A FRAMEWORK FOR USING THE PENTIUM'S PERFORMANCE MONITORING HARDWARE

BY

KEVIN DAVID SAFFORD

B.S., University of Illinois, 1995

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

ACKNOWLEDGMENTS

My advisor, Professor Wen-Mei Hwu, has been wonderful throughout my career. His enthusiasm for teaching and his passion for research were my main inspiration for going into computer architecture. I thank him for giving me the opportunity to work with a wonderful group.

I wish to thank the entire IMPACT research group for providing an excellent infrastructure with which to do this research. In particular, Brian Dietrich served as my mentor throughout graduate school, helping me every step of the way. John Gyllenhaal and Teresa Johnson have provided many ideas on how this software will be used. Matt Merten taught me many aspects about x86 code and served as a sounding board for many of my ideas. He also helped greatly in debugging this software.

Finally, I wish to thank my parents for their love and support through six years of university study. Without them, none of this would have been possible.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. INTRODUCTION

1.1 Overview

Analyzing a program's run-time behavior on a real system is a difficult challenge. Basic UNIX tools only provide data such as execution time. Conventional profilers, such as *gprof*, only give information on how many times a function might be called, or how long a function might take to execute. Neither of these tools, however, provides insight as to why a particular section of code takes as long to execute as it does. Modern processors contain multiple execution units, high-speed caches, and other performance-enhancing features. Obtaining the greatest possible performance requires following a complicated set of rules during compilation. In an effort to characterize how real code runs on a real processor, special hardware can be added to monitor performance. In the case of Intel's Pentium processor, the architects added two performance monitoring counters that can count 38 different events related to processor performance. The designers provided an interface to allow users to access these counters and measure various aspects of system performance.

This thesis serves several purposes. First, it is designed to give some background information about the monitoring hardware found in the Pentium. Second, it is a source of documentation for both low-level and high-level software developed as part of this thesis work to provide access to these counters. Finally, it offers insight on how to use this hardware and software to obtain meaningful results. The rest of this thesis is organized as follows: Chapter 1 provides motivation for this work, as well as describing some related work. Chapter 2 then describes the performance monitoring hardware found in the Pentium processor. Chapter 3 describes the low-level software and library routines developed to provide user-level access to the hardware. Chapter 4 documents a high-level program, run_p5mon, which gathers data on program execution. Chapter 5 provides some examples of using this software to monitor various programs. Finally, Chapter 6 discusses conclusions as well as possible future work.

1.2 Motivation

There are several reasons for using performance monitoring hardware to analyze code behavior. Traditional benchmarking has only looked at real execution time of entire programs on a particular system. Little additional detail has been available. To gather finer-grain information, some sort of simulator or emulator must be used.

Simulators, while useful to evaluate new architectural features to obtain a rough idea of potential benefits, have several drawbacks when used to analyze code on an existing system. The first is the execution speed of a simulator. At best, it is several times slower than the real hardware. On average, the cycle-accurate simulator used by the IMPACT group is about 1000 times slower if all dynamic program instructions are simulated. This can be reduced to about a 20 times slowdown if sampling is used. Second, the accuracy of a simulator is directly dependent on the accuracy of the processor and memory models. Finally, while some simulators and emulators, such as Stanford's SimOS [2], can easily measure operating system and library code, it is difficult for trace-driven simulators to measure this code because of the difficulty in obtaining traces. The effects of library and operating system code can be far-reaching. First, the simulator will generally generate optimistic results, since the actual program with library and system calls will execute more instructions. These extra instructions can displace other program instructions and data in both the instruction and data caches, displace branches in the branch target buffer (BTB), cause more pipeline flushes, and have other negative effects on the program's execution time.

To overcome the problems of a simulator, it is desirable to analyze code behavior on a real system in order to include both library and operating system code when running a particular program. In addition to gathering data on all executing code, hardware monitors give the added benefit of helping verify a simulator's accuracy by allowing comparisons of simulated results with the actual measured results. In a benchmark that has few library calls, the simulator results should be very close to the measured results. If they are not, then there is a possibility that something is not being modeled correctly within the simulator.

The other benefit of using the Pentium hardware counters is to analyze compiler-generated x86 code. Through analysis, we can determine what portions of the code lose performance by measuring different events on small regions of code. Then, the reasons for the performance loss — be they stalls due to cache misses, cache bank conflicts, address generation interlocks, or other factors — can be identified and addressed. In general, this type of analysis can be useful to any software or compiler writer trying to tune code to run on that particular processor.

1.3 Related Work

Little is published concerning hardware performance monitors. When Intel released the Pentium processor, they merely mentioned that there were several performance monitoring counters in the Pentium, but did not disclose how to access them or what they measured. However, Terje Mathisen reverse-engineered this information and published the results in *BYTE Magazine* [1]. Following this, Intel released the Pentium's performance monitoring hardware information. They also disclosed this information for both the Pentium Pro and the Pentium with MMX processors when they were released.

Two papers have been published concerning the use of the Pentium's performance monitoring counters. Chen et al, used the Pentium's hardware counters to compare the performance of the Microsoft Windows 3.1, Microsoft Windows 95, and NetBSD operating systems [3], [4]. Intel itself recently published a paper comparing the performance of the Pentium and Pentium Pro running various benchmarks [5]. These comparisons contained information that was gathered from the hardware monitoring counters on both processors. However, no information was given on how they actually gathered the data.

Presumably, other processors contain this type of hardware, as monitoring counters are an excellent source of information for the designers of the processor in determining how their processor will behave on real code. This hardware is also helpful for the company to provide code generation guidelines for external compiler writers.

2. PENTIUM PERFORMANCE MONITORING COUNTERS

2.1 Overview

In order to monitor the performance of the Pentium, the architects added several registers and counters to the Pentium architecture. The performance monitoring registers in the Pentium processor are a subset of the Model Specific Registers (MSRs). To provide access, two instructions, RDMSR (Read Model Specific Register) and WRMSR (Write Model Specific Register), were added to the Pentium's instruction set. This chapter describes the Pentium's MSRs. Section 2.2 first describes the function of the MSRs, and Section 2.3 details their performance monitoring features. Finally, Section 2.4 describes the various events that may be counted using the MSRs. All of this information concerning the MSRs comes from the *Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual* [6].

2.2 Model Specific Registers (MSRs)

The Pentium Processor includes model specific registers to provide access to features that are tied to implementation-dependent aspects of the Pentium. In addition to registers that help test internal chip structures, such as the cache or BTB, the MSRs include performance monitoring counters that allowed the architects and developers to investigate how the Pentium behaves while executing real code. In addition to being useful to the architects and designers, the model specific registers are useful to compiler writers, software developers, and system designers.

In order to access the MSRs, two instructions were added to the Pentium's instruction set, both of which are described in Chapter 25 of the *Pentium Programming Manual* [6]. The RDMSR (Read Model Specific Register) instruction reads one of the model specific registers and returns the value. The value in the ECX register specifies one of the processor's MSRs. The value of the MSR is copied into the EDX and EAX registers, with EDX containing the high-order 32 bits and EAX the low-order 32 bits. This instruction takes anywhere from 20 to 24 clock cycles to complete. This instruction must be executed from privilege level 0, otherwise a protection fault will occur. No processor flags are affected by this instruction.

Similarly, the WRMSR (Write Model Specific Register) instruction writes one of the model specific registers. The value in the ECX register specifies which one of the 64-bit MSRs to write. The value to be written is specified in the EDX and EAX registers, with EDX containing the high-order 32 bits and EAX the low-order 32 bits of the value. This instruction takes anywhere from 30 to 45 clock cycles to complete. It too must be executed from privilege level 0. No processor flags are affected by the WRMSR instruction.

2.3 Performance Monitoring Features of the MSRs

Four MSRs are related to performance monitoring in the Pentium. They are summarized in Table 2.1. It is important to note that none of the performance monitoring counters (PMCs) are saved and restored when using the low-level task switch instruction normally executed by the operating system. If the user wishes to have these registers saved and restored, the operating system itself must actually save and restore the MSRs. This is discussed in some detail later.

Table 2.1. Pentium performance monitoring MSRs

MSR (in hex)	Register Name
0x10	Time Stamp Counter
0x11	Control and Event Select
0x12	Programmable Event Counter 0
0x13	Programmable Event Counter 1

2.3.1 Time Stamp Counter (TSC)

The Time Stamp Counter (TSC) is a dedicated, 64-bit register that is incremented once every clock cycle. A separate instruction, RDTSC (Read Time Stamp Counter) is used to access this counter. The TSC was not used during the course of this work.

2.3.2 Event counters

There are two 40-bit programmable event counters (CNT0 and CNT1) that may be programmed to count any event from a predetermined set, as described in Section 2.4. When switching to a new event, the user must clear or preset the counters using the WRMSR instruction. Each of these counters can also signal the occurrence of an event on an external pin to allow system designers to perform hardware monitoring of the processor.

2.3.3 Control and Event Select Register (CESR)

The Control and Event Select Register (CESR) is a 32-bit register used to control the two event counters described above. The layout of the CESR is shown in Figure 2.1. Each counter has a 6-bit Event Select (ES) field, 3-bit Counter Control (CC) field, and Pin Control (PC) bit. The shaded areas are reserved and may not be changed by the user. Since it is not possible to selectively write a part of the CESR, the CESR must first be read, the appropriate fields changed, and all 32 bits written back in order to make a configuration change.

2.3.3.1 Event select

The six-bit Event Select field controls the events to be counted by each counter. The two events are independent of one another. The individual events that can be counted are described in Section 2.4.

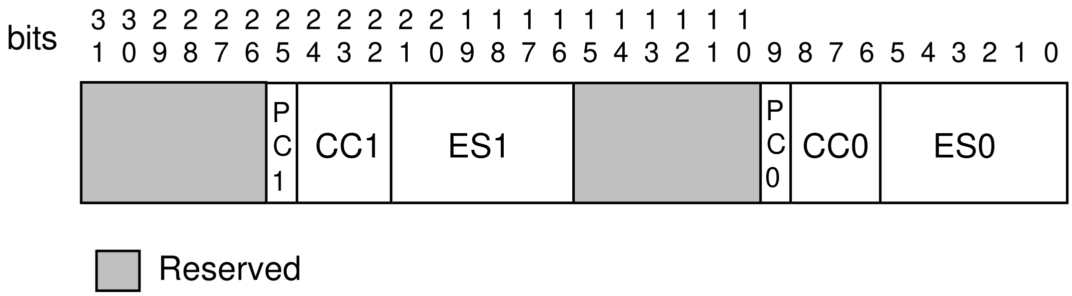

Figure 2.1. Control and Event Select Register layout

2.3.3.2 Counter control

A 3-bit Counter Control field is used to control the operation of the two counters. The counter control determines whether to count the specified event or just clock cycles, as well as what "ring" in which to count the events. The Pentium processor has four rings of operation in order to support varying degrees of protection. Ring 0 (also known as CPL 0) is the most privileged level of operation, typically where the operation system runs. Rings 1 and 2 (CPLs 1 and 2) are additional rings used by the operating system. Finally, Ring 3 (CPL 3) is the level where all user processes run. In addition to counting events, both counters can count just clock cycles, transforming the counter into a time stamp counter.

The three bits of the control field determine what mode each counter is operating in, with each bit in the field controlling one aspect. The highest order bit selects between counting events and counting clock cycles. The middle bit enables counting in Ring 3 (the user level). Finally, the low order bit enables counting when executing in rings 0, 1, or 2 (the operating system, or kernel, level). Using these three bits, eight modes of operation are possible, as summarized in Table 2.2.

Table 2.2. Counter modes

CC	Description
000	disable counter
001	count event while CPL=0,1, or 2
010	count event while CPL=3
011	count event regardless of CPL
100	disable counter
101	count clocks while CPL=0,1, or 2
110	count clocks while CPL=3
111	count clocks regardless of CPL

2.3.3.3 Pin control

As mentioned before, each counter can control an external pin to signal an event occurrence or counter overflow. For the purposes of this research, these are ignored and set to 0.

2.4 Events

The Pentium can count 38 different events in each of its performance monitoring counters. These events range from data reads and writes, to branches and BTB hits. Each of the events may be classified as either among those that count an *occurrence*, or among those that count a *duration*. An *occurrence* event is counted each time the event takes place. Some events in the Pentium can occur twice in one clock cycle, which causes the counter to be incremented by two. For a *duration* event, the counter counts the total number of clock cycles during which a particular condition is true. The events, including their event code, a brief description, and whether it is an *occurrence* or *duration* event, are summarized in Table 2.3. Each event, or group of events, is described in detail in Appendix A.

Table 2.3. Performance monitoring events

Encoding (in hex)	Event	Occurrence/ Duration
0x00	data read	occurrence
0x01	data write	occurrence
0x28	data read or write	occurrence
0x02	data TLB miss	occurrence
0x03	data read miss	occurrence
0x04	data write miss	occurrence
0x29	data read or write miss	occurrence
0x05	write (hit) to M or E state lines	occurrence
0x06	data cache lines written back	occurrence
0x07	external snoops	occurrence
0x08	data cache snoop hits	occurrence
0x09	memory access in both pipes	occurrence
0x0A	bank conflicts	occurrence
0x0B	misaligned data memory or I/O references	occurrence
0x0C	code read	occurrence
0x0D	code TLB miss	occurrence
0x0E	code cache miss	occurrence
0x0F	any segment register loaded	occurrence
0x12	branches	occurrence
0x13	BTB hits	occurrence
0x14	taken branch or BTB hit	occurrence
0x15	pipeline flushes	occurrence
0x16	instructions executed	occurrence
0x17	instructions executed in the v-pipe	occurrence
0x18	clocks while a bus cycle is in progress	duration
0x19	number of clocks stalled due to full write buffers	duration
0x1A	pipeline stalled waiting for data memory read	duration
0x1B	stall on write to an E or M state line	duration
0x1C	locked bus cycle	occurrence
0x1D	I/O read or write cycle	occurrence
0x1E	non-cacheable memory reads	occurrence
0x1F	pipeline stalled due to address generation interlock	duration
0x22	floating point operations (FLOPs)	occurrence
0x23	breakpoint match on DR0 register	occurrence
0x24	breakpoint match on DR1 register	occurrence
0x25	breakpoint match on DR2 register	occurrence
0x26	breakpoint match on DR3 register	occurrence
0x27	hardware interrupts	occurrence

3. DEVICE DRIVER AND LIBRARY

3.1 Overview

The performance monitoring hardware in the Pentium is useless without some sort of software to configure the counters, then read the values and provide them to the user. Since the PMCs can only be accessed by use of privileged instructions, such software support must be integrated into the operating system itself. However, the operating system interface can be awkward to use, so various levels of abstraction form the software control desired by a user. These levels of abstraction are shown below in Figure 3.1. There are three distinct layers: the hardware itself, the operating system, and the user process.

