Conversely, infrequent analysis results in a need for larger table sizes and the potential for

stale data to be analyzed in conjunction with recent data. Therefore care should be taken in when and how the data is accumulated and processed.

3.2 Implementation

With an understanding of what our design goals are and what information is desired from Section 3.1, design choices become easier to make. With the goals as a guide, PMPT can be developed to maximize usefulness while minimizing complexity, execution overheads and the impact it has on the monitored application. To aid in explaining the implementation, the following figures are provided: a flowchart of PMPT activity (Figure 2) and a high-level diagram of the design's organization (Figure 3). As shown below, execution is divided into three phases, where the second and third phases alternate until the monitored application completes execution.

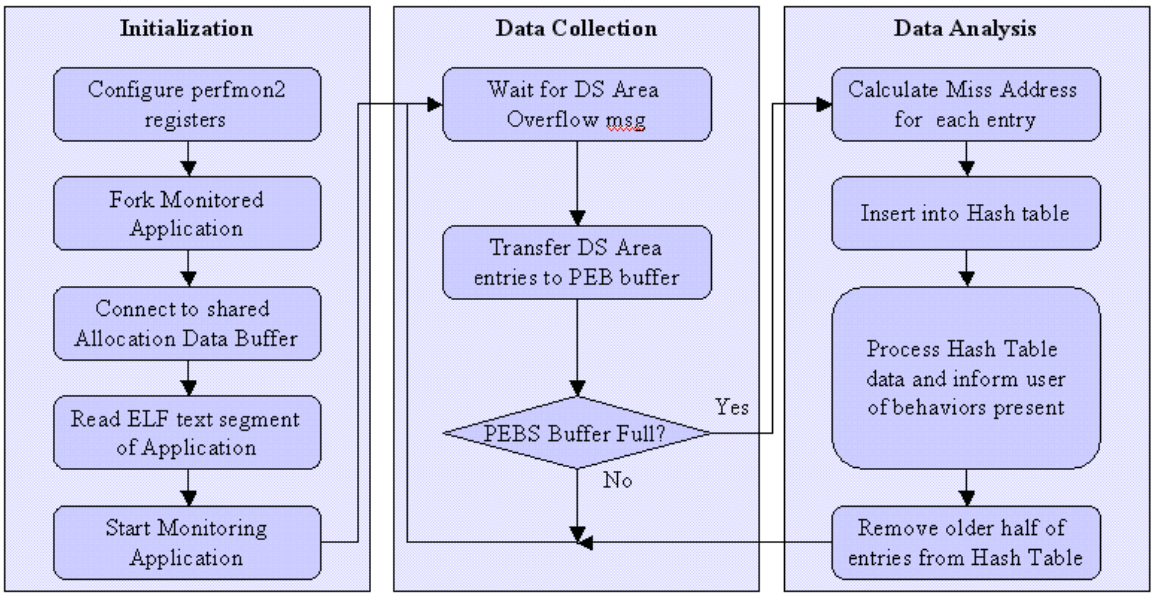

Figure 2 – Flowchart of PMPT Execution

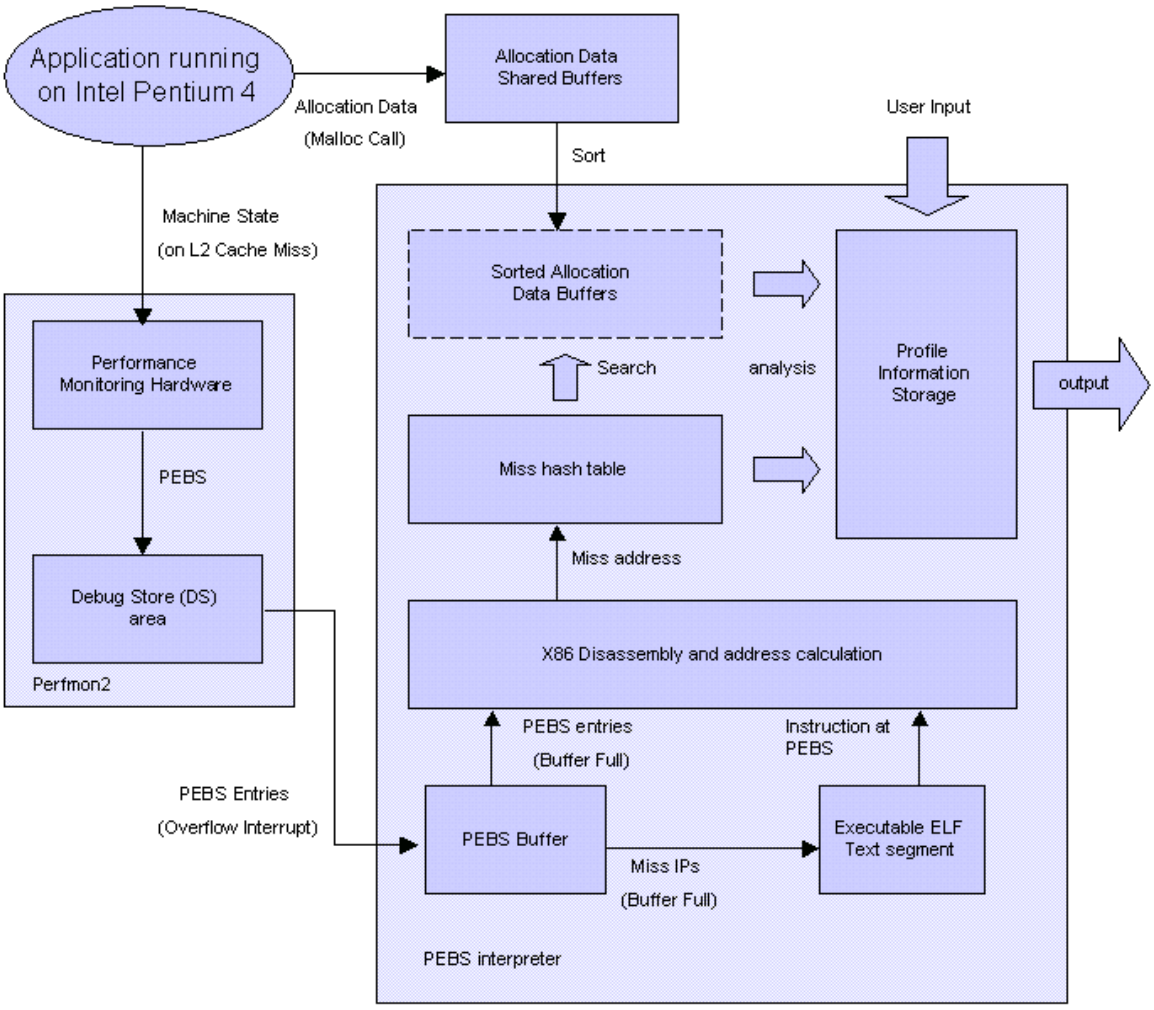

Figure 3 – High-level organization of PMPT

3.2.1 Initialization Phase

The initialization phase is primarily based on Stephane Eranian's PEBS example provided with the perfmon2 package [5, 6, 7]. In the example, after processor and version verification is complete, the DS area is setup and configured to interface with the performance monitoring hardware present in the Intel Pentium 4 processor via a custom sampling format provided with perfmon2 (installed as a module). The perfmon2 contexts for writing to the internal machine specific registers are then configured as outlined in the Intel Software Developer's Manual. This includes setting the Event Select Control Registers

(ESCR), the Counter Configuration Control Registers (CCCR), and the special purpose PEBS_MATRIX_VERT and IA32PEBS_ENABLE machine specific registers to enable cache miss event counting and PEBS acquisition. Minor modifications were made to enable L2 cache miss rather than L1 cache miss monitoring.

