Advanced Event-Sampling Support for PAPI

Forrest Smith and Vincent M. Weaver(✉)

University of Maine, Orono, ME 04469, USA
{forrest.smith,vincent.weaver}@maine.edu

Abstract. The PAPI performance library is a widely used tool for gathering performance data from running applications. Modern processors support advanced sampling interfaces, such as Intel's Precise Event Based Sampling (PEBS) and AMD's Instruction Based Sampling (IBS). The current PAPI sampling interface predates the existence of these interfaces and only provides simple instruction-pointer based samples.

We propose a new, improved, sampling interface that provides support for the extended sampling information available on modern hardware. We extend the PAPI interface to add a new `PAPI_sample_init` call that uses the Linux `perf_event` interface to access the extra sample information. A pointer to these samples is returned to the user, who can either decode them on the fly, or write them to disk for later analysis.

By providing extended sampling information, this new PAPI interface allows advanced performance analysis and optimization that was previously not possible. This will greatly enhance the ability to optimize software in modern extreme-scale programming environments.

1 Introduction

When conducting performance analysis, the easiest type of data to collect is total, aggregate results. This includes information such as the total number of cycles a program ran, the total number of cache misses that occurred, and the total wall clock time. While all of this information is of interest, often more detail is wanted: *what* function takes the most cycles, *which* data structure causes the cache misses, *why* is the code taking so long to run.

The most straightforward way to get this detailed information is via *sampling*; to periodically interrupt the program's execution and gather machine state about what it is happening at the time of the interruption. Overall program behavior can be extrapolated based on these representative samples. There is a tradeoff between overhead and accuracy: a higher sample rate leads to more accurate results, but if you sample *too* frequently you will add overhead that can interfere with the results being measured. Some of this overhead can be mitigated if the sampling is done in hardware rather than in software.

A. Bhatele et al. (Eds.): ESPT/VPA 2017/2018, LNCS 11027, pp. 144–161, 2019.
https://doi.org/10.1007/978-3-030-17872-7_9

1.1 Hardware Performance Counters

Most modern processors support hardware performance counters; these counters are internal to the system and increment when certain architectural events occur. Total aggregate counts can be gathered by starting the counters at the beginning of the code of interest, and stopping them afterward. Traditionally these counters are found in CPUs, but their use has expanded to other pieces of hardware such as the disk, network, and memory systems.

Typically there are only a handful of counters available, often in the range from two to eight (though this varies by vendor, architecture, and processor generation). The counters are typically between 32 and 64 bits in size. Each counter can measure an event, chosen from a large list (potentially hundreds on some architectures [4,13]).

Usually the counters can be configured to trigger a hardware interrupt if the register overflows. This can be used to notice and account for large counts generated by frequent events; if the counter overflows multiple times between readings it would not be possible to determine the exact count. The overflow mechanism is also useful for sampling. An event can be set to overflow periodically, for example, every 100,000 cycles. Once the interrupt triggers, the operating system interrupt handler takes over and can construct a sample that includes additional useful information, such as where the instruction pointer is currently located. If your CPU lacks performance counter overflow interrupt support, sampling can still be done by using some other regular interrupt source (such as a periodic timer). However usually the performance counters are used for this purpose if they are available.

1.2 Advanced Sampling

While you can learn a lot about a program by gathering instruction pointer samples, there is a lot more to program behavior than just instruction traces. Recent processors from Intel and AMD support more advanced sampling modes. These allow gathering extra information on an overflow, such as detailed cache miss and cache latency values.

The sampling features are grouped together under a large number of processor features with sometimes confusing acronyms. The more well known are Intel's Precise Event Based Sampling (PEBS) and AMD's Instruction Based Sampling (IBS). There are a few common sampling related interfaces:

- **Sampled Profiling** – traditional sampling, as defined previously. A periodic interrupt is used to sample the instruction pointer and any other info that can be easily obtained, such as register values. Most CPUs can do this purely in 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 Intel PEBS do this.
- **Extra CPU State** – PEBS and IBS log additional CPU state that cannot be obtained in software from the operating system. 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 the skid). PEBS and IBS provide support for low-skid sampling, at the expense of some additional time overhead.
- **Last Branch Sampling** – The hardware keeps track of the last branches taken, and allows generating call stacks. Intel Last Branch Record (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.

