A Prototype Sampling Interface for PAPI

Ivonne Lopez

University of Texas at El Paso
500 West University Avenue
El Paso, TX 79968
+1 915-747-5480

ielopez3@miners.utep.edu

Shirley Moore

University of Texas at El Paso
500 West University Avenue
El Paso, TX 79968
+1 915-747-5883

svmoore@utep.edu

Vincent Weaver

University of Maine
Electrical and Computer Engineering
Orono, Maine 04469
+1 207-581-2227

vincent.weaver@maine.edu

ABSTRACT

PAPI is a widely used portable library for accessing hardware counters on modern microprocessors. PAPI offers both counting and sampling interfaces, but the sampling interface is extremely limited, consisting of a simple interrupt-driven interface that can periodically report processor state. In the past few years, the hardware and operating systems of modern processors have added support for new more advanced sampling features. These features enable information about non-uniform memory access (NUMA) behavior to be obtained. Currently, performance tool developers who want to provide sampling data to their users must make use of a complex low-level kernel interface, sometimes developing their own kernel patch to access the features they need. This paper reports on initial efforts to develop a middleware layer that will serve as a stable interface and enable tool developers to access sampling data through standard PAPI calls and to obtain data important for NUMA analysis.

Categories and Subject Descriptors

D.3.3 [Performance of Systems]: Measurement techniques, performance attributes.

General Terms

Measurement, Performance.

Keywords

Hardware counters, sampling, non-uniform memory access (NUMA)

1. INTRODUCTION

PAPI is a widely used portable library for accessing hardware counters on modern microprocessors [7]. PAPI offers both counting and sampling interfaces, but the sampling interface is extremely limited, consisting of a simple interrupt-driven interface that can periodically report processor state. In the past few years, the hardware and operating systems of modern processors have added support for new more advanced sampling features. For memory access events, information such as access latency, operand address, and data source for loads can be returned. Such information is extremely useful for performance tuning on modern multicore systems. For example, sampling memory accesses that exceed a specified latency threshold and mapping these accesses

to the data addresses responsible can help code developers analyze and improve data layout on a non-uniform memory access (NUMA) system.

We have implemented a prototype low-level interface for precise sampling. The prototype is implemented as a thin layer over the Linux `perf_event` sampling interface [9] that is part of the Linux kernel. We present preliminary results from sampling cache and memory latency events for multithreaded codes. The prototype is a first step towards implementing an advanced sampling interface for PAPI. A PAPI sampling interface will serve as middleware that can be used by performance analysis tool developers to access the hardware performance information needed for cache and memory performance analysis.

2. BACKGROUND

Most modern microprocessors contain performance counters, architectural registers that allow low-level analysis of program execution. Performance counters can be used in two ways: for aggregate measurement and for statistical sampling.

With aggregate measurement, or counting, the counters are enabled just before a program starts running (or the part of the program that is of interest), and total results are gathered when it finishes. The operating system aids measurement by saving and restoring values on context switch and also handling overflows where a value exceeds the counter width. This methodology provides total measurements, showing high-level overviews of program behavior. For detailed per-function or per-instruction results other methods must be used.

With statistical sampling, the counters are programmed to periodically gather measurements at regular intervals via the generation of timer or overflow interrupts. By recording the instruction pointer at time of sampling, function-granularity performance results can be extrapolated statistically. This has the advantage that the program does not have to be modified with calls to measurement routines, but has the disadvantage that the values are not exact and resolution can be coarse. Setting the sampling rate higher can help, but if set too high leads to increased interrupt and operating system overhead that can overwhelm the actual measurements. Not all hardware supports sampling; some systems lack a working overflow interrupt (for example various ARM systems).

Recently, the hardware and operating systems of modern systems have added support for new more advanced sampling features. Hardware features such as Intel PEBS and PEBS-LL [3], AMD IBS [1], and IBM marked events [8] enable gathering more detailed information for a sample, including register state, pipeline status, and stall accounting. For memory access events, information such as access latency, operand address, data source

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

XSEDE'15, July 26-30, 2015, St. Louis, Missouri, USA. (submitted)
Copyright 2015 ACM

for loads (e.g., cache level or memory and whether the access is local or remote) can be returned.