After perfmon2 is properly configured, PMPT executes a fork to create a child process, which becomes the monitored application. The tool is connected to the allocation buffer created by the child process and reads the ELF text segment of the application's executable, which contains the assembly instructions as mentioned in Section 2.4. Once these steps are completed, execution of the application begins and the tool is suspended, awaiting a message that either a new set of samples are present in the DS area or that the monitored application has finished execution.

3.2.2 Data Collection Phase

This phase of execution is very straightforward. Its primary purpose is to facilitate control over the frequency of sample processing without disturbing the configuration of the DS area. As a result the DS area can be the size of one page of memory for optimal performance and fast emptying, while the PEBS buffer's size can be adjusted to fit a coarse-grained sampling frequency. This is beneficial since processing of large, contiguous sample groups is easier and has a lower impact on the monitored application as explained earlier in Section 3.1.

Additional information required for effective analysis are the malloc request values. These are not collected in any phase of execution, but rather are collected on the fly by the monitored application itself. Whenever the application makes a call to malloc, realloc or

free, the call is intercepted by the inserted hook functions discussed in Section 2.5. Relevant information like the start address of the block, the requested size, the returned size and call chain information are stored in shared memory segments so the data can be accessed by PMPT for later analysis. Since malloc references may continue throughout execution, segments are allocated as necessary, creating a linked list of malloc information segments.