Ideally all of these types of sample data could be easily returned to the user through a straightforward interface.

1.3 Software Interfaces

Hardware counter accesses are privileged by the hardware, so usually the operating system is responsible for enforcing access. On Linux this is done by the `perf_event` [10] subsystem. Over the years Linux has gradually added support for the more advanced sampling modes. Directly accessing these results from userspace involves using the `perf_event` interface which is complicated to set up and use [33]. Most users instead opt to use the `perf` command-line tool which abstracts away some of the low-level interface.

PAPI [26] is a portable, cross-platform library for accessing hardware performance counters. Many higher-level tools, such as VAMPIR [19] and HPCToolkit [1] build on PAPI. PAPI has supported simple event sampling for a long time, but has lacked the ability to gather advanced samples from modern processors. In this paper we describe the existing PAPI support for sampling, and how we plan to add support for the more advanced hardware sampling interfaces supported by perf_event.

2 Hardware Sampling Interfaces

As with general performance counter support, sampling interfaces are not part of any x86 standard and thus have completely different implementations between vendors. What follows is a quick overview of support found on recent processors.

2.1 Intel x86_64

Intel processors introduced performance counter support with the original Pentium processor. Since the beginning they have supported hardware interrupt on counter overflow, allowing sampled profiling. More advanced sampling interfaces began appearing starting with the Pentium 4 processor.

Intel 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) [13]. PEBS support originated in Pentium 4 and Core architectures. It is available on all subsequent Core-derived processors as well as some Atom models.

Only a subset of events can be used as PEBS events, and sometimes only a certain counter slot can be used. A suitable Data Store (DS) area must be set up in memory; samples will be directly written to this area without any operating system involvement. When PEBS is enabled for an event, the PEBS circuitry is armed when the counter overflows. The next instruction that triggers this event had a record with sample information written out to the DS area. The DS area can be configured to generate an interrupt when full (or nearing being full) so that multiple samples worth of data can be queued up and processed at once by the operating system, reducing overall overhead.

The information that can be recorded on a PEBS sample varies by architecture but can include:

- trap vs fault (whether the event recorded is the next or the current one),
- a full set of processor registers (in addition to the instruction pointer),
- store latency data,
- transactional memory data
- TSC value, and
- the counter value.

Nehalem processors add more features. Now you can record load latency information: the latency in cycles from first dispatch to final retirement of the instruction. When enabled, load instructions are randomly chosen to accumulate the load latency info. The value recorded is the latency for the last randomly tagged event, not necessarily the one that triggered the PEBS operation. The information gathered includes the Data Linear Address (usually the same as the virtual address of value being loaded), latency value, and data source (which indicates what part of the cache memory hierarchy was involved with returning the loaded value).

Sandybridge processors add more PEBS features, and enable PEBS for more events. In addition to loads, now store instructions can also be measured (but this is limited in some ways, including not being able to get latency values). Additional info is returned on whether loads hit in the TLB. Precise store support is added, where information is returned on the very next store rather than a randomly selected one. Sandybridge also adds support for low-skid measurement

via the Precise Distribution of Instructions Retired (PDIR) interface. It notices when an overflow interrupt is about to happen and prepares for it and enters a slower high-accuracy mode that allows it to exactly determine which instruction caused the overflow.

With Haswell precise store was replaced by Data Linear Address Profiling (DataLA); the full linear (virtual) destination address of the load or store is stored in the sample. Additionally information on whether the access hit in the closest level of cache is stored. The eventing instruction pointer (the address of the instruction that caused PEBS to trigger) is also recorded. Finally, various transactional memory related sample types were added.

Skylake processors add a field for recording the TSC timestamp value from when that event occurred, and adds additional front-end events (iTLB and iCache misses).

PEBS support was originally designed for desktop and server chips, but some of the Atom class chips also have support for PEBS. On Goldmont Atom chips, PEBS records can be recorded for all events. However for non-precise events there is no guarantee about what instruction actually generates the sample. Other information recorded includes the time stamp counter (TSC) and info on which event caused the overflow (if multiple are enabled). Reduced skid and linear address support is also available.

