Reprinted from the

Proceedings of the Linux Symposium

Volume One

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

IA64-Linux perf tools for IO dorks

Examples of IA-64 PMU usage

Grant Grundler
Hewlett-Packard
iod00d@hp.com
grundler@parisc-linux.org

Abstract

Itanium processors have very sophisticated performance monitoring tools integrated into the CPU. McKinley and Madison Itanium CPUs have over three hundred different types of events they can filter, trigger on, and count. The restrictions on which combinations of triggers are allowed is daunting and varies across CPU implementations. Fortunately, the tools hide this complicated mess. While the tools prevent us from shooting ourselves in the foot, it's not obvious how to use those tools for measuring kernel device driver behaviors.

IO driver writers can use pfmon to measure two key areas generally not obvious from the code: MMIO read and write frequency and precise addresses of instructions regularly causing L3 data cache misses. Measuring MMIO reads has some nuances related to instruction execution which are relevant to understanding ia64 and likely ia32 platforms. Similarly, the ability to pinpoint exactly which data is being accessed by drivers enables driver writers to either modify the algorithms or add prefetching directives where feasible. I include some examples on how I used pfmon to measure NIC drivers and give some guidelines on use.

q-syscollect is a "gprof without the pain" kind of tool. While q-syscollect uses the same kernel perfmon subsystem as pfmon, the former works at a higher level. With some knowledge about how the kernel operates, q-syscollect can collect call-graphs, function call counts, and percentage of time spent in particular routines. In other words, pfmon can tell us how much time the CPU spends stalled on d-cache misses and q-syscollect can give us the call-graph for the worst offenders.

Updated versions of this paper will be available from `http://iou.parisc-linux.org/ols2004/`

1 Introduction

Improving the performance of IO drivers is really not that easy. It usually goes something like:

1. Determine which workload is relevant
2. Set up the test environment
3. Collect metrics
4. Analyze the metrics
5. Change the code based on theories about the metrics
6. Iterate on Collect metrics

This paper attempts to make the collect-analyze-change loop more efficient for three

obvious things: MMIO reads, MMIO writes, and cache line misses.

MMIO reads and writes are easier to locate in Linux code than for other OSs which support memory-mapped IO—just search for *readl()* and *writel()* calls. But `pfmon` [1] can provide statistics of actual behavior and not just where in the code MMIO space is touched.

Cache line misses are hard to detect. None of the regular performance tools I've used can precisely tell where CPU stalls are taking place. We can guess some of them based on data usage—like spin locks ping-ponging between CPUs. This requires a level of understanding that most of us mere mortals don't possess. Again, `pfmon` can help out here.

Lastly, getting an overview of system performance and getting run-time call graph usually requires compiler support that gcc doesn't provide. q-tools[4] can provide that information. Driver writers can then manually adjust the code knowing where the "hot spots" are.

1.1 `pfmon`

The author of `pfmon`, Stephane Eranian [2], describes `pfmon` as "the performance tool for IA64-Linux which exploits all the features of the IA-64 Performance Monitoring Unit (PMU)." `pfmon` uses a command line interface and does not require any special privilege to run. `pfmon` can monitor a single process, a multi-threaded process, multi-processes workloads and the entire system.

`pfmon` is the user command line interface to the kernel perfmon subsystem. perfmon does the ugly work of programming the PMU. Perfmon is versioned separately from `pfmon` command. When in doubt, use the perfmon in the latest 2.6 kernel.

There are two major types of measurements: counting and sampling. For counting, `pfmon` simply reports the number of occurrences of the desired events during the monitoring period. `pfmon` can also be configured to sample at certain intervals information about the execution of a command or for the entire system. It is possible to sample any events provided by the underlying PMU.

The information recorded by the PMU depends on what the user wants. `pfmon` contains a few preset measurements but for the most part the user is free to set up custom measurements. On Itanium2, `pfmon` provides access to all the PMU advanced features such as opcode matching, range restrictions, the Event Address Registers (EAR) and the Branch Trace Buffer.

1.2 `pfmon` command line options

Here is a summary of command line options used in the examples later in this paper:

–us-c use the US-style comma separator for large numbers.

–cpu-list=0 bind `pfmon` to CPU 0 and only count on CPU 0

–pin-command bind the command at the end of the command line to the same CPU as `pfmon`.

–resolve-addr look up addresses and print the symbols

–long-smpl-periods=2000 take a sample of every 2000th event.

–smpl-periods-random=0xfff:10 randomize the sampling period. This is necessary to avoid bias when sampling repetitive behaviors. The first value is the mask of bits to randomize (e.g., 0xfff) and the second value is initial seed (e.g., 10).

-k kernel only.

–system-wide measure the entire system (all processes and kernel)

Parameters only available on a to-be-released `pfmon` v3.1:

–smpl-module=dear-hist-itanium2 This particular module is to be used ONLY in conjunction with the Data EAR (Event Address Registers) and presents recorded samples as histograms about the cache misses. By default, the information is presented in the instruction view but it is possible to get the data view of the misses also.

-e data_ear_cache_lat64 pseudo event for memory loads with latency $\geq$ 64 cycles. The real event is `DATA_EAR_EVENT` (counts the number of times Data EAR has recorded something) and the pseudo event expresses the latency filter for the event. Use "`pfmon -ldata_ear_cache*`" to list all valid values. Valid values with McKinley CPU are powers of two ($4 - 4096$).

1.3 q-tools

The author of q-tools, David Mosberger [5], has described q-tools as "gprof without the pain."

q-tools package contains `q-syscollect`, `q-view`, `qprof`, and `q-dot`. `q-syscollect` collects profile information using kernel perfmon subsystem to sample the PMU. `q-view` will present the data collected in both flat-profile and call graph form. `q-dot` displays the call-graph in graphical form. Please see the `qprof` [6] website for details on `qprof`.

`q-syscollect` depends on the kernel perfmon subsystem which is included in all 2.6 Linux kernels. Because `q-syscollect` uses the PMU, it has the following advantages over other tools:

- no special kernel support needed (besides perfmon subsystem).
- provides call-graph of kernel functions
- can collect call-graphs of the kernel while interrupts are blocked.
- measures multi-threaded applications
- data is collected per-CPU and can be merged
- instruction level granularity (not bundles)

2 Measuring MMIO Reads

Nearly every driver uses MMIO reads to either flush MMIO writes, flush in-flight DMA, or (most obviously) collect status data from the IO device directly. While use of MMIO read is necessary in most cases, it should be avoided where possible.

2.1 Why worry about MMIO Reads?

MMIO reads are expensive—how expensive depends on speed of the IO bus, the number bridges the read (and its corresponding read return) has to cross, how "busy" each bus is, and finally how quickly the device responds to the read request. On most architectures, one can precisely measure the cost by measuring a loop of MMIO reads and calling `get_cycles()` before/after the loop.

I've measured anywhere from 1μs to 2μs per read. In practical terms:

- $\sim$ 500–600 cycles on an otherwise-idle 400 MHz PA-RISC machine.

- $\sim$ 1000 cycles on a 450 MHz Pentium machine which included crossing a PCI-PCI bridge.
- $\sim$ 900–1000 cycles on a 800 MHz IA64 HP ZX1 machine.

And for those who still don't believe me, try watching a DVD movie after turning DMA off for an IDE DVD player:

```
hdparm -d 0 /dev/cdrom
```

By switching the IDE controller to use PIO (Programmed I/O) mode, all data will be transferred to/from host memory under CPU control, byte (or word) at a time. `pfmon` can measure this. And `pfmon` looks broken when it displays three and four digit "Average Cycles Per Instruction" (CPI) output.

2.2 Eh? Memory Reads don't stall?

They do. But the CPU and PMU don't "realize" the stall until the next memory reference. The CPU continues execution until memory order is enforced by the acquire semantics in the MMIO read. This means the **Data Event Address Registers record the next stalled memory reference due to memory ordering constraints, not the MMIO read**. One has to look at the instruction stream carefully to determine which instruction actually caused the stall.

This also means the following sequence doesn't work exactly like we expect:

```
writel(CMD,addr);
readl(addr);
udelay(1);
y = buf->member;
```

The problem is the value returned by `read(x)` is never consumed. **Memory ordering imposes no constraint on non-load/store instructions.** Hence `udelay(1)` begins before the CPU stalls. The CPU will stall on `buf->member` because of memory ordering restrictions if the `udelay(1)` completes before `readl(x)` is retired. Drop the `udelay(1)` call and `pfmon` will always see the stall caused by MMIO reads on the next memory reference.

Unfortunately, the IA32 Software Developer's Manual[3] Volume 3, Chapter 7.2 "MEMORY ORDERING" is silent on the issue of how MMIO (uncached accesses) will (or will not) stall the instruction stream. This document is very clear on how "IO Operations" (e.g., IN/OUT) will stall the instruction pipeline until the read return arrives at the CPU. A direct response from Intel(R) indicated `readl()` does not stall like IN or OUT do and IA32 has the same problem. The Intel® architect who responded did hedge the above statement claiming a "udelay(10) will be as close as expected" for an example similar to mine. Anyone who has access to a frontside bus analyzer can verify the above statement by measuring timing loops between uncached accesses. I'm not that privileged and have to trust Intel® in this case.