3.2.3 Data Analysis Phase

The core of this work resides within the data analysis phase of execution, where tasks from conversion of raw samples into addresses to the organization and processing of the data for the user takes place. Since address calculation has already been discussed in Section 2.3, this sub-section will focus primarily on the hash table implementation, the processing of the data present in the hash table, the shared-memory allocation buffer segments, and the interaction between the hash table and the allocation buffer data.

A hash table was chosen for the speed in which searching can be accomplished. It uses the calculated address of the miss for the key and linked lists for collision handling. Any recurrence of an address replaces the older entry with the newer version. Thus, only one copy of an address can be present in the hash table at any time. In order to meet the design goal of avoiding dynamic allocation, a static array of entries is leveraged as both an active/free list and a time ordered list of entries. This enables decaying of samples to remove older samples as newer ones are generated by the application and PEBS hardware. Figure 4 depicts the organization of the hash table.

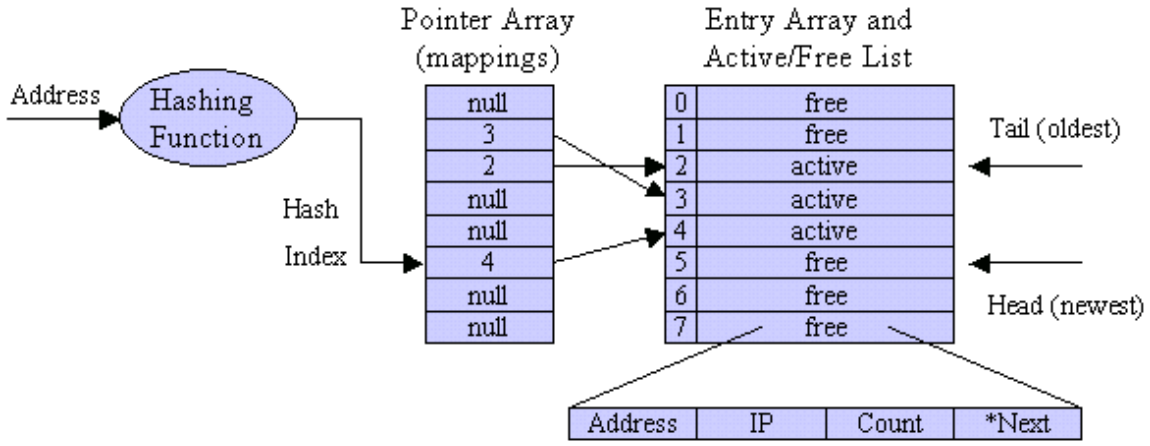

Figure 4 – High-level view of hash table organization

The replacement policy for when a recurrent address is detected results in 'active' entries that are not associated with any key. Although these entries no longer have a valid mapping, they cannot be discarded or freed for two reasons. First, when all entries have been inserted into the hash table, the data in the replaced hash entries is still required for proper counting of address recurrences and other analyses. Second, allocating entries from the head and freeing from the tail significantly simplifies the design while ensuring that the list is chronologically ordered. Maintaining a time ordered list is important; as samples are collected, older entries need to be freed to make room for newer entries. If the entry at the tail is guaranteed to be the oldest entry in the list, this process is as simple as freeing the tail entry.

Since the analysis phase is processing samples in bulk, care must be taken to ensure that all PEBS samples are compared against sufficient previous and future samples. If the hash table were to be completely emptied at the end of an analysis phase, any samples collected immediately preceding an analysis phase will not be correlated with future samples. Similarly the first samples collected after the end of an analysis phase will have no concept of the past to compare against. To resolve this problem, only the older half of the entry array

is freed when transitioning from analysis to collection. This guarantees that each entry will be compared against a minimum of (# of entries / 2) older and newer samples in consecutive phases, as shown in Figure 5.

