Hardware Performance Monitoring Counters on non-X86 Architectures

George V. Neville-Neil
Neville-Neil Consulting
gnn@neville-neil.com

February 9, 2010

Abstract

Hardware Performance Monitoring Counters provide programmers and systems integrators with the ability to gather accurate, low level, information about the performance of their code, both at the user and kernel levels. Until recently these counters were only available on Intel and AMD chips but they have now been made available on alternate, embedded, architectures such as MIPS and ARM. This paper discusses the motivation, design and implementation of counters using the hwpmc(4) driver in the *FreeBSD*operating system with an eye towards easing future porting efforts.

1 Motivation

As long as there have been computers there have been attempts to measure the performance of the code that is running on them. In the early days of computing the cost of owning and maintaining a computer was so high, and the speeds of the systems were so slow that it was vitally important to squeeze every last ounce of performance out of a system. More recent computers are vastly faster than their ancestors but there are still many reasons to measure and improve the performance of software. Any programmer worthy of the title will, by nature, wish to produce the fastest and tightest code, but this kind of personal concern for the quality of one's code is not the force behind current efforts to improve the performance of software. In the first decade of the 21st century the driving force behind the amazing improvements in the speed CPUs abruptly died. Moore's Law, which stated that transistor density on a chip would double every 18 months, and by which every computer user from 1965 until 2005 benefited, can no longer be counted on to cover up the poor performance of software[6]. Newer chips do not bring higher clock speeds, but instead they now pack multiple copies of the CPU onto the same die, giving a higher core count but not, overall, improving the speed of any single-threaded programs. The performance of a program that is not easily broken down into components which can execute independently on several cores will not improve with a new processor but remain mostly static, which makes performance analysis a more pressing concern than at any time in the history of computing.

Analyzing the performance of software is a tedious and difficult process which encompasses experimental design, data collection and data analysis. This paper will not address experimental design or data analysis but instead discuss improvements in the area of data collection.

2 History of Performance Data Collection

Collecting performance data while running a piece of software presents several problems, the most difficult of which is making sure that the data collection process has a minimal effect on the performance of the system that is being measured. For many years the

two most common ways to measure the performance of a system were to run it in a simulator or to have the compiler insert special pieces of code into the final executable which told the operating system to periodically collect performance data[3].

Anyone who has used a simulator, in particular one that is implemented purely in software, knows that the correlation between how the software runs in the simulator and on real hardware is relatively poor. It is very difficult to have a software simulator that shows all of the perverse corners cases that are found in hardware and unfortunately it is just these corner cases which can be important to understanding and improving the performance of a system. Interactions between memory buses, caches, and devices are notoriously difficult to simulate, and yet, in modern computer architectures it is these components which may make or break a piece of software.

By far the most common way to collect performance data is to have the compiler insert special code into the executable version of a program and then to have the operating system collect the information periodically while the program is running. While this second approach gives a far more accurate picture of what is going on when a program executes it has several drawbacks. Inserting extra code into the program changes how the program executes, which is a computer science version of the observer paradox from physics. In this case the observing of the program changes its performance characteristics. While the fidelity of the data collected will be higher than in a software simulator it may not be of sufficiently high fidelity to make appropriate decisions about performance trade-offs. Collecting data in this way also leaves out an important component, the operating system itself. If the system cannot be viewed in its totality then it is quite possible that the developer will wind up optimizing the wrong piece of code.

3 Hardware Performance Monitoring Counters

With the huge growth in the number of transistors that could be placed on a single silicon wafer, hardware engineers went looking for things to do with these transistors. Over time, functionality that used to reside outside of the CPU, such as floating point co-processors, caches, and I/O devices were moved onto the CPU, which sped up the overall operation of the system while reducing the problems of hooking the components together. It is far easier to make sure the results of some operation show up in the right place and at the right time if all of the components involved in the operation are on a single die and do not have to go off chip. In the early 1990s the first performance monitoring counters appeared on commercial CPUs.[2] Unlike other features that had been integrated onto the CPU the counters were there purely to collect data on the performance of the software that executed on the chip.