Intel Last Branch Record (LBR). Starting with the Pentium 4 most Intel hardware supports logging a trace of the last branches that were executed via the Last Branch Record (LBR) interface. Full details can be found in Chapter 17 of the Vol3b documentation. The number of branches recorded varies from 4 up to 32. The LBR record contains detailed information about the branch, such as the last location branched from, the last location branched to, and whether the branch was predicted correctly or not. This is not strictly a sampling feature, but the data is recorded to MSR registers and under Linux is reported via the perf_event interface.

Intel Branch Trace Store (BTS). Intel processors can also support the Branch Trace Store (BTS), where the last N branch records can be written out to a circular buffer called the Debug Store (DS) which should not be confused with the PEBS Data Store. This feature lets you track the branch behavior of your program, but is known to slow down program execution when enabled.

Nehalem chips added the ability to filter based on branch type. 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. It has 32 entries now and can capture length of time spent in a basic block with the TSC time. Atom Goldmont allows you to obtain the number of cycles since last branch.

Intel Resource Director Technology (RDT). The Intel Resource Director Technology (RDT) is available on server machines, Haswell Xeon E5 v3 and newer. It supports a number of technologies. Cache Monitoring Technology (CMT) can measure cache occupancy of program in last level cache. Memory bandwidth monitoring (MBM) [14] can monitor memory bandwidth between cache levels. You can assign a resource monitoring ID (RMID) to a task, processor, or group of processors and monitor them.

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.

Some machines have Cache Quality-of-service Monitoring (CQM) but it is not documented, and while Linux has some initial support for it, it was later removed.

Intel Processor Trace (PT). Intel Processor Trace (PT) [18] lets you record program execution traces. The first implementation is control flow tracing and can log enough information to give an exact program flow trace. It can also generate basic block vectors and trace power events. It aims for less than 5% overhead, and records latency info. It can reconstruct program flow by recording the taken/not-taken path of conditional branches. There is a possibly related technology called Intel Architectural Event Trace (AET) but information on how to use this is not publicly released.

2.2 AMD x86_64

AMD processors support simple sampling using hardware interrupts on counter overflow. Recent processors also support some more advanced sampling interfaces, but not quite as many nor as varied as supported by Intel.

AMD Instruction Based Sampling (IBS). AMD chips support Instruction Based Sampling (IBS), which is described in the various BIOS and Kernel Development Guides [2,3] as well as in some research papers [6,7].

IBS was introduced with Barcelona (fam10h) to aid in creating low-skid profiles. It selects a random instruction or micro operation (uop) and records information, generating an interrupt when completed. There are two types of sample: one that happens on instruction fetch (involving TLB and instruction cache behavior) and one that happens on instruction execution.

For instruction fetch the following information is logged:

- if the fetch was completed or aborted,
- number of cycles spent on the fetch,
- if the fetch hit in the caches and TLB, and
- the linear/physical address corresponding to the fetch.

For instruction execution the following is logged:

– if 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,
– data cache statistics (hit or not, latency),
– clocks from tag until retire,
– clocks from execution until retire, and
– DRAM and MMIO source info.

Unlike PEBS these values aren't stored in a memory buffer, but in a set of MSRs. Only one record can be pending at a time. Only three events are supported: cycles, cycles:p, and uops.

2.3 Other Processors

Most other modern processors support performance counters, and again most of these support simple sampling via counter-overflow interrupt (although notably various ARM based platforms might not, such as the original ARM1176 Raspberry Pi systems).

Support for more advanced sampling is not as widespread as it is on x86. ARM has no PEBS or IBS equivalent, but it does have something similar to Processor Trace called CoreSight. Newer 64-bit ARM models optionally support the Statistical Profiling Extension (SPE) [5]; perf_event added support for this with Linux 4.15.

The IBM s390 class of machines has a sampling facility as part of the CPU Measurement Facility [12] that will write samples into a buffer that will trigger an interrupt when full.

3 Software Interface for Sampling

Advanced hardware sampling interfaces are complex and vendor specific. Some of this complexity can be abstracted away by the operating system (in our case we will assume the OS is Linux). On Linux the `perf_event` interface used for accessing regular hardware performance counters is also used for accessing sample data. This interface itself is complex and hard to use, so we develop the PAPI library which is yet another layer of abstraction on top of `perf_event`. This allows existing users of PAPI to gain access to the sampling interface using familiar PAPI interfaces, without needing to majorly restructure their code.

3.1 Linux perf_event Interface

Access to performance counter registers requires supervisor or privileged access to the hardware, in order to initialize the model-specific registers (MSRs) and set up the sampling memory buffers. Because of this the operating system is usually responsible for the interface. In addition access to the underlying hardware might

be further restricted for security reasons. A clever user can monitor in detail what a system is doing based on fine grained performance information, and this can leak information. This was once considered a mostly theoretical attack, such as being able to reverse engineer encryption happening on other cores by monitoring cycle or cache miss counts; this has recently become a much more critical worry with the advent of the Meltdown and Spectre vulnerabilities [22].

The standard performance counter interface provided by Linux is known as `perf_event` and the primary way of accessing it is the `perf_event_open()` system call [33]. This system call is used to configure and open a performance counter event; it is a complex call with over forty interacting parameters. The system call returns a file descriptor which can be used to control and access the event. Values can be read with the `read()` system call, and memory can be set up with `mmap()` that allows both sampling to a circular buffer as well as gathering additional information about the event. Various `ioctl()` calls are used to start and stop the events. Advanced features, such as event scheduling, event multiplexing, and save/restore on context switch are all provided by the interface.

Linux perf_event supports most of the advanced hardware sampling interfaces described in Sect. 2.

perf_event Sampled Profiling. As long as your system supports overflow interrupts you can do statistical sampling with perf_event. You can specify the event, the frequency, and a whole host of other options. On overflow, a user-specified signal handler can be called that your code can use to find the register state, including instruction pointer location.

perf_event Low-Latency Sampling. The perf_event interface can provide access to low-latency sampling, which is gathering multiple samples into a buffer without program intervention. The samples are gathered until a watermark threshold is crossed, and only then will your program be interrupted to let it know that the buffer is full and ready to be processed. There is still some operating system overhead involved, as some events need to be handled in the kernel even if userspace code is not bothered. When using an interface such as Intel PEBS even this can be avoided, as the hardware can store PEBS records to a memory buffer directly without any operating system involvement at all.

By default perf_event does not support low-latency sampling, and instead runs in "single-entry" mode. This is because the perf records require some values that only the OS can provide, such as pid/tid. It is possible to enable the N-entry PEBS mode if you are willing to sacrifice some features: you must use a fixed period, no timestamp if pre-Skylake, the PEBS buffer flushed on context-switches, and no LBR [8].

perf_event 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
- PERF_SAMPLE_READ – counts for all events in group
- PERF_SAMPLE_CALLCHAIN – callchain info
- 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 (useful 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.

perf_event Low-Skid Interrupts. The perf_event interface supports various levels of low-skid measurements on an event. This is enabled via the *precise_ip* field, which is indicated in both perf and PAPI 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").

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 precise distribution support gives you level 3. Note Level 2 support uses the LBR for accuracy, so it might not be able if you are also attempting to use LBR for branch sampling.

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, cycles, and uops. Previously you needed to specify you want to run system wide `-a` not just per-task to do this (which often requires root) but on a recent machines this is no longer necessary.

perf_evet Branch Sampling. This info can be gathered with the raw perf_event PERF_SAMPLE_BRANCH_STACK option. It can report the last N branches (16 on recent machines), the address and target, and whether it was properly predicted. On some machines you can filter by branch type.

The related Branch Trace Store functionality has its own PMU driver and uses a special AUX area of the mmap buffer which is mostly independent from the normal sample buffer. It can return branches, their ip, their target, and whether they were a branch hit or miss.

Other, Non-sampling Interfaces. Intel Processor Trace is a whole tracing subsystem, and does much more than sampling [16]. It uses the AUX mmap buffer just like BTS does.