For IA64, we considered putting an extra burden on `udelay` to stall the instruction stream until previous memory references were retired. We could use dummy loads/stores before and after the actual delay loop so memory ordering could be used to stall the instruction pipeline. That seemed excessive for something that we didn't have a bug report for.

Consensus was adding `mf.a` (memory fence) instruction to `readl()` should be sufficient. The architecture only requires `mf.a` serve as an ordering token and need not cause any delays of its own. In other words, the implementation is platform specific. `mf.a` has not been added to `readl()` yet because every-

thing was working without so far.

2.3 `pfmon -e uc_loads_retired`

IO accesses are generally the only uncached references made on IA64-linux and normally will represent MMIO reads. The basic measurement will tell us roughly how many cycles the CPU stalls for MMIO reads. Get the number of MMIO reads per sample period and then multiply by the actual cycle counts a MMIO read takes for the given device. One needs to measure MMIO read cost by using a CPU internal cycle counter and hacking the kernel to read a harmless address from the target device a few thousand times.

In order to make statements about per transaction or per interrupt, we need to know the cumulative number of transactions or interrupts processed for the sample period. `pktgen` is straightforward in this regard since `pktgen` will print transaction statistics when a run is terminated. And one can record `/proc/interrupts` contents before and after each `pfmon` run to collect interrupt events as well.

Drawbacks to the above are one assumes a homogeneous driver environment; i.e., only one type of driver is under load during the test. I think that's a fair assumption for development in most cases. Bridges (e.g., routing traffic across different interconnects) are probably the one case it's not true. One has to work a bit harder to figure out what the counts mean in that case.

For other benchmarks, like SpecWeb, we want to grab `/proc/interrupt` and networking stats before/after `pfmon` runs.

2.4 tg3 Memory Reads

In summary, Figure 1 shows tg3 is doing $2749675/(1834959 - 918505) \approx 3$ MMIO reads per interrupt and averaging about $5000000/(1834959 - 918505) \approx 5$ packets per interrupt. This is with the BCM5701 chip running in PCI mode at 66MHz:64-bit.

Based on code inspection, here is a break down of where the MMIO reads occur in temporal order:

1. `tg3_interrupt()` flushes MMIO write to `MAILBOX_INTERRUPT_0`
2. `tg3_poll()` → `tg3_enable_ints()` → `tw32(TG3PCI_MISC_HOST_CTRL)`
3. `tg3_enable_ints()` flushes MMIO write to `MAILBOX_INTERRUPT_0`

It's obvious when inspecting `tw32()`, the BCM5701 chip has a serious bug. Every call to `tw32()` on BCM5701 requires a MMIO read to follow the MMIO write. Only writes to mailbox registers don't require this and a different routine is used for mailbox writes.

Given the NIC was designed for zero MMIO reads, this is pretty poor performance. Using a BCM5703 or BCM5704 would avoid the MMIO read in tw32().

I've exchanged email with David Miller and Jeff Garzik (tg3 driver maintainers). They have valid concerns with portability. We agree tg3 could be reduced to one MMIO read after the last MMIO write (to guarantee interrupts get re-enabled).

One would need to use the "tag" field in the status block when writing the mail box register to indicate which "tag" the CPU most recently

```
gsyprf3:~# pfmon -e uc_loads_retired -k --system-wide \
-- /usr/src/pktgen-testing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:     918505          0  IO-SAPIC-level  eth1
Result: OK: 7613693(c7613006+d687) usec, 5000000 (64byte) 656771pps 320Mb/sec
(336266752bps)  errors: 0
 57:    1834959          0  IO-SAPIC-level  eth1
CPU0                     2749675 UC_LOADS_RETIRED
CPU1                        1175 UC_LOADS_RETIRED
}
```

Figure 1: tg3 v3.6 MMIO reads with pktgen/IRQ on same CPU

saw. Using Message Signaled Interrupts (MSI) instead of Line based IRQs would guarantee the most recent status block update (transferred via DMA writes) would be visible to the CPU before `tg3_interrupt()` gets called.

The protocol would allow correct operation without using MSI, too.

2.5 Benchmarking, `pfmon`, and CPU bindings

The purpose of binding `pktgen` to CPU1 is to verify the transmit code path is NOT doing any MMIO reads. We split the transmit code path and interrupt handler across CPUs to narrow down which code path is performing the MMIO reads. This change is not obvious from Figure 2 output since tg3 only performs MMIO reads from CPU 0 (`tg3_interrupt()`).

But in Figure 2, performance goes up 30%! Offhand, I don't know if this is due to CPU utilization (pktgen and `tg3_interrupt()` contending for CPU cycles) or if DMA is more efficient because of cache-line flows. When I don't have any deadlines looming, I'd like to determine the difference.

