OPTIMIZING PAPI FOR LOW-OVERHEAD COUNTER MEASUREMENT

By Yan Liu

B.A., Central South University of Forestry and Technology (China), 2010

M.S., Central South University of Forestry and Technology (China), 2012

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Engineering)

The Graduate School

The University of Maine

December 2017

Advisory Committee:

Vincent Weaver, Assistant Professor, Advisor

Bruce Segee, Henry R. and Grace V. Butler Professor

Yifeng Zhu, Dr. Waldo "Mac" Libbey '44 Professor & Graduate Coordinator

OPTIMIZING PAPI FOR LOW-OVERHEAD COUNTER MEASUREMENT

By Yan Liu

Thesis Advisor: Dr. Vincent Weaver

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Engineering)

December/2017

Performance analysis is an essential step for better software optimization, which is critical for embedded systems, desktop applications and scientific computing. Most modern microprocessors contain hardware performance counters that can help with performance analysis. The PAPI library is a widely used self-monitoring performance measurement interface that supports the performance counter hardware found in most major microprocessors. PAPI supports self-monitoring: letting programs instrument chunks of code and gather detailed performance values.

A key aspect of self-monitoring is reading hardware performance counters with minimum possible overhead. Any overhead in the measurements can affect the accuracy of the results. In perf_event, the Linux interface to performance counters, the values are read via the read system call. This involves a large overhead when entering and exiting the operating system kernel.

In this work, we modify PAPI to use the rdpmc instruction that allows userspace measurement of counters on x86 systems. This replaces the use of the high-overhead *read()* system call. We tested the result across 14 modern systems and 4 benchmarks. We find that the performance measurement latency is improved by at least a factor of three (and often a factor of six or more) in our test cases.

ACKNOWLEDGEMENTS

The greatest gratitude is sent to my advisor, Dr. Vincent M. Weaver, for his complete support on this dissertation. Dr. Weaver has always been very encouraging, patient, knowledgeable, inspiring and supportive. The door to Dr. Weaver's office was always open whenever I had a question about my research and thesis writing. Without the help of him, this thesis would have never been written or completed. It is hard to imaging having a better advisor and mentor for my research and study on performance measurement.

Also, acknowledged are my committee members: Dr. Bruce Segee, Dr. Yifeng Zhu for their comments, suggestions and revisions provided to my research work.

It has been a good experience working and studying at the Department of Electrical & Computer Engineering. I want to express my gratefulness to the professors that have taught me professional knowledge during these two years. I thank Dr. Hummels, Dr. Segