

# Understanding Performance with HWPMC

George Neville-Neil

[gnn@neville-neil.com](mailto:gnn@neville-neil.com)

DCBSDCon 2009

# What you will learn in this talk

- ▶ Configure HWPMC
- ▶ Run programs under HWPMC
- ▶ Measure user and kernel code
- ▶ Understand the output of the tools
- ▶ Tune the driver for better results

# What you will *not learn in this talk*

- ▶ Everthing there is to know about performance analysis
- ▶ The necessary math to perform a complete analysis
- ▶ How to optimize your code

# Why Optimize?

- ▶ Moore's Law is Dead
- ▶ Power consumption is your enemy
- ▶ Cooling isn't free
- ▶ Fast code is better than slow code

# What can be Optimized?

- ▶ Time and/or Space
- ▶ Transactions per second
- ▶ Memory usage

# Perils of Optimization

- ▶ Fixing wrong piece of code
- ▶ Spending too much time optimizing
- ▶ Doing it too early

# The Optimization Process

- ▶ Design an experiment
- ▶ Choose a workload
- ▶ Perform a measurement
- ▶ Form a hypothesis
- ▶ Make a change
- ▶ Take the same measurement
- ▶ Evaluate your result
- ▶ Repeat

# Designing an Experiment

- ▶ What problem are you trying to solve?
- ▶ Choose your variables wisely
- ▶ Make sure your measurements are repeatable
- ▶ Start small, simpler is often better

# Choosing a Workload

- ▶ Try to choose a realistic workload
- ▶ Easy to store
- ▶ Easy to replay
- ▶ Easy to explain

# Recording your measurements

- ▶ How you record a measurement can bias the test
  - ▶ Probe effect
- ▶ Recording framework needs to be as low impact as possible
- ▶ Easy to store, replay and explain

# Studying Performance without HWPMC

- ▶ `gprof` has been the tool for over 20 years
- ▶ Special compiler option, `-pg` builds code for profiling
- ▶ `gprof` has a demonstrable probe effect

# How the Processor Can Help

- ▶ What if performance counters were built into every CPU?
- ▶ Counters on a chip would be lower impact than in the operating system
- ▶ The CPU has the most knowledge of what is happening to it, and to other components in the system
  - ▶ Memory
  - ▶ I/O
  - ▶ Bus Transfers
  - ▶ Special Instructions

# HWPMC History

- ▶ Hardware Performance Monitoring Counters have been a part of CPUs since early Pentium and AMD K processors
- ▶ Earlier processors had fewer counters and counter types
- ▶ The newer the processor the more things it can track

# Availability

- ▶ AMD
  - ▶ K7, K8
- ▶ Intel
  - ▶ Pentium Pro, Pentium II, Pentium III, Pentium M
  - ▶ Xeon, Core2 Duo

# Decoding Processor Names

- ▶ Both AMD and Intel have not made it easy to map the name of a processor to a processor model.
- ▶ e.g. A Xeon 5400 processor is really a Core Micro-architecture CPU

# Events Have to be Mapped to Processors

- ▶ Which counters and events are available depend on the processor, down to the type. *You must refer to the processor manual if you want to use a specific counter.*
- ▶ *e.g. A Xeon 5400 is different than a Xeon 5100*

# How HWPMC Works

- ▶ Counters are kept on the chip
- ▶ The user selects what type of counter to use
- ▶ The kernel driver programs the chip to start counting events
- ▶ Chip generates an interrupt
- ▶ HWPMC driver reads out the counters into a memory buffer
- ▶ User level program, `pmcstat`, reads information from the kernel

# Counter Types and Modes

- ▶ Process Mode Counters
- ▶ System Mode Counters
- ▶ Counting
- ▶ Sampling

# Example Counters

- ▶ BR\_MAC\_MISSP\_EXEC
- ▶ BR\_CALL\_EXEC
- ▶ INST\_RETIRED.ANY\_P
- ▶ INST\_RETIRED.LOADS
- ▶ L2\_LS
- ▶ L2\_LINES\_IN
- ▶ L2\_LINES\_OUT
- ▶ ...
  - ▶ There are 217 of these on a modern Intel processor

# Convenient Aliases

- ▶ branches
- ▶ branch-mispredicts
- ▶ cycles
- ▶ dc-misses
- ▶ ic-misses
- ▶ instructions
- ▶ interrupts
- ▶ unhalted cycles

# Demonstration

Please see second screen.