2.6 e1000 Memory Reads

e1000 version 5.2.52-k4 has a more efficient implementation than tg3 driver. In a nut shell, MMIO reads are pretty much irrelevant to the pktgen workload with e1000 driver using default values.

Figure 3 shows e1000 performs $173315/(703829 - 622143) \approx 2$ MMIO reads per interrupt and $5000000/(703829 - 622143) \approx 61$ packets per interrupt.

Being the curious soul I am, I tracked down the two MMIO reads anyway. One is in the interrupt handler and the second when interrupts are re-enabled. It looks like e1000 will always need at least 2 MMIO reads per interrupt.

3 Measuring MMIO Writes

3.1 Why worry about MMIO Writes?

MMIO writes are clearly not as significant as MMIO reads. Nonetheless, every time a driver writes to MMIO space, some subtle things happen. There are four minor issues to think about: memory ordering, PCI bus utilization, filling outbound write queues, and stalling MMIO reads longer than necessary.

```
gsyprf3:~# pfmon -e uc_loads_retired -k --system-wide \
-- /usr/src/pktgen-testing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    5809687          0  IO-SAPIC-level  eth1
Result: OK: 5914889(c5843865+d71024) usec, 5000000 (64byte) 845451pps 412Mb/se
c (432870912bps)  errors: 0
 57:    6427969          0  IO-SAPIC-level  eth1
CPU0                    1855253 UC_LOADS_RETIRED
CPU1                        950 UC_LOADS_RETIRED
```

Figure 2: tg3 v3.6 MMIO reads with pktgen/IRQ on diff CPU

```
gsyprf3:~# pfmon -e uc_loads_retired -k --system-wide \
-- /usr/src/pktgen-testing/pktgen-single-e1000
Configuring devices
Running... ctrl^C to stop
 59:     622143          0  IO-SAPIC-level  eth3
Result: OK: 10228738(c9990105+d238633) usec, 5000000 (64byte) 488854pps 238Mb/
sec (250293248bps)  errors: 81669
 59:     703829          0  IO-SAPIC-level  eth3
CPU0                     173315 UC_LOADS_RETIRED
CPU1                       1422 UC_LOADS_RETIRED
```

Figure 3: MMIO reads for e1000 v5.2.52-k4

First, memory ordering is enforced since PCI requires strong ordering of MMIO writes. This means the MMIO write will push all previous regular memory writes ahead. This is not a serious issue but it can make a MMIO write take longer.

MMIO writes are short transactions (i.e., much less than a cache-line). The PCI bus setup time to select the device, send the target address and data, and disconnect measurably reduces PCI bus utilization. It typically results in six or more PCI bus cycles to send four (or eight) bytes of data. On systems which strongly order DMA Read Returns and MMIO Writes, the latter will also interfere with DMA flows by interrupting in-flight, outbound DMA.

If the IO bridge (e.g., PCI Bus controller) nearest the CPU has a full write queue, the CPU will stall. The bridge would normally queue the MMIO write and then tell the CPU it's done. The chip designers normally make the write queue deep enough so the CPU never needs to stall. But drivers that perform many MMIO writes (e.g., use door bells) and burst many of MMIO writes at a time, could run into a worst case.

The last concern, stalling MMIO reads longer than normal, exists because of PCI ordering rules. MMIO reads and MMIO writes are strongly ordered. E.g., if four MMIO writes are queued before a MMIO read, the read will wait until all four MMIO write transactions have completed. So instead of say 1000 CPU cycles, the MMIO read might take more than 2000 CPU cycles on current platforms.

3.2 `pfmon -e uc_stores_retired`

`pfmon` counts MMIO Writes with no surprises.

3.3 tg3 Memory Writes

Figure 4 shows tg3 does about 10M MMIO writes to send 5M packets. However, we can break the MMIO writes down into base level (feed packets onto transmit queue) and `tg3_interrupt` which handles TX (and RX) completions. Knowing which code path the MMIO writes are in helps track down usage in the source code.

Output in Figure 5 is after hacking the `pktgen-single-tg3` script to bind `pktgen` kernel thread to CPU 1 when eth1 is directing interrupts to CPU 0. The distribution between TX queue setup and interrupt handling is obvious now. CPU 0 is handling interrupts and performs $3013580/(5803789 - 5201193) \approx 5$ MMIO writes per interrupt. CPU 1 is handling TX setup and performs $5000376/5000000 \approx 1$ MMIO write per packet.

Again, as noted in section 2.5, binding pktgen thread to one CPU and interrupts to another, changes the performance dramatically.

3.4 e1000 Memory Writes

Figure 6 shows $248891/(991082 - 908366) \approx 3$ MMIO writes per interrupt and $5001303/5000000 \approx 1$ MMIO write per packet. In other words, slightly better than tg3 driver. Nonetheless, the hardware can't push as many packets. One difference is the e1000 driver is pushing data to a NIC behind a PCI-PCI Bridge.

