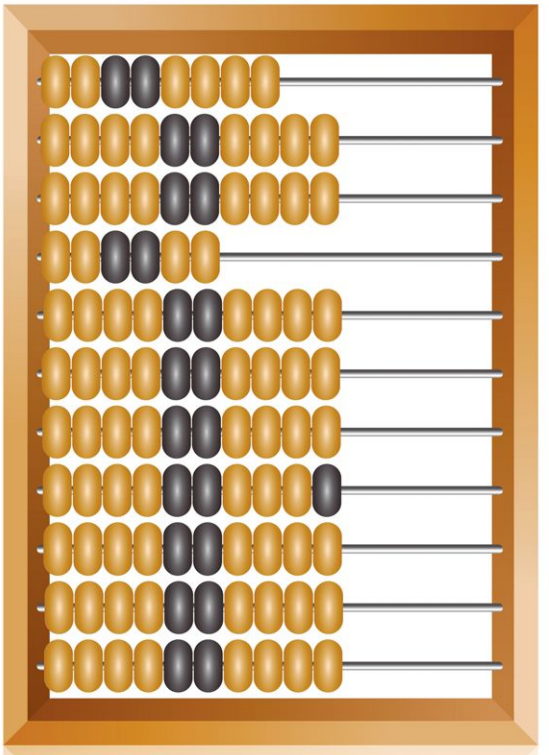
UNDERSTANDING APPLICATION AND SYSTEM PERFORMANCE WITH HWPMC(4)

BY GEORGE NEVILLE-NEIL



Architecture and memory have become more complex over the last several years. This added complexity has made it harder than ever to understand software performance.

Fortunately, CPU designers have added features to their chips to enable developers and administrators to better understand the performance of software with a very small amount of overhead. The Hardware Performance Monitoring Counters, hwpmc(4), driver and associated tools on FreeBSD provide software developers—and anyone else who is interested in system performance—a way to better understand how efficiently their software is utilizing the underlying hardware, their applications and the operating system itself. The hwpmc subsystem takes advantage of hardware specific registers set aside by the CPU designer purely for the purpose of understanding the run time performance of the system.

It Was 20 Years Ago...

In early versions of BSD software, performance was measured using a software based profiling system and an associated tool, named gprof. The g stands for graph, as in call graph. Software that was to be profiled was compiled with a set of flags indicating that the resulting program should have special hooks to tell the operating system to periodically collect performance information. Once the program had been executed, this profiling information was collated by the gprof program and presented to the user.

A software based profiling system has several problems. First, software profiling is computationally expensive. Depending on how often the profiler runs, it may introduce an overhead penalty on the order of 10% to 20%. For a short program, the profiling can easily outweigh the code being profiled, resulting in measurements that are useless. A second problem is that a software based profiling solution changes the flow of the resulting binary program, meaning that the code being profiled is not a one to one representation of the final software as it would be shipped to a customer. While it is possible to ship profiled binaries to customers, the overhead incurred in a profiled binary would result in worse overall system performance, which would be unacceptable. The third problem presented by software based profiling is that it is impossible for an end user to measure the performance of their system on their own. A customer with a non-profiled binary application to run has

no way of adding profiling to the binary to find out if the program is a source of system inefficiency. Hardware based performance solutions ameliorate some of these problems.

Hardware Based Performance Monitoring Counters

As CPUs got to be more densely packed with transistors, and as the feature sets of CPUs got larger, it became possible for CPU designers to include specific registers to count events relating to system performance. At first the types and numbers of events that could be counted were small, with only a handful of events and one or two counting registers. It was only possible to count instructions that were executed or the number of level one cache misses. On a modern Intel CPU, hundreds of event types can be counted, and there are enough counting registers to record 7 different events simultaneously.