3.2 PAPI Library Interface

The PAPI performance library [26] is a cross-platform library designed to allow access to performance counters on a wide variety of machines. On current Linux machines PAPI uses the perf_event interface. We will briefly describe the old sampling methods available in PAPI prior to the forthcoming 6.0 release expected in 2019.

PAPI Statistical Sampling. The current PAPI interface used when sampling is `PAPI_overflow()`. There are two key parameters: an overflow threshold and a signal handler. Once the event in question hits the threshold, the hardware triggers an overflow interrupt which is then passed by the operating system to the Linux system handler. It is up to the user to do something useful in the signal handler (such as read out the instruction pointer value) before returning. 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.)

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

The signal handler looks like:

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

PAPI_profil(). There are two legacy PAPI sampling interfaces, `PAPI_profil()` and `PAPI_sprofil()`, which are meant to provide interfaces compatible with the UNIX "profil" system call.

```
int PAPI_profil(void *buf, unsigned bufsiz,
        caddr_t offset, unsigned scale,
        int EventSet, int EventCode,
        int threshold, int flags);

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. This interface is not as widely used as the much more popular `PAPI_overflow()`.

PAPI Low-Skid Interrupts. PAPI currently support perf_event low skid interrupts. To do this you use the `PAPI_add_named_event()` interface and when specifying the event name include one of the `:p` suffixes to indicate you want a more precise event.

4 Related Work

Other interfaces besides PAPI offer ways to read hardware performance counters. Many of these interfaces also support sampling.

4.1 Existing Profiling Tools

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 a timer that triggers every 10 ms. PAPI has existing code to emulate this interface. While profiles can be generated, no advanced sampling information is available.

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

Valgrind. Valgrind [27] 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.

4.2 NUMA Profiling

numap [31] presents an API for gathering sampled data for use when analyzing NUMA systems. First `init_samp_session()` is 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. The primary downside of this, at least to PAPI users, is that it is a separate tool and not integrated into the PAPI interface.

Memphis [25] is a tool that talks to the AMD IBS registers directly via a kernel module in order to gather the extended sample information. MemProf [20] is another AMD IBS-based NUMA memory profiler. Again, neither of these is integrated into the PAPI infrastructure.

4.3 GPU Profiling

Some GPU hardware supports profiling interfaces too, specifically recent NVIDIA devices [30]. For MAXWELL GPUs and CUDA 7.5 you can use CUPTI to 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. While useful for analyzing GPU code, in our work we are more concerned with the advanced sampling interfaces provided on modern CPUs.

4.4 Other Tools with Sampling Interfaces

LIKWID [32] is a hardware performance measurement interface that is capable of reading performance counters on supported x86 processors. Using the likwid-perfctr command with the -t option, the user can measure performance results

from LIKWID at a specified time interval. The interface recommends using an interval no smaller than 100 ms, otherwise the results are considered invalid. Achieving fine-grained sampling results from the LIKWID interface is not possible due to this constraint. LIKWID does not support PEBS as it is a userspace tool and cannot setup the kernel buffers needed to hold the PEBS records.

HPCToolkit [1] is a large suite of tools for analyzing the performance of multithreaded applications. It can be used for anything from a home computer to a super computer. HPCToolkit interfaces directly with PAPI to read hardware performance counters and gather samples. The samples do not contain the extra data that is available from PEBS events; they are merely counter readings using the `PAPI_overflow()` code.

4.5 Other Proposed PAPI Sampling Interfaces

Lopez, Moore, and Weaver [24] were the first to propose an enhanced sampling interface for PAPI that gathered the PEBS cache latency values. Their sampling interface is similar to the one that we propose in this paper. Their proposed interface was never implemented and remained a proof of concept. They used raw perf_event calls to show it was possible to measure both single thread and multithreaded applications. They were successfully able to gather STREAM sample results using OpenMP with eight threads.

5 Proposed Advanced PAPI Sampling API

It is not possible to retrofit the existing `PAPI_overflow()` method of gathering samples to handle extended sample information in a backwards compatible way.

We propose two new enhanced interfaces. One stays true to the historical cross-platform layer-of-abstraction nature of PAPI, but only provides limited information. The other acts as a thin layer on top of the perf_event interface that provides all sampling info, but is very Linux specific.

5.1 Abstracted Interface

This interface attempts to provide access specifically to the cache latency values that can be found in PAPI. This is the most requested feature, and in theory can be made cross-platform although currently only Intel PEBS provides this information.

This interface involves a `PAPI_sample_init()` call shown in Fig. 1 which internally inside of PAPI will take the event selected and set up a sampling buffer. Once the buffer is full, PAPI will gather the data and create an array of sample data that will be passed back to the user.