Figure 7 shows a ≈40% improvement in throughput[1] for pktgen without a PCI-PCI Bridge in the way. Note the ratios of MMIO writes per interrupt and MMIO writes per

[1] This demonstrates how the distance between the IO device and CPU (and memory) directly translates into latency and performance.

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/pktgen-test
ing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    4284466          0  IO-SAPIC-level  eth1
Result: OK: 7611689(c7610900+d789) usec, 5000000 (64byte) 656943pps 320Mb/sec
(336354816bps)  errors: 0
 57:    5198436          0  IO-SAPIC-level  eth1
CPU0                    9570269 UC_STORES_RETIRED
CPU1                        445 UC_STORES_RETIRED
```

Figure 4: tg3 v3.6 MMIO writes

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/
pktgen-testing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    5201193          0  IO-SAPIC-level  eth1
Result: OK: 5880249(c5811180+d69069) usec, 5000000 (64byte) 850340pps 415Mb
/sec (435374080bps)  errors: 0
 57:    5803789          0  IO-SAPIC-level  eth1
CPU0                    3013580 UC_STORES_RETIRED
CPU1                    5000376 UC_STORES_RETIRED
```

Figure 5: tg3 v3.6 MMIO writes with pktgen/IRQ split across CPUs

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/
pktgen-testing/pktgen-single-e1000
Running... ctrl^C to stop
 59:     908366          0  IO-SAPIC-level  eth3
Result: OK: 10340222(c10104719+d235503) usec, 5000000 (64byte) 483558pps 236Mb
/sec (247581696bps)  errors: 82675
 59:     991082          0  IO-SAPIC-level  eth3
CPU0                     248891 UC_STORES_RETIRED
CPU1                    5001303 UC_STORES_RETIRED
```

Figure 6: MMIO writes for e1000 v5.2.52-k4

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/pktgen-test
ing/pktgen-single-e1000
Running... ctrl^C to stop
 71:          3          0  IO-SAPIC-level  eth7
Result: OK: 7491358(c7342756+d148602) usec, 5000000 (64byte) 667467pps 325Mb/s
ec (341743104bps)  errors: 59870
 71:      59907          0  IO-SAPIC-level  eth7
CPU0                     180406 UC_STORES_RETIRED
CPU1                    5000939 UC_STORES_RETIRED
```

Figure 7: e1000 v5.2.52-k4 MMIO writes without PCI-PCI Bridge

packet are the same. I doubt the MMIO reads and MMIO writes are the limiting factors. More likely DMA access to memory (and thus TX/RX descriptor rings) limits NIC packet processing.

4 Measuring Cache-line Misses

The Event Address Registers[2] (EAR) can only record one event at a time. What is so interesting about them is that they record precise information about data cache misses. For instance for a data cache miss, you get the:

- address of the instruction, likely a load
- address of the target data
- latency in cycles to resolve the miss

The information pinpoints the source of the miss, not the consequence (i.e., the stall).

The Data EAR (DEAR) can also tell us about MMIO reads via sampling. The DEAR can only record *loads* that miss, not stores. Of course, MMIO reads always miss because they are uncached. This is interesting if we want to track down which MMIO addresses are "hot." It's usually easier to track down usage in source code knowing which MMIO address is referenced.

Collecting with DEAR sampling requires two parameters be tweaked to statistically improve the samples. One is the frequency at which Data Addresses are recorded and the other is the threshold (how many CPU cycles latency).

Because we know the latency to L3 is about 21 cycles, setting the EAR threshold to a value higher (e.g., 64 cycles) ensures only the load misses accessing main memory will be captured. This is how to select which level of cacheline misses one samples.

While high threshholds (e.g., 64 cycles) will show us where the longest delays occur, it will not show us the worst offenders. Doing a second run with a lower threshold (e.g., 4 cycles) shows all L1, L2, and L3 cache misses and provides a much broader picture of cache utilization.

When sampling events with low threshholds, we will get saturated with events and need to reduce the number of events actually sampled to every 5000th. The appropriate value will depend on the workload and how patient one is. The workload needs to be run long enough to be statistically significant and the sampling period needs to be high enough to not significantly perturb the workload.

4.1 tg3 Data Cache misses > 64 cycles

For the output in Figure 8, I've iteratively decreased the smpl-periods until I noticed the total pktgen throughput starting to drop. Figure 8 output only shows the tg3 interrupt code path since `pfmon` is bound to CPU 0. Normally, it would be useful to run this again with `cpu-list=1`. We could then see what the TX code path and pktgen are doing.