The purpose of a hardware monitoring counter is to keep track of how many operations, of various types, a processor completes during the execution of a piece of software. There are three parts to any system that provides a set of performance monitoring counters; the counters themselves, which are special registers that are incremented when some known event takes place, the set of events which may cause the registers to increment, and the registers which are programmed to tell the chip which events to count and where to deposit the results.

Early implementations of performance monitoring counters provided space to count only one or two events at a time, and only supported a few events. As the transistor count on CPUs increased, the number of simultaneous counters the number of events that could be tracked increased. Where a Pentium processor from Intel had only two registers to store counters and 75 events, a Nehalem processor has space to record up to seven simultaneous counters and over 300 events that can be tracked.

4 Driver and Library API and Architecture

In the *FreeBSD*operating system the hwpmc(4) driver, user space libraries, and controlling programs, pmcstat(8) and pmccontrol(8) are responsible for

Directory	Contents
src/lib/libpmc	PMC Libraries
src/sys/dev/hwpmc	Driver Source Code

Figure 1: Source Code Locations

programming the underlying hardware, periodically collecting the results from the CPU, and delivering the information to the user. The libraries allow developers to write their own performance tools or to adapt other libraries or performance tools to the *FreeBSD*system[1].

The code that implements the hwpmc subsystem is broken up into two major groups, the library and the driver. The driver is responsible for talking to the hardware, programming the registers, and collecting the data. The library is responsible for translating the collected data into something that is easily consumable by programs that wish to analyze other programs. While events can be read and processed directly from the driver it is more common to have the driver write the events into a log file. The user level code opens a file and then hands the open file descriptor to the driver via the `pmc_configure_logfile` function in the pmc library. Once the log file is configured the driver will place all the collected events into the log file until the driver is told to stop, after which the file is closed. The post processing of events is also supported by the pmc library which allows programs to open previously recorded log files and to process them in any way they choose.

The code for the driver is kept in the src/sys/dev/hwpmc/ directory and is organized into files separated by architecture and CPU as seen in Figure 2. Some of the files, such as those for ARM and Sparc64, are simply stubs that do not have any active code in them, while others, such as those for AMD and Intel processors, as well as XScale and MIPS now have rudimentary support for performance monitoring counters.

All of the events supported on all of the different CPUs are collected into a single file, `pmc_events.h`, which acts as the bridge between the driver and the library. If an event is not present in the `pmc_events.h` file then it can not be programmed

File	Purpose
hwpmc_amd.c	AMD K7, K8
hwpmc_arm.c	ARM
hwpmc_core.c	Intel Core Architecture
hwpmc_ia64.c	Itanium
hwpmc_intel.c	Generic Intel Code
hwpmc_logging.c	Internal Logging Code
hwpmc_mips.c	MIPS
hwpmc_mod.c	Kernel Module Routines
hwpmc_pentium.c	Pentium
hwpmc_piv.c	Pentium IV
hwpmc_powerpc.c	PowerPC
hwpmc_ppro.c	Pentium Pro
hwpmc_sparc64.c	SPARC64
hwpmc_tsc.c	Time Stamp Counter
hwpmc_x86.c	Intel Callchain Support
hwpmc_xscale.c	Intel XScale (ARM)
pmc_events.h	All Events for All Processors

Figure 2: Driver Support Files

Method	Meaning
pmd_pcpu_init	Do per-CPU initializtion
pmd_pcpu_fini	Per-CPU teardown
pmd_switch_in	Context switch in
pmd_switch_out	Context switch out
pmd_intr	Handle interrupt

Figure 3: PMC Machine Dependent Methods

or collected by the system. We will come back to this file later when we add new events for the MIPS architecture.

The driver is split up into two major groups of functions that control the underlying hardware. Like many other subsystems in the *FreeBSD*kernel, the hwpmc driver uses a form of Objects in C to encapsulate both state and functions together within a structure. The two major structures present in the hwpmc driver are the `class` and `machine dependent` components which contain pointers to the functions that implement the CPU specific code which actually does the work of talking to the hardware. Whenever a new architecture or CPU is supported the routines in Figures 3 and 4 must be implemented.