Support for hardware sampling has recently been added to the `perf_event` interface to the hardware counters that is part of the Linux kernel. The `perf_event` subsystem was created in 2009 as a response to the proposed merge of existing `perfctr` and `perfmom2` interfaces [7]. `perf_event` entered Linux 2.6.31 as “Performance Counters for Linux” and was subsequently renamed `perf_event` in the 2.6.32 release. The interface is built around file descriptors allocated with the new `perf_event_open()` system call [9]. Events are specified at open time in a complicated `perf_event_attr` structure; this structure has over forty different fields that interact in complex ways. Counters are enabled and disabled via calls to `ioctl()` or `prctl()` and values are read via the standard `read()` system call. Sampling can be enabled to periodically read counters and write the results to a ring buffer which can be accessed via `mmap()`. New data availability can be determined via a signal or by `poll()`. The `perf_event` interface undergoes frequent changes and thus tools that use it directly would need to port their tools to each new release.

Non-uniform memory access (NUMA) is used in multiprocessing as a computer memory design to determine memory access time depending on the memory location in the processor. The main advantage of processors under NUMA is the capacity to access its own local memory faster than remote memory (non-local memory). NUMA allows multiprocessing systems to share memory locally and improve performance. This is also used in symmetric multiprocessing (SMP) system, a “share everything” system, in which multiple processors work under a single operating system and have the possibility of accessing each other’s memory over a interconnect path or common bus. However, a problem arises as microprocessors are added; the common bus or data path gets overloaded and eventually comes a performance bottleneck. NUMA approaches this problem by providing an intermediate level of memory shared between microprocessors so that all data accesses do not travel on the main bus. The problem with NUMA is that it can be difficult for the programmer to control where the data accessed by a thread running on a core gets placed. Ideally the data should be close to the core on which the thread is running, and if this is not the case, there can be adverse effects on performance.

3. RELATED WORK

In order to develop an efficient program, programmers require performance tools to guide the design of codes for NUMA architectures. To approach this problem, several extensions were done to HPCToolkit – an open source performance tool for call path profiling of parallel programs – in order to enhance measurement and analysis of performance problems on multiprocessors with multiple NUMA domains. The improvements done to HPCToolkit allow the user to distinguish, quantify and analyze NUMA bottlenecks in the execution of multithreaded programs [5]. The tool evaluates the obtained metrics in order to assess the severity of bottlenecks, analyzes memory accesses, and provides plenty of information to improve optimization. Also, it includes information about the most adequate way to distribute data to minimize contention and reduce memory access latency. The HPCToolkit-NUMA version of HPCToolkit has not been released for general use. The reason is that the underlying access to the NUMA data uses low-level interfaces that are difficult to program. This tool would be more portable and easier to install and use if it could be built on top of a

PAPI middleware layer for accessing NUMA-related data, and this layer is exactly what we are proposing to provide.

The MemAxes tool described in [2] provides visualizations of NUMA data in semantic contexts, including node hardware topology, source code, and application-specific. These visualizations assist the programmer in finding and fixing NUMA performance bottlenecks. The visualization part of the MemAxes tool has been released for general use, but not the measurement part, because the authors have written their own kernel module to access the low-level NUMA performance data. Users of a production computer system do not normally have privileges to install such a kernel module. Thus, having this data provided by a community-accepted PAPI middleware layer would enable users to install and use MemAxes on any system that had this version of PAPI installed.

Other tools that have relied on instruction sampling to do memory performance analysis include Memphis [6] and Memprof [4]. Again, these tools require a kernel patch to obtain the data they need and require low-level programming of the hardware counter interfaces by the tool developers.

Our work differs from the above related work in that we are not trying to provide a tool that encompasses the entire measurement and analysis process, but rather middleware on which the above tools can build their analysis and visualization components. Furthermore, we are trying to do this as part of the widely used PAPI middleware library and to work with the Linux kernel developers to get the necessary functionality to support per-thread sampling into the mainstream kernel. Our effort will allow tool developers to focus their effort on support for interpreting and analyzing the data, and if we are successful, will enable their tools to be installed and usable on standard Linux platforms without the need for a kernel patch.

