UMaine VMW Group Tech Report
UMAINE-VMW-TR-PEBS-IBS-SAMPLING-2016-08

Advanced Hardware Profiling and Sampling (PEBS, IBS, etc.): Creating a New PAPI Sampling Interface

Vincent M. Weaver
University of Maine
vincent.weaver@maine.edu

August 2, 2016

Abstract

One common way of conducting performance analysis is profiling via sampling, where periodically the CPU is interrupted and its state is recorded, and full system behavior is extrapolated. This can be easily done if the counter hardware supports an overflow interrupt so that processor state can be periodically polled. This methodology can give detailed program behavior at the expense of overhead and accuracy.

Modern CPUs often support more advanced forms of sampling. This includes Intel's Precise Event-Based Sampling (PEBS) and AMD's Instruction Based Sampling (IBS). These allow gathering the samples in hardware, possibly multiple at a time, with potentially much lower overhead. In addition, the sampling hardware provides other benefits, such as low "skid" events, which improves accuracy.

We would like PAPI to support these new interfaces, but it will require an overhaul to the existing PAPI sampling interfaces.

1 Introduction

When conducting performance analysis the easiest type of report to get is for the total, aggregate results. The total number of cycles a program ran, the total number of cache misses, the total wall clock time the program ran. While this is of interest, often more detail is wanted. *Where* did the cache misses happen, *what* function took the most cycles, etc. The most straightforward way to get this type of information is to periodically interrupt the program's execution and gather performance information. A detailed breakdown of program behavior can be extrapolated based on these representative samples. There is a tradeoff between overhead and accuracy; the more often you sample the more accurate the results, but at the same time if you sample *too* frequently you will start to add overhead and affect the program being measured.

Modern processors try to help with sampling and provide hardware interfaces. However the features provided by the hardware overlap a bit and there can be some confusion about what one means by "sampling". It can be any of the following features:

- **Sampled Profiling** – as described earlier, periodically interrupting program execution to grab a hardware event count or register state. Most CPUs can do this purely hardware or can emulate it in software (by using some sort of timer).
- **Low-latency Sampling** – instead of having periodic interrupts and manually gather program state, some hardware allows automatically sampling multiple times to a dedicated memory buffer without any operating system (interrupt) involvement. This has lower latency than traditional interrupt-based sampling. Intel PEBS does this.
- **Hardware Profiling** – at regular intervals the CPU is interrupted and detailed information about the current instruction is logged. Often the

actual instruction logged is randomly chosen after a certain trigger point. AMD IBS and PEBS do this.

- **Extra CPU State** – PEBS and IBS log additional CPU state that cannot be obtained from the context structure provided by the overflow signal handler. This includes register state, kernel register state (if the interrupt happened in the kernel), branch predictor outcome, instruction latencies, sources of cache misses, etc.
- **Low-skid Interrupts** – One issue with measurements involving interrupts is "skid": once an overflow interrupt happens, it takes a CPU (especially modern complex out-of-order designs) some amount of time to stop the processor and pinpoint exactly which instruction was active at time of the overflow. Often there is an offset between the instruction indicated versus the one causing the interrupt (this offset is called skid). PEBS and IBS provide support for low-skid sampling, at the expense of some additional overhead.
- **Last Branch Sampling** – The hardware keeps track of the last branches taken, and allows generating call stacks. Intel LBR allows this.
- **Processor Trace** – The CPU logs to a buffer details on all instructions being executed (although usually this is filtered, as the raw data stream can be huge otherwise). Intel Processor Trace and ARM CoreSight are examples of this.

This document describes in more detail these interfaces as well as how to access them on Linux machines. We then describe how the PAPI [1] performance library currently supports sampling, and how it can be improved.

2 Hardware Sampling Interfaces

Sampling interfaces vary by vendor and processor model. This is a quick overview of support found on recent processors.

2.1 Intel x86_64

2.1.1 Traditional Sampling

Intel chips have supported hardware interrupts on counter overflow since the introduction of hardware performance counters in the original Pentium processor.

To conduct sampling, one programs a counter so that it will overflow after a certain number of counts (say 100,000 cycles). This will generate a hardware interrupt. The hardware interrupt can then gather information on the program state at this time (such as registers, program counter, event counts, etc) and return it to the user.

Under Linux a NMI (non-maskable) interrupt is used, which means sampling can (with certain limitations) also be used to profile the operating system kernel.

Intel hardware with architectural performance monitoring 2 or later support "PMI Overhead Mitigation" which allows freezing counts automatically when a performance monitoring interrupt (PMI) happens. This allows measurements to not include the counts imposed by the interrupt handler in with the rest of the results.

2.1.2 Precise Event-based Sampling (PEBS)

Recent Intel chips support Precise Event Based Sampling (PEBS), as described in Chapter 18 of the Intel 64 and IA-32 Architectures Software Developer's Manual (Volume 3)[2]. PEBS support originated in Pentium 4 and Core architectures. It is available on all subsequent processors as well as some Atom processors (Silvermont and later?). Note: support is broken on some Sandybridge-EP machines due to a firmware bug, so if you see a message in `dmesg` describing this be sure you update your CPU microcode.

When a processor is configured for a PEBS event overflow info can be logged efficiently (possibly without even triggering a costly interrupt?). When the overflow happens, PEBS is armed. The next instruction that triggers the event is logged, and assuming there is space in the relevant DS (data store) area a record is logged holding information on the sample.

Only a subset of events can be used as PEBS events, and sometimes only a certain counter slot can be used.

Things that can be configured vary by architecture but include: Trap vs fault: specifies whether the event recorded is the next or the current one. Registers to save: can specify whether all registers are saved, or just the instruction pointer/flags. Can also store latency data, transactional memory data, TSC value, and counter value.

Nehalem

Nehalem lets you gather PEBS in all four event slots and load latency. The PEBS results indicate the state *after* the instruction has executed.

It also supports load latency, which reports latency in cycles from first dispatch to final retirement. When enabled, load instructions are randomly chosen to accumulate the load latency info. The returned value for latency is the last randomly tagged event, not necessarily the one that triggered the PEBS operation. Info returned is Data Linear Address (linear address (physical mem?) of the value being loaded), latency value, and data source which indicates where in the memory hierarchy the load happened from.

Sandybridge

When hyperthreading is enabled up to 8 performance counters are available, but PEBS only works in the lower 4. Now supports store instructions for load latency at least as far as linear address and type, but cycle count always zero. More PEBS events supported. When load latency is enabled it has to be the only active PEBS event. In addition info is returned on whether the load hit the DTLB/STLB.

Precise store is now available. Only in counter3. After PEBS triggers, info on the very next store are recorded.

Low-skid. Precise Distribution of Instructions Retired (PDIR). Provides low-skid results. Senses when an interrupt is about to happen and prepares for it (possibly slows down execution noticeably too). Only INST_RETIRED.ALL in counter slot 1. Enables trap on the actual instruction that caused the event.

Haswell

Precise store replaced by data address profiling. DataLA or Data Linear Address Profiling. When the PEBS record is generated, also records the linear address of the destination of load or source of store. Also indicates if it was a hit in closest (likely L1) cache.

Eventing IP. The address of the instruction that caused the PEBS event is also recorded.

Also support for transactional memory in PEBS.

Skylake

Adds a field that reports TSC value. Adds additional front-end events (iTLB and iCache misses).

Xeon Phi

Possibly has PEBS support(?).

Atom

On Goldmont Atom can record PEBS records for all events. However for non-precise events there's no guarantee about what instruction actually generated the sample. It can also record the TSC, and info on which event caused the overflow (if multiple are enabled). Reduced skid and linear address are also available.

2.1.3 Last Branch Record (LBR)

Starting with the Pentium 4 most Intel hardware supports logging a trace of the last branches that were executed (Last Branch Record). See Chapter 17 of the Vol3b documentation. The number of branches recorded varies from 4 up to 32.

This is not strictly a sampling interface but at least under Linux the data is handled similarly.

If the Branch Trace Store (BTS) is enabled, the last branch record of the last N records can be written out to memory in the BTS area, part of the Debug Store (DS) save area. It can be a circular buffer, and can cause an interrupt when full. This is documented as having the possibility to slow down program execution.

The LBR record contains various information on the branch. Last branch from (location branched from), Last branch to (location branched to), predicted (whether the branch was correctly predicted or not).

Nehalem

You can filter the branch types you are interested in.

Haswell

Supports call-stack recording, where you can configure it to record the branches

in a LIFO setup (i.e. when you return from a function call, the branches that have happened since the initial call to the function are backed off). This allows generating a call stack more easily especially with programming languages that have deep call trees.

Skylake

Changes the format a bit, and includes transactional memory info as well as cycle counts. Has 32 entries now. Can capture length of time in basic block with TSC.

Atom Goldmont

You can also obtain number of cycles since last branch.

2.1.4 Intel Resource Director (RDT, CMT, MBM, CQM)

Available on server machines (Haswell Xeon E5 v3 and newer).

Cache monitoring technology (CMT). Can measure cache occupancy of program in last level cache.

Memory bandwidth monitoring (MBM). Monitor memory bandwidth between cache levels.

Resource monitoring ID (RMID) to a task, processor or group of processors, used for monitoring. Then the values can be read out.

On Xeon E5 v4 processors (Broadwell) RDT also supports cache allocation technology (CAT) and code data prioritization (CDP). This allows one to give hints on how much cache a program should be allowed to use.

CQM is Cache Quality-of-service Monitoring but the most recent volume 3B does not mention it? Related to Memory Bandwidth Monitoring (MBM) [3].

2.1.5 Processor Trace (PT)

Intel Processor Trace [4] lets you record program execution traces. The first implementation is control flow tracing. Can log enough info to give an exact program flow trace. Can generate basic block vectors. Can trace power events too.

Aims for less than 5% overhead, records latency info. Records one bit taken/not-taken on conditional branches, enough to reconstruct program flow. Mode changes also logged.

2.1.6 Data Save Area (DS)

This is a memory region where LBR and PEBS data is recorded. It is divided into three parts; a management area which has all the configuration information, the BTS area and the PEBS area. Each area has a given size, and a threshold when an interrupt is generated. For PEBS it also holds the value of what to reset the PEBS threshold to be.

2.2 AMD x86_64

2.2.1 IBS

AMD chips support Instruction Based Sampling (IBS). This is described in the various BIOS and Kernel Development Guides [5, 6]. There have also been various other papers describing it in more detail [7, 8].

IBS was introduced with Barcelona (fam10h) to aid in creating low-skid profiles.

IBS selects a random instruction (or micro op) and record info. An interrupt is generated after this has happened. Two types; instruction fetch (TLB and instruction cache behavior) and instruction execution.

For instruction fetch the following information is logged: if fetch completed or was aborted, number of cycles spent on the fetch, if it hit in the caches and TLB, the linear/physical address corresponding to the fetch. For instruction execution the following is logged: only one micro-op of the instruction can be tagged, branch status of the instruction, linear/physical address of instruction, linear/physical address of load/store destination, (linear address is address after segmentation) data cache statistics (hit or not, latency), clocks from tag until retire, clocks from execution until retire, DRAM and MMIO source info.

Unlike PEBS these values aren't stored in a memory buffer, but in a set of MSRs. The counter is only 20 bits wide? Only one pending at a time, cannot write multiple samples to a buffer.

Only works on three events, cycles, cycles:p, and uops.

2.3 Intel Itanium

Itanium supports an Execution Trace Buffer for recording traces.

It also supports Instruction EAR and Data EAR (address range constraints) which PAPI supports when profiling.

2.4 ARM

ARM has no PEBS or IBS equivalent, but it does have something similar to processor trace called CoreSight.

2.5 POWER

I am sure POWER has some advanced features, have not had time to look into them yet.

3 Existing Tool Support

3.1 Software Profiling

3.1.1 profil

On some UNIX implementations there is a `profil()` system call that will periodically interrupt program execution and generate a profile histogram.

Linux does not support this system call, although the C library implements it in software via an itimer timer that triggers every 10ms.

3.1.2 gprof

gprof lets you instrument your program at compile time (with the `-pg` option) and then at run time it will report how long each function was called and how much time was spent in it. This is a bit intrusive overhead wise, and requires you have access to the source code.

3.1.3 valgrind

Valgrind [9] does dynamic-binary instrumentation. One of its tools is "callgrind" which will instrument basic blocks on the fly and allow creating profiles which can be viewed with the "callgrind_annotate" tool. It also has "cachegrind" which runs the code through a cache simulator.

The primary downside to valgrind and similar tools is the slowdown which ranges from 10-100x slower than natively running.

3.2 NUMA Profiling

3.2.1 numap

numap [10] presents an API for gathering sampled data for use when analyzing NUMA systems.