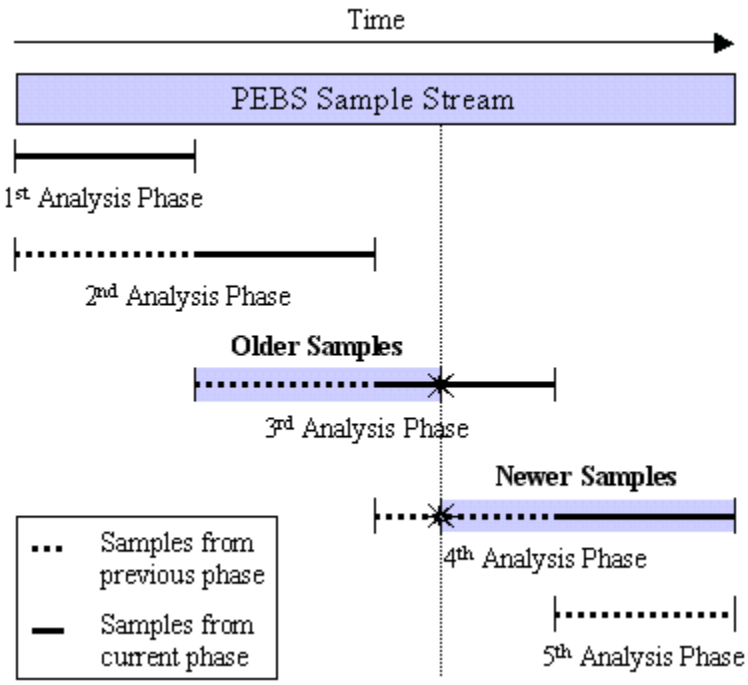

Figure 5 - Window of samples available for analysis

Once the hash table is populated with miss addresses extracted from PEBS samples, trends among the values can be detected. Perhaps the most obvious trend is the recurrence of an address within the collected misses. This can be accomplished by traversing the active list, doing a hash table lookup for each entry and incrementing the count of the address-matching hash table entry. Since only one copy of an address can be present in the hash table, all addresses in the active list with the same address will increment the same entry, providing a total count. This is valuable because it helps isolate the truly delinquent addresses. Further analysis can be applied to these, saving execution time compared to an approach that applies full analysis to all samples present in the hash table.

Determining a recurrence count is also useful since it can provide a sense of the cache miss distribution to the developer. Inferences can be drawn from the distribution about whether capacity or conflicts are more prohibitive to performance. For example, many unique addresses (a high density of single counts) indicate that capacity is potentially the problem. While calculating the recurrence count, a count of address-to-cache set mappings is also taken to determine the distribution of miss events across cache sets. This is useful in determining whether conflict misses are dominating the cache. Information about the EIP when the miss event occurred can provide insights as well, such as whether the delinquent structure is consistently missing in the same function or in several places throughout the code. It can also be leveraged to determine if an algorithmic behavior is responsible for the misses.

The shared allocation buffer also contains a wealth of information. As previously mentioned, simple statistics about sizing, both in object size distribution and malloc request vs. returned block sizes, can make a developer aware of optimization opportunities. By incorporating the malloc information with that available in the hash table, even more opportunities become evident. By combining the miss address with the start address of the allocation, we can see what fields within a structure are responsible for the misses. Mapping delinquent structures back to their initial allocation can reveal trends and provide the means for more intelligent data placement and organization. In order to enable these linkages, the shared memory segments, which comprise the allocation buffer, are sorted individually and sequentially searched. Address ranges are maintained for each segment to accelerate the searching (hash entries can skip over segments that cannot possibly contain their allocation data).

3.3 Limitations and future work

Although this tool is capable of many things, it does have limitations and room for growth. The reason behind these limitations will be discussed, followed by the future direction of this work.

Perhaps the most detrimental limitation of this work is the lack of multithreaded support. Perfmon2 was designed to enable per-thread sampling and as a result requires processes to be attached to the context independently. However, this is problematic since the tool cannot autonomously inject code to do this context attachment for multithreaded workloads as they spawn threads. The alternative is to apply global sampling, but this has limitations as well since it will not only cause data from all running processes to be collected, but also cause malfunctions with regard to perfmon2 message passing for sampling termination. In global mode, PMPT sees itself as a monitored thread and therefore is never passed the 'monitored task terminated' message from the perfmon2 infrastructure. As a result, PMPT gets locked in an infinite loop.

Furthermore, even if these problems were mitigated, there is the dilemma of determining what executable to read so ELF information can be acquired for address calculation. Another issue related to this phenomenon is shared library misses. Some programs exhibit misses caused by shared library usage, but since these libraries are generally dynamically loaded and unloaded, it is extremely difficult to capture the assembly instructions required for address calculation. The PEBS record is of little use without the information to extract the instruction and in turn the data address..

Despite these problems, there is hope for the future. One of the more useful improvements would be to integrate some form of communication between multiple instances of the tool. If the multithreaded limitations can be overcome, this tool could be leveraged to run on multiprocessor machines and assist in the detection of false sharing and data migration. Through this communication system, the home node of a miss could run allocation search routines on distant nodes to discover what home and use patterns exist. Depending on the lock implementation, lock behaviors could even be studied and shared between nodes.