Also, the `pin-command` option in this example doesn't do anything since `pktgen-single-tg3` directs a `pktgen` kernel thread bound CPU 1 to do the real work. I've included the option only to make people aware of it.

4.2 tg3 Data Cache misses > 4 cycles

Figure 9 puts the `lat64` output in Figure 8 into better perspective. It shows tg3 is spending more time for L1 and L2 misses than L3 misses

[2] **`pfmon` v3.1 is the first version to support EAR and is expected to be available in August, 2004.**

```
gsyprf3:~# pfmon31 --us-c --cpu-list=0 --pin-command --resolve-addr \
           --smpl-module=dear-hist-itanium2 \
           -e data_ear_cache_lat64 --long-smpl-periods=500 \
           --smpl-periods-random=0xfff:10 --system-wide \
           -k -- /usr/src/pktgen-testing/pktgen-single-tg3
added event set 0
only kernel symbols are resolved in system-wide mode
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    7209769          0  IO-SAPIC-level  eth1
Result: OK: 5915877(c5845032+d70845) usec, 5000000 (64byte) 845308pps 412Mb/sec
(432797696bps)  errors: 0
 57:    7827812          0  IO-SAPIC-level  eth1
# total_samples 672
# instruction addr view
# sorted by count
# showing per per distinct value
# %L2  : percentage of L1 misses that hit L2
# %L3  : percentage of L1 misses that hit L3
# %RAM : percentage of L1 misses that hit memory
# L2   :  5 cycles load latency
# L3   : 12 cycles load latency
# sampling period: 500
#count %self   %cum    %L2    %L3    %RAM   instruction addr
  38  5.65%   5.65%  0.00%  0.00% 100.00% 0xa000000100009141 ia64_spinlock_contention
                                                            +0x21<kernel>
  36  5.36%  11.01%  0.00%  0.00% 100.00% 0xa00000020003e580 tg3_interrupt[tg3]+0xe0<kernel>
  32  4.76%  15.77%  0.00%  0.00% 100.00% 0xa000000200034770 tg3_write_indirect_reg32[tg3]
                                                            +0x90<kernel>
  32  4.76%  20.54%  0.00%  0.00% 100.00% 0xa00000020003e640 tg3_interrupt[tg3]+0x1a0<kernel>
  30  4.46%  25.00%  0.00%  0.00% 100.00% 0xa000000200034e91 tg3_enable_ints[tg3]+0x91<kernel>
  29  4.32%  29.32%  0.00%  0.00% 100.00% 0xa00000020003e510 tg3_interrupt[tg3]+0x70<kernel>
  28  4.17%  33.48%  0.00%  0.00% 100.00% 0xa00000020003d1a0 tg3_tx[tg3]+0x2e0<kernel>
  27  4.02%  37.50%  0.00%  0.00% 100.00% 0xa00000020003cfa0 tg3_tx[tg3]+0xe0<kernel>
  24  3.57%  41.07%  0.00%  0.00% 100.00% 0xa00000020003cfd1 tg3_tx[tg3]+0x111<kernel>
  21  3.12%  44.20%  0.00%  0.00% 100.00% 0xa000000200034e60 tg3_enable_ints[tg3]+0x60<kernel>
   .
   .
   .
# level 0 : counts=0 avg_cycles=0.0ms   0.00%
# level 1 : counts=0 avg_cycles=0.0ms   0.00%
# level 2 : counts=672 avg_cycles=0.0ms 100.00%
approx cost: 0.0s
```