When working with hardware based performance monitoring counters there are a few pieces of terminology to keep in mind. An "event" is anything that the chip can count for you, such as the number of instructions retired, branch predictions that were missed, cycles required to fetch memory, etc. A counting register is a place where an event can be counted. Events can be recorded in two different modes, and counted in two different scopes. An event may be simply "counted" or the CPU may be configured to interrupt the operating system when a counter has hit a set level, an event which the hwpmc(4) driver records in its log for later analysis. A counted event gives a raw number that tells the programmer how many of some event happened over a particular unit of time. A sampled event is more complicated than a counted event. In event sampling, the system is programmed to take a sample of the instruction pointer and possibly the program's call chain, whenever some number of events has occurred. Sampling allows the system to show where an event occurred in the software, helping the programmer pinpoint the source of a performance problem. Events can be counted and sampled in one of two scopes. Process scope records events only when the target program is currently executing. System scope records events at all times and when coupled with sampling mode will show the performance not only of the program that is being tested, but of all programs in the system, including the operating system itself. It is possible to count events in either system or process mode as well as sample events in system or process mode.

Measuring Performance with hwpmc

The easiest way to learn to use hwpmc in your own programs is by trying with a few contrived examples. The unixbench[cite] system of benchmarks is a well-known, easy to understand, set of programs that try to determine the speed of both software and hardware. We will measure the performance of several programs from unixbench with the pmc tools in order to give some clear examples.

Before working with the hwpmc(4) driver it must be loaded into your kernel. To measure system level performance you will also need root privileges on the machine on which you wish to use hwpmc. The default (GENERIC) kernel does not have hwpmc loaded at boot time. In order to load hwpmc issue the following commands as root (Figure1).

Fig. 1

```
# kldload hwpmc

hwpmc: SOFT/16/64/0x67<INT,USR,SYS,REA,WRI> TSC/1/64/0x20<REA>
IAP/4/48/0x3ff<INT,USR,SYS,EDG,THR,REA,WRI,INV,QUA,PRC>
IAF/3/48/0x67<INT,USR,SYS,REA,WRI>
```

When the hwpmc(4) driver is loaded it reports the number, type and width of the counting registers it finds on the CPU. The output varies widely from processor to processor, even within the same vendor family. In the example shown above there are sixteen soft registers, one time stamp counter, four programmed counters and three fixed counters. Certain events can only be counted in certain types of registers, and you will be given an error if you try to count events in a register that does not accept the event you are asking for. For the most part you only need to know the number of registers in each class, as the system attempts to assign events correctly when you ask for them. If a user tries to count four events that are only possible in the fixed type registers (IAF) then the tools will report an error and exit without counting any events.

The types of events that can be counted are listed with the `pmccontrol -L` command. There are 194 possible events that can be counted on this host, but don't worry, we will not go through all of them.

One of the first program measurements that is typically made is the raw number of instruc-

tions that it executes in a particular amount of time. The event, INSTR_RETIRED_ANY, counts instructions executed. The term retired is used because the end of executing an instruction is called retiring in CPU parlance.

In order to count or sample events the

```
int main(argc, argv)
int   argc;
char  *argv[];
{
 ... (Intentionally Left Blank) ...

      while(1) {
            mov(disk,1,3);
            iter++;
            }

      exit(0);
}

void mov(int n, int f, int t)
{
      int o;
      if(n == 1) {
            num[f]—;
            num[t]++;
            return;
      }
      o = other(f,t);
      mov(n-1,f,o);
      mov(1,f,t);
      mov(n-1,o,t);
}
```

pmcstat command is used.

Figure 2 shows a simple, cumulative, count of instruction's retired when the hanoi benchmark is run for ten seconds. We need not concern ourselves with the details of the hanoi program at this point, but we will dig into it more later. The first column is the number of instructions that were retired during the entire run of the program, the last number showing the cumulative total of 27527655088. In the ten seconds that the hanoi program was running it executed 1343590 loops, and this output is from the hanoi program itself, it has nothing to do with hwpmc.

Examples in this article were run on a Lenovo Thinkpad X230 with a Core i7 running at 2.9GHz.

For comparison, we ran the hanoi program without the performance monitoring system. Based on the number of loops that hanoi was able to execute in ten seconds (Figure 3) we see that hwpmc introduces less than 2% of overhead.

One common measure of code efficiency is Cycles Per Instruction (CPI) which is derived by counting both instructions retired and the number of cycles during a test run, and then dividing the two results.