4. PAPI SAMPLING COMPONENT

The PAPI sampling interface is designed as a PAPI component that can be configured and compiled when PAPI is installed. It is implemented as a thin layer over the Linux `perf_event` interface. The design uses signals to determine when data is available and to collect it. The hardware event to be sampled, the sample data that is collected, and how the interrupt will be handled are determined by arguments to the `PAPI_sample_init()` call.

```
int PAPI_sample_init(
    int EventSet,
    int EventCode,
    int sample_type,
    int sample_period,
    int threshold, /* user-defined threshold for latency events */
    PAPI_sample_handler_t handler);

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

`PAPI_sample_init()` sets up a specific `EventCode` in an `EventSet` for sampling. The bits in the `sample_type` argument specify what information to include in the sample. Possible values for this argument are listed on the `perf_event_open()` man page [5] and include `PERF_SAMPLE_IP` for the instruction pointer, `PERF_SAMPLE_TID` for the process and thread IDs, `PERF_SAMPLE_WEIGHT` for a measure of how costly the sampled event was (e.g., the memory latency), and `PERF_SAMPLE_DATA_SRC` for where in the memory hierarchy the data associated with a sampled instruction came from. A sampled event generates an overflow notification every N

events, where N is given by `sample_period`. The handler argument is a user-defined signal handler. When an overflow occurs, the sample data is recorded in an mmap buffer. The threshold argument is optional and can specify a threshold value for the latency of memory operations above which instructions will be sampled – if not specified, a default value will be used. The `PAPI_sample_init()` call returns a file descriptor. The file descriptor is included in the `siginfo_t` data structure passed to the handler. The mmap buffer must be set up using the file descriptor prior to starting sampling. Because `PAPI_sample_init()` is intended as a low-level interface, we have left setting up the buffer, handling the interrupt, and parsing the buffer to the user. However, we provide example test programs and a utility routine to parse the buffer. The low-level sampling interface is intended for use by tool developers. In the future, we plan to provide a higher-level interface suitable for end users.

Once sampling for an event has been set up, it can be started by calling `PAPI_sample_start(fd)`, where `fd` is the file descriptor that was returned by `PAPI_sample_init()`, and sampling can be stopped by calling `PAPI_sample_stop(fd)`.

The `PAPI_sample_init()` call sets a number of additional `perf_event_attr` options by default. For example, it sets the `exclude_kernel` bit to exclude events that happen in kernel space. It sets `precise_ip` to 2 to request zero skid in the instruction pointer. These as well as several other fields may be modified by the expert user by calling `PAPI_set_opt()` and passing an opaque data structure containing the `perf_event_attr` structure with the desired values filled in.

It is highly desirable to obtain per-thread samples, since multithreaded codes may need to be analyzed for NUMA effects. When threads run on a shared-memory NUMA system, having the memory accessed by a thread be remote can degrade performance. Because UNIX signals are per-process, there is not a way to ensure that the thread that causes the interrupt will receive the overflow signal for the sampled event. It is possible for threads to mask off the signals so that only the threads that have the signals unmasked will receive it. Thus, it would be possible to set up a monitoring thread that received all the signals. By including `PERF_SAMPLE_TID` in the `sample_type` argument to `PAPI_sample_init()`, the recorded samples will contain the thread ID. Thus, when the buffer is parsed, the samples can be attributed to the proper thread.

There is still a problem however, in that samples are collected only for the calling process and thread. At first we thought that getting per-thread samples would be a simple matter of flipping on the "inherit" bit. This doesn't work however, because the `perf_event` kernel code specifically blocks getting mmap samples if `inherit` is enabled (with a comment about how getting that many samples would hurt performance). With the help of Linux kernel developers, we found a solution. To gather the per-thread information, we needed to set up a counter on each core, each with its own mmap buffer. In this way we don't have the large bus overhead of all the threads trying to write sample data back to the main thread's buffer. It's still necessary to have a hand-patched kernel in order for the overflow signals to come in from the child threads.

Thus, we needed to modify the `PAPI_sample_init` calling sequence so that it returns an array of file descriptors, one for each core's counter, as shown below.