Method	Meaning
pcd_config_pmc	Configure a PMC
pcd_get_config	Read the configuration
pcd_read_pmc	Read the counter value
pcd_write_pmc	Write a counter value
pcd_allocate_pmc	Allocate a PMC for use
pcd_release_pmc	Release an allocated PMC
pcd_start_pmc	Start counting events
pcd_stop_pmc	Stop counting events
pcd_describe	Text description
pcd_pcpu_init	Initialize the class
pcd_pcpu_fini	End the class
pcd_get_msr	Machine specific interface

Figure 4: PMC Class Methods

The class methods handle all of the generic work required by the hwpmc driver, including setting up and tearing down state for each CPU in a multi-processor system as well as handling context switches and interrupts. The hwpmc system needs to know when a process is running because the user is allowed to ask the system to record events only for a specific process and therefore event collection may be started when a process is context switched in and stopped when it is switched out.

The class methods are used to directly control the underlying counter hardware. Because it is possible to have different classes of counters, for instance, Intel Core processors have two different types, it is necessary to encapsulate the state and methods into a structure that is independent from the `pmc_mdep` structure. Briefly, a CPU has only one `pmc_mdep` structure but may have several `pmc_classdep` structures.

In order to support a new CPU or architecture both of these structures must be filled in with working code. These structures give some clues as to how the pmc driver works with the underlying hardware. A pmc goes through a particular life-cycle when it is used. Before events can be counted the system must allocate a counter for use. Allocating a counter merely reserves is, but does not count any events. After the counter is allocated it is started. While the counter is running it is counting events. The driver collects the events and writes them to an internal log that is read by the library. When the counter is no longer needed it is stopped and then released. This life cycle is implemented by the `pcd_allocate_pmc`, `pcd_start_pmc`, `pcd_stop_pmc` and `pcd_release_pmc` functions.

- Add events to pmc_events.h
- Add all methods
- Modify `libpmc.c`
- Document counters
- Test

Figure 5: Porting Process

5 MIPS Counters

Like other vendors in the embedded systems space MIPS has added support for performance monitoring counters only recently. CPUs in the MIPS32 24K processor family support only two simultaneous counters and more than 90 events. The counters are provided as two pairs of registers, one half of the pair is the control register which indicates which event to record and whether to record the event when the CPU is in kernel, user or interrupt mode. The second register in the pair is a 32 bit wide container which is incremented whenever an event is triggered[4]. Programming these registers is complicated by the fact that an event takes on a different meaning depending on whether it is programmed into pmc 0 or 1. For example when event 5 is programmed into pmc 0 it counts instruction TLB accesses, but when it is programmed into pmc 1 it counts instruction TLB misses.

Porting the hwpmc driver to the MIPS architecture required several steps, which are outlined in Figure 5.

The most time consuming and tedious part of porting the hwpmc driver to a new architecture is adding and tracking the new events. The system has a well defined set of macros which help organize the events,

```
#define __PMC_EV_MIPS()                         \
        __PMC_EV(MIPS, CYCLE)                   \
        __PMC_EV(MIPS, INSTR_EXECUTED)          \
        __PMC_EV(MIPS, BRANCH_COMPLETED)        \
        __PMC_EV(MIPS, BRANCH_MISPRED)          \
        __PMC_EV(MIPS, RETURN)                  \
        __PMC_EV(MIPS, RETURN_MISPRED)          \
        __PMC_EV(MIPS, RETURN_NOT_31)           \
        __PMC_EV(MIPS, RETURN_NOTPRED)          \
        __PMC_EV(MIPS, ITLB_ACCESS)             \
        __PMC_EV(MIPS, ITLB_MISS)               \

...
```

Figure 6: MIPS events in pmc_events.h

```
/*
 * MIPS Events from "Programming the MIPS32 24K Core Family",
 * Document Number: MD00355 Revision 04.63 December 19, 2008
 * These events are kept in the order found in Table 7.4.
 * For counters which are different between the left hand
 * column (0/2) and the right hand column (1/3) the left
 * hand is given first, e.g. BRANCH_COMPLETED and BRANCH_MISPRED
 * in the definition below.
 */
```

Figure 7: Event Documentation Reference

but with more than 90 of them to add, many of which have similar names, it is still easy to make mistakes. The `pmc_events.h` file contains all the events for all the architectures and CPUs that are supported by the hwpmc driver. Adding a new set requires defining the events and their names. The events *must* be added in the same order in which they are enumerated in the manual. Figure 6 shows the first 9 events, of 92, being defined for the driver.