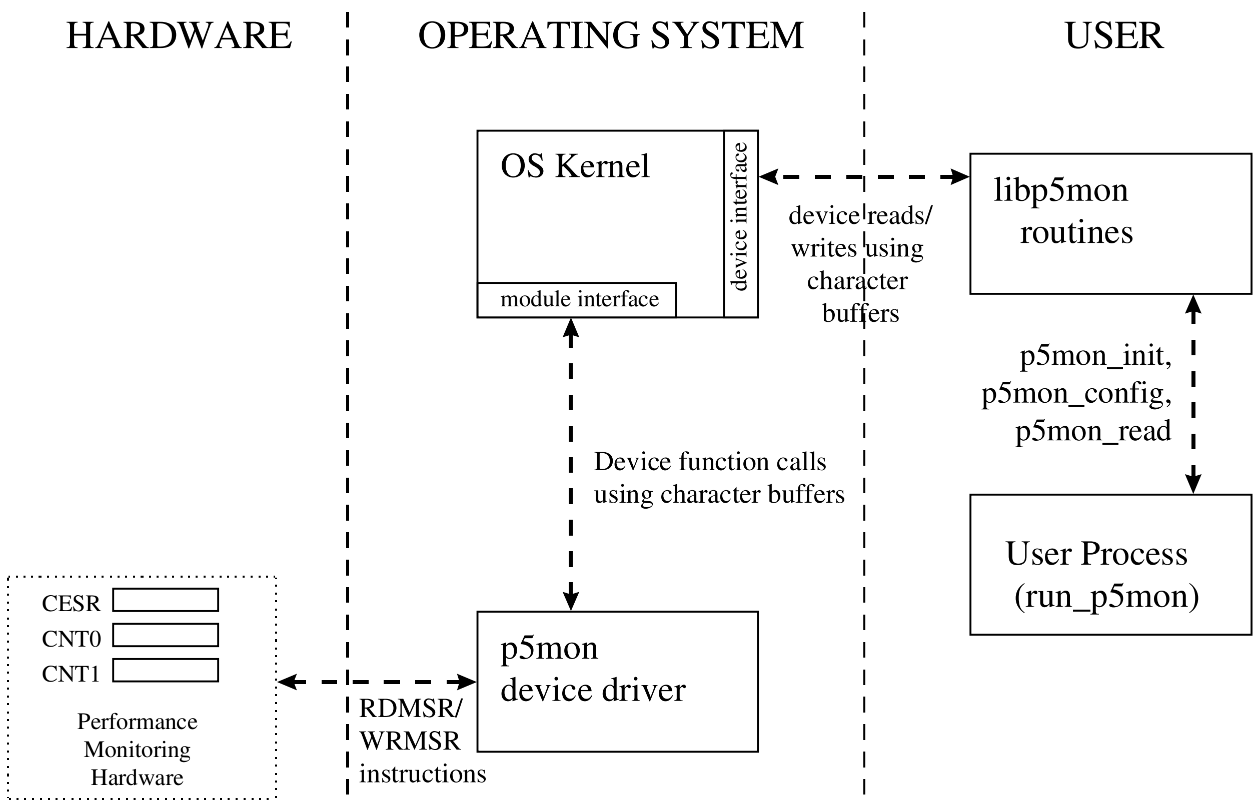

Figure 3.1. Software abstraction levels

The purpose of this chapter is to provide the reader with insight into how the various counters are controlled by the operating system. Two levels of abstraction, a device driver and a library, form the interface between the user and the hardware. Section 2 describes the device driver itself, which forms the lowest level of software support required in the operating system. Section 3 then describes the high-level library routines that provide an interface to the device driver. The highest level of abstraction depicted in Figure 3.1 is described in Chapter 4.

3.2 Linux Device Driver Module

As mentioned in Chapter 2, both the Read Model Specific Register (RDMSR) and Write Model Specific Register (WRMSR) instructions must be executed from a privileged mode. The Intel x86 architecture provides four "rings" of operation, ranging from 0 to 3, with ring 0 being the most privileged level. Generally, the operating system kernel is the only program executing in ring 0. Thus, both the RDMSR and WRMSR instructions must be executed from within the operating system kernel in some fashion. However, in order for the instructions to be useful to the user, there must be some type of interface to allow the user to both read and write the performance monitoring counters.

3.2.1 Access methods

There are two methods of having the operating system kernel provide this type of access to the user. The first method involves using system calls that are built into the kernel itself. In order to add system calls, the code must be compiled directly into the operating system kernel, which involves changing the kernel source code. The second method involves treating the counters as a device in the system. Creating a device driver to act as an intermediary between the operating system and the hardware allows the user to pass “commands” through the operating system’s device interface; this is the approach taken in this thesis work.

3.2.2 Device driver approach

There are several advantages to the device driver approach. The first, and perhaps most important, advantage is that no changes to the operating system kernel source code are necessary. In the case of Linux, a widely-distributed “free” UNIX operating system, modifying the kernel source is relatively easy. However, it would be nearly impossible for a commercial UNIX system. Most UNIX operating systems provide some sort of third-party device driver support to allow hardware developers to control their hardware. Under Linux, the device driver is implemented as a “module” that can attach to the kernel. Furthermore, this module is somewhat independent of the operating system kernel. Thus, if support is to be added to another Linux system, the module would be merely copied over and compiled.

Second, it is simple to create new modules that operate in much the same way for the performance counters that are available in the Pentium Pro or the Pentium with MMX technology. Both processors can monitor different events and have slightly different instructions to access the information. Using a device interface, the paradigm for using the counters remains consistent regardless of the platform.

There are two minor disadvantages to using a device driver over system calls. First, since the hardware is considered a device, it is controlled through a rigidly defined interface that ordinary users might find difficult to use when compared to a custom-designed system call interface. A set of higher-level library routines that actually communicate with the device driver overcomes this problem. The second disadvantage is that a device driver might have more overhead than a system call. The operating system has to go through various layers of functions to get from the device interface to the hardware itself. This second point, as well as overhead in general, is addressed in later chapters.

3.2.3 P5mon device driver

The device driver to control the Pentium's performance monitoring counters is called *p5mon*. The user uses character buffers to send configuration information to write into the MSRs and to read values from the MSRs. Linux provides a common interface to all devices through a set of system calls. During a system call, the operating system relays these buffers to the appropriate function in the p5mon device driver. The device driver itself consists of an initialization function, an open function, a read device function, and a write device function. The device driver itself was written with flexibility and efficiency in mind.

3.2.3.1 Opening p5mon

The p5mon device, usually installed as */dev/p5mon*, is opened using the *open* system call [7]. The only important item of note here is that the device does not lock itself to prevent another process from reading or writing the PMCs. This support is needed to allow two processes to use the device at the same time, which is essential for doing some types of measurement. The user on the system must ensure that no other process uses the counters while trying to measure events in a program.

3.2.3.2 Configuring the PMCs

The PMCs are configured through the *write* system call [8]. As mentioned earlier, the configuration information is passed through a character buffer. Note that in the Intel architecture, a character is one byte in length. Three values must be sent via this buffer: a value used to configure the CESR, a flag used to control initialization of the counters, and, optionally, two

40-bit initialization values to write into the PMCs. The format of this character buffer is shown in Figure 3.2. In all cases where a field spans multiple bytes, the field is stored in little endian format, meaning the lowest byte of the field is stored in the first byte of the buffer, the next byte in the second byte of the buffer, and so on.

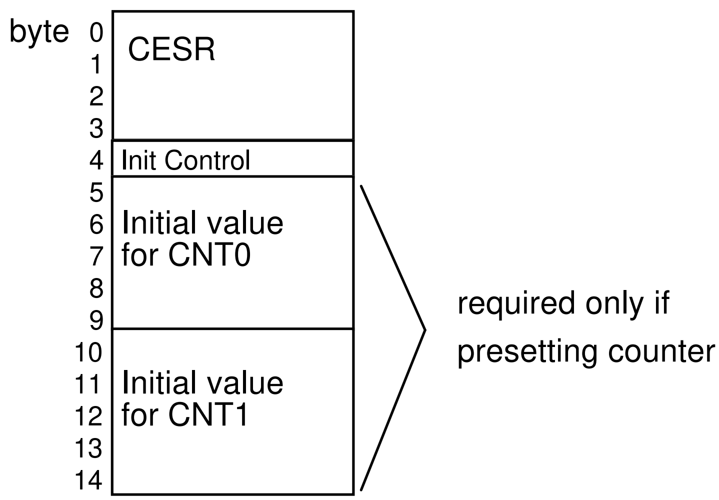

Figure 3.2. P5mon device write buffer

The user is responsible for setting up the CESR correctly by providing two events and a counter control for each event in the correct positions within the 32-bit CESR word as shown in Figure 2.1 (page 8). However, the user does not have to worry about assigning the reserved positions of the CESR since the device driver will actually first perform a read of the CESR, combine only the unreserved bits with the user-specified value, and finally write the new value back to the CESR. The fifth byte in the buffer is an integer, represented as an enumerated type, specifying the type of counter initialization to perform. Since writing the CESR does not automatically clear the two counters, the user must specify whether the counters are to be initialized to zero, preset with some specified value, or left alone. Furthermore, the user can specify different types of initialization for each of the two counters. The various initialization modes are summarized in Table 3.1. Finally, if the counters are to be preset, the values are provided in the next ten bytes of the buffer.

Table 3.1. Counter initialization control

Value	Name	Description
0	clr_none	leave both counters alone
1	clr_0	clear CNT0, leave CNT1 alone
2	clr_1	clear CNT1, leave CNT0 alone
3	clr_0_1	clear both CNT0 and CNT1
4	init_0_clr_1	initialize CNT0, clear CNT1
5	init_1_clr_0	initialize CNT1, clear CNT0
6	init_0	initialize CNT0, leave CNT1 alone
7	init_1	initialize CNT1, leave CNT0 alone
8	init_0_1	initialize both CNT0 and CNT1
9	clear_only	clear both CNT0 and CNT1, do not reconfigure CESR

3.2.3.3 Reading the PMCs

Reading the PMCs, using the **`read`** system call, is much simpler than writing the counters [9]. The user passes a pointer to an empty 24-byte character buffer to the read device routine. This routine then places the values of counter 0, counter 1, and CESR into that buffer. The buffer format is shown in Figure 3.3.

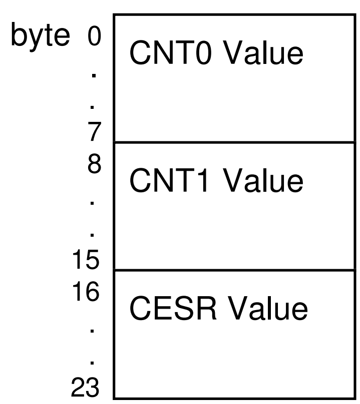

Figure 3.3. P5mon device read buffer

While it is possible to create a smaller buffer, forming the buffer in this way allows much easier creation and access to the three elements in the buffer, either by using an array of six 32-bit unsigned integer types or an array of three 64-bit “long long” unsigned integer types. There is one caveat about using the 64-bit integer type. While GCC does support the 64-bit integer type internally, it is not supported by any library functions, such as *printf*. In order to display the values, they must first be converted to “double” (floating point) types.

3.2.3.4 Example p5mon device driver usage

Figure 3.4 contains a C program that illustrates how to use system calls to directly to access the PMCs through the p5mon device driver. This simple example creates a configuration buffer, opens the p5mon device, writes to the device, reads values out of the device, converts the values to “doubles”, and finally displays the values to the screen. Comments within the code explain what is occurring at each step.

3.3 Library Routines

Using the device driver interface can be very difficult. It is prone to errors with incorrect character buffer sizes and formats, as well as incorrect command usage. In the interest of efficiency, very little error checking is done in the device driver itself. In order to provide some sort of error checking, as well as a more powerful interface, a set of library routines has been created to allow a user-level process to easily manage the p5mon device. In addition to providing higher-level “configuration” and “read” functions that operate on structures, the library routines provide a set of symbolic names to relate the event codes to the events they actually measure. By having the library routines carefully build up the character buffers, fatal errors involving the device driver will not occur. Ideally, any user interaction with the p5mon device will occur only through

```
int main() {
      int P5MON;  /* holds the p5mon device ID */
      unsigned long long int buf[3]; /* 24-byte reading buffer */
      char outbuf[5];  /* writing buffer */
      double temp1, temp2; /* holds the results from the buffer */

      /* Form the buffer to send to the device */
      /* Since clearing the counters, only need a 5-byte buffer */
      outbuf[0]=0x83; /* Event 0 is Data Read Miss, */
      outbuf[1]=0;    /*   and count user-level events */
      outbuf[2]=0x80; /* Event 1 is Data Read, */
      outbuf[3]=0;    /*   and count user-level events */
      outbuf[4]=3;    /* want to clear both counters */

      /* Open the device using the "open" system call */
      P5MON = open("/dev/p5mon", O_RDWR);
      if(P5MON == 0) {
          printf("opening unsuccessful ");
          exit(1);
      }

      write(P5MON, outbuf, 5);  /* Configure the counters */

      read(P5MON, buf, 24);     /* Read values back out */

      /* Convert the unsigned long longs into 40-bit doubles */
      temp1 = buf[0];
      temp2 = buf[1];

      printf("Counter 0 value: %f\n", temp1);
      printf("Counter 1 value: %f\n", temp2);
      printf("Ratio: %f\n", temp1/temp2);

      close(P5MON);  /* close the device */
      return 0;
}
```

Figure 3.4. Example C program using p5mon device directly

this library. In the sections that follow, the library is first described, then is followed by an example using the library routines to obtain information from the PMCs.