# Test Goal

- ▶ Display various counters for a small, easy to understand program
- ▶ Become familiar with how to run the tools

# Testing Method

- ▶ Pick a simple to understand program
- ▶ Choose a workload
- ▶ Measure the program with pmc
- ▶ Evaluate the results

# Test Program

- ▶ `context.c` from the `unixbench` set of benchmarks
- ▶ Measures number of context switches per unit time
- ▶ Uses a set of pipes to ping/pong between threads

# Our test program

## Parent Process

```

if (pipe(p1) || pipe(p2)) {
    perror("pipe_create_failed");
    exit(1);
}
if (fork()) { /* parent process */
    /* master, write p1 & read p2 */
    close(p1[0]); close(p2[1]);
    while (1) {
        if (write(p1[1], (char *)&iter, sizeof(iter)) != sizeof(iter)) {
            if ((errno != 0) && (errno != EINTR))
                perror("master_write_failed");
            exit(1);
        }
        if (read(p2[0], (char *)&check, sizeof(check)) != sizeof(check)) {
            if ((errno != 0) && (errno != EINTR)) {
                perror("master_read_failed");
                exit(1);
            }
        }
        if (check != iter) {
            printf("Master_sync_error:_expect_%lu,_got_%lu\n",
                iter, check);
            exit(2);
        }
        iter++;
    }
}
}

```

# Our Testing Program

## Child Process

```

else { /* child process */
    unsigned long iter1;
    iter1 = 0;
    /* slave, read p1 & write p2 */
    close(p1[1]); close(p2[0]);
    while (1) {
        if (read(p1[0], (char *)&check, sizeof(check)) != sizeof(check)) {
            if ((errno != 0) && (errno != EINTR))
                perror("slave_read_failed");
            exit(1);
        }
        if (check != iter1) {
            printf("Slave_sync_error:_expect_%ld ,_got_%ld\n",
                iter, check);
            exit(2);
        }
        if (write(p2[1], (char *)&iter1, sizeof(iter1)) != sizeof(check)) {
            if ((errno != 0) && (errno != EINTR))
                perror("slave_write_failed");
            exit(1);
        }
        iter1++;
    }
}
}

```

# Testing Platform

- ▶ All tests were carried out on the same machine
- ▶ ThinkPad X60 XXXXX PROCESSOR
- ▶ FreeBSD CURRENT as of January 2009

# Command Line Programs

- ▶ Several programs are used to control and use `hwpmc`
- ▶ `pmccontrol`
- ▶ `pmcstat`

# pmccontrol

- ▶ A program for controlling the underlying device driver
- ▶ Used to enable and disable PMCs on processors
- ▶ Can list all the supported counters on a system
  - ▶ `-L`

# pmcstat

- ▶ The program you will use most to take measurements
- ▶ Has many possible arguments
- ▶ We will only cover the basics today
- ▶ Full documentation in `pmcstat(8)`

# Choosing Your Counters

- ▶ You need to tell `pmcstat` what type of counter you're using
- ▶ Sampling uses capitals `-P -S`
- ▶ Counting uses lower case `-p -s`
- ▶ System mode counters can only be used with root privileges

# Counting Instructions

- ▶ Process mode counting
- ▶ Executable and its arguments follows the `pmcstat` command and its arguments
- ▶ The output of `pmcstat` is shown last.