```
int PAPI_sample_init(
    int EventSet,
```

```
    int EventCode,
    int sample_type,
    int sample_period,
    int threshold, /* user-defined threshold for latency events */
    PAPI_sample_handler_t handler,
    int *fds);
```

Each of these file descriptors is given a value by a call to `perf_event_open` for its core `i`, as shown below.

```
fds[i] = perf_event_open(&pe, 0, i, -1, 0);
```

The client code, which could be the application using PAPI directly, or an end-user performance analysis tool that is performing the instrumentation, then sets of the mmap buffers for each core, associating each with its file descriptor, as shown below (with error checking code omitted).

```
for (i=0; i<NUM_CORES; i++) {
    mmap_info[i].count=0;
    mmap_info[i].fd=fds[i];
    mmap_info[i].mmap=mmap(NULL, mmap_pages*4096,
        PROT_READ|PROT_WRITE, MAP_SHARED, fds[i],0);
}
```

The client code then calls `PAPI_sample_init(fd)` for each counter to start per-thread sampling. Our client code interrupt handler checks what file descriptor is passed into it and reads the mmap buffer associated with that file descriptor. The client code calls `PAPI_sample_stop(fd)` for each counter when the code being monitored has finished execution.

TEST PROGRAM RESULTS

We are developing our prototype PAPI sampling interface and running our tests on a 4-core Intel Haswell machine at the University of Maine that is running Linux 4.0.0-rc1 with the patch described above applied. We have instrumented the STREAM benchmark code (<https://www.cs.virginia.edu/stream/>) to collect memory latency events. We sampled the native event `MEM_TRANS_RETIRED:LATENCY_ABOVE_THRESHOLD`. We set the sample type to `PERF_SAMPLE_IP | PERF_SAMPLE_WEIGHT | PERF_SAMPLE_DATA_SRC`. We set the sample period to 100000 and used the default threshold. The run collected 65 samples, two of which are shown below.

```
PERF_SAMPLE_IP, IP: 401308
PERF_SAMPLE_WEIGHT, Weight: 40
PERF_SAMPLE_DATA_SRC, Raw: 68100242
Load Hit Line fill buffer No snoop Hit Level 1 TLB Level 2 TLB

PERF_SAMPLE_IP, IP: 401360
PERF_SAMPLE_WEIGHT, Weight: 7
PERF_SAMPLE_DATA_SRC, Raw: 68100142
Load Hit L1 cache No snoop Hit Level 1 TLB Level 2 TLB
```

Since higher level performance analysis tools typically have ways to convert the instruction pointer to a source code address, we do not address this issue. In order to obtain these results in a human-readable form, we had to parse the mmap buffer, which is not a trivial undertaking. The mmap values start with a header that contains a type field. If the type is `PERF_RECORD_SAMPLE`, then the record is a sample. If `PERF_SAMPLE_IP` was specified, then a 64-bit instruction pointer value is included. If `PERF_SAMPLE_TID` is enabled, then a 32-bit process ID and 32-bit thread ID are included. If `PERF_SAMPLE_DATA_SRC` was specified, then a 64-bit value is recorded that includes the type of memory operation (e.g., load, store, prefetch), the memory hierarchy level hit or miss, the snoop mode, and TLB access hit or miss. Detailed information on the layout of the `perf_event` mmap

buffer can be found on the Linux `perf_event_open` man page [9]. Although our example mmap buffer parser is not part of the PAPI library, we include it as a utility routine that can be modified as desired.

To test per-thread counts, we instrumented the OpenMP version of STREAM with the second version of `PAPI_sample_init()` described above. We set up eight counters for the eight logical CPUs and ran with eight OpenMP threads. We collected per-thread samples, two of which as shown below.

```
PERF_SAMPLE_IP, IP: 4012c0
PERF_SAMPLE_TID, pid: 3144, tid: 3144
PERF_SAMPLE_WEIGHT, Weight: 7
PERF_SAMPLE_DATA_SRC, Raw: 68100142
Load Hit L1 cache No snoop Hit Level 1 TLB Level 2 TLB