The API:

- `init_samp_session()` – initialize data structures
- `samp_read_start()`, `samp_write_start()` – setup the perf_event() mmap buffer, once per thread
- `get_count()` – returns results
- `samp_read_stop()`, `samp_write_stop()` – when finished
- `print_rd()`,`print_wr()` – pretty print the results from the samples

First `init_samp_session()` called to specify threads to be profiled. Then `samp_read_start()` called to setup the mmap buffer. The code of interest happens. Then `samp_read_stop()` called to stop sampling. Finally the results printed with `print_rd()` which decodes the binary blob returned by the kernel.

It is also possible to get the data results directly (the data of interest is mostly the PEBS data: instruction pointer of the instruction, address of the load/store, "weight" which is the number of cycles, and data_src which is the part of the hierarchy causing the result.

3.2.2 Others

Memphis [11] and MemProf [12] (AMD only, IBS). Need to fill in with more details.

HPCToolkit [13]

I am sure I am missing some.

3.3 GPU Profiling

Some GPU hardware supports a profiling interface too, specifically recent NVIDIA devices [14].

For MAXWELL GPUs and CUDA 7.5. With CUPTI you create a sampling data structure `PC_SAMPLING_ACTIVITY`, `SOURCE_LOCATOR`, and `KERNEL_ACTIVITY`.

To use the Activity API you initialize, register callbacks, enable the activities, and set the sample rate.

4 Linux perf_event interface

Linux perf_event, as of the 4.7 kernel (current as of the writing of this), supports much of the advanced hardware sampling interfaces.

4.1 Sampled Profiling

As long as your system supports overflow interrupts you can do statistical sampling with perf. `perf record -e cpu-cycles` followed by `perf report` You can specify the event, the frequency, and a whole host of other options.

Even if you do not have a proper overflow interrupt (for example, one is not available on the original BCM2835 Raspberry Pi machines) you can usually simulate it using the `task-clock` software event.

4.2 Low-latency Sampling

Perf will do low-latency sampling automatically if you have a PEBS capable CPU and you are sampling on a PEBS event. The samples are gathered until a watermark threshold is crossed, and only then will you receive a notification that the buffer is full and ready to be processed.

Perf can do something similar even without PEBS, in that you can queue up multiple samples before having userspace notified. This avoids the overhead of your process being woken up with a signal every time the event triggers, but there is still the overall overhead of an interrupt happening every overflow.

By default perf cannot do multiple PEBS samples at once, as it needs to record other values that only the OS can provide, such as pid/tid. This is known as "single-entry" mode.

To enter N-entry mode need to set fixed period, no timestamp (except Skylake), PEBS buffer flushed on context-switches, no LBR [15].

4.3 Hardware Profiling

AMD IBS is a bit different than PEBS. Rather than a memory buffer, the sampled info is stored in a set of MSRs that need to be read before the next sample comes in. There are a much more limited set of events that can be used. Also there are two types of IBS profiles, frontend and backend.

The IBS results are presented as separate perf PMUs, ibs_op and ibs_fetch. The sampled results will have the instruction pointer and register results, but the other extended values are *not* folded back into the normal sampling interface like PEBS results (no weight, ip, etc field), but the results are dumped as a RAW result and you will need a special decoder to gather.

The various IBS parameters (as described in the event docs) can be set manually, such as below setting the random low bit enable (to slightly randomize where overflows happen).

```
perf record -e ibs_fetch/rand_en=1/GH
perf record -e ibs_op/cnt_ctl=1/GH
```

4.4 Extra Processor State

Linux perf_event supports returning a large amount of data with each sample. Some of the sample types are extended with PEBS data when available.

Currently any of the following can be dumped into a sample by perf:

- PERF_SAMPLE_IP – instruction pointer
- PERF_SAMPLE_TID – thread ID
- PERF_SAMPLE_TIME – a timestamp
- PERF_SAMPLE_ADDR – effective address (requires PEBS?)
- PERF_SAMPLE_READ – counts for all events in group
- PERF_SAMPLE_CALLCHAIN – helps if you have LBR?
- PERF_SAMPLE_ID – a unique id for the group leader
- PERF_SAMPLE_CPU – current CPU
- PERF_SAMPLE_PERIOD – current sampling period
- PERF_SAMPLE_STREAM_ID – another unique ID