```
struct sample_struct {
        uint64_t        type;
        uint64_t        instruction_address;
        uint64_t        memory_access_address;
        uint64_t        cache_access_type;
        uint64_t        latency;
};

int   PAPI_sample_init(int EventSet, int EventCode,
        long long sample_period, long long buffer_size,
        PAPI_overflow_handler_t handler);

typedef void (*PAPI_overflow_handler_t)
        (int EventSet, void *sample_struct,
        long long num_samples);
```

Fig. 1. Proposed abstract interface

5.2 Direct perf_event Interface

This option for the interface does not try to abstract away the samples. It operates on the assumption that most HPC work happens on Linux kernels via the perf_event interface and as much information as possible provided by this interface should be passed back to PAPI if requested. While this is the most powerful interface, it requires a lot of internal `perf_event` knowledge. The example interface is shown in Fig. 2.

```
int   PAPI_sample_init(int EventSet, int EventCode,
        long long sample_period, long long buffer_size,
        struct perf_event_attr *attr,
        PAPI_overflow_handler_t handler);

typedef void (*PAPI_sample_handler_t)
        (int EventSet, void *buffer_address);
```

Fig. 2. Proposed perf_event interface

This interface provides a pointer to the raw perf_event mmap() sample buffer, and it is up to the user code to interpret this and get the samples out. For performance reasons, the Linux kernel enforces a rule that to gather PEBS-type sample data, each individual core needs to have its own `mmap()` buffer. Currently it is up to the user to open one event per core as needed, but we are planning an interface to simplify this.

Existing PAPI code using `PAPI_overflow()` can be used with few changes. You still need to create an eventset, add an event (note: only some events are capable of providing extra sampling information). Then initialize sampling using the proposed interface. Finally, start/stop events as per normal.

When a threshold is crossed and a sample is gathered, PAPI will activate the signal handler that was set up by the user. It is then up to the user to access the `mmap()` buffer and do something useful with the contents before returning.

In PAPI we provide two sample programs: one writes out the raw sample data to disk for later analysis, and one that prints out the sample results on the fly.

The low level changes required to PAPI are mostly about making sure the mmap() buffers get set up properly. A lot of the hard work involving internal PAPI management of mmap() buffers was already done when fast rdpmc read support was added [23]. The PAPI code manages setting up the mmap()s and making sure that the events are opened properly.

The types of sample information available can be found in the perf_event_open manpage [33]. For PEBS latency information use one specifies `PERF_SAMPLE_IP | PERF_SAMPLE_WEIGHT | PERF_SAMPLE_DATA_SRC` which asks for the instruction pointer, the weight (latency) and the source of data. For IBS events one would ask for PERF_SAMPLE_RAW and you have to parse the IBS values yourself.

Limitations of this Interface. The primary limitation of this proposed interface is how Linux-centric it is. PAPI is in theory supposed to be platform agnostic. In addition the samples are in the raw perf_event sample record format which requires the users to have some non-trivial code to decode the results.

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.

Unsupported Sample Types. The perf_event interface returns most sample data through the mmap() interface, so anything supported by perf_event can be gathered that way. This means results such as LBR records can also be obtained through this interface.

Some values, such as Intel BTS and Intel Processor Trace, use an additional, auxiliary, mmap() buffer to store the results. PAPI currently does not support gathering data via that mechanism.

Data Format. Once the user signal handler is called, the program can read out the samples in the mmap() buffer and interpret them. There are two straightforward ways to deal with the data. One is to immediately write it to disk, interpreting it offline. The other is to decode and act on the results immediately. Both methods of gathering data will require some sort of library to interpret the fields in the samples. We provide examples that do both types of analysis, but this code is currently not part of PAPI, but separate code to be included in the analysis routines.

For the save to disk case, there is a standard on-disk format for perf records, the a `perf.data` file [9,17,28]. Programs that write out data in this format can then be analyzed by other compatible perf tools. There are various existing tools that can parse raw `perf.data` files:

- pmu-tools parser [15],
- quipper C++ parser (part of chromiumos-wide-profiling),
- gooda [21], and
- flame graphs [11].

6 Preliminary Results

We have been developing the advanced PAPI sampling interface on a number of machines, with the primary testing happening on an Intel Skylake machine with four cores. The test machine is running Linux 4.4.0-127-generic and our primary benchmark is a PAPI instrumented version 2.2 of the High-Performance Linpack (HPL) benchmark [29]. Samples have been recorded and verified for all PEBS events in the Skylake, Haswell, and Broadwell architectures.

Figure 3 shows results gathered on a Skylake machine when using sample types `PERF_SAMPLE_IP`, `PERF_SAMPLE_READ`, and `PERF_SAMPLE_CPU`. The native `FRONTEND_RETIRED.L1I_MISS` event was used, which counts instruction cache misses. Each sample contains the value of the performance counter, which can be seen next to "Value:". The samples also record the CPU on which the event is occur ING and the instruction pointer at the time of the event. The samples were collected with a sample period of 10000. Two captured samples are shown; it is a multithreaded benchmark and it can be seen that the samples were gathered from two different cores. In this example, the raw data is gathered in a signal handler and this is parsed and printed each time a signal occurs.

Figure 4 shows results gathered on a Haswell machine that include cache latency and source results. These were gathered using the event:
`MEM_TRANS_RETIRED:LATENCY_ABOVE_THRESHOLD`
and the sample type:
`PERF_SAMPLE_IP | PERF_SAMPLE_WEIGHT | PERF_SAMPLE_DATA_SRC`.