3.3.1 Enumerated types

The library defines three sets of enumerated types that allow the user to specify symbolic names instead of numbers to reference certain key aspects of controlling the PMC configuration. These enumerated types are useful for any application using the p5mon device.

3.3.1.1 Events (EVENT)

The first enumerated type, defined to be the type EVENT, includes all possible events (described in Table 2.3, page 10) that the PMCs can measure, providing a mapping between event names and their corresponding event codes. This mapping is shown in Table 3.2. The number associated with each enumerated event is identical to the hex code for the event. Two additional special events, one that counts nothing and one that counts clock cycles, are also available. In addition to the enumerated type names themselves, a static character array containing the event names is available. This allows the user to easily output the name of the event based on the event number.

Table 3.2. Enumerated event names

Event Code	Enumerated Event Name	Event Code	Enumerated Event Name
0x00	data_read	0x14	taken_branch_or_btb_hit
0x01	data_write	0x15	pipeline_flushes
0x28	data_read_or_write	0x16	instructions_executed
0x02	data_tlb_miss	0x17	instructions_executed_in_v_pipe
0x03	data_read_miss	0x18	clocks_while_bus_cycle_in_progress
0x04	data_write_miss	0x19	clocks_stalled_full_write_buffers
0x29	data_read_or_write_miss	0x1A	clocks_stalled_waiting_data_read
0x05	write_to_M_or_E_lines	0x1B	clocks_stalled_write_to_M_or_E_line
0x06	data_cache_lines_written_back	0x1C	locked_bus_cycle
0x07	external_snoops	0x1D	IO_read_or_write_cycle
0x08	data_cache_snooping_hits	0x1E	non_cacheable_memory_reads
0x09	memory_accesses_in_both_pipes	0x1F	pipeline_stalled_address_generation_interlock
0x0A	bank_conflicts	0x22	flops
0x0B	misaligned_data_memory_or_IO_references	0x23	breakpoint_match_DR0
0x0C	code_read	0x24	breakpoint_match_DR1
0x0D	code_tlb_miss	0x25	breakpoint_match_DR2
0x0E	code_cache_miss	0x26	breakpoint_match_DR3
0x0F	segment_register_loaded	0x27	hardware_interrupts
0x12	branches	-none-	count_clocks
0x13	btb_hits	-none-	event_disable (count nothing)

3.3.1.2 Counter control (CNT_CTRL)

The second enumerated type, CNT_CTRL, specifies how each counter should be controlled. As discussed in Chapter 2, each counter may be configured to count events or clocks within the four different processor rings. Both counters can be configured individually. For example, the user may wish to count a certain event within the kernel level in one of the counters and count the same event within the user level in the other counter, thus allowing a comparison to be made between the user and kernel (i.e., operating system) levels. The different control modes and their associated enumerated type names are shown in Table 3.3.

Table 3.3. Counter control

Name	**Description**
evt_disable	disable counter
evt_kernel	count kernel-level events
evt_user	count user-level events
evt_all	count both kernel- and user-level events
clk_kernel	count kernel-level clock cycles
clk_user	count user-level clock cycles
clk_all	count both kernel- and user-level clock cycles

3.3.1.3 Counter initialization (CNT_CLR)

The final enumerated type, CNT_CLR, describes how the counters are to be initialized. As mentioned in Section 3.2, each counter can be independently cleared or preset to a specified value. An additional name called *clear_only* signals the library routines to only clear the counters and not write to the CESR. Table 3.4 lists the various initialization combinations and their associated enumerated type names.

Table 3.4. Counter initialization control

Name	Description
clr_none	leave both counters alone
clr_0	clear CNT0, leave CNT1 alone
clr_1	clear CNT1, leave CNT0 alone
clr_0_1	clear both CNT0 and CNT1
init_0_clr_1	initialize CNT0, clear CNT1
init_1_clr_0	initialize CNT1, clear CNT0
init_0	initialize CNT0, leave CNT1 alone
init_1	initialize CNT1, leave CNT0 alone
init_0_1	initialize both CNT0 and CNT1
clear_only	clear both CNT0 and CNT1, do not reconfigure CESR

3.3.2 Structures

A configuration structure, CNT_CONFIG, is used to allow the user to configure the PMCs. The user merely fills in the fields of the structure, then the library functions convert the structure into the character buffer format that the p5mon device understands. Similarly, another routine can generate a configuration structure given the value of the CESR. The structure itself contains seven different fields. Table 3.4 lists the various fields in the structure, their types, and a brief description.

Table 3.5. CNT_CONFIG structure fields

Structure Field	Type	Description
event0	EVENT	event to be counted in CNT0
event1	EVENT	event to be counted in CNT1
ctrl1	CNT_CTRL	CNT1 operation control
ctrl0	CNT_CTRL	CNT0 operation control
clr	CNT_CLR	configuration mode to use
init0	64-bit integer	40-bit initial value for CNT0
init1	64-bit integer	40-bit initial value for CNT1

3.3.3 Functions

The other features in the library are the functions that act as an interface between the user and the device driver. The prototypes for six different functions are shown in Figure 3.5 and are described in the following sections.

```
int p5mon_init();
int p5mon_close();
int p5mon_clear();
int p5mon_config(CNT_CONFIG *config);
int p5mon_read(ULONG *result);
void p5mon_gen_config(ULONG cesr, CNT_CONFIG *config);
```

Figure 3.5. Libp5mon function prototypes

3.3.3.1 p5mon_init and p5mon_close

These two functions open and close the p5mon device. The device must be initialized before it is used, and should be closed before the program exits.

3.3.3.2 p5mon_clear

This function clears the two counters but does not reconfigure the CESR. This allows the user to reset the counters whenever desired without knowing what events are actually being counted.

3.3.3.3 p5mon_config

This function is the most complicated function in the library. As its name suggests, it is responsible for configuring the CESR to count various events. It takes the configuration structure described in Section 3.3.2 as input, builds the character write buffer described in Section 3.2.3.2, and writes the character buffer to the p5mon device. This function also provides some error checking on the configuration structure as well.

3.3.3.4 p5mon_read

This is the companion function to *p5mon_config*. It simply reads both counters and the CESR, and puts the results into an array of three 64-bit integers in that order.

3.3.3.5 p5mon_gen_config

This function fills in the configuration structure fields based on the CESR value passed to the function, so that the user doesn't have to know the format of the CESR to interpret what is being counted.

3.3.4 Example p5mon library routines usage

Figure 3.6 contains a C program that illustrates how to use the p5mon library routines to access the PMCs. This example has the same function as the example shown in Figure 3.4 (page 18) and is useful for seeing the differences in using the library routines versus directly accessing the p5mon device. This simple example initializes the p5mon device, creates a simple configuration structure, calls the configuration routine, uses the read function to get the value of the counters, and finally prints the values to *stdout* (standard out). The primary difference is in the method of configuring the CESR. Comments within the code explain what is occurring at each step.

3.4 Summary

The device driver and the library provide the interface needed to access the Pentium's performance monitoring counters. Appendix B describes how to install the p5mon device and library onto a Linux system. For more detailed information, the reader is encouraged to look at the source code. These software routines provide access to the counters, but they don't provide power. The real power of the counters comes from the ability to measure different events in real programs, look at multiple results to compute averages and other statistics, and examine smaller regions of a program without having to worry about all the details of creating configuration

```
#include "libp5mon.h"

int main() {

      CNT_CONFIG p5mon_cfg;  /* configuration structure */
      unsigned long long results[3]; /* array for results */
      double temp1, temp2;   /* temp values */

      /* initialize the p5mon device */
      if(p5mon_init()) {
           printf("Unable to initialize p5mon device\n");
            exit(-1);
      }
      /* Build up the configuration structure */
      p5mon_cfg.event0 = data_read_miss;
      p5mon_cfg.ctrl0 = evt_user;
      p5mon_cfg.init0 = (unsigned long long) 0x0;
      p5mon_cfg.event1 = data_read;
      p5mon_cfg.ctrl1 = evt_user;
      p5mon_cfg.init1 = (unsigned long long) 0x0;
      p5mon_cfg.clr = init_0_1;

      p5mon_config(&p5mon_cfg); /* Configure PMCs */
      p5mon_read(results); /* Read PMCs */

      /* Convert the unsigned long longs into 40-bit doubles */
      temp1 = (double) results[0]
      temp2 = (double) results[1]

      printf("Counter 0 value: %f\n", temp1);
      printf("Counter 1 value: %f\n", temp2);
      printf("Ratio: %f\n", temp1/temp2);

      p5mon_close();  /* close the device */
      exit(0);
}
```

Figure 3.6. Example C program using p5mon library routines

structures and reading the counters. A program called run_p5mon uses the library routines described in this chapter to create the power of this software.

4. RUN_P5MON

4.1 Motivation

Run_p5mon is a C program that allows a user to easily measure different events in an executable with little or no modifications to the source code of the program. Run_p5mon acts as an interface to the p5mon library routines to allow the user to easily turn on and off various events, run each event multiple times, and compute statistics for each event. By using run_p5mon, the user doesn't have to worry about configuring the performance monitoring counters (PMCs), managing buffers, or keeping track of what events have been measured.

Run_p5mon has been designed with two goals in mind. It allows a user to measure events on a per-executable basis without any modification to the program's source code. Since there is no source code modification, no recompilation of the program is required to measure events. By using this method, the user can obtain statistics for the entire program. However, sometimes the user would like to obtain finer-grain data on the program. In order to support this, run_p5mon's second goal is to allow the user to make some minor additions to the assembly-level source code, recompile this modified program, and then use run_p5mon as a driver for this modified program.

4.2 Overview of run_p5mon

Run_p5mon acts as a controlling process for measuring events in a program. Fully configurable by the user, it is responsible for running the program to be monitored, gathering and computing statistics, and displaying the results. It uses the p5mon library functions to access the p5mon device driver as described in Chapter 3. In fact, it can be thought of as the "user process" portion within the abstraction layers shown in Figure 3.1 (page 11). Figure 4.1 is a high-level algorithm showing how run_p5mon works.

```
read parameters file
     while (current_repetition < total_repetitions) {
          while (events_remaining) {
               get the next two events to be counted
               configure the PMCs
               execute the program
               read and store the value of the PMCs
          }
          current_repetition++
     }
     compute statistics
     output results
```

Figure 4.1. Run_p5mon algorithm

Run_p5mon takes an arbitrary length event list, as specified by the configuration options, and pairs up the events to measure two at a time, since the PMCs allow at most two events to be measure simultaneously. In order to measure many events, run_p5mon will actually execute the program multiple times. In addition, since the results may not be precise, the user may wish to measure each individual event multiple times. This is termed the number of repetitions. This should not be confused with having to run the program multiple times to gather data for all the events. Run_p5mon can support both unmodified and modified executables.

4.2.1 Unmodified executable

As the name suggests, running an unmodified executable requires no modification to the machine-level code. Events are measured from the program's entry to its exit, including the operating system overhead in order to create and destroy the process.

4.2.2 Modified executable

If the user desires finer-grain information, probes can be inserted into the assembly code of the program to define regions. These regions can overlap or be executed multiple times (e.g., loops). To define the regions, the user inserts one-line function calls in the assembly code, called "assembly stubs," that act as probes. These one-line function calls are actually wrappers to simple C functions. Up to 100 regions may be defined in a program in the current version of run_p5mon. More regions may be added by changing a constant within run_p5mon and recompiling.

As the modified program is being executed, the assembly stubs and, in turn, the C functions are called. The begin and end region function calls merely sample the PMCs as the program executes and store the difference in an accumulator corresponding to the region. When the modified program exits, all data are written to a temporary file that run_p5mon then uses to compute statistics to output to the user. Just as with the unmodified executable, run_p5mon acts as a driver by configuring the PMCs and controlling the number of times the program must be executed to measure all of the events. In order to use these stubs, the user must have access to the assembly code and be willing to recompile the code after inserting the stubs. These function calls add a relatively constant overhead to the data being gathered each time they are called.

All of the assembly stub routines are defined in two files. The file *p5mon_asm.c* contains all of the C functions and the file *p5mon_asm_elf.s* contains the assembly stubs themselves. Both files are automatically compiled into the p5mon library (libp5mon). When the modified assembly code is recompiled, it must be linked to the p5mon library by adding the *-lp5mon* flag to the compiler command line.

Before defining any regions, it is necessary to call the initialization function, *p5mon_asm_init*, in order to create some data structures to hold information. Regions themselves are defined between a "begin region" and "end region" function call. These two functions take the form of *p5mon_asm_begreg_X* and *p5mon_asm_endreg_X*, where *X* is a number between 0 and 99. The begin-region function samples the PMCs and stores the value in a temporary array. The end-region function again samples the PMCs, subtracts the starting value, and adds the result into an accumulator. In order to have the correct value in the accumulator, the user should make sure that the number of times that a region is entered is the same as the number of times that it is exited. A detailed example using the assembly regions is provided in Chapter 5.

4.3 Configuring run_p5mon

The power of run_p5mon lies in its ability to be easily configured to perform a variety of tasks, freeing the user from having to manipulate the PMCs, keeping track of buffers, and computing statistics.