Intelligent automated analysis is an area currently under investigation. The concept behind this is to develop a set of 'fingerprints' for some common problems and monitor the data stream for their presence. If successful in this endeavor, the tool would not only become much more user friendly, but could even be used to perform dynamic optimization during runtime. It could even provide highly detailed information to the compiler, behaving as an advanced profiling mechanism.

Other possibilities include more advanced allocation buffer analysis to assist in memory leak detection by monitoring malloc/free patterns. Stack analyses based on EBP and ESP values collected from PEBS samples may be useful to users who are concerned about the stack usage of their applications. The inclusion of stack variables in the allocation buffers for analysis would also be a useful addition since not all misses map to dynamic data structures.

Chapter 4 - Case Studies

The following are discussions of some discoveries made using this tool with a simple verification microbenchmark and the SPEC2K benchmarks *ammp, gcc,* and *mcf* [10]. These are followed by a presentation of the impact the tool overhead has on execution time.

4.1 Microbenchmark

The primary purpose of the microbenchmark is to verify basic functionality of the tool. It was designed specifically to cause consistent L2 cache misses by allocating an array and performing strided accesses such that they all coincide to the same set. The tool could capture this behavior and allow the user to clearly see the trend.

This initially did not function correctly due to the Pentium 4's built in stride prefetcher; the expected miss pattern was not present in the PEBS sample stream. This required the introduction of a random number generator to effectively 'break' the prefetecher. As a result, it was found that the usage of the shared libraries for the *rand()* function produced a substantial portion of the overall misses detected. This is being investigated as an area of future research in hopes of discovering techniques to correct this behavior.

4.2 *Ammp*

Ammp's usage of malloc routines is rather frequent, having a somewhat fine-grained allocation scheme. As a result the collected miss addresses, when mapped to their corresponding malloc references, provided an interesting insight – highly delinquent miss addresses early in execution all shared the same offset into the allocated objects. This

implies that the same field within each object is being accessed. Upon further investigation it was discovered that the field causing misses is actually the *next* field of the data structures, with the misses occurring in a linked-list traversal for loop.

What is unusual about this is that these misses should not be present, or at least not in the manner being represented. Earlier in the same code segment the *serial* field of the object is accessed to determine whether to progress down the linked-list or halt at the current node. The *next* field is stored adjacent to the *serial* field and is not separated by cache-line boundaries, and therefore should already be present in the cache. Since the source field is being read first, it should be incurring the misses, but no samples correspond to the *serial* field of the object. This implies some manner of out-of-order execution is taking place.

Further study of the code shows these same data structures are used later, but not inducing misses. This implies that these initial miss samples may be representing the compulsive misses for initially bringing the data into the cache, due to the sample's presence at the beginning of execution and absence later. A possible optimization would be to introduce a pre-loading function that assists in filling the cache with the data in a predictable manner, enabling the hardware prefetcher to reduce the compulsive miss penalty prior to the actual usage of the data.

Later in execution another behavior can be observed. The miss count for specific addresses increases dramatically, most of which map to a local variable on the stack. Again the code responsible is a linked list traversal function; however, this time another phenomena may be responsible. A single local variable, a pointer, is repeatedly being read and written to as the linked list is being traversed (first being read for the address of the object being pointed to and later being overwritten with the address of the next object in the list). Since

the misses are mapping to the local variable address and not the object it points to, the frequent reading and writing to the stack is the source of the problem. Most likely store-to-load forwarding is not properly taking place, perhaps due to the load-store queue overflowing because of the rapid nature of the reads and writes present in the loop. This could result in writes causing cache lines to be invalidated, incurring a miss upon the next read to the same line.

4.3 *Gcc*

The reference inputs for *gcc* resulted in far fewer cache misses than the other benchmarks studied. As a result, few analyses could be made upon the memory behaviors present. It is worth noting that this was made even more difficult due to the quality of the samples that were collected. A large portion of the samples collected by the hardware could not properly be interpreted since their IP values did not fall into the range of the executable, preventing miss address calculation. It is assumed this is the result of shared library usage, emphasizing the importance of this area of future research.

4.4 *Mcf*

Mcf is different in that large portions of memory are allocated then broken into smaller components within the program, most likely to avoid fragmentation and frequent malloc calls. This makes the job of interpreting the results difficult, however, as most addresses map to the same allocation site. Moreover, because of this organization offsets are not easy to interpret – the offset is now a combination of the offset into the object and the offset of the

object into the allocated chunk of memory. Even with these challenges, certain trends are clearly visible.