```
PERF_RECORD_SAMPLE [91], MISC=2
(PERF_RECORD_MISC_USER), Size=64
PERF_SAMPLE_IP, IP: 7f9b5f1bc439
PERF_SAMPLE_CPU, cpu: 2  res 0
PERF_SAMPLE_READ, read_format
Number: 1
enabled: 4827080
running: 4827080
Value: 10000 id: 2084

PERF_RECORD_SAMPLE [91], MISC=2
(PERF_RECORD_MISC_USER), Size=64
PERF_SAMPLE_IP, IP: 7f9b6f03b7fc
PERF_SAMPLE_CPU, cpu: 7  res 0
PERF_SAMPLE_READ, read_format
Number: 1
enabled: 12203500
running: 4517409
Value: 10001 id: 2089
```

Fig. 3. Example advanced sampling, with IP, CPU, and READ samples shown.

```
PERF_RECORD_SAMPLE [c001],
MISC=16386
PERF_SAMPLE_IP, IP: 55fb7799a730
PERF_SAMPLE_WEIGHT, Weight: 48
PERF_SAMPLE_DATA_SRC,
Raw: 668100842
Load Hit L3 cache No snoop
Hit Level 1
TLB Level 2 TLB Hardware walker

PERF_RECORD_SAMPLE [c001],
MISC=16386
PERF_SAMPLE_IP, IP: 55fb7799a730
PERF_SAMPLE_WEIGHT, Weight: 67
PERF_SAMPLE_DATA_SRC,
Raw: 668100842
Load Hit L3 cache No snoop
Hit Level 1
TLB Level 2 TLB Hardware walker
```

Fig. 4. Example of advanced sampling, with IP, WEIGHT, and DATA_SRC samples. The weight indicates the latency of the sampled instruction.

7 Conclusion and Future Work

We have designed an improved sampling interface for PAPI. It integrates advanced sampling support into the PAPI interface while abstracting away some of the difficulty of using the perf_event_open sampling interface. We provide code that can be used to parse samples found in the mmap() buffer which is not a trivial task.

The interface is currently under test for architectures other than Broadwell, Haswell and Skylake. Once testing is completed, the interface will be included and released with the upcoming 6.0 PAPI release.

By adding extended sampling support to PAPI we have opened new avenues for code analysis that will greatly aid users trying to optimize for performance in current and future extreme-scale systems.

Acknowledgment. This work was supported by the National Science Foundation under Grant No. SSI-1450122.

References

1. Adhianto, L., et al.: HPCToolkit: tools for performance analysis of optimized parallel programs. Concurrency Comput.: Practice Exp. **22**(6), 685–701 (2010)
2. Advanced Micro Devices: BIOS and Kernel Developer's Guide (BKDG) For AMD Family 15h Models 00h–0Fh Processors, January 2013
3. Advanced Micro Devices: BIOS and Kernel Developer's Guide (BKDG) For AMD Family 15h Models 30h–3Fh Processors, March 2014
4. AMD: AMD Family 15h Processor BIOS and Kernel Developer Guide (2011)
5. ARM: ARM Architecture Reference Manual Supplement Statistical Profiling Extension, for ARMv8-A (2017)
6. Drongowski, P., Yu, L., Swehosky, F., Suthikulpanit, S., Richter, R.: Incorporating instruction-based sampling into AMD CodeAnalyst. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, pp. 119–120, March 2010
7. Drongowski, P.: Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. Advanced Micro Devices, Inc. (2007)