4.3.1 IMPACT parameters facility

All modules within the IMPACT (Illinois Microarchitecture Project Utilizing Advanced Compiler Technology) research compiler use the same parameter facility for configuration. Each IMPACT module can have its own set of parameters, located in what is known as a PARMS (or parameters) file. Run_p5mon has its own section, called "p5mon," that specifies its configuration. These parameters control all aspects of run_p5mon, from the events to measure to the executable to use. In addition to the PARMS file itself, the user may use "-P" and "-F" command line switches to control run_p5mon configuration. For more information on the IMPACT parameter facility, the reader is encouraged to look at the PARMS tutorial [10]. Using this parameter passing facility allows a user to customize run_p5mon for a given benchmark, just as the user would for any other aspects of compilation within IMPACT.

4.3.2 Basic configuration options

Below are the six basic configuration options for run_p5mon. The heading for each subsection is also the parameter name. Most configuration values have some sort of default value that is indicated. All configuration parameters, as well as a sample PARMS file, are summarized in Appendix C.

4.3.2.1 command

Quite simply, this is the program measured by run_p5mon. The command parameter should contain the full command line used to invoke this program, including any command line arguments or redirection. If the executable is not in the current directory, the full path should also be specified. The executable format of the binary, a.out or ELF, does not matter. This parameter has no default value and must be specified.

4.3.2.2 repetitions

This parameter specifies how many times each event should be measured. This parameter has a default value of one.

4.3.2.3 user_events, kernel_events

These two parameters control the level, user, kernel, or both, in which the events should be counted. The default is to count only user-level events. If both user- and kernel-level events are enabled, both are counted. If neither is enabled, a warning message will be printed because nothing is being counted.

4.3.2.4 output_all_results

This parameter allows the user to see the results obtained for each repetition of each event, as well as a final average and confidence interval based on all the repetitions. This parameter might be used to analyze results both when there is a relatively wide confidence interval and when the confidence interval is equal to zero. If the number of repetitions is large, a great deal of information will be printed. The default for this parameter is "no," meaning the extra information is not printed.

4.3.2.5 confidence_level_99

This parameter is used to trigger which statistical confidence interval, 95% or 99%, is provided in the results. The default confidence interval is 95%. If the user wishes to have a higher confidence interval, this parameter can be enabled to calculate the 99% confidence interval. The statistical confidence intervals are discussed in Section 4.5.1.

4.3.3 Advanced configuration options

In addition to the six basic configuration parameters, five advanced configuration options allow even more control of run_p5mon.

4.3.3.1 use_assembly_regions

This parameter instructs run_p5mon to run an executable that contains assembly stubs. By default, it is disabled. If assembly stubs are being used, this parameter must be enabled in order to instruct run_p5mon to look for the file containing the data gathered on the regions generated by the modified executable.

4.3.3.2 max_regions

This parameter is used in conjunction with the previous parameter. It informs run_p5mon of the maximum number of regions present in the modified executable, which cannot be greater than 100. By default, this parameter is set to ten. It helps run_p5mon to limit memory usage while computing statistics.

4.3.3.3 output_file

This parameter specifies the name of a file in which to write the results. The results are written out in a format that can be easily parsed to allow the data to be used by other programs. Even when an output file is specified, results are written to *stdout*. The format of the output file is given in Section 4.5.3.

4.3.3.4 skip_initialization_run

By default, before gathering data on an executable, run_p5mon will first execute the program to try to minimize some of the start-up effects, such as cold-start cache misses. If the user does not wish to perform this initialization, this parameter should be set to "no." Be aware, however, that the values of the first two events measured will be affected.

4.3.3.5 verbose_level

This integer parameter is used to see the status of run_p5mon as it executes. This parameter is useful if run_p5mon takes a long time to execute. By default, this parameter is set to zero, which means all messages are turned off. Level 1 prints a message at the beginning of each repetition. Level 2 prints a message at the start of each execution and specifies the events being measured. All messages are printed to *stderr* (standard error) so that normal program output may be redirected.

4.3.3.6 Events

Thirty-nine different events can be counted, including the 38 that the hardware directly supports, plus one event to count total clock cycles. Each event can be turned on or off as needed by setting its parameter to a "yes" or a "no" value. By default, all events are disabled. The name of each parameter is very similar to the actual event names. These parameters are summarized in Appendix C.

4.4 Results

After gathering results for all the specified events, run_p5mon generates a report containing the raw data as well as averages and confidence intervals on the data gathered.

4.4.1 Run_p5mon statistics

Run_p5mon computes two different statistics for each event that it measures. First, run_p5mon computes an average value of the event based on the data for each repetition. Second, run_p5mon computes a two-sided confidence level on the data. Run_p5mon can compute both a 95% and a 99% confidence interval, based on the value of the *conf_level_99* parameter. This confidence interval is computed using standard statistical techniques [4]. The meaning of the 95% confidence interval is that the probability that the "real" value lies within the range of the average, plus or minus the confidence interval, is equal to 95%. Thus, there is a 95% probability that if the same event were measured under the same set of conditions, including system load and input parameters, the result obtained would lie within this range.

4.4.2 Output report format

After computing statistics, run_p5mon generates a report. The two sections of the report, the header and the detailed information, are described below. The information contained in the detailed information section depends on the values of the *output_all_results* and *use_assembly_regions* parameters.

4.4.2.1 Header information

The first part of the report, the header, consists of the date and time run_p5mon started, the directory in which it was invoked, the parameter file read, the command-line arguments passed to run_p5mon (such as the -F arguments), and the program actually executed. A sample header is shown in Figure 4.2.

```
run_p5mon started at Thu Jan 30 16:02:20 1997
   current directory: /home/fixed/safford/p5mon
   parameter file: ./STD_PARMS
   arguments:
   command: hello
```

Figure 4.2. Sample header from run_p5mon

4.4.2.2 Summary information

The second part of the report contains the data gathered on the program execution behavior. The exact format of this section depends on the parameters described in Section 4.3. The first line of this section will contain the number of repetitions and the confidence interval, then the results of each event are given on a single line, with the event name first, followed by the average value. After the average, the confidence interval is displayed as “+/- <conf int> (<percentage>%)” where “<conf_int>” is the raw confidence interval, and “<percentage>” is the confidence interval expressed as a percentage of the average. The final lines of the report show the total run time for the entire execution of the run_p5mon program, as well as the number of times the program actually executed to gather the data. A sample results section is shown in Figure 4.3.

```
Results (for 5 repetitions with a 95% confidence level):
   Data Read: 201900.0 +/- 2.8 (0.001%)
   Data TLB Miss: 269.0 +/- 0.0 (0.000%)
   Code Read: 202666.4 +/- 38.8 (0.019%)
   Code TLB Miss: 525.4 +/- 10.2 (1.941%)
   Instructions Executed in the v-pipe: 141021.6 +/- 21.5 (0.015%)
   Clock cycles: 201900.0 +/- 2.8 (0.001%)

Total elapsed time: 1.0 seconds
   (program executed 15 times, with an average execution time of 0.1s)
```

Figure 4.3. Sample results section from run_p5mon

4.4.2.3 Detailed information

Setting the parameter *output_all_results* to "yes" displays the data collected for each repetition of each event, in addition to the summary information. The value for each repetition of the event is shown on its own line. Averages and confidence intervals follow in the same format as described in Section 4.4.2.2. A sample results section containing this information is shown in Figure 4.4.

```
Results (for 5 repetitions with a 95% confidence level):
   Event: Data Read
     rep  0:       201902
     rep  1:       201902
     rep  2:       201902
     rep  3:       201902
     rep  4:       201907
     ----------------------
     Average:      201903.0 +/- 2.3 (0.001%)
   Event: Data TLB Miss
     rep  0:          269
     rep  1:          269
     rep  2:          269
     rep  3:          269
     rep  4:          271
     ----------------------
     Average:         269.4 +/- 0.9 (0.341%)
Total elapsed time: 1.3 seconds
   (program executed 5 times, with an average execution time of 0.3s)
```

Figure 4.4. Sample results section containing results from each repetition

4.4.2.4 Regions

If the executable contains assembly regions, the output will have information broken down by regions. The first line of the results section contains the number of regions, the number of repetitions, and the confidence level computed. Each region itself has a header, containing both the region number and the number of times the particular region was entered and exited. Within each region, each event measured results in a corresponding summary line, in a format similar to the summary line information described in Section 4.4.2.2. The only addition is the number within brackets at the end of the line, which is the average value divided by the number of times the region was entered. If the *output_all_results* parameter is set to "yes," each event within each region will also have a corresponding output line, as described in Section 4.4.2.3. The next two figures, Figures 4.5 and 4.6, show the results using multiple regions, both with and without this parameter set to "yes."

```
Results (for 3 regions, 3 repetitions, 95% confidence level):
   Region 0,  entered 1 times and exited 1 times:
       Branches: 797.0 +/- 0.0 (0.000%) [797.0]
       BTB Hits: 502.3 +/- 11.3 (2.241%) [502.3]
   Region 1,  entered 100 times and exited 100 times:
       Branches: 3900.0 +/- 0.0 (0.000%) [39.0]
       BTB Hits: 2993.0 +/- 6.5 (0.218%) [29.9]
   Region 2,  entered 1 times and exited 1 times:
       Branches: 174.0 +/- 0.0 (0.000%) [174.0]
       BTB Hits: 57.3 +/- 3.1 (5.446%) [57.3]
```

Figure 4.5. Sample results with multiple regions

4.4.3 Output file format

In addition to outputting the results to *stdout*, run_p5mon also creates a file containing data that is suitable for parsing when a file is specified in the *output_file* parameter. The data contained in this file will differ depending on the value of the *output_all_results* parameter and whether regions have been defined. Any line beginning with a “#” in the output file is considered

```
Results (for 3 regions, 3 repetitions, 95% confidence level):

   Region 0,  entered 1 times and exited 1 times:
       Event: Branches
          rep 1:           797
          rep 2:           797
          rep 3:           797
          ----------------------
          Average:         797.0 +/- 0.0 (0.000%)
                           797.0 per time region entered
       Event: BTB Hits
          rep 1:           498
          rep 2:           498
          rep 3:           511
          ----------------------
          Average:         502.3 +/- 11.3 (2.241%)
                           502.3 per time region entered

   Region 1,  entered 100 times and exited 100 times:
       Event: Branches
          rep 1:          3900
          rep 2:          3900
          rep 3:          3900
          ----------------------
          Average:        3900.0 +/- 0.0 (0.000%)
                            39.0 per time region entered
       Event: BTB Hits
          rep 1:          2995
          rep 2:          2996
          rep 3:          2988
          ----------------------
          Average:        2993.0 +/- 6.5 (0.218%)
                            29.9 per time region entered

   Region 2,  entered 1 times and exited 1 times:
       Event: Branches
          rep 1:           174
          rep 2:           174
          rep 3:           174
          ----------------------
          Average:         174.0 +/- 0.0 (0.000%)
                           174.0 per time region entered
       Event: BTB Hits
          rep 1:            59
          rep 2:            58
          rep 3:            55
          ----------------------
          Average:          57.3 +/- 3.1 (5.446%)
                            57.3 per time region entered
```

Figure 4.6. Sample results with multiple regions, displaying the results for each repetition

a comment. The first part of the file is a header specifying the number of regions (if regions are being used), the number of repetitions, and the confidence level. The next line contains a listing of the fields present in this file, to help guide parsing the file. Following the header lines are the results. In order to conserve space and simplify parsing, the events are specified by their actual event codes.

4.4.3.1 All results with no regions defined

When the user uses an unmodified executable and requests the results from each repetition, each result will lie on a separate line. The event code is the first number on each line, followed by the repetition that the data corresponds to, and finally, the value of the counter for that repetition. However, a repetition number of "-1" indicates that the line contains summary information. These lines have a slightly different format. The third number is the average result from all the repetitions of this event. The fourth and fifth numbers are the confidence interval expressed as a number and then a percentage of the average value. Figure 4.7 contains a sample output file with this information.

```
# run_p5mon results file
# Results (5 repetitions, 95% confidence level):
# Event Repetition Value [conf.int] [percentage]
0x3 0          11113
0x3 1          11003
0x3 2          10962
0x3 3          10975
0x3 4          10979
0x3 -1        11006.4 63.2 0.574
0x4 0           8856
0x4 1           8876
0x4 2           8816
0x4 3           8850
0x4 4           8845
0x4 -1         8848.6 22.3 0.252
```

Figure 4.7. Sample output file containing all results with no regions

4.4.3.2 Summary information only, with no regions defined

When the user requests only summary information for each event, only the summary lines are printed. Sample output is shown in Figure 4.8.

```
# run_p5mon results file
# Results (5 repetitions, 95% confidence level):
# Event Repetition Value [conf.int] [percentage]
0x3 -1        11800.4 50.9 0.431
0x4 -1        10565.6 25.9 0.245
```

Figure 4.8. Sample output file with no regions, and summary information only

4.4.3.3 Regions defined

When a modified executable containing regions is used, the region number is simply added to the beginning of each line. The rest of the fields do not change. To make reading easier, comment lines are inserted at the beginning of each region. Figure 4.9 shows all of the results of an executable with multiple regions.

```
# run_p5mon results file
# Results (for 2 regions, 5 repetitions, 95% confidence level):
# Region Event Repetition Value [conf.int] [percentage]
# Region 0: entered 1 exited 1
0 0x3 0          5
0 0x3 1          5
0 0x3 2          5
0 0x3 3          5
0 0x3 4          5
0 0x3 -1        5.0 0.0 0.000 5.0
0 0x4 0          6
0 0x4 1          6
0 0x4 2          6
0 0x4 3          6
0 0x4 4          6
0 0x4 -1        6.0 0.0 0.000 6.0
# Region 1: entered 1 exited 1
1 0x3 0          0
1 0x3 1          0
1 0x3 2          0
1 0x3 3          0
1 0x3 4          0
1 0x3 -1        0.0 0.0 NaN 0.0
1 0x4 0          2
1 0x4 1          2
1 0x4 2          2
1 0x4 3          2
1 0x4 4          2
1 0x4 -1        2.0 0.0 0.000 2.0
```

Figure 4.9. Sample output file with multiple regions

4.5 Interpreting the Results

When examining the results obtained from run_p5mon, the user must realize that there are several sources of error in the values measured. First, and most important, since the PMCs are not context switched, they measure the activity in the entire system. Thus, background noise caused by other running processes in the system will cause the results to vary. Results with wide confidence intervals are one sign of an excessive amount of background noise in the system. There are several ways of removing the effect of background processes. The first, and simplest, is to simply not run any other jobs in the system while gathering data using run_p5mon. This includes any system processes that are not crucial to system operation. Another method is to lower the priority of the other processes with the *nice* or *renice* system commands, and to raise the priority of the run_p5mon process. This will, in effect, stop the other jobs in the system without having to kill them.