- PERF_SAMPLE_RAW – raw data (PMU specific).
  On IBS this contains the raw MSR dumps which include the below (and other) info:
  - Fetch: Randomize event enabled, TLB miss, TLB size, icache miss, fetch addresses
  - Execute: address, microcode, branch fused, branch predicted, cache hit, offcore (northbridge) source, tlb latency, memory width, l2 cache miss, load or store, TLB stats, alignment, branch target access, physical address
- PERF_SAMPLE_BRANCH_STACK – branch stack from LBR
- PERF_SAMPLE_REGS_USER – current user level register state.
- PERF_SAMPLE_STACK_USER – user stack, to allow stack unwinding (usefull for call traces)
- PERF_SAMPLE_WEIGHT – for PEBS this is the cycle time
- PERF_SAMPLE_DATA_SRC – this is the PEBS cache miss hierarchy info
- PERF_SAMPLE_IDENTIFIER – another unique ID, but in a fixed location
- PERF_SAMPLE_TRANSACTION – has to do with Intel TSX transactional memory
- PERF_SAMPLE_REGS_INTR – current register state at interrupt, can be in userspace. If PEBS enabled and a precise event is being measured then the registers here are the ones gathered by PEBS.

Note that the PEBS weight and data source data can be hard to interpret and often gives non-intuitive results, such as it reporting a cache miss taking more cycles to complete than an L3 cache miss. This is (at least in part) because the cycles count can take into account other things going on in the chip unrelated to the memory hierarchy.

A quick way to access the PEBS results with perf is:

```
perf mem record
perf mem report
```

Skylake Frontend:

```
perf record -e cpu/event=0xc6,umask=0x1,frontend=0x12/pp
```

Skylake TSC:

```
perf record -c 1000003 -e cpu/event=0xc0,umask=1/upp
perf script -F ip,time,
```

4.5 Low-skid Interrupts

The perf_event_open interface [16] supports various levels of low-skid measurements on an event. This is set by the `precise_ip` field, which is indicated by putting `:p` values on the end of events (`:p`, `:pp`, `:ppp`). Only a subset of events support precise reporting, and it varies by processor model.

The following precise settings are supported:

- Level 0 – an event can have arbitrary skid
- Level 1 – request constant skid
- Level 2 – request zero skid (but the processor might not always be able to deliver)
- Level 3 – require zero skid (or equivalent, such as "randomization to avoid shadowing effects").

Using low skid events can slow down your measurements (we have experimentally shown this as part of a cluster computing class project). The problem of skid is caused by out-of-order processors having so many instructions in flight that it would severely complicate the processor to try to track the exact instruction causing a perf interrupt, especially since it's somewhat unusual to have them enabled. So what the chip apparently does is monitor the overflow counter, and when it gets close to overflowing it slows down the processor (possibly reducing the number of instructions in flight simultaneously) until the event actually triggers, making it easy to target the exact instruction.

To use a precise event under perf you can do something like this:
`perf stat -e instructions:p`
although it is of limited use for aggregate counts.

Usually you will use it in something like this:
`perf record -e instructions:pp /bin/ls`
followed by
`perf report`.

The `perf record` command sets up a buffer for PEBS to sample in, and the samples recorded should have lower skid than if you had used the event without the `:p` specifiers.

On Intel chips, PEBS support gives you level 1 of precise events, LBR and PEBS format v2 gives you level 2 (IP Fixup), and PEBS prec dist support gives you level 3.

Note Level 2 support uses the LBR to help adjust things, so if you are doing branch sampling you might not be able to get such support.

On AMD machines precise IP is supported through the IBS interface. Both Level 1 and Level 2 are supported. Only three events are supported, cpu-cycles, r076 (also cycles) and r0C1 (uops). Previously you possibly needed to specify you want to run system wide `-a` not just per-task to do this (which often requires root) but at least on a recent machine this does not seem to be necessary.
`perf stat -a -e cpu-cycles:p`

4.6 Last Branch Sampling

This info can be gathered with the raw perf_event PERF_SAMPLE_BRANCH_STACK option. Lists the last so many branches (16 on recent machines), address and target, and whether predicted.

Also which branches are recorded can be filtered. Some machines can do this in hardware, others the kernel does it in software for you.

See `perf record -b -e cpu/event=0xc0,umask=0x1/upp triad` `perf report --no-branch-stack`

4.7 Branch Trace Store

This has its own PMU driver and uses a special AUX area of the mmap buffer away from (and mostly independent from) the normal sample buffer.