The command in Figure 4 counts instructions retired and clock cycles simultaneously. Dividing the clock cycles by the number of instructions retired gets us a CPI of 0.44, which, according to Intel's optimization manuals, is an acceptable value. Higher values of CPI, for instance those greater than 1, indicate that instructions are taking longer than they ought to. For a full discussion of CPI and general performance tuning using Intel's PMC events see:

(http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf)

Counting events gives an overall idea of the efficiency of a complete program. To look more deeply into where a piece of code is spending its time we need to use sampling mode.

The command shown in Figure 5 uses the instruction retired event from the previous example, but switches to sampling mode, indicated by the capital P command line switch, and stores the resulting output in a log file in /tmp/hanoi.log. Without the output option, the entire log would be dumped to stdout, which wouldn't be very useful.

Once the event collection is complete we analyze the log file with pmcstat to see what functions were taking up the most instructions.

We process the collected log file in Figure 6 to produce a graph file. The output in the graph file (hanoi.graph) shows the functions that took up the largest percentage of events, starting with the largest and proceeding to the smallest.

The output in Figure 7 shows that the mov() routine, see the code in Listing X, takes up the largest number of samples, whereas the main() routine for the program had very few. The result is what we'd expect from this program. Output from pmcstat can be shown in another way, as gprof(1) output `pmcstat -R /tmp/hanoi.log -g`. (Figure 7).

Processing the same log with the -g argument creates a per-event directory, INSTR_RETIRED_ANY/ which contains output files for each program, library, and the kernel that were in use when the samples were taken.

Processing the hanoi.gmon file gives the output shown in Figure 8. Time, in this case, is misleading. The numbers in the seconds columns represent events counted, and not seconds, but the output is convenient and brief to read. We still see that the mov() routine is the biggest consumer of events, taking up 99.8%

```
pmcstat -C -p INSTR_RETIRED_ANY ./hanoi 10
# p/INSTR_RETIRED_ANY
           14201054051 1343590 loops
           27527655088
```

Fig. 2

```
./hanoi 10
1357889 loops
```

Fig. 3

```
pmcstat -C -p INSTR_RETIRED_ANY -p CPU_CLK_UNHALTED_CORE ./hanoi 10
# p/INSTR_RETIRED_ANY p/CPU_CLK_UNHALTED_CORE
           13318387828               5962151280 1324620 loops

           27139612697              11987228985
```

Fig. 4

```
> pmcstat -O /tmp/hanoi.log -P INSTR_RETIRED_ANY ./hanoi 10
1013645 loops
```

Fig. 5

```
> pmcstat -R /tmp/hanoi.log -G /tmp/hanoi.graph
```

Fig. 6

```
@ INSTR_RETIRED_ANY [365189 samples]

99.17%  [362173]    mov @
/usr/home/gnn/svn/headports/benchmarks/unixbench/work/unixbench-4.1.0/pgms/hanoi
 99.61%  [360744]     mov
  97.57%  [351963]      mov
   90.90%  [319928]       mov
   09.10%  [32035]        main
  02.43%  [8781]        main
   100.0%  [8781]         _start
 00.39%  [1429]       main
  100.0%  [1429]        _start
```

Fig. 7

```
> ls
hanoi.gmon                kernel.gmon               libc.so.7.gmon
hwpmc.ko.gmon             ld-elf.so.1.gmon
> gprof ../hanoi hanoi.gmon
granularity: each sample hit covers 4.00673 byte(s) for 0.00% of 362181.00 seconds

 %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.8  361564.31 361564.31    42245  8558.75  8558.75  mov [3]
  0.0  361572.29      7.99    10210     0.78 35413.54  main [1]
  0.0  361572.29      0.00        0     0.00%           _start [2]
```

Fig. 8

of all events found relating to the program.

Up to this point we have shown the hwpmc system working only in process scope. The `hanoi` program is meant to show only the performance of the CPU and has no interaction with the underlying operating system. We will now move on to the syscall benchmark which shows the performance of one aspect of the operating system itself, the speed of a system call.