The second source of error is the operating system overhead incurred when creating and destroying a process. This overhead can be eliminated by defining one assembly-level region covering the entire program. This is the most accurate method to measure events on an entire executable, and should be used whenever possible. Also, since this effect can be easily measured to an extent, a user can subtract the overhead from the results obtained in order to find the true values. Appendix D lists the approximate overhead values observed when using these counters.

Another source of error is the overhead in accessing the counters themselves. In order to read the PMC values, several layers of function calls are required, all of which will have effects that are counted in the PMCs. These function calls can also have effects on the instruction and data caches, as they may replace values needed by the program, as well as effects on the Branch Target Buffer. The Heisenburg principle does hold with these counters. The finer-grain results the user desires, the higher the relative overhead to get the data. The overhead of accessing the PMCs will also change as the program executes, because the code may or may not be in the cache, and branches may or may not be in the BTB. Appendix D also lists the approximate overhead for accessing the counters. Some of this error, however, will be eliminated in the Pentium with MMX and Pentium Pro implementations of this software. This is due to the addition of a user-level Read Performance Monitoring Counter (RDPMC) instruction which will allow the assembly routines to directly sample the counters without having to go through the device driver.

The results obtained on all the events can be considered an upper bound. The real values may be lower than indicated by the counters, since every source of error described above will cause the counter values to be greater than they would otherwise be.

5. USING RUN_P5MON

5.1 Overview

This chapter will help the reader understand how to use run_p5mon to gather data on real programs. The chapter begins with two simple examples, both with an unmodified executable (Section 5.2) and a modified executable (Section 5.3.) Section 5.4 then presents three techniques for best gathering data using run_p5mon. Section 5.5 then uses two small synthetic benchmarks to show how to use the overhead data in Appendix D to calculate the true results. Finally, Sections 5.6 and 5.7 use two programs, *word count* and *compress*, to show how run_p5mon can be used to analyze real programs. Some of the examples described in this chapter are a part of the run_p5mon tutorial [11].

5.2 Basic Usage of an Unmodified Executable

This section contains an example using run_p5mon on an unmodified program. Consider the simple C code presented in Figure 5.1. This program prints a "Hello World" message and executes a loop one million times. This example will use run_p5mon to determine the effects of invoking GCC compiler optimizations.

```
int main() {
  int i,j;
  printf("Hello World!\n");
  for (i=0; i<1000000; i++)
    j += 2;
  printf("j is: %d", j);
  return(0);
}
```

Figure 5.1. Sample hello.c program

After setting the parameters in the PARMS file, run_p5mon is used to gather data on ten events for both an unoptimized and an optimized executable. The commands used to generate and run these two executables are shown in Figure 5.2. The results, without considering the overhead described in Appendix D, are summarized in Table 5.1. The data given are the averages of 20 with a 95% confidence level. As shown by the results, the optimized executable had ten times few data cache reads and writes, about half the branches, and about 2,000,000 fewer instructions than the unoptimized executable. These three factors contributed to the optimized executable requiring three times fewer clock cycles to execute.

```
> gcc -o hello_unopti hello.c
> run_p5mon -Fcommand=hello_unopti
> gcc -o hello_opti -O3 hello.c
> run_p5mon -Fcommand=hello_opti
```

Figure 5.2. Commands executed to gather data on unmodified program

Table 5.1. Results for sample unmodified program

Event	Unoptimized	Optimized
Data read	3305545.2 ± 17.9	305264.5 ± 15.6
Data read miss	11635.5 ± 9.5	11641.2 ± 5.1
Data write	2112898.2 ± 16.5	112737.7 ± 14.0
Data write miss	9226.0 ± 4.8	9240.4 ± 3.4
Memory Accesses in Both Pipes	44983.1 ± 6.2	44937.0 ± 2.4
Branches	2149681.2 ± 5.6	1149340.4 ± 0.5
BTB Hits	2106274.2 ± 20.2	1106090.4 ± 23.0
Instructions Executed	5772170.3 ± 37.0	3770972.5 ± 20.8
Instructions Executed in the v-pipe	1211495.1 ± 105.3	1211047.5 ± 5.8
Clock cycles	10464525.2 ± 3287.7	3456044.4 ± 1957.7

5.3 Basic Usage of a Modified Executable

This example is based on the C code shown in Figure 5.1 (page 43). The optimized assembly code for this program, including line numbers, generated by the command *gcc -S -O3 hello.c*, is shown in Figure 5.3. In Figure 5.4 (page 44), two assembly regions have been added to this code, one region around the entire program and the other around the second call to the *printf* function. The first line added, 5a, is a call to *p5mon_asm_init*, used to set up some data structures as well as register a function to be called when the program exits to dump out the data. The next line, 5b, begins the first region of the program. The first region is ended on line 23b just before the program exits. The second region begins on line 17a and ends on line 20a, covering the entire call to the *printf* function.

```
1     .LC0:
2           .string "Hello World!\n"
3     .LC1:
4           .string "j is: %d"
5     main:
6           pushl %ebp
7           movl %esp,%ebp
8           pushl %ebx
9           pushl $.LC0
10          call printf
11          addl $4,%esp
12          movl $999999,%eax
13          .align 4
14    .L11:
15          addl $2,%ebx
16          decl %eax
17          jns .L11
18          pushl %ebx
19          pushl $.LC1
20          call printf
21          xorl %eax,%eax
22          movl -4(%ebp),%ebx
23          leave
24          ret
```

Figure 5.3. Optimized hello.c assembly code

After modifying the code, the user must recompile the program and link to the p5mon library (libp5mon). Run_p5mon then uses this executable to gather data. The commands used are shown in Figure 5.5. The results for both regions, based on twenty repetitions, a 95% confidence level, and without the overhead removed, are given in Table 5.2. Due to the lesser amount of overhead, region zero's results, covering the entire program, are smaller than the results shown in column two of Table 5.1 (page 43). The results of region 1 show the data for a single call to the *printf* function. Note, however, that the overhead still needs to be removed from these values in order to obtain the true results.

```
.LC0:
        .string "Hello World!\n"
.LC1:
        .string "j is: %d\n"
main:
        call p5mon_asm_init
        call p5mon_asm_begreg_0
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        pushl $.LC0
        call printf
        addl $4,%esp
        movl $999999,%eax
        .align 4
.L11:
        addl $2,%ebx
        decl %eax
        jns .L11
        call p5mon_asm_begreg_1
        pushl %ebx
        pushl $.LC1
        call printf
        call p5mon_asm_endreg_1
        xorl %eax,%eax
        movl -4(%ebp),%ebx
        leave
        call p5mon_asm_endreg_0
        ret
```

Figure 5.4. Modified hello.c assembly code

```
> gcc -o hello_mod hello_mod.s -lp5mon
> run_p5mon -Fcommand=hello_mod -Fuse_assembly_regions=yes
```

Figure 5.5. Commands executed to gather data on modified program

Table 5.2. Results for sample modified program

Event	Region 0	Region 1
Data read	1953.0 ± 15.7	319.0 ± 0.0
Data read miss	126.8 ±- 1.2	9.3 ± 0.3
Data write	931.2 ± 14.1	232.0 ± 0.0
Data write miss	102.3 ±- 2.5	25.4 ± 0.3
Memory Accesses in Both Pipes	310.1 ± 0.5	76.2 ± 0.4
Branches	1000970.0 ± 0.0	166.0 ± 0.0
BTB Hits	1000539.0 ± 2.3	42.8 ± 0.9
Instructions Executed	3004899.0 ± 0.0	817.0 ± 0.0
Instructions Executed in the v-pipe	1001145.8 ± 1.3	161.5 ± 1.3
Clock cycles	2014853.1 ± 498.2	3029.4 ± 70.9

5.4 Techniques for Using Run_p5mon

In this section, three simple yet powerful techniques utilizing assembly regions are presented. The techniques will allow the user to gather the most accurate data on program execution behavior using run_p5mon. For each of the techniques, a small example is presented to illustrate how to use it on a program.

5.4.1 One assembly region covering the whole program

Defining a single assembly region covering the entire program is the most accurate method of obtaining results for the entire program. Since this method uses a modified executable, the operating system overhead of creating and destroying a process is not present. As described in Appendix D, the overhead for using one assembly region seems to be very consistent. To do this, the user must insert three lines into the assembly code. The first two instructions to be executed in the modified program are the *p5mon_asm_init* and *p5mon_asm_begreg_0* functions. These should be placed directly after the label *main* in the assembly code. The final instruction to be executed in the modified program is the *p5mon_asm_endreg_0* function. This function call should be placed directly before the *ret* instruction at the end of the program. If a program has more than one exit point, then this function call should be placed directly before all program exit points. Figure 5.6 illustrates this technique on a small assembly program.

5.4.2 Defining a region around a loop

When analyzing a program, it might be useful for the user to define a region covering a loop. Before doing this, the user must identify all entry and exit points to the loop. Normally, loops have a single entry point, so only one begin-region function call is necessary, and it should become the first instruction of the loop. The user then must find the target of each of the exit

```
.LC0:
        .string "%d\n"
main:
         call p5mon_asm_init
         call p5mon_asm_begreg_0
         pushl %ebp
         movl %esp,%ebp
         movl $9,%eax
         .align 4
.L11:
         addl $2,%edx
         decl %eax
         jns .L11
         pushl %edx
         pushl $.LC0
         call printf
         xorl %eax,%eax
         leave
         call p5mon_asm_endreg_0
         ret
```

Figure 5.6. Using one region around entire program

points and place an end-region function call at that location in the code, with all of the end-region function calls using the same region number as the entry point. The easiest way to verify that all entry and exit points have been properly identified is to look at the results of run_p5mon. If the number of times that a region was entered and exited do not match, then all the entry or exit points are not properly defined. Figure 5.7 presents a simple example of a program showing an assembly region covering a loop with one entry and two exit points.

5.4.3 Nonoverlapping regions

The user may wish to analyze different parts of a program using regions. For example, the user may want to gather data for the entire program, but would also like to gather data on a particular loop within the program. One way to accomplish this would be to define one region covering the entire program and another region around the loop. However, by doing this, the region covering the entire program will also contain all of the overhead for reading the counters around the loop. If the loop is executed many times, this overhead could get very large.

```
main:
        call p5mon_asm_init
        pushl %ebp
        movl %esp,%ebp
        xorl %ecx,%ecx
        movl $9,%eax
        .align 4
.L11:
        call p5mon_asm_begreg_0
        incl %edx
        addl $2,%ecx
        jns .L15                 # This will exit loop
        decl %eax
        call p5mon_asm_endreg_0  # before loopback edge
        jns .L11                 # Loopback edge
pushl %edx
        pushl $.LC0
        call printf
        jmp .L20
.L15
        call p5mon_asm_endreg_0  # Exit target
        pushl %ecx
        pushl $.LCO
        call printf
.L20
        xorl %eax,%eax
        leave
        ret
```

Figure 5.7. Defining region around a loop

A more accurate method of accomplishing the same thing would be to define three nonoverlapping regions. The first region would contain the part of the program leading up to the loop, the second would contain the loop, and the third would contain the rest of the program. To calculate the values for the entire program, the user would add the results from the three regions. In order to illustrate this technique, a simple example, using the same program in Figure 5.7, is given in Figure 5.8.

5.5 Synthetic Benchmarks

This section presents two small synthetic programs to help illustrate how to use the overhead values given in Appendix D to determine the “real” value for an event. The programs are simple enough to calculate the actual values by hand, allowing a comparison to be made

```
main:
        call p5mon_asm_init
        call p5mon_asm_begreg_0
        pushl %ebp
        movl %esp,%ebp
        xorl %ecx,%ecx
        movl $9,%eax
        call p5mon_asm_endreg_0
        .align 4
.L11:
        call p5mon_asm_begreg_1
        incl %edx
        addl $2,%ecx
        jns .L15                  # This will exit loop
        decl %eax
        jns .L11                  # Loop backedge
        call p5mon_asm_endreg_1   # Exit target 1
        call p5mon_asm_begreg_2
        pushl %edx
        pushl $.LC0
        call printf
        jmp .L20
.L15
        call p5mon_asm_endreg_1   # Exit target 2
        call p5mon_asm_begreg_2
        pushl %ecx
        pushl $.LCO
        call printf
.L20
        xorl %eax,%eax
        leave
        call p5mon_asm_endreg_2
        ret
```

Figure 5.8. Using nonoverlapping regions in a program

between the measured and theoretical values for some events. Each of these programs is measured using the technique of one assembly region covering the entire program. The data gathered is based on 10 repetitions with a 95% confidence level for each event.

5.5.1 Program 1

The first program to be considered is a program containing a small loop that executes 100 times. The C code for this program is given in Figure 5.9, and the optimized GCC assembly code with one assembly region defined is given in Figure 5.10.

```
int main() {
  int i,j;
  for (i=0; i<100; i++) {
    j++;
  }
  printf("%d\n", j);
  return(0);
}
```

Figure 5.9. Program 1 C code

```
main:
        call p5mon_asm_init
        call p5mon_asm_begreg_0
        pushl %ebp
        movl %esp,%ebp
        movl $99,%eax
        .align 4
.L11:
        incl %edx
        decl %eax
        jns .L11
        xorl %eax,%eax
        leave
        call p5mon_asm_endreg_0
        ret
```

Figure 5.10. Program 1 modified assembly code

Six events were measured in this small program. For each of the events, Table 5.3 lists the value from run_p5mon, the overhead given in Appendix D, the value calculated by subtracting the overhead from the measured value, and the theoretical value. The results show that the overhead measured is not exact, causing the slight difference between the corrected and theoretical values.