The `__PMC_EV` macro defines an event for use by the driver and is used to define events for all of the architectures. The events are defined in the documentation for each chip and that documentation is referenced directly in the code, as seen in Figure 7.

At the very end of the `pmc_events.h` file is where the set of all known PMC events is defined. In order to complete the addition of the new events an entry must be added to the `__PMC_EVENTS` macro which defines a range for the new events. The range bears no relation to the event number used by the hardware, it is simply a convenient way to reserve blocks of numbers for the driver to use to map events into the system. The current set of known events is shown in Figure 8.

One of the major complications in handling the MIPS events is that events with the same number

```
#define __PMC_EVENTS()                          \
        __PMC_EV_BLOCK(TSC,     0x01000)        \
        __PMC_EV_TSC()                          \
        __PMC_EV_BLOCK(K7,      0x2000)         \
        __PMC_EV_K7()                           \
        __PMC_EV_BLOCK(K8,      0x2080)         \
        __PMC_EV_K8()                           \
        __PMC_EV_BLOCK(IAF,     0x10000)        \
        __PMC_EV_IAF()                          \
        __PMC_EV_BLOCK(IAP,     0x10080)        \
        __PMC_EV_IAP()                          \
        __PMC_EV_BLOCK(P4,      0x11000)        \
        __PMC_EV_P4()                           \
        __PMC_EV_BLOCK(P5,      0x11080)        \
        __PMC_EV_P5()                           \
        __PMC_EV_BLOCK(P6,      0x11100)        \
        __PMC_EV_P6()                           \
        __PMC_EV_BLOCK(XSCALE,  0x11200)        \
        __PMC_EV_XSCALE()                       \
        __PMC_EV_BLOCK(MIPS,    0x11300)        \
        __PMC_EV_MIPS()

...
```

Figure 8: Event Table

```
        enum pmc_event  pe_ev;
        uint8_t         pe_code;
};

/*
 * MIPS event codes are encoded with a select bit.  The
 * select bit is used when writing to CP0 so that we
 * can select either counter 0/2 or 1/3.  The cycle
 * and instruction counters are special in that they
```

Figure 9: Event Table

have a different meaning depending on which counter register they are programmed into. A subset of the events, including those for counting cycles and instructions, are the same in either counter 0 or counter 1.

Figure 9 shows a portion of the code from `hwpmc_mips.c` which defines the event codes specific to the MIPS architecture. Only the cycle and instructions executed have a unique number while the events that follow show up as pairs. The code which controls the counters must make sure to program the correct counter in order to record the correct events at run time.

Once all of the codes have been listed in the appropriate structures the supporting methods for the machine dependent and class structures must be written. This code is not reproduced here but it is important to point out a few of the salient features of the MIPS architecture as they relate to hwpmc.

MIPS is like many RISC CPUs in that it is not so much a single processor as it is a collection of pro-

cessor functionality that can be tied together by a systems designer in order to provide a very targeted set of functionality. The way that this is achieved is that there is a small defined subset of functionality which acts as an irreducible core, tied with a simple method of extending the core to add new functions. The MIPS architecture is actually flexible enough to allow for extensions that could not have been envisioned by the original MIPS designers. The extension mechanism used by the MIPS architecture is to provide a set of coprocessor registers which can be read and written using the core instructions, `MFC0` and `MTC0`, which stand for "move from co-processor 0" and "move to co-processor 0" respectively. Extended functionality, such as performance monitoring counters, is implemented using this co-processor system.

The instructions used for interacting with the co-processor have one interesting side effect, they are memory barrier instructions. RISC architectures depend heavily on instruction pipelining to achieve high performance. Talking to a co-processor presents a hazard because "resources controlled via Coprocessor 0 affect the operation of various pipeline stages of a MIPS32 processor, manipulation of these resources may produce results that are not detectable by subsequent instructions for some number of execution cycles. When no hardware interlock exists between one instruction that causes an effect that is visible to a second instruction, a CP0 hazard exists."[5] The use of these instructions has the effect of flushing the pipeline and also of possibly impacting performance, so they *must* be used sparingly otherwise we run the risk of the performance counting system having a significant negative effect on performance. All of the code used in the MIPS specific parts of the hwpmc driver attempts to minimize the use of these potentially disruptive instructions, and developers working with similar architectures must pay close attention to the same issues.