First, as execution progresses we see a kind of migration in the values of the samples – one address's miss frequency diminishes while another's increases. These misses take place primarily in a function doing tree traversal, implying that these represent highly traveled nodes of the tree, perhaps the root or other high level nodes, as the tree evolves during execution. Since this traversal and updating is input dependent, it may be difficult to improve the performance of this. Software prefetching of the root or dedicating special memory so it is not easily evicted from the cache may prove beneficial.

Mcf also exhibits some of the same behaviors as *ammp* with regard to local variable usage. Toward the middle of execution the cache miss behavior is dominated by a single local variable, with over 100 samples mapping to it in a single analysis phase (compared to 7 mappings for the next highest address reappearance). This recurring behavior prompts the need for a solution to improve the load-store queue for applications with traversal algorithms.

Even so, throughout execution, the distribution of the misses in the cache is relatively uniform. Aside from the addresses recently mentioned, most others have very low recurrence, implying that capacity is a problem rather than conflict. This is to be expected since *mcf* has a very large working set [19].

4.4 PMPT induced overhead

As mentioned in Chapter 3, one of the design goals of this project was to provide useful data with little overhead. To show the overheads associated with this tool, each benchmark studied was run as a stand alone, then with PMPT installed and running with a sampling

period of 10,000 (record machine state once every 10,000 miss events). Below are the execution times of running the tool, normalized against the base case. Of those studied, the worst-case slowdown in execution time is 3%. The runtimes of the benchmarks evaluated ranged from 23 seconds for the custom microbenchmark to over 11 minutes for *ammp*.

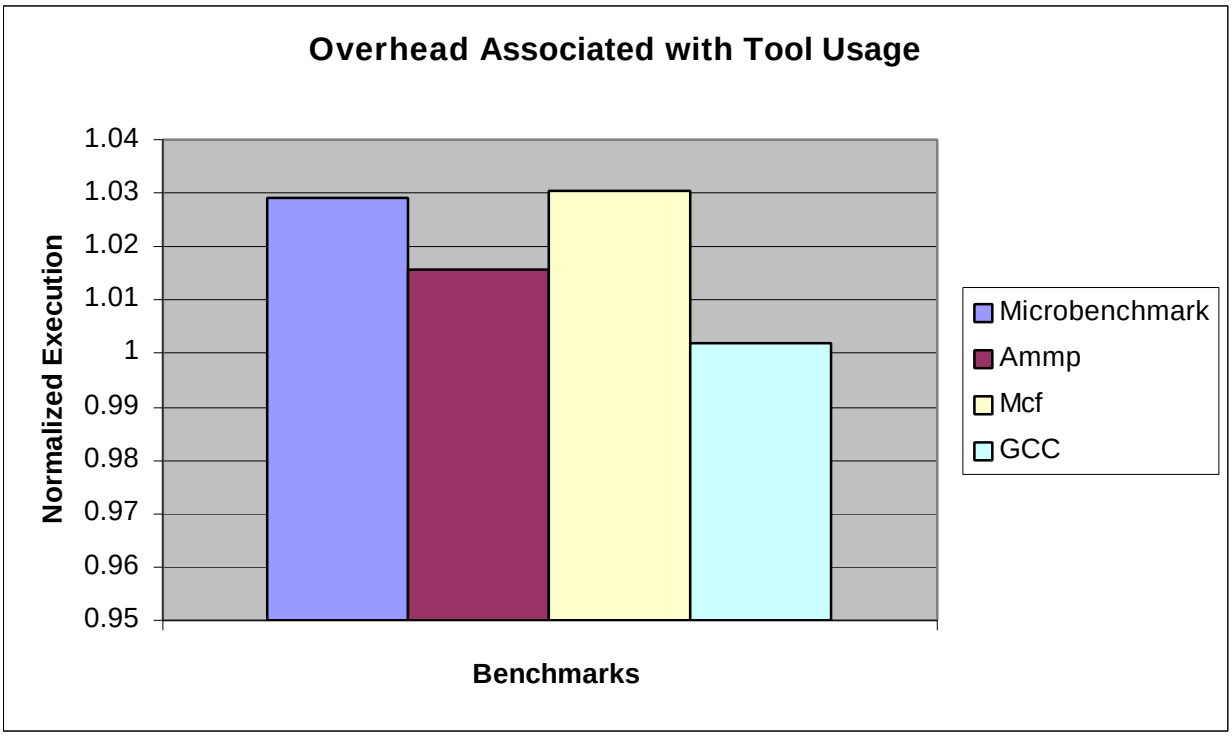

Figure 6 – Cost of running PMPT.

Chapter 5 - Related Work

Currently there are several tools available to programmers that can provide feedback about their application. These can generally be divided into two broad categories, those that use the performance monitoring hardware and those that are primarily software based.