Table 5.3. Program 1 results

Event	run_p5mon	overhead	corrected	theoretical
Branches	126.0 ± 0.0	26.0 ± 0.0	100.0	100
BTB Hits	105.0 ± 0.0	5.0 ± 0.0	100.0	98
Taken Branch or BTB Hit	122.0 ± 0.0	22.0 ± 0.0	100.0	100
Pipeline flushes	19.0 ±- 0.0	17.0 ± 0.0	2.0	2
Instructions Executed	453.0 ± 0.0	147.0 ± 0.0	306	305
Instructions Executed in v-pipe	126.5 ± 1.1	30.0 ± 0.0	96.5	~100

5.5.2 Program 2

This second synthetic program to be considered uses some events related to data cache performance. This program has a small loop, executed three times, that contains a misaligned access. This example is also useful because it shows the errata, as described in Appendix A, with the "data read" and "data write" events being incorrectly incremented twice on a misaligned access. The assembly code for this program is given in Figure 5.11.

```
main:
       call p5mon_asm_init
       call p5mon_asm_begreg_0
       pushl %ebp
       movl %esp,%ebp
       subl $1024,%esp
       xorl %edx,%edx
      .align 4
.L16:
       incl -511(%ebp,%edx,4)  # misaligned access
       incl %edx
       cmpl $2,%edx
       jle .L16
       .align 4
       leave
       call p5mon_asm_endreg_0
       ret
```

Figure 5.11. Program 2 assembly code

Seven events were measured in this small program. Table 5.4 lists the measured results, the overhead values, the corrected results, and the theoretical results. Because there are three misaligned reads and three misaligned writes in the program, it can be seen that the measured results did incorrectly increment the *data read* and *data write* events twice, because they are both three greater than the theoretical value, one for each loop iteration.

Table 5.4. Program 2 results

Event	run_p5mon	overhead	corrected	theoretical
Data Read	76.0 ± 0.0	69.0 ± 0.0	7.0	4
Data Read Miss	1.0 ± 0.0	0.0 ± 0.0	1.0	1
Data Write	73.0 ± 0.0	66.0 ± 0.0	7.0	4
Data Write Miss	4.0 ± 0.0	4.0 ± 0.0	0.0	0
Data Cache Lines Written Back	0.0 ± 0.0	0.0 ± 0.0	0.0	0
Misaligned Data Memory Accesses	14.0 ± 0.0	8.0 ± 0.0	6.0	6
Branches	30.0 ± 0.0	26.0 ± 0.0	4.0	4

5.6 Case Study: Word Count

This section describes using run_p5mon to gather data on the UNIX utility *word count (wc)*. First, the first-level (L1) cache results obtained with the PMCs are compared with those from the IMPACT simulator. This is useful to learn how much effect the library code, which is not included in the trace sent to the simulator, has on the program. Second, the inner loop of *wc* is analyzed to learn about its run-time behavior.

Wc is a utility that counts the number of characters, words, and lines in an input file For all of the experiments, a 4036 line input file containing 105196 characters was used. In the first experiment, the IMPACT cycle-based simulator was configured to match the hardware L1 data cache, L2 data cache, and main memory configuration as closely as possible [6], [12]. The actual cache and memory parameters used are given in Table 5.5. One region covering the entire program body was used in the assembly code generated by the IMPACT optimizing compiler. The results obtained using 20 iterations, a 95% confidence level, and removing the overhead are presented in Table 5.6. The differences between the simulated and measured values are due to the library and system calls present in *wc*, such as *printf* and *getc*.

Table 5.5. Parameters for cache and memory configuration

Parameter	L1 Cache	L2 Cache	Main Memory
Size	8 KB	256 KB	32 MB
Line size	32 bytes	32 bytes	N/A
Write policy	write back	writeback	N/A
Set associativity	2	direct mapped	N/A
Replacement algorithm	LRU	LRU	N/A
Ports	2	N/A	N/A
Data bus width	N/A	64 bit	64 bit
Latency[1]	N/A	3-1-1-1-1-1-1-1 (pipeline burst)	10-3-3-3 (read) 3-1-1-1 (write)
Clock Frequency	100 MHz	66 MHz	66 MHz

[1]The latencies are expressed as follows. The first number represents the cycles required to return the first 64-bit quantity. Subsequent numbers represent the cycles required to return the next three (or seven in the case of the L2 cache) 64-bit chunks of data. Data is returned in critical order, meaning the 64-bit chunk actually requested is returned first.

Table 5.6. Simulated versus PMC results for *wc*

Event	Simulator	PMCs	Difference
Data read	441127	448702.5 ± 1.1	7575
Data read miss	138	1726.8 ± 11.2	1588
Data write	230755	234355.5 ± 1.1	3600
Data write miss	10	28.0 ± 6.3	18
Lines written back	0	200.7 ± 9.2	200
Total instructions executed	2248201	2266817.5 ± 0.2	18616

The second experiment looked at the behavior of *wc*'s inner loop, which actually counts the number of characters, words, and lines for an entire file. A single region was defined around this loop in the IMPACT-generated assembly code. The results are given in Table 5.7. One interesting result is that the branches within the inner loop are hard to predict correctly. On average, there are 2.5 BTB misses per iteration. Improving this loop's branch performance would decrease execution time.

Table 5.7. Results for *wc* inner loop

Event	Value
number of times loop body executed	105198
average total branches per loop iteration	5.5
average total BTB hits per loop iteration	3
instructions executed in each loop iteration	24
instructions executed in the v-pipe in each loop iteration	14

5.7 Case Study: Compress

The final example presents using run_p5mon on *compress*, another UNIX utility that implements a compression algorithm from the SPEC95 benchmark suite. This program has a very large input that requires about 20 minutes to run. Running this program on the IMPACT simulator would take roughly two to three weeks. However, run_p5mon can complete one pass, measuring two events, in only 20 minutes.

This program was run with no modifications. Since the measured values are so large, the overhead has little effect. The results are given in Table 5.8. As is shown, about 50% of the program's execution time is spent waiting for data that misses in the cache. Also, the results show that the number of address generation interlocks is quite high, offering an insight into a possible improvement in the code being generated. For a more detailed analysis, the user could define multiple regions within the code, which might provide even more information behind the processor's high execution time.

Table 5.8. Results for *compress*

Event	Value
Data read	12,080,864,478
Data read miss	1,401,807,536
Data write	6,697,887,424
Data write miss	964,277,224
Data cache lines written back	106,338,997
Data TLB miss	369,024,858
Memory accesses in both pipes	3,676,637,852
Bank conflicts	518,178,042
Misaligned data memory or I/O references	4,651
Pipeline stalled waiting for data memory read	33,765,207,119
Pipeline stalled due to address generation interlock	1,413,460,755
Number of clock cycles stalled due to full write buffers	757,887,506
Stall on write to an E or M state line	0
Code read	13,258,708,528
Code TLB miss	29,554
Code cache miss	326,766
Branches	6,448,792,619
BTB Hits	4,109,481,354
Pipeline flushes	1,329,905,915
Instructions executed	46,304,777,427
Instructions executed in v-pipe	17,247,147,592
Clock cycles	90,949,512,186

6. SUMMARY

This thesis offers a powerful method of using the Pentium's performance monitoring counters (PMCs) to help analyze an application's run-time behavior on a real system.

The current implementation of the p5mon device, library routines, and run_p5mon are also an excellent basis for future work. The first extension will be to add support for the new events in the Pentium with MMX and Pentium Pro processors. Both of these add many new events as well as a new instruction, the Read Performance Monitoring Counter instruction, that allows a user-level process to directly read the counters. This instruction will greatly reduce the overhead while using the assembly probes to gather data. Any additions or changes to the device driver, library routines, or run_p5mon program will be documented in another publication.

The second extension is to modify the Linux operating system kernel to save and restore the PMCs during a context switch. Using the information published about the Linux kernel [13], it should be possible to modify the kernel to keep the PMC information on a per-process basis. This will reduce the amount of error in the results. It is unclear how much more accurate the results will be, since other programs running in the system will still have effects on the data cache and BTB.

Another extension to this work is to add this support to Microsoft Windows 95 and Windows NT. This involves first writing a virtual device driver, much like the Linux device driver, to control the counters, and, second, building a DLL library on top of this device. Run_p5mon should then run with little modification. The most difficult aspect here is measuring the overhead in accessing the counters and the effect of background noise on the results.

APPENDIX A

DESCRIPTION OF PERFORMANCE MONITORING COUNTER EVENTS

This appendix contains a brief description of each event that can be counted in the Pentium processor. The newer events that were added for the Pentium Pro and the Pentium with MMX are not included. With each event is the corresponding event code in hex. Also note that for all cache reference related events, only the internal caches (L1 cache) are considered. The performance monitoring counters do not have any information about whether references hit or miss in the L2 cache. For more information concerning Pentium performance monitoring counter events, consult the *Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual, Chapter 26* [6]. Some events have errata, found in the *Pentium Processor Specification Update* [14]. While none of the errata are very serious, most of them are explained along with the event description. For more information on the new events contained in the Pentium Pro processor, consult [15]-[17]. For more information on the new events contained in the Pentium with MMX, consult [18].

- **data read [0x00], data write [0x01], data read or write [0x28]**

  These events count data read or write accesses, regardless of whether they hit or miss in the internal data cache. I/O reads and writes are not included. In the case of split accesses, each individual component read or write is counted individually. Data reads that are part of processing a Translation Lookaside Buffer (TLB) miss are not included.

- **data TLB miss [0x02]**

  This event counts the number of misses in the data cache TLB. This event incorrectly gets incremented twice if the data that misses the data cache TLB also causes an exception.

- **data read miss [0x03], data write miss [0x04], data read or write miss [0x29]**

  These events count the number of data memory read or write accesses that miss in the internal data cache whether or not the access is cacheable or non-cacheable. Data reads that are part of processing a TLB miss are not included. Accesses directed to I/O space are also not included in the count. Both the *Data read miss* and *Data write miss* events incorrectly get incremented twice if the access to the cache is misaligned.

- **write (hit) to M or E state lines [0x05]**

  This event counts the number of write hits to lines in the data cache that are in either the "exclusive" or "modified" state.

- **data cache lines written back [0x06]**

  This event counts all dirty data cache lines that are written back to external caches or memory regardless of the cause. These include both replacing a dirty cache line, as well as having to provide data as a result of an internal or external snoop.

- **external snoops [0x07]**

  This event counts external snoops that are accepted by the processor, regardless of whether they hit in the code or data caches. No internal snoops are counted.

- **data cache snoop hits [0x08]**

  This event is similar to the *external snoops* event, except that it only counts hits on those snoops directed at the data cache. In addition to the data cache itself, hits in the data line fill buffers or one of the writeback buffers are counted as hits for the data cache.

- **memory accesses in both pipes [0x09]**

  This event counts the number of data memory reads or writes which are paired together in the pipelines. Because of cache misses or cache bank conflicts, these accesses are not necessarily executed in parallel.

- **bank conflicts [0x0A]**

  This event counts the number of cache bank conflicts. This event may be incremented more than once if a v-pipe cache access takes more than one clock cycle to execute.

- **misaligned data memory or I/O references [0x0B]**

  This event counts the number of memory or I/O reads or writes that are misaligned. The definition of "misaligned" depends on the length of access. A 2- or 4-byte access is misaligned when it crosses a 4-byte boundary, while an 8-byte access is misaligned when it crosses an 8-byte boundary. Finally, 10-byte accesses are treated internally as two separate accesses of 8 and 2 bytes. This event incorrectly gets incremented twice if the access was caused by a FST or FSTP instruction.

- **code read [0x0C]**

  This event counts the total number of instructions read, whether or not the instruction is cacheable or non-cacheable.

- **code TLB miss [0x0D]**

  This event counts the total number of instruction reads that miss the code TLB. This event incorrectly gets incremented twice if the instruction that misses the code TLB also causes an exception.

- **code cache miss [0x0E]**

  This event counts the total instruction reads that miss in the internal code cache, whether or not the read is cacheable or non-cacheable.

- **any segment register loaded [0x0F]**

  This event counts the total number of writes into any segment register, whether in real or protected mode. The segment registers include CS, DS, ES, and DS, as well as the LDTR, GDTR, IDTR, and TR. A segment load instruction will explicitly trigger this event. In addition, far control transfers (which may be explicit or caused by interrupts or exceptions) and task switches both implicitly cause segment register loads. Furthermore, far control transfers and task switches which cause a privilege-level change will signal this event twice.

- **branches [0x12]**

  This event counts the total number of taken and not-taken branches. The number of branches actually executed is measured, rather than the number of predicted branches. In addition to taken conditional branches, jumps, calls, returns, software interrupts, and interrupt returns, the Pentium processor treats the following operations as causing taken branches: serializing instructions, some segment descriptor loads, hardware interrupts (including FLUSH#), and exceptions that invoke a trap or fault handler.

- **Branch Target Buffer (BTB) hits [0x13]**

  This event counts those branch instructions that are executed and hit in the BTB.

- **taken branch or BTB hit [0x14]**

  This event counts the number of taken branches and BTB hits. It represents an event that may cause a hit in the BTB. Specifically, the branch is already in the BTB or is a candidate for a space in the BTB.

- **pipeline flushes [0x15]**

  This event counts the total number of pipeline flushes. BTB misses on taken branches, BTB mispredictions, exceptions, interrupts, and some segment descriptor loads all cause pipeline flushes. This event does not occur on a serializing instruction, as serializing instructions only cause the prefetch queue to be flushed. However, this event may incorrectly be incremented for some segment descriptor loads and the VERR instruction.