8. Eranian, S.: Linux perf_events status update. In: Scalable Tools Workshop, August 2016
9. Fässler, U., Nowak, A.: Perf file format. Technical report, CERN Openlab, September 2011
10. Gleixner, T., Molnar, I.: Performance counters for Linux (2009)
11. Gregg, B.: FlameGraphs. http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
12. IBM: Linux on Z and LinuxONE: Device Drivers, Features, and Commands (2018)
13. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide, June 2015
14. Juvva, K.: Memory bandwidth monitoring in Linux for HPC applications. In: Linux Con North America 2015, August 2015
15. Kleen, A.: Intel PMU profiling tools. https://github.com/andikleen/pmu-tools

16. Kleen, A.: Adding processor trace support to Linux. Linux Weekly News (2015). https://lwn.net/Articles/648154/
17. Kleen, A.: perf.data file format specification draft (2015). https://lwn.net/Articles/644919/
18. Kleen, A., Strong, B.: Intel®processor trace on Linux. In: Tracing Summit 2015 (2015)
19. Knüpfer, A., et al.: The Vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68564-7_9
20. Lachaize, R., Lepers, B., Quéma, V.: Memprof: a memory profiler for NUMA multicore systems. In: USENIX Annual Technical Conference, June 2012
21. Levinthal, D.: Gooda PMU event analysis package. https://github.com/David-Levinthal/gooda
22. Lipp, M., et al.: Meltdown. ArXiv e-prints, January 2018
23. Liu, Y., Weaver, V.: Enhancing PAPI with low-overhead rdpmc reads. In: Proceedings of the 6th Workshop on Extreme-Scale Programming Tools, November 2017
24. Lopez, I., Moore, S., Weaver, V.: A prototype sampling interface for PAPI. In: Extreme Science Engineering Discovery Environment Conference, July 2015
25. McCurdy, C., Vetter, J.: Finding and fixing NUMA-related performance problems on multi-core platforms. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, pp. 87–96, March 2010
26. Mucci, P.J., Browne, S., Deane, C., Ho, G.: PAPI: a portable interface to hardware performance counters. In: Proceedings of Department of Defense HPCMP User Group Conference, June 1999
27. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 89–100, June 2007
28. Olsa, J.: Perf & CTF. In: Tracing Summit 2014 (2014)
29. Petitet, A., Whaley, R., Dongarra, J., Cleary, A., Luszczek, P.: HPL – a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Innovative Computing Laboratory, Computer Science Department, University of Tennessee, v2.2, December 2017. http://www.netlib.org/benchmark/hpl/
30. Ragate, S.: GPU PC sampling utility. Technical report, Innovative Computing Lab, University of Tennessee (2015)
31. Selva, M., Morel, L., Marquet, K.: numap: a portable library for low level memory profiling. Technical report RR-8879, INRIA, March 2016
32. Treibig, J., Hager, G., Wellein, G.: LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures, September 2010
33. Weaver, V.: perf_event_open manual page. In: Kerrisk, M. (ed.) Linux Programmer's Manual, February 2018