It can return branches, their ip, their target, and whether they hit or miss.

```
perf record -e branches:u -c 1 -d
```

The following is supposed to print some of that info but it doesn't (?)

```
perf script -f ip,addr,sym,symoff,dso
```

4.8 Processor Trace

For more info see [17]. It uses the AUX mmap buffer just like BTS does.

```
perf record -e intel_pt//u
```

4.9 Memory Bandwidth Monitoring

In theory can do this:

```
perf stat -e intel_cqm/llc_local_bw/ -a
perf stat -e intel_cqm/llc_total_bw/ -a
```

but unclear if this has made it to mainline kernel yet? (check on that).

4.10 Offcore / Uncore / RAPL

perf_event supports measuring most of the new event types on chips, including offcore and uncore (measuring events off the main core, usually involving the memory hierarchy) as well as estimated power and others.

Often these events do not have interrupts dedicated too them (or if so, are limited or buggy in some way). So they cannot easily be used for sampling.

However the readings from the events themselves can be read at sample time.

5 Current PAPI

PAPI has various existing sampling interfaces.

5.1 Statistical Sampling

`PAPI_overflow()` will set up a routine that is run at overflow time.

```
int PAPI_overflow(int EventSet, int EventCode,
                  int threshold, int flags,
                  PAPI_overflow_handler_t handler);
```

When an overflow happens, a UNIX signal is sent which PAPI handles and then calls the user's handler.

```
typedef void (*PAPI_overflow_handler_t)
                  (int EventSet, void *address,
                  long long overflow_vector, void *context);
```

This returns the address that the overflow happened at (for profiling) as well as the CPU context which on Linux has the various register state as they were the last time they were in userspace (note that the overflow might have happened in kernel space, you need to do more advanced work in kernel to get the register state of the actual overflow in that case).

If there is no hardware overflow available, a software timer can be used instead.

PAPI uses the perf_event sampling interface, but triggers on every overflow, which can cause a lot of overhead. Also PAPI is the only known user of this interface, so it sometimes breaks and the linux-kernel developers do not always notice right away.

5.1.1 Underlying Interface

The user calls `PAPI_overflow()`.
papi.c

- Parameter sanity checking is conducted.
- If `PAPI_OVERFLOW_FORCE_SW` is set, or the hardware does not support hardware overflow, then this is simulated using a timer.
- If threshold is 0, it means disable overflow
- Otherwise initialize the event to take overflows
- Call the set_overflow() function of the proper PAPI component
  components/perf_event/perf_event.c
  - Set up attr structure for event to be a sampled event
  - Call `_papi_hwi_start_signal()` to setup signal handler
  - Call `_pe_update_control_state` to tear down and restart all of the events to reflect the updated attr fields. It calls open_pe_events() which will call tune_up_fd() if it is a sampling event (tune_up_fd() allocates the MMAP sampling buffer).

5.2 Low-latency Sampling

PAPI currently does not support using the multiple samples per interrupt functionality provided by Linux and by PEBS.

5.3 Extended Sample Data

When using `PAPI_overflow()` PAPI does not support returning info besides the instruction pointer, although in theory the register state can also be manually gathered from the signal context on Linux.

Currently it is not possible to get the advanced sample info (kernel register state, latencies, branch predictor outcome, cache hierarchy extra info, etc.)

5.4 Profile Interface

5.4.1 PAPI_profil()

This interface is meant to be identical to the UNIX "profil" system call.

`PAPI_profil()`
`PAPI_sprofil()`

```
int PAPI_sprofil( PAPI_sprofil_t *prof, int profcnt,
        int EventSet, int EventCode, int threshold, int flags );
```

A range of addresses to watch is given, and then there is a regular overflow which stops, notes the instruction pointer, and then increments the value in a set of "bins". This can be used to generate a profile of where the code has been executing.

As far as I can tell this interface is not widely used, `PAPI_overflow()` is much more popular.

Internally this interface uses the `PAPI_overflow()` interface.

5.5 Low-skid Interrupts

In theory PAPI can use these events if you use `PAPI_add_named_event()` and use a proper event with one of the `:p` qualifiers tagged on the end.

Note and TODO: I have not tried this recently.

5.5.1 Last Branch Sampling

PAPI has no support for this currently.

5.5.2 Processor Trace

PAPI has no support for this currently.

6 Proposed PAPI Interface Concerns

We plan to enhance PAPI to have a better sampling interface.

6.1 Sampled Profiling

PAPI currently supports this, so most of the future work is how to extend this without breaking backward compatibility.