- **instructions executed [0x16]**

  This event counts the total number of instructions executed, up to two per clock cycle. This includes instructions that are executed as part of a fault handler as well as for hardware and software interrupts and exceptions. All instructions that have a "Repeat" prefix (which is one of REP, REPE, REPZ, REPNE, REPNZ) will only be counted once, despite the fact that the repeat loop executes the instruction multiple times. Finally, this event only increments once for each HALT instruction that is executed, regardless of how many cycles the processor is in the HALT state.

- **instructions executed in v-pipe [0x17]**

  This event counts the number of instructions actually executed in the v-pipe. When combined with the *instructions executed* event, the user can determine the parallelism of the code being executed. The rules concerning the REP prefix and HALT instruction in the *instructions executed* event also apply here. Both the *instructions executed* and this event should be incremented when any exception is recognized. However, if the instruction in the v-pipe generates an exception, and a second exception occurs before the execution of the first exception's handler, the counter incorrectly does not increment for the first exception. Unless the code that the user is measuring causes exceptions, this should not pose a problem.

- **clocks while a bus cycle is in progress [0x18]**

  This event measures bus utilization by counting the number of clock cycles during which a bus cycle is in progress, including the clock cycles when HLDA, AHOLD, and BOFF# are asserted.

- **number of clocks stalled due to full write buffers [0x19]**

  This event counts the number of clock cycles the internal pipeline is stalled due to full write buffers. Full write buffers stall the pipeline in the following cases: data memory read misses, data memory write misses, and data memory write hits to "shared" state cache lines. Stalls on I/O accesses are not included.

- **pipeline stalled waiting for data memory read [0x1A]**

  This event counts the number of clock cycles the processor is stalled while a data memory read is in progress. This includes both data TLB miss processing as well as read attempts that are not bypassed while a line is being filled in the cache. This event incorrectly counts a misaligned access as 2 clock cycles instead of 3 clock cycles unless the access misses the TLB.

- **clocks stalled on write to M or E line [0x1B]**

  This event counts the number of clock cycles the processor is stalled waiting to write to a "modified" or "exclusive" cache line. It only counts those clock cycles while #EWBE is not asserted.

- **locked bus cycle [0x1C]**

  This event counts the number of times that a locked bus cycle occurs. This includes the LOCK instruction, instructions with a LOCK prefix, Page Table updates, and Descriptor Table updates. In a locked Read-Modify-Write instruction, only the read portion is counted. Split

Locked cycles count as two separate accesses. Cycles restarted due to the BOFF# signal are not counted.

- **I/O read or write cycle [0x1D]**

  This event counts the number of bus cycles directed to I/O space. Misaligned I/O accesses will generate two bus cycles. Bus cycles restarted due to BOFF# are not recounted.

- **noncacheable memory reads [0x1E]**

  This event counts the number of bus cycles while reading noncacheable instructions or data memory. It includes the read cycles caused by TLB misses. Also, cycles restarted due to BOFF# are not recounted. Finally, read cycles to I/O space are not supposed to be counted, but the counter incorrectly gets incremented for reads to I/O space.

- **pipeline stalled due to address generation interlock [0x1F]**

  This event counts the number of address generation interlocks (AGIs) occurring in a pipeline. An AGI occurring in both the U and V pipelines during the same clock signals this event twice. An AGI occurs when the instruction in the execute stage of either pipeline is writing to either the index or base address register of an instruction in the D2 (address generation) stage of either pipeline.

- **FLOPs (Floating Point Operations) [0x22]**

  This event counts the number of floating point adds, subtracts, multiplies, divides, remainders, and square roots. The instructions implementing transcendental functions consist of multiple adds and multiplies, and will signal this event multiple times. However, instructions generating the divide by zero, negative square root, special operand, or stack exceptions will be not be counted. Instructions generating all other floating point exceptions will be counted.

Any instructions (such as the integer multiply instruction) that use the floating point arithmetic circuitry will be counted as floating point operations.

- **breakpoint match DR0 [0x23], breakpoint match DR1 [0x24], breakpoint match DR2 [0x25], breakpoint match DR3 [0x26]**

  These event counts the number of breakpoint matches, whether or not the breakpoints are enabled. (These events correspond to the signals driven on the BP[3:0] pins. Refer to the chapter on debugging in the *Pentium Processor Developer's Manual, Vol. 3* for more information. Also, there are several errata pertaining to these events. As these events are not likely to be used, they are not described here. Consult the *Pentium Processor Specification Update* for more information.)

- **hardware interrupts [0x27]**

  This event counts only the number of taken interrupts (INTR) and nonmaskable interrupts (NMI).

APPENDIX B

INSTALLING P5MON SOFTWARE

This appendix contains instructions for installing the p5mon software onto a Linux system. Section B.1 describes installing the p5mon device driver. Section B.2 describes installing the p5mon library and run_p5mon. All of the code described in this thesis is available from the IMPACT web page (*http://www.crhc.uiuc.edu/Impact/*).

B.1 Installing P5mon Device Driver

All of the steps below should be executed as "root" on a Linux system. The user may change any of the pathnames as desired. Note that this software could present a possible security risk, because two processes could use the MSRs to communicate with one another without the operating system knowing.

1. Compiling a Linux kernel

   In order for p5mon to function correctly, the Linux kernel must have support for modules compiled in it. If the kernel running on the system does not have this support, the user will need to build a new kernel.

   a) Obtain the latest kernel source code. As of this writing, this is version 2.0.28.

   b) Configure the kernel to match your system. Pay special attention to two of the options that appear in the *Loadable module support* section when *make config* is run to configure the kernel.

      i) Answer "yes" to *Enable loadable module support (CONFIG_MODULES).*

      ii) Answer "no" to *Set version information on all symbols for modules (CONFIG_MODVERSIONS).*

c) Compile the kernel and test it before continuing.

2. Compiling the p5mon device source code

a) Obtain the p5mon device source code. In the IMPACT group, the code is stored under CVS control, and can be checked out by typing *cvs checkout p5mon* on one of the HPs or Suns. The p5mon directory then must be copied over onto the Linux system, presumably in the directory */root*.

b) Go to the directory *p5mon_driver* and compile the p5mon driver by typing *make all*.

c) Create a new device entry by entering *mknod -m 666 /dev/p5mon c 52 0*.

d) Load the module into the operating system kernel by typing *./install_p5mon*. If successful, the message "p5mon: initialization successful" will be displayed. If unsuccessful, one of several errors may have occured.

   i) The p5mon device may be loaded only on a system with a Pentium processor. The "p5mon: Invalid processor" message will appear, and the module will not be loaded if the user attempts to load p5mon on a machine without a Pentium processor.

   ii) If the message "kernel version needed, but can't be found" is displayed, the option *Set Version information on all symbols for modules* was not set properly. See Step 1 for details.

   iii) If the message "Can't get major number" appears, then the system needs to be rebooted.

e) Verify that the module is loaded into memory by typing *lsmod* to list all modules in memory. The p5mon module should appear in this list.

3. Testing the p5mon device

   a) A small test program is included with the p5mon device. This test program, called p5mon_test, should have been compiled along with the device. To run it, type *p5montest*.

   b) The program simply sets the counters to monitor two events, and then samples the events five times. The output should be similar to Figure B.1. If the device is working correctly, no errors should occur when executing this simple program.

```
[1]>p5montest
p5montest: Attempting to open p5mon
p5montest: p5mon device opened with ID 5
p5montest: attempt to write p5mon device
p5montest: write of 0x00800083 to CESR successful
p5montest: attempting to read p5mon device 5 times
p5montest: read successful
p5montest: CESR: 00800083, PMC0: 96, PMC1: 2166
p5montest: read successful
p5montest: CESR: 00800083, PMC0: 436, PMC1: 20008811
p5montest: read successful
p5montest: CESR: 00800083, PMC0: 592, PMC1: 40013102
p5montest: read successful
p5montest: CESR: 00800083, PMC0: 749, PMC1: 60017304
p5montest: read successful
p5montest: CESR: 00800083, PMC0: 882, PMC1: 80021521
p5montest: attempting to close p5mon device
p5montest: device closed normally
[2]>
```

Figure B.1. Output of p5mon device test program

4. Automatically loading the p5mon device

   a) To automatically load the p5mon device at boot time, add the line */root/p5mon/p5mon_driver/install_p5mon* to the end of the */etc/rc.d/rc.local* file.

   b) Test this addition by rebooting the system and watching for the "p5mon: initialization successful" message at the end of the boot sequence. This message will also be printed in the kernel log, located in the */var/log/messages* file. In addition, the module should be visible in memory when the *lsmod* command is used.

5. Making changes to the p5mon device

   If the p5mon device itself is changed, the system must be rebooted in order for the changes to take effect.

B.2 Installing P5mon Library and Run_p5mon

The p5mon library and run_p5mon program may be installed in two ways. First, the system administrator may wish to install these programs in a common place on the system, such as */usr/local/bin* and */usr/local/lib*, and have all the users on the system use these copies. Alternatively, each user may wish to compile the two programs and install them in their own directory. The instructions that follow may be used in either fashion.

1. Compiling the p5mon source code

   a) Obtain the p5mon library and run_p5mon source code. In the IMPACT group, the code is stored under CVS control, and can be checked out by typing *cvs checkout p5mon* on one of the HPs or Suns. In addition, the IMPACT parameter facility located in *libimpact.a* must be present to compile run_p5mon.

   b) The p5mon.README file in the directory contains a brief description of what each of the source files contains.

   c) Go to the directory *p5mon_src* and compile both the p5mon library (*libp5mon.a*) and run_p5mon by typing *make install*. In addition to the library functions described in Chapter 3, the p5mon library also contains all of the functions needed to use the assembly-level stubs.

   d) If desired, copy the run_p5mon executable and the file libp5mon.a to a convenient location, such as a */bin* or */lib* directory.

2. Compiling an executable with assembly-level stubs

   In order to compile a program to use the assembly-level stubs, in addition to any other parameters required, be sure to link to the p5mon library by adding *-lp5mon* to the compiler parameters.

APPENDIX C

SUMMARY OF RUN_P5MON PARAMETERS

Table C.1 lists all the configuration parameters available for run_p5mon. For each parameter, the parameter name, type (integer, string, or boolean (yes/no)), default value, and brief description are included. For the parameters that correspond to events in the performance monitoring counters (PMCs), the event code is given to conserve space. Figure C.1 contains a sample PARMS file showing how all the parameters can be used.

Table C.1 Run_p5mon parameters

BASIC OPTIONS			
Parameter	**Type**	**Default**	**Description**
command	string	NONE	program to be executed
repetitions	integer	1	number of times to count each specified event
user_events	boolean	yes	count user-level events
kernel_events	boolean	no	count kernel-level events
output_all_results	boolean	no	output results from each time event is counted
confidence_level_99	boolean	no	use a 99% confidence interval instead of a 95% level
ADVANCED OPTIONS			
Parameter	**Type**	**Default**	**Description**
use_assembly_regions	boolean	no	using modified assembly code with stubs to define regions
max_regions	integer	100	maximum regions defined in modified assembly code
output_file	string	NONE	file to write results in a parseable format
skip_initialization_run	boolean	no	specifies whether or not to skip the initialization run
verbose_level	integer	0	levels of debugging information
EVENTS			
Parameter	**Type**	**Default**	**Description**
count_clocks	boolean	no	counts total number of clock cycles for the program
data_read	boolean	no	counts PMC event 0x00
data_write	boolean	no	counts PMC event 0x01
data_read_or_write	boolean	no	counts PMC event 0x28
data_tlb_miss	boolean	no	counts PMC event 0x02

Table C.1 Continued

Parameter	Type	Default	Description
data_read_miss	boolean	no	counts PMC event 0x03
data_write_miss	boolean	no	counts PMC event 0x04
data_read_or_write_miss	boolean	no	counts PMC event 0x29
write_to_M_or_E_lines	boolean	no	counts PMC event 0x05
data_cache_lines_written_back	boolean	no	counts PMC event 0x06
external_snoops	boolean	no	counts PMC event 0x07
data_cache_snooping_hits	boolean	no	counts PMC event 0x08
memory_accesses_in_both_pipes	boolean	no	counts PMC event 0x09
bank_conflicts	boolean	no	counts PMC event 0x0A
misaligned_data_memory_or_IO_references	boolean	no	counts PMC event 0x0B
code_read	boolean	no	counts PMC event 0x0C
code_tlb_miss	boolean	no	counts PMC event 0x0D
code_cache_miss	boolean	no	counts PMC event 0x0E
segment_register_loaded	boolean	no	counts PMC event 0x0F
branches	boolean	no	counts PMC event 0x12
btb_hits	boolean	no	counts PMC event 0x13
taken_branch_or_btb_hit	boolean	no	counts PMC event 0x14
pipeline_flushes	boolean	no	counts PMC event 0x15
instructions_executed	boolean	no	counts PMC event 0x16
instructions_executed_in_v_pipe	boolean	no	counts PMC event 0x17
clocks_while_bus_cycle_in_progress	boolean	no	counts PMC event 0x18
clocks_stalled_full_write_buffers	boolean	no	counts PMC event 0x19
clocks_stalled_waiting_data_read	boolean	no	counts PMC event 0x1A
clocks_stalled_write_to_M_or_E_line	boolean	no	counts PMC event 0x1B
locked_bus_cycle	boolean	no	counts PMC event 0x1C
IO_read_or_write_cycle	boolean	no	counts PMC event 0x1D
non_cacheable_memory_reads	boolean	no	counts PMC event 0x1E
pipeline_stalled_address_generation_interlock	boolean	no	counts PMC event 0x1F
flops	boolean	no	counts PMC event 0x22
breakpoint_match_DR0	boolean	no	counts PMC event 0x23
breakpoint_match_DR1	boolean	no	counts PMC event 0x24
breakpoint_match_DR2	boolean	no	counts PMC event 0x25
breakpoint_match_DR3	boolean	no	counts PMC event 0x26
hardware_interrupts	boolean	no	counts PMC event 0x27

```
(p5mon
      command = hello;
      repetitions = 5;
      user_events = yes;
      kernel_events = no;
      output_all_results = no;
      conf_level_99 = no;
      use_assembly_regions = no;
      max_regions = 1;
      output_file = p5mon_output;
      skip_initialization_run = no;
      verbose_level = 0;
      repeat_first_run = no;
      data_read = no;
      data_write = no;
      data_read_or_write = no;
      data_tlb_miss = no;
      data_read_miss = yes;
      data_write_miss = yes;
      data_read_or_write_miss = no;
      write_to_M_or_E_lines = no;
      data_cache_lines_written_back = no;
      external_snoops = no;
      data_cache_snooping_hits = no;
      memory_accesses_in_both_pipes = no;
      bank_conflicts = no;
      misaligned_data_memory_or_IO_references = no;
      code_read = no;
      code_tlb_miss = no;
      code_cache_miss = no;
      segment_register_loaded = no;
      branches = no;
      btb_hits = no;
      taken_branch_or_btb_hit = no;
      pipeline_flushes = no;
      instructions_executed = no;
      instructions_executed_in_v_pipe = no;
      clocks_while_bus_cycle_in_progress = no;
      clocks_stalled_full_write_buffers = no;
      clocks_stalled_waiting_data_read = no;
      clocks_stalled_write_to_M_or_E_line = no;
      locked_bus_cycle = no;
      IO_read_or_write_cycle = no;
      non_cacheable_memory_reads = no;
      pipeline_stalled_address_generation_interlock = no;
      flops = no;
      breakpoint_match_DR0 = no;
      breakpoint_match_DR1 = no;
      breakpoint_match_DR2 = no;
      breakpoint_match_DR3 = no;
      hardware_interrupts = no;
      count_clocks = no;

      dump_parms = no;
      warn_parm_not_defined = yes;
      warn_parm_defined_twice = yes;
      warn_parm_not_used = yes;
      parm_warn_file_name = stderr;
      parm_dump_file_name = stderr;
end)
```