5.1 Tools leveraging and/or providing hardware support

The most basic capability supplied by performance monitoring hardware is hardware counting. Oprofile [15] leverages hardware counters to provide a variety of statistics about where execution time is being spent in an application. Hardware counters are present on most modern processors and in the event they are not, Oprofile provides support via software real-time clocks (with less functionality), making it highly portable. Oprofile is also unobtrusive, installed as a kernel module, removing the need for a kernel rebuild, and requiring no recompilation of the monitored applications. Due to its use of the hardware counters, the overheads for monitoring are also relatively low, on average "1-8%, dependent on the sampling frequency and workload". It does not, however, have PEBS support and thus cannot provide information about memory behaviors with respect to application data structures.

Performance application programming interface, or PAPI [21], provides a standard interface for accessing the performance monitoring hardware available on most modern processors. It has both high-level and low-level interfaces to enable a range of counting behaviors based on the developer's needs. The high level interface provides generic functionality whereas the low level interface provides more specific capabilities based upon the underlying architecture's monitoring hardware support. PAPI relies on perfctr [18] for

compatibility with Linux kernels running on Intel Pentium processors. As a result, PAPI does not provide PEBS support and therefore cannot map cache miss events back to the relevant data structures as PMPT does.

Perfmon2 [5, 6, 7] also provides a standardized interface for accessing performance monitoring hardware across a variety of platforms. Perfmon2, however, was designed with the goal of becoming part of the kernel. As a result, it is highly flexible and generic to enable interfacing with current architectures as well as future processors. This is possible because all modern performance monitoring units (PMUs) are controlled via a register interface. As mentioned in Section 2.2, perfmon2 configures these by adding a layer of abstraction, perfmon contexts, which configure the mapping from perfmon configuration (PMC) and perfmon data (PMD) registers to the underlying PMU control and data registers. This allows for a certain level of configurability, enabling the PEBS support required for this work.

Another interface specifically for accessing the Pentium 4 performance counters and hardware is Brink and Abyss [20]. This tool is divided into two portions, brink, a configuration front-end, and abyss, a program that initializes and collects samples from the hardware via a custom device driver. Brink is a perl script that takes two files as inputs: an event monitoring configuration file, which contains a description of the monitoring hardware registers, and an experiment configuration file, which determines what applications to run and what performance counter configuration to use. Upon processing of these, brink supplies the proper inputs required by abyss in order to run the desired applications and configurations specified by the experiment configuration file. Brink and Abyss, being designed specifically for the Pentium 4, does provide support for PEBS sampling. It does not, however, enable

straightforward access to the hardware as perfmon2 does, making it less attractive for use in this work.

5.2 Software performance and profiling tools

In addition to those that use hardware, some of the more useful developer tools are pure software implementations. Perhaps one of the most widespread tools available is the Valgrind [1] binary rewrite tool with debugging and profiling capabilities. Because Valgrind is reading the binary, source code is not needed, nor is any recompilation or re-linking. It is compatible with programs written in any language as a result of the binary analysis. Due to its widespread use and extensibility, there are also a variety of tools available for use with Valgrind to further enhance the profiling capabilities, as well as support for custom tool development. The overheads associated with Valgrind, however, are rather extreme, with a minimum slowdown of 4 times for the nulgrind tool (which adds no instrumentation) to upwards of 50 times slower for tools like Memcheck. Nonetheless, Valgrind is a very powerful tool that can provide vital insights into an application's behaviors.

Pin [2] is another binary rewrite tool available for profiling of applications. It does so by dynamically injecting small code segments (which may be written in C or C++) into the executable while it is running. The developers describe it as being similar to a "just in time" compiler, where it generates instruction sequences using the original application's code combined with instrumentation code. These sequences transfer control back to Pin upon completion where another instruction sequence is generated and executed. Included with Pin are examples of a basic block profiler and cache simulator to assist in the writing of custom

tools. As a result, Pin has the potential to provide very specific profile information, but at the cost of development and execution time.

Another option for collecting profile information is gprof [8]. Gprof can provide both a flat profile, showing the amount of execution time spent in each function, and call graphs, showing the amount of execution time spent in each function and its children. This profile information is collected by periodically sampling the program counter and determining what function the current instruction resides in. Function compilation is modified such that as functions are called, information about their caller is recorded in specialized data structures, providing a better view of where costly functions reside in the call chain. As a result, gprof requires recompilation of the application, with the appropriate flags set, to function correctly. Although useful, the simplistic nature of the profile information collected by gprof does not enable memory behavior analysis; it can only provide data about where execution time is being spent, not why.