Figure 8: tg3 v3.6 lat64 output

```
gsyprf3:~# pfmon31 --us-c --cpu-list=0 --resolve-addr --smpl-module=dear-hist-itanium2 \
-e data_ear_cache_lat4 --long-smpl-periods=5000 --smpl-periods-random=0xfff:10 \
--system-wide -k -- /usr/src/pktgen-testing/pktgen-single-tg3
added event set 0
only kernel symbols are resolved in system-wide mode
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    8484552          0  IO-SAPIC-level  eth1
Result: OK: 5938001(c5866437+d71564) usec, 5000000 (64byte) 842034pps 411Mb/sec
            (431121408bps)  errors: 0
 57:    9093642          0  IO-SAPIC-level  eth1
# total_samples 795
# instruction addr view
# sorted by count
# showing per per distinct value
# %L2  : percentage of L1 misses that hit L2
# %L3  : percentage of L1 misses that hit L3
# %RAM : percentage of L1 misses that hit memory
# L2   :  5 cycles load latency
# L3   : 12 cycles load latency
# sampling period: 5000
# #count %self    %cum     %L2     %L3    %RAM   instruction addr
      95  11.95%  11.95%   0.00%  98.95%   1.05% 0xa00000020003d150 tg3_tx[tg3]+0x290<kernel>
      83  10.44%  22.39%  93.98%   4.82%   1.20% 0xa00000020003d030 tg3_tx[tg3]+0x170<kernel>
      21   2.64%  25.03%   0.00%  95.24%   4.76% 0xa0000001000180f0 ia64_handle_irq+0x170<kernel>
      20   2.52%  27.55%   5.00%  80.00%  15.00% 0xa00000020003d040 tg3_tx[tg3]+0x180<kernel>
      18   2.26%  29.81%  50.00%  11.11%  38.89% 0xa00000020003cfa0 tg3_tx[tg3]+0xe0<kernel>
      17   2.14%  31.95%   0.00%   0.00% 100.00% 0xa00000020003e671 tg3_interrupt[tg3]
                                                                    +0x1d1<kernel>
      17   2.14%  34.09%   0.00% 100.00%   0.00% 0xa00000020003e700 tg3_interrupt[tg3]
                                                                    +0x260<kernel>
      16   2.01%  36.10%  56.25%  43.75%   0.00% 0xa000000100012160 ia64_leave_kernel
                                                                    +0x180<kernel>
      16   2.01%  38.11%  62.50%   0.00%  37.50% 0xa00000020003cf60 tg3_tx[tg3]+0xa0<kernel>
      15   1.89%  40.00%  86.67%   6.67%   6.67% 0xa00000020003cfd0 tg3_tx[tg3]+0x110<kernel>
      15   1.89%  41.89%   0.00%   0.00% 100.00% 0xa000000100016041 do_IRQ+0x1a1<kernel>
      15   1.89%  43.77%   0.00%  53.33%  46.67% 0xa00000020003e370 tg3_poll[tg3]+0x350<kernel>
      .
      .
      .
# level 0 : counts=226 avg_cycles=0.0ms  28.43%
# level 1 : counts=264 avg_cycles=0.0ms  33.21%
# level 2 : counts=305 avg_cycles=0.0ms  38.36%
approx cost: 0.0s
```

Figure 9: tg3 v3.6 lat4 output

and in only two locations. Adding one prefetch to pull data from L3 into L2 would help for the top offender. One needs to figure out which bit of data each recorded access refers to and determine how early one can prefetch that data.

We can also rule out MMIO accesses as the top culprit. `tg3_interrupt+0x1d1` could be an MMIO read but it doesn't show up in Figure 8 like `tg3_write_indirect_reg32` does.

Note `smpl-periods` is 10x higher in Figure 9 than in Figure 8. Collecting 10x more samples with `lat4` definitely disturbs the workload.

5 q-tools

`q-syscollect` and `q-view` are trivial to use. An example and brief explanation for kernel usage follow.

Please remember most applications spend most of the time in user space and not in the kernel. q-tools is especially good in user space.

5.1 `q-syscollect`

```
q-syscollect -c 5000 -C 5000 -t
20 -k
```

This will collect system wide kernel data during the 20 second period. Twenty to thrity seconds is usually long enough to get sufficient accuracy[3]. However, if the workload generates a very wide call graph with even distribution, one will likely need to sample for longer periods to get accuracy in the $\pm 1\%$ range. When in doubt, try sampling for longer periods to see if the call-counts change significantly.

[3]See Page 7 of the David Mosberger's Gelato talk [4] for a nice graph on accuracy which *only applies to his example.*

The `-c` and `-C` set the call sample rate and code sample rate respectively. The call sample rate is used to collect function call counts. This is one of the key differences compared to traditional profiling tools: q-syscollect obtains call-counts in a statistical fashion, just as has been done traditionally for the execution-time profile. The code sample rate is used to collect a flat profile (`CPU_CYCLES` by default).

The `-e` option allows one to change the event used to sample for the flat profile. The default is to sample CPU_CYCLES event. This provides traditional execution time in the flat profile.

The data is stored in the current directory under `.q/` directory. The next section demonstrates how `q-view` displays the data.

5.2 `q-view`

I was running the netperf [7] TCP_RR test in the background to another server when I collected the following data. As Figure 10 shows, this particular TCP_RR test isn't costing many cycles in tg3 driver. Or, at least not ones I can measure.

`tg3_interrupt()` shows up in the flat profile with 0.314 seconds time associated with it. The time measurement is only possible because `handle_IRQ_event()` re-enables interrupts if the IRQ handler is not registered with `SA_INTERRUPT` (to indicate latency sensitive IRQ handler). `do_IRQ()` and other functions in that same call graph do NOT have any time measurements because interrupts are disabled. As noted before, the call-graph is sampled using a different part of the PMU than the part which samples the flat profile.