Figure C.1. Sample p5mon PARMS file

APPENDIX D

PMC ACCESS OVERHEAD

As mentioned in previous chapters, there are several sources of overhead associated with measuring events in a program using run_p5mon. If the entire program is being measured, the values in the PMCs will include the overhead of configuring and reading the counters as well as the operating system overhead of creating and destroying a process. If assembly stubs are being used, the functions that define a region, both of which read the PMCs, add overhead to the results.

It is extremely important to note that the values given in this appendix are approximate. Many factors, including system configuration, can change the overhead in accessing the counters. When first using this software on a system, it may be useful to measure the overhead for the system.

Table D.1 lists the measured overhead values for using run_p5mon on an unmodified executable. Twenty repetitions were used to compute an average value with a 95% confidence interval. Results are given separately for the user-level only and for the user- and kernel-levels together. An empty C program that contained no useful instructions was used to determine the overhead of using run_p5mon on an unmodified executable. This represents the overhead required for the operating system to create a process, execute the process, and then destroy the process. Observation has demonstrated that many factors can affect this overhead. These include, but are not limited to, the number of command-line arguments, the size of the program, and the number of other processes running on the system in the background.

Table D.1. PMC overhead for an unmodified executable

Event	User-level	User- & Kernel-level
data read	322972.0 ± 1.4	444409.3 ± 31.3
data write	109913.0 ± 1.4	216049.9 ± 19.6
data read or write	432886.5 ± 3.4	660479.7 ± 50.7
data TLB miss	349.0 ± 0.0	685.2 ± 0.3
data read miss	11000.8 ± 7.7	23664.5 ± 21.4
data write miss	8896.4 ± 13.4	71583.0 ± 26.3
data read or write miss	19887.0 ± 22.7	95273.6 ± 28.3
write (hit) to M or E state lines	101016.9 ± 13.5	144482.9 ± 35.5
data cache lines written back	1491.3 ± 5.6	4353.4 ± 13.8
external snoops	16.1 ± 2.4	26.1 ± 2.0
data cache snoop hits	0.0 ± 0.0	0.0 ± 0.0
memory access in both pipes	45710.4 ± 3.6	92023.8 ± 5.9
bank conflicts	3626.3 ± 0.5	5898.6 ± 3.3
misaligned data memory or I/O references	96.0 ± 0.0	398.0 ± 0.0
code read	318381.3 ± 40.8	440211.7 ± 62.1
code TLB miss	585.9 ± 5.5	1378.8 ± 7.3
code cache miss	7535.2 ± 869.2	18240.2 ± 20.5
any segment register loaded	992.0 ± 91.6	4017.2 ± 2.5
branches	156084.6 ± 0.3	229068.5 ± 17.4
BTB hits	115131.3 ± 13.9	152622.5 ± 28.9
taken branch or BTB hit	124941.1 ± 9.9	173992.1 ± 82.1
pipeline flushes	32786.4 ± 20.0	47608.9 ± 62.2
instructions executed	803766.2 ± 0.3	1144670.3 ± 80.1
instructions executed in the v-pipe	220958.8 ± 7.3	315723.8 ± 15.4
clocks while a bus cycle is in progress	339094.3 ± 2492.7	1025924.6 ± 2835.7
number of clocks stalled due to full write buffers	11259.5 ± 61.6	163819.2 ± 412.5
pipeline stalled waiting for data memory read	166717.7 ± 963.7	519362.3 ± 2589.6
stall on write to an E or M state line	0.0 ± 0.0	0.0 ± 0.0
locked bus cycle	26.1 ± 4.4	533.5 ± 0.9
I/O read or write cycle	0.0 ± 0.0	13.0 ± 1.1
non-cacheable memory reads	1902.9 ± 15.4	3557.1 ± 12.5
pipeline stalled because of an address generation interlock	104075.9 ± 4.6	138659.5 ± 10.5
floating point operations (FLOPs)	194.0 ± 0.0	205.0 ± 0.0
breakpoint match on DR0 register	0.0 ± 0.0	0.0 ± 0.0
breakpoint match on DR1 register	0.0 ± 0.0	0.0 ± 0.0
breakpoint match on DR2 register	0.0 ± 0.0	0.0 ± 0.0
breakpoint match on DR3 register	0.0 ± 0.0	0.0 ± 0.0
hardware interrupts	1.9 ± 0.4	2.8 ± 0.3
clock cycles	1453038.1 ± 1472.8	2515425.1 ± 60414.0

Table D.2 contains similar results for a modified executable. Again, 20 repetitions were used to compute an average and a 95% confidence interval. Results for both the user-level only as well as the user- and kernel-levels together are given. When assembly regions are in use, overhead is caused by reading the PMCs in the begin-region and end-region function calls. Two sets of values are presented in Table D.2. The first set, labeled "Not in cache," is the overhead in accessing the counters once, as would occur when defining a single assembly region around an entire program. To obtain these numbers, a program with a single assembly region containing no instructions was used. The second set of values, labeled "In cache," is the results obtained for a second region defined in the same program. In this case, all of the code to read the PMCs is already in the cache and the branches are already in the BTB. This can affect many events, including the data cache performance, the BTB performance, the instruction pairing, and the number of clock cycles stalled. When defining multiple regions in a real program, the user will have to think about whether the code for the begin and end region functions is likely to be in the cache, and whether the branches are likely to remain in the BTB when determining the overhead associated with their use. One aspect of assembly regions that has not been considered is their effect on code layout. The "true" value will lie somewhere between these two cases.

When looking at these overhead values, the user must keep several things in mind. First, the results obtained using assembly regions are much more accurate than the results for an entire executable. Furthermore, using one assembly region in a program is the most accurate method of using the counters. Second, the results for the user-level only are more accurate than the results for both the user- and kernel-levels. Finally, the most inaccurate overhead values are associated with the events that count clock cycles, such as "clock cycles while a bus cycle is in progress" or "number of clocks stalled waiting for a data memory read."

Table D.2. PMC access overhead using assembly regions

EVENT	Not In Cache (first time)		In Cache (other times)	
	USER	BOTH	USER	BOTH
data read	69.0 ± 0.0	274.0 ± 0.0	69.0 ± 0.0	274.0 ± 0.0
data write	66.0 ± 0.0	224.0 ± 0.0	66.0 ± 0.0	224.0 ± 0.0
data read or write	135.0 ± 0.0	498.0 ± 0.0	135.0 ± 0.0	498.0 ± 0.0
data TLB miss	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
data read miss	0.0 ± 0.0	6.4 ± 0.6	0.0 ± 0.0	6.4 ± 0.6
data write miss	4.0 ± 0.0	4.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
data read or write miss	4.0 ± 0.0	10.2 ± 0.4	1.0 ± 0.0	7.3 ± 0.4
write (hit) to M or E state lines	62.0 ± 0.0	220.0 ± 0.0	65.0 ± 0.0	223.0 ± 0.0
data cache lines written back	0.0 ± 0.0	0.1 ± 0.1	0.0 ± 0.0	0.1 ± 0.1
external snoops	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
data cache snoop hits	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
memory access in both pipes	25.0 ± 0.0	84.0 ± 0.0	32.0 ± 0.0	91.0 ± 0.0
bank conflicts	5.0 ± 0.0	4.8 ± 0.5	5.0 ± 0.0	5.0 ± 0.0
misaligned data memory or I/O references	8.0 ± 0.0	8.0 ± 0.0	8.0 ± 0.0	8.0 ± 0.0
code read	50.3 ± 0.4	254.4 ± 0.7	52.9 ± 0.2	254.8 ± 0.5
code TLB miss	1.0 ± 0.0	1.0 ± 0.0	0.0 ± 0.0	1.0 ± 0.0
code cache miss	12.0 ± 0.0	22.2 ± 0.3	3.0 ± 0.0	14.0 ± 0.0
any segment register loaded	3.0 ± 0.0	12.0 ± 0.0	3.0 ± 0.0	12.0 ± 0.0
branches	26.0 ± 0.0	149.0 ± 0.0	26.0 ± 0.0	149.0 ± 0.0
BTB hits	5.0 ± 0.0	78.8 ± 0.6	16.8 ± 0.2	93.0 ± 0.7
taken branch or BTB hit	22.0 ± 0.0	104.5 ± 0.3	23.0 ± 0.0	105.2 ± 0.3
pipeline flushes	17.0 ± 0.0	38.9 ± 0.4	9.3 ± 0.2	26.9 ± 0.5
instructions executed	147.0 ± 0.0	771.0 ± 0.0	147.0 ± 0.0	771.0 ± 0.0
instructions executed in the v-pipe	30.0 ± 0.0	185.3 ± 0.3	43.0 ± 0.0	197.8 ± 0.4
clocks while a bus cycle is in progress	186.7 ± 3.5	337.1 ± 11.1	29.0 ± 0.0	189.2 ± 11.1
number of clocks stalled due to full write buffers	1.0 ± 0.0	1.5 ± 0.7	0.0 ± 0.0	0.3 ± 0.7
pipeline stalled waiting for data memory read	11.0 ± 0.0	61.2 ± 7.0	11.0 ± 0.0	58.5 ± 7.9
stall on write to an E or M state line	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
locked bus cycle	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
I/O read or write cycle	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
non-cacheable memory reads	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0
pipeline stalled because of an address generation interlock	16.0 ± 0.0	72.5 ± 0.2	19.0 ± 0.0	75.5 ± 0.2
floating point operations (FLOPs)	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0
breakpoint match on DR0 register	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
breakpoint match on DR1 register	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
breakpoint match on DR2 register	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
breakpoint match on DR3 register	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
hardware interrupts	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
clock cycles	423.7 ± 1.5	1292.5 ± 23.7	304.1 ± .6	1154.2 ± 15.4

REFERENCES

[1] T. Mathisen. "Pentium Secrets," *BYTE Magazine*, pp. 191-192, July 1994.

[2] M. Rosenblum et al., "Complete Computer Simulation: The SimOS Approach," in *IEEE Parallel and Distributed Technology*, vol. 3, no. 4, pp. 34-43, Winter 1995.

[3] J. B. Chen et al., "The Measured Performance of Personal Computer Operating Systems," in *ACM Transactions on Computer Systems,* vol. 14, no. 1, pp. 23-40, Feb 1996.

[4] J. B. Chen et al., "The Impact of Operating System Structure on Personal Computer Performance," Center for Research in Computing Technology, Harvard University, Cambridge, MA, Tech. Rep TR-09-95.

[5] D. Bhandarkar and J. Ding, "Performance Characterization of the Pentium Pro Processor," in *Proceedings, Third International Symposium on High-Performance Computer Architecture*, 1997, p. 288.

[6] *Pentium Processor Family Developers Manual, Volume 3: Architecture and Programming Manual*. Intel Corporation, 1996.

[7] "write," Linux 2.0.27 man page.

[8] "read," Linux 2.0.27 man page.

[9] "open," Linux 2.0.27 man page.

[10] IMPACT Parameters Tutorial, IMPACT Research Group, University of Illinois, 1996. Located under CVS in *impact/tutorials/parms_tutorial*.

[11] Run_p5mon Tutorial, IMPACT Research Group, University of Illinois, 1997. Located under CVS in *impact/src/x86_tools/p5mon/p5mon_src/tutorial*.

[12] *82430FX PCIset Datasheet, 82437FX System Controller (TSC) and 82438FX Data Path Unit (TDP).* Intel Corporation, 1995.

[13] M. Beck et al., *Linux Kernel Internals*. Harlow, England: Addison Wesley Longman, 1996.

[14] *Pentium Processor Specification Update, January 1997*. Intel Corporation, 1997.

[15] *Pentium Pro Family Developer's Manual, Volume 3: Operating System Writer's Manual.* Intel Corporation, 1996.

[16] *Pentium Pro Family Developer's Manual, Volume 2: Programmer's Reference Manual.* Intel Corporation, 1996.

[17] *Pentium Pro Processor Specification Update, February 1997.* Intel Corporation, 1997.

[18] *Pentium Processor Family Developer's Manual*. Intel Corporation, 1997.