6.2 Low-latency Sampling

This mostly means queuing up multiple events at once without triggering an interrupt or signal. This could be easy to do, modify the PAPI_overflow() interface so it only overflows after a certain number of overflows, and then it's up to the handler to read multiple overflow values.

6.3 Extended Sample Data

Many users are interested in getting extended sample data that modern hardware preserves. This includes register state, cache miss info, instruction latencies, etc.

It is possible in software currently to get the IP and register state from the signal overflow context, however that only gives the *user* context (i.e. the last executing userspace code before the interrupt happened). If the overflow actually happened in the operating system you need the help of perf to return those values.

6.3.1 numap Proposal

See the info on numap in Section 3.2.1.

6.3.2 XSEDE'15 Paper Proposal

The paper by Lopez et al. [18] discussed a new interface that would allow gathering the extended info on a sample. It is a thin interface over the perf_event PEBS support. The results returned are similar to those from the numap proposal.

```
int PAPI_sample_init(int EventSet, int EventCode,
                int sample_type, int sample_period,
                int threshold,
                PAPI_sample_handler_t handler);

typedef void PAPI_sample_handler_t(int signum,
                siginfo_t *info,void *ucontext);
```

The sample_type passed in would just be the values the raw Linux interface is expecting. The result would be a binary blob of encoded perf records. Of most interest to NUMA profiling is what Linux currently returns for PEBS data: instruction pointer of the instruction, address of the load/store, "weight" which is the number of cycles, and data_src which is the part of the hierarchy causing the result. (Note the previously described limitations with these results).

Limitations of this proposed interface is how Linux-centric it is (PAPI is in theory supposed to be platform agnostic), as well as requiring the user to have a perf_event record decoder (a non-trivial piece of code) either included in PAPI or else hand constructed.

Another concern is how to remain forward compatible, as Intel adds more features to PEBS how can we return those too without requiring tools to be recompiled.

In addition the interface is Intel specific, for example AMD IBS results cannot be returned via the interface as it stands. Should PAPI have a layer of indirection that provides a more generic interface on top? Or just report raw perf_event data?

6.4 Low-skid Interrupts

Currently PAPI relies on libpfm4's perf_event parsing to handle the `:p` precise event notation.

We could add infrastructure to handle precise event information in the traditional PAPI way of `PAPI_set_opt()` but it is likely no one would want to use it as it requires an extra 2-3 calls for each event you add.

The other solution is just deprecate the `PAPI_set_opt()` calls and enforce all option setting by event name strings (the various :u and :k for setting domain have already started us down this path). This is fairly easy to do on Linux perf_event systems but it would mean extra custom string parsing on all non-Linux platforms (are they relevant anymore?)

6.5 Last Branch Sampling

Should we add a PAPI interface to return call chains? This might require linking against a library that handles printing backtraces.

6.6 Processor Trace

Again this might be useful, but the resulting data would likely be a perf-specific binary blob, and the main problem would be somehow translating this to what the user wants.

6.7 Other Concerns

6.7.1 System-Wide Profiling

Currently PAPI has a very process and thread-specific profiling mode.

Often it would be more interesting to have a system-wide view of what is going on, especially if one is worrying about NUMA performance.

By its nature the perf_event sampling interface is per-processor (need to open one buffer per core) so it can handle this, but how to express this back to the PAPI code?

6.7.2 Real Time or Later Analysis

Another thing to consider: will most users be trying to analyze their code while the program is running? Or will they be storing it to disk for later analysis?

If in real time, PAPI must provide libraries for dumping perf_data, generalizing it, and providing a fixed format that can be handled on the fly.

If the users really just want to store raw perf_event data (but self-monitored, unlike perf's full-program granularity) for later analysis it simplifies things as a separate suite of tools can be developed for use independent of the core PAPI library.

7 Proposed PAPI Interface

7.1 Thin perf_event wrapper, write direct to file

This would be the simplest interface for the user.

```
int PAPI_sample_init(
                int EventSet,
                int EventCode,
                int sample_type,
                int sample_period,
                char filename);
```

You would use an existing eventset, pick which event to sample on, pick a sample_type that is component specific (so on perf_event would just be the types perf_event supports), a sample frequency, and the name of a file to dump everything to.

Values could not be easily read on-the-fly.

Once the program is done it is up to the user to read the file and interpret the values (PAPI could provide sample routines to help do this).

The data could be written out in the existing `perf.data` format that the perf utility uses [19, 20, 21].