We see the main loop of the syscall program in Listing Y. The benchmark measures the speed of the operating system's system calls by repeatedly duplicating a file descriptor, getting the process ID and the user ID, and setting the umask. Each of these calls does a very small amount of work compared to the work required to enter and exit the kernel, and therefore

Y

```
while     (1) {
          close(dup(0));
          getpid();
          getuid();
          umask(022);
          iter++;
      }
```

make good targets for measuring the overhead of the system call mechanism itself.

We collect the samples and generate a graph file in Figure 9. The graph file contains over 5000 lines of output, including functions that have no relation to the syscall benchmark program. For this example, we are using system-wide scope for even collection, and so we have collected events for all the various processes currently executing on the system, including Emacs, in which this article is being written. The first several lines of the graph file are shown in Figure 10.

The largest number of samples, 10%, come from the witness_unlock() kernel routine. As we move down the graph we see the constituent components that contributed to the 12263 events recorded against witness_unlock(), including do_dup(), closefp(), and sys_umask() which are the kernel side routines that are called by the dup(), close() and umask() system calls. The cheapest system calls, getuid() and getpid() do not occur until much farther down in the file. An interesting comparison is the number of events that are counted against libc vs. those that are counted against the kernel.

Figure 11 shows the number of events counted against the library version of getuid vs. those counted against the kernel side of the system call, sys_getuid. The boilerplate in the C library is minimal compared to that seen in the kernel, which tells us that the largest speed up in this code would be afforded by improving the kernel side code. Both getpid() and getuid() are trivial, but are used in benchmarks to determine the overhead of a system call.

Counters Counters Everywhere

Originally only available on a small number of Intel and AMD processors, hwpmc has now been extended to cover ARM and MIPS processors as well, giving developers the ability to profile their code on popular embedded systems. The events and counters are architecture specific, but the basic concepts remain the same. The hwpmc system also provides convenient aliases for common events, such as cycles, for whatever the CPU's cycle counter is, and instructions, for instructions retired. Aliases are always lower case, and architecture specific counters, like INSTR_RETIRED_ANY, are upper case. Event aliases are present across almost all processors supported by FreeBSD, which makes writing portable performance analysis scripts easier.

As new processors are put into the market by CPU vendors the hwpmc system gets extended to add support for newer events and newer sets of counting registers. If you're trying to understand the performance characteristics of the software you're writing or the system as a whole, the hwpmc system and its tools are a great place to start. •

George Neville-Neil works on networking and operating system code for fun and profit. He also teaches various courses on subjects related to computer programming. His professional areas of interest include code spelunking, operating systems, networking, and security. He is the co-author with Marshall Kirk McKusick of *The Design and Implementaion of the FreeBSD Operating System* and is the columnist behind ACM Queue magazine's *Kode Vicious*. Neville-Neil earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts. He is a member of the ACM, the Usenix Association, and the IEEE. He is an avid bicyclist and traveler and currently resides in New York City.

```
> sudo pmcstat -O /tmp/syscall.log -S INSTR_RETIRED_ANY ./syscall 10
3232709 loops
> sudo pmcstat -R /tmp/syscall.log -G /tmp/syscall.graph
CONVERSION STATISTICS:
 #exec/elf                                  1
 #samples/total                             262735
 #samples/unclaimed                         6
 #samples/unknown-function                  41
 #callchain/dubious-frames                  6
```

Fig. 9

```
@ INSTR_RETIRED_ANY [113032 samples]

10.85%  [12263]     witness_unlock @ /boot/kernel/kernel
 54.50%  [6683]       _sx_xunlock
  47.36%  [3165]       do_dup
   100.0%  [3165]        amd64_syscall
  32.71%  [2186]       closefp
   100.0%  [2186]        amd64_syscall
  19.93%  [1332]       sys_umask
   100.0%  [1332]        amd64_syscall
```

Fig. 10

```
 01.45%  [1642]      sys_getuid @ /boot/kernel/kernel
 00.47%  [526]       getuid @ /lib/libc.so.7
```

Fig. 11