# Interpreting the Results

- ▶ The output of this program tells us very little.
- ▶ A single execution is insufficient to tell us anything.
- ▶ We need multiple trials to figure out what might be going on
- ▶ Simply counting instructions for a program will not tell us a bottle neck.

# Multiple Trials

# More Interpretation

- ▶ We do not get the same count each time.
  - ▶ Why not?
- ▶ At least one count is 2x the rest of them

# Other Types of Counters

- ▶ Instructions are not the only thing to count
- ▶ Cycles are not instructions
- ▶ Different forms of cache misses
- ▶ Counts of special instructions

# Cycle Counting

## Why the close correlation?

```
/* set up alarm call */
    iter = 0;
    wake_me(duration , report);
void wake_me(seconds, func)
    int seconds;

void (*func)();
{
    /* set up the signal handler */
    signal(SIGALRM, func);
    /* get the clock running */
    alarm(seconds);
}
```

# Interpreting the Results Again

- ▶ The program executes for 1 second
- ▶ The number of cycles per second should be well correlated
- ▶ You need to have some understanding of how your code *should behave*

# Branching Instructions and Branch Misses

# Why count branches?

- ▶ Branches are expensive
- ▶ Missing predicted branches is even more expensive

# Counters for Caches

- ▶ There are several types of caches
- ▶ Many types of cache counters
  - ▶
  - ▶
  - ▶
  - ▶
  - ▶
  - ▶
  - ▶

# Counting Special Instructions

- ▶ Several CPU specific instructions can be counted
- ▶ SIMD
- ▶ SSE
- ▶ Math

# Division

# Multiplication

# A new tool, pmcannotate

- ▶ Written by Attilio Rao
- ▶ Can show hot spots in code
- ▶ Requires non-stripped executable to retrieve function names
- ▶ Can work on any code, kernel or user application

# Example of pmcannotate

# A Brief Introduction to gprof

- ▶ FreeBSD already has a performance gathering tool
- ▶ `gprof` was originally written in the 1980s as a profiling tool for code
- ▶ Gives two different types of output about code execution
  - ▶ Flat
  - ▶ Hierarchical
- ▶ Uses the OS to gather information on each clock tick
- ▶ Significant probe effect

# Deriving gprof output from pmcstat

- ▶ **Collect the samples**

- ▶ `pmcstat -O /tmp/samples.out -P instructions \`  
`hanoi 1 8`

- ▶ **Convert the samples to gprof format**

- ▶ `pmcstat -R /tmp/samples.out -g`

- ▶ **A directory is produced based on the counter name**  
`ipm-v2-instructions/`

- ▶ **Use gprof on the executable and the instruction file**

- ▶ `cd ipm-v2-instructions`

- ▶ `gprof ../hanoi hanoi.gmon`

# Instruction Count Output

# The Hanoi Program

```

int main(argc , argv)
int argc;
char *argv[];
{
  ...
  while(1) {
    mov(disk ,1 ,3);
    iter++;
  }
  ...
}

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);
}

```

# An Important Reminder

- ▶ What you're counting is what shows up in the output!!!
- ▶ If you are counting instructions then instructions are what's counted.
- ▶ If you are counting cache misses then cache misses are counted

# Comparing Three Counters

- ▶ Instructions
- ▶ Multiplications
- ▶ Divisions
- ▶ We use the Whetstone benchmark for this comparison

# Whetstone Instructions

# Whetstone Multiplications

# Whetstone Divisions

# Whetstone Analysis

- ▶ There is one function that never shows up in the math counts
- ▶ Why is that?