In theory various existing tools can parse the raw `perf.data` files:

- pmu-tools parser `https://github.com/andikleen/pmu-tools`
- quipper C++ parser, part of chromiumos-wide-profiling
- gooda `https://github.com/David-Levinthal/gooda`
- flame graphs `http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html`

7.2 Thin perf_event wrapper, user gets notice of data

This proposal is the one from the XSEDE paper.

```
int PAPI_sample_init(
                int EventSet,
                int EventCode,
                int sample_type,
                int sample_period,
                int threshold,
                void *buffer,
                int buffer_size,
                PAPI_sample_full_callback handler);

typedef void PAPI_sample_full_callback(void);
```

Again the sample_type is just a component specific value, for perf_event just the ones Linux supports.

In this case the user creates a memory buffer and size. When the Linux MMAP sample buffer is full, it will copy the results to this buffer and call the sample handler callback. It's up to the user to do something with the data (write it to disk, parse it, etc.)

The user can also set a "threshold" value so that with a value of 1 it is called each sample, or else it can be set higher and only be notified of every X samples.

This is much more flexible for live self-monitoring, at the expense of a much more complex interface.

7.3 Others?

Most other likely suggestion is something that abstracts things more, something like instead of a raw memory buffer PAPI provides a structure

```
PAPI_sample_info_t {
        long addr;
        long regs[32];
        long effective_address;
        int access_type;
        int latency;
};
```

and this gets filled in, ideally in a cross-platform manner. The problem with this is while useful for NUMA profiling, it ignores most of the other interesting data fields that perf_event is capable of providing.

8 Conclusion

New hardware has some pretty impressive advanced sampling hardware, however creating a generic software interface for it remains difficult. PAPI users would like access to the newer data, so we need to come up with a useful way to do this that does not break too many existing programs.

9 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. SSI-1450122.

References

[1] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. Department of Defense HPCMP User Group Conference*, June 1999.

[2] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, June 2015.

[3] K. Juvva, "Memory bandwidth monitoring in linux for HPC applications," in *Linux Con North America 2015*, Aug. 2015.

[4] A. Kleen and B. Strong, "Intel®processor trace on Linux," in *Tracing Summit 2015*, 2015.

[5] Advanced Micro Devices, *BIOS and Kernel Developers Guide (BKDG) For AMD Family 15h Models 00h-0Fh Processors*, Jan. 2013.

[6] Advanced Micro Devices, *BIOS and Kernel Developers Guide (BKDG) For AMD Family 15h Models 30h-3Fh Processors*, Mar. 2014.

[7] P. Drongowski, *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. Advanced Micro Devices, Inc., 2007.

[8] P. Drongowski, L. Yu, F. Swehosky, S. Suthikulpanit, and R. Richter, "Incorporating instruction-based sampling into AMD CodeAnalyst," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 119–120, Mar. 2010.

[9] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 89–100, June 2007.

[10] M. Selva, L. Morel, and K. Marquet, "numap: A portable library for low level memory profiling," Tech. Rep. RR-8879, INRIA, Mar. 2016.

[11] C. McCurdy and J. Vetter, "Finding and fixing numa-related performance problems on multi-core platforms," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 87–96, Mar. 2010.

[12] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for NUMA multicore systems," in *USENIX Annual Technical Conference*, June 2012.

[13] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multi-threaded programs on NUMA architectures," in *Proc. of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 259–272, Feb. 2014.

[14] S. Ragate, "GPU PC sampling utility," tech. rep., Innovative Computing Lab, University of Tennessee, 2015.

[15] S. Eranian, "Linux perf_events status update," in *Scalable Tools Workshop*, Aug. 2016.

[16] V. Weaver, "perf_event_open manual page," in *Linux Programmer's Manual* (M. Kerrisk, ed.), Dec. 2013.

[17] A. Kleen, "Adding processor trace support to Linux," *Linux Weekly News*, July 2015.

[18] I. Lopez, S. Moore, and V. Weaver, "A prototype sampling interface for PAPI," in *Extreme Science Engineering Discovery Environment Conference*, July 2015.

[19] U. Fässler and A. Nowak, "perf file format," tech. rep., CERN Openlab, Sept. 2011.

[20] J. Olsa, "perf & CTF," in *Tracing Summit 2014*, 2014.

[21] A. Kleen, "perf.data file format specification draft." https://lwn.net/Articles/644919/, 2015.