With the work in the driver complete, the last step is to make the libpmc library aware of the new architecture and CPU support. The library only needs to know about the events that are available on a particular CPU, and does not need to know how to program the hardware, as that is the job of the driver. Event names are communicated to the library via the

```
static struct pmc_event_alias mips_aliases[] = {
        EV_ALIAS("instructions",        "INSTR_EXECUTED"),
        EV_ALIAS("branches",            "BRANCH_COMPLETED"),
        EV_ALIAS("branch-mispredicts",  "BRANCH_MISPRED"),
        EV_ALIAS(NULL, NULL)
};
```

Figure 10: MIPS Aliases

`pmc_events.h` file which is effectively shared between the driver and the library.

The link between the library and driver is, in part, through the `pmc.h` file which is part of the kernel source and is installed when a new kernel is installed. The CPU and the class must be declared in this include file to make the driver and the library aware of the new architecture. The declarations are made available via a pair of macros `__PMC_CPU` and `__PMC_CLASS` respectively.

The final set of modifications are to the library itself, `libpmc.c`, where various macros are used to declare the various class and machine dependent tables that are used to describe the low level hardware and events to high level programs. With the proliferation of profiling events available on various architectures it is nice to be able to depend on there being a mapping from a well known event, such as the one which counts instructions, to the specific event as it is implemented on a particular CPU. The pmc library makes it easy for a programmer to map an event to an alias, which is usually a common, or well known, term for performance analysis.

In Figure 10 we see three aliases which map the MIPS specific events `INSTR_EXECUTED`, `BRANCH_COMPLETED`, and `BRANCH_MISPRED` to the aliases "instructions", "branches", and "branch-mispredicts." It is a convention in the hwpmc system that CPU and architecture specific events are presented in *ALL_CAPS* while aliases are *lower-case*.

With all of the code in place we can now go on to documenting and testing. The hwpmc system has always been well documented, which is a necessity in a system that has so many possible events to explain. The documentation is contained in architecture specific man pages that accompany the library and which are installed with it. A final thing to note is that the library and the driver *must* match, installing two dif-

ferent versions will not work and in the best case will give obvious errors, but in the worst case may give misleading data.

6 Related Work and Conclusions

The hwpmc driver, libraries and user land programs were originally developed by Joseph Koshy. The first non x86 architecture to be added to the hwpmc driver was *Intel XScale (ARM)*, which was committed by Rui Paulo in late 2009. Other operating systems, including Linux and Solaris provide access to performance monitoring counters via their own methods.

With the spread of performance monitoring counters onto new hardware platforms porting the hwpmc driver to new and different architectures is now an important part of bringing the full functionality of *FreeBSD*to developers and systems integrators. Not having performance monitoring counters, and the insight they can provide into system performance, both for the kernel and user space applications, can lead to poor performance and clumsy attempts at optimization. This paper presented an example of porting the driver and adapting the libraries to the MIPS architecture. It is hoped that this will spur other developers to add other architectures to the system.

7 Acknowledgements

The author would like to thank Rui Paulo and Fabien Thomas for their comments on this paper, as well as Joseph Koshy who has written and maintained the hwpmc code from the start.

8 Bibliography

References

[1] Papi programmer's reference. *Manuals*, pages 1–212, May 2008.

[2] Intel Corporation. Intel(r) 64 and ia32 architectures software developer's manual vol 3a. pages 1–756, Jun 2009.

[3] Susan Graham, Peter Kessler, and Kirk McKusick. gprof - a call graph execution profiler. page 10, May 1998.

[4] MIPS Technologies Inc. Programming the mips32® 24k® core family. *Manuals*, pages 1–116, Dec 2008.

[5] MIPS. Mips32® architecture for programmers volume ii: The mips32® instruction set. *Manuals*, pages 1–296, Dec 2008.

[6] Gordon E. Moore. Cramming more components onto integrated circuits. page 4, Jan 1965.