# Whetstone Functions

```
void p3(SPDP *x, SPDP *y, SPDP *z, SPDP t, SPDP t1, SPDP t2)
{
    *x = *y;
    *y = *z;
    *x = t * (*x + *y);
    *y = t1 * (*x + *y);
    *z = (*x + *y)/t2;
    return;
}
```

# Whetstone Functions

```

void p3(SPDP *x, SPDP *y, SPDP *z, SPDP t, SPDP t1, SPDP t2)
{
    *x = *y;
    *y = *z;
    *x = t * (*x + *y);
    *y = t1 * (*x + *y);
    *z = (*x + *y)/t2;
    return;
}

```

```

void pa(SPDP e[4], SPDP t, SPDP t2)
{
    long j;
    for(j=0;j<6;j++) {
        e[0] = (e[0]+e[1]+e[2]-e[3])*t;
        e[1] = (e[0]+e[1]-e[2]+e[3])*t;
        e[2] = (e[0]-e[1]+e[2]+e[3])*t;
        e[3] = (-e[0]+e[1]+e[2]+e[3])/t2;
    }
    return;
}

```

# Whetstone Functions

```

void p3(SPDP *x, SPDP *y, SPDP *z, SPDP t, SPDP t1, SPDP t2)
{
    *x = *y;
    *y = *z;
    *x = t * (*x + *y);
    *y = t1 * (*x + *y);
    *z = (*x + *y)/t2;
    return;
}

```

```

void pa(SPDP e[4], SPDP t, SPDP t2)
{
    long j;
    for (j=0;j <6;j++) {
        e[0] = (e[0]+e[1]+e[2]-e[3])* t;
        e[1] = (e[0]+e[1]-e[2]+e[3])* t;
        e[2] = (e[0]-e[1]+e[2]+e[3])* t;
        e[3] = (-e[0]+e[1]+e[2]+e[3])/ t2;
    }
    return;
}

```

```

void po(SPDP e1[4], long j, long k, long l)
{
    e1[j] = e1[k];
    e1[k] = e1[l];
    e1[l] = e1[j];
    return;
}

```

# The Kernel is just another program

- ▶ The `hwpmc` driver can measure the kernel as well
- ▶ The same tools are used with different options
- ▶ Of course you have to be `root` to profile the kernel

# Capturing the Data

- ▶ You need to use a system wide counter
- ▶ Need to store the samples somewhere
- ▶ `sudo pmcstat -O /tmp/samples.out -S instructions`
- ▶ Interrupt after some number of seconds
- ▶ `pmcstat -R /tmp/samples.out -g`

# Idle Kernel Instruction Count

# Idle Kernel Cache Miss Count

## hwpmc using a sampling model

- ▶ How many samples you take determines your accuracy
- ▶ Sampling presents a trade off
- ▶ Too often and you increase the probe effect
- ▶ Not often enough and you miss important data

# Controlling Your Sample Rate

- ▶ `pmcstat` provides a `-n` option
- ▶ Sample after every `n` instructions
- ▶ If you sample too often you can *panic the kernel*
- ▶ *Start with a large number*
- ▶ *The default is to sample after every 65535 events*

# Kernel Tunables

- ▶ Call Chain Depth
- ▶ Number of Buffers
- ▶ Number of Samples
- ▶ Need to be set in `/boot/loader.conf`
- ▶ Require a reboot to take effect

# Manual Pages

- ▶ **Driver manual page** `hwpmc` (4)
  - ▶ Useful for understanding and tuning the driver and its resources
- ▶ **Control Program** `pmccontrol` (8)
- ▶ **Statistics Gathering** `pmcstat` (8)
- ▶ **Library routines** `pmc` (3)
  - ▶ If you want to add `hwpmc` directly to your program

# More Processor Support

- ▶ Adding support for new processors is a tedious job
- ▶ Support for the latest Xeons is in CURRENT
- ▶ There is a patch that can be applied to 7.1.
- ▶ Full support will be in FreeBSD 7.2

## Further Reading

- ▶ “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2”
- ▶ “gprof: A Call Graph Execution Profiler”, by Graham, S.L., Kessler, P.B., McKusick, M.K.; Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982.
- ▶ R. Jain, “The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,”

# Question and Answer Session