Chapter 6 - Conclusion

Through the use of PMPT, some of the mystery surrounding performance degradations can be unraveled. In the worst case, problems can at least be brought to the developer's attention, and in better scenarios PMPT is capable of revealing otherwise hidden opportunities for improvement. Furthermore, by incorporating information available from malloc calls, we can add depth to the collected data to extract even more information about the memory behaviors present during execution.

Additionally, because PMPT uses hardware already present in the processor, modifications to code, based on the data, can be made and tested rapidly. As a result, the development cycle of improving an application is accelerated considerably over other alternative performance monitors that can provide comparable information. This makes the Performance Monitor PEBS Tool a valuable addition to the programs available for improving application performance.

Bibliography

1. Armour-Brown, Cerion, Jeremy Fitzhardinge, Tom Hughes, Nicholas Nethercote, Paul Mackerras, Dirk Mueller, Julian Seward, and Robert Walsh, developers. *Valgrind*. Valgrind Developers. Apr. 2005 <http://valgrind.org/>.

2. Cohn, Robert, Geoff Lowney, C.k. Luk, Steven Wallace, Harish Patil, Artur Klauser, Manuel Fernandez, Greg Lueck, Mark Charney, Kim Hazelwood, and Vijay Janapareddi, developers. *Pin*. University of Colorado. <http://rogue.colorado.edu/Pin/index.html>.

3. Culler, David E., Jaswinder Pal Singh, with Anoop Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, © 1999 Morgan-Kauffman, ISBN 1-55860-343-3.

4. Executable and Linking Format (ELF). *Tool Interface Standards Committee*, May 1995.

5. Eranian, Stephane. *Perfmon2: a Performance Monitoring Interface for Linux*. Hewlett Packard. 2005. Aug. 2006 <http://openlab-mu-internal.web.cern.ch/openlab-mu-internal/Documents/3_Presentations/Slides/2005/05-01_SE_perfmon.pdf>.

6. Eranian, Stephane. *The perfmon2 Interface Specification*. Hewlett Packard. 2005. Jan. 2006 <http://www.hpl.hp.com/techreports/2004/HPL-2004-200R1.pdf>.

7. Eranian, Stephane. "Quick Overview of the perfmon2 Interface." perfctr-devel. 19 Dec. 2005. 10 July 2006 <http://sourceforge.net/mailarchive/forum.php?forum_id=2237>.

8. Fenlason, Jay, and Richard Stallman, developers. *GNU Gprof, the GNU Profiler*. University of Utah. <http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html>.

9. Hennessey, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2002.

10. Henning, J. L. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. IEEE Computers. 33, 7 (July), 28-35.

11. Hill, Mark D. and Alan Jay Smith. “Evaluating associativity in CPU caches.” *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

12. *Intel, An Overview of Cache*. July 2006 <http://www.intel.com/design/intarch/papers/cache6.pdf>.

13. Intel, *IA-32 Intel Architecture Software Developer's Manual Volume 2A: Instruction Set Reference*, A-M, available at http://www.intel.com/design/Pentium4/documentation.htm, 2005.

14. Intel, *IA-32 Intel Architecture Software Developer's Manual Volume 3B:* System Programming Guide, Part 2, available at http://www.intel.com/design/Pentium4/documentation.htm, 2005.

15. Levon, John, and Philippe Elie, developers, *Oprofile*. Mar. 2006 <http://oprofile.sourceforge.net>.

16. Ludloff, Christian. "IA-32 Architecture." *Sandpile.Org*. 2006. May 2006 <http://www.sandpile.org/ia32/index.htm>.

17. Mohan, Vivek, *Libudis86*. Vers. 0.91. <http://sourceforge.net/projects/udis86>.

18. Pettersson, Mikael, comp. *Perfctr*. Uppsala University. July 2005 <http://user.it.uu.se/~mikpe/linux/perfctr/>.

19. Sharma, S., J. G. Beu and T. M. Conte, "Spectral prefetcher: An effective mechanism for L2 cache prefetching," ACM Transactions on Architecture and Code Optimization, vol. 2 , no. 4, Dec. '05, pp. 423-450.

20. Sprunt, Brinkley, comp. *Brink and Abyss*. Bucknell University. Aug. 2005 <http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtm>.

21. Terpstra, Dan, Keith Seymour, Haihang You, Phil Mucci, Jack Dongarra, and Shirley Moore, developers. *PAPI*. University of Tennessee. Aug. 2005 <http://icl.cs.utk.edu/papi/index.html>.

22. Wilkes, M.V. The memory wall and the CMOS end-point. *Computer Architecture News*, 23(4):4–6, September 1995.