PERF_SAMPLE_IP, IP: 401a78
PERF_SAMPLE_TID, pid: 3144, tid: 3167
PERF_SAMPLE_WEIGHT, Weight: 28
PERF_SAMPLE_DATA_SRC, Raw: 68100242
Load Hit Line fill buffer No snoop Hit Level 1 TLB Level 2 TLB
```

5. CONCLUSIONS AND FUTURE WORK

We have been unable so far to test our PAPI sampling interface on a NUMA machine, since our development machine consists of a single NUMA node. We can modify the OpenMP version of the isbenchmark to have poor NUMA behavior on a NUMA architecture by turning off first touch initialization of the data. For example, when we do this on the Stampede system at TACC, we see a drastic reduction in the memory bandwidth. Results from running STREAM with sixteen threads on a single node on Stampede, with and without first touch, are shown below.

With first touch:

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	44840.9	0.005951	0.003568	0.017847
Scale:	47127.0	0.004679	0.003395	0.012240
Add:	52849.9	0.005304	0.004541	0.011292
Triad:	53368.3	0.005225	0.004497	0.010981

Without first touch:

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	7387.9	0.023044	0.021657	0.026183
Scale:	7259.9	0.023979	0.022039	0.028078
Add:	10768.7	0.025722	0.022287	0.030115
Triad:	10942.4	0.026642	0.021933	0.034551

We obtain similar results when running with eight threads. The poor performance happens because the data are initialized by the main thread and reside in memory local to its core, but the data are then accessed remotely by other threads. Unfortunately, none of the NUMA machines to which we have access are running a late enough Linux kernel to provide the features our PAPI sampling interface needs. We expect to have access to such a machine soon so that we can carry out our tests with STREAM and other multithreaded codes such as the OpenMP version of LULESH.

Our initial prototype of a PAPI sampling interface is a low-level interface intended for performance tool developers. We plan to make our implementation, which we call PAPI-NUMA, available to tool developers in a development branch of PAPI so that they can try it out and we can get feedback on its usefulness and on features they would like to have added or modified. We also plan to design a higher-level interface that will not require the user to provide the signal handler nor parse the mmap buffer.

Having per-thread sampling of memory events available on machine with stock Linux kernels through the PAPI interface will help make the tools discussed in the related work section that do memory performance analysis available to users of these machines. Furthermore, performance analysis tools that are already widely deployed on XSEDE machines and that use PAPI underneath, such as TAU and PerfExpert, will be able to expand their analysis capabilities to include NUMA performance issues.

There is an unavoidable problem that the capabilities needed rely on a recent Linux kernel and on the required patch becoming part of the kernel. Production machines typically run older kernel versions, and there is unfortunately no way that we can make PAPI-NUMA compatible with older kernel versions.

6. ACKNOWLEDGMENTS

This work is partially supported by the Department of Energy SciDAC program under grant number DE-SC00006722 and by the Air Force Office of Scientific Research under AFOSR Award No. FA9550-12-1-0476.

REFERENCES

- [1] Drongowski, Paul J. *Instruction-Based Sampling: A new performance analysis technique for AMD Family 10h Processors*. 2007. http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf
- [2] Giménez, Alfredo, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, Bernd Hamann. *Dissecting On-Node Memory Access Performance: A Semantic Approach*. SC 2014: 166-176.
- [3] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2.
- [4] Lachaize, R., B. Lepers, and V. Quema. MemProf: A memory profiler for NUMA multicore systems. In *Proc. Of the 2012 USENIX Annual Technical Conference*, Berkeley, CA, 2012.
- [5] Liu, Xu, and Mellor-Crummey, John M. *A tool to analyze the performance of multithreaded programs on NUMA architectures*. PPOPP 2014: 259-272.
- [6] McCurdy, C. and J.S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. of the 2010 IEEE International Symposium on Performance Analysis of Systems Software*, pages 87-96, March 2010.
- [7] Moore, S. V., D.K. Terpstra, V.M. Weaver. *Chapter 3: Software Interfaces to Hardware Counters*. D.H. Bailey, R.F. Lucas (Eds.) *Performance Tuning of Scientific Applications*, Taylor and Francis, 2010.
- [8] Srinivas et. al. IBM POWER7 performance modeling, verification, and evaluation. *IBM J. Res. & Dev.* 55(3): May/June 2011.
- [9] Weaver, Vince. *Perf_event_open manual page*, `perf_event_open.2`, in *Linux Programmer's Manual*, M. Kerrisk, Ed., Apr. 2014.