Lastly, I've omitted the trailing output of `q-view` which explains the fields and columns more completely. Read that first be-

```
gsyprf3:~# q-view .q/kernel-cpu0.info | more
Flat profile of CPU_CYCLES in kernel-cpu0.hist#0:
 Each histogram sample counts as 200.510u seconds
% time      self     cumul     calls self/call  tot/call name
 68.88     13.41     13.41      215k     62.5u     62.5u default_idle
  2.90      0.56     13.97      431k     1.31u     1.31u finish_task_switch
  2.50      0.49     14.46      233k     2.09u     4.89u tg3_poll
  1.77      0.35     14.80     1.38M      251n      268n ipt_do_table
  1.61      0.31     15.12      240k     1.31u     1.31u tg3_interrupt
  1.51      0.29     15.41      240k     1.22u     5.95u net_rx_action
  .
  .
  .
Call-graph table:
index %time      self  children         called     name
                                                    <spontaneous>
[176]  69.4     30.5m      13.4         -           cpu_idle
                29.5m     0.285      231k/457k      schedule [164]
                10.0m      0.00      244k/244k      check_pgt_cache [178]
                 13.4      0.00      215k/215k      default_idle [177]
-----------------------------------------------
   .
   .
   .
-----------------------------------------------
                0.293      1.14      240k           __do_softirq [40]
[56]    7.4     0.293      1.14      240k           net_rx_action
                0.487     0.649      233k/233k      tg3_poll [57]
-----------------------------------------------
                0.487     0.649      233k           net_rx_action [56]
[57]    5.9     0.487     0.649      233k           tg3_poll
                    -      0.00      229k/229k      tg3_enable_ints [133]
                97.7m     0.552      225k/225k      tg3_rx [61]
                    -      0.00      227k/227k      tg3_tx [58]
-----------------------------------------------
   .
   .
   .
-----------------------------------------------
                    -      1.88      348k           ia64_leave_kernel [10]
[11]    9.7         -      1.88      348k           ia64_handle_irq
                    -      1.52      239k/240k      do_softirq [39]
                    -     0.367      356k/356k      do_IRQ [12]
-----------------------------------------------
   .
   .
   .
```

Figure 10: `q-view` output for TCP_RR over tg3 v3.6

fore going through the rest of the output.

6 Conclusion

6.1 More pfmon examples

CPU L2 cache misses in one kernel function
`pfmon --verb -k \ --irange=sba_alloc_range \ -el2_misses --system-wide \ --session-timeout=10`
Show all L2 cache misses in `sba_alloc_range`. This is interesting since `sba_alloc_range()` walks a bitmap to look for "free" resources. One can instead specify `-el3_misses` since L3 cache misses are much more expensive.

CPU 1 memory loads
`pfmon --us-c \ --cpu-list=1 \ -e loads_retired \ -k --system-wide \ -- /tmp/pktgen-single` Only count memory loads on CPU 1. This is useful for when we can bind the interrupt to CPU 1 and the workload to a different CPU. This lets us separate interrupt path from base level code, i.e., when is the load happening (before or after DMA occurred) and which code path should one be looking more closely at.

List EAR events supported `pfmon -lear` List all EAR types supported by `pfmon`[4].

More info on Event `pfmon -i DATA_EAR_TLB_ALL` `pfmon` can provide more info on particular events it supports.

[4] EAR isn't supported until `pfmon v3.1`

6.2 And thanks to...

Special thanks to Stephane Eranian [2] for dedicating so much time to the perfmon kernel driver and associated tools. People might think the PMU does it all—but only with a lot of SW driving it. His review of this paper caught some good bloopers. This talk only happened because I sit across the aisle from him and could pester him regularly.

Thanks to David Mosberger[5] for putting together q-tools and making it so trivial to use.

In addition, in no particular order:
Christophe de Dinechin, Bjorn Helgaas, Matthew Wilcox, Andrew Patterson, Al Stone, Asit Mallick, and James Bottomley for reviewing this document or providing technical guidance.

Thanks also to the OLS staff for making this event happen every year.

My apologies if I omitted other contributors.

References

[1] perfmon homepage, `http://www.hpl.hp.com/research/linux/perfmon/`

[2] Stephane Eranian, `http://www.gelato.org/community/gelato_meeting.php?id=CU2004#talk22`

[3] The IA-32 Intel(R) Architecture Software Developer's Manuals, `http://www.intel.com/design/pentium4/manuals/253668.htm`

[4] q-tools homepage, `http://www.hpl.hp.com/`

research/linux/
q-tools/

[5] David Mosberger,
http://www.gelato.org/
community/gelato_
meeting.php?id=CU2004#
talk19

[6] qprof homepage,
http://www.hpl.hp.com/
research/linux/qprof/

[7] netperf homepage, http:
//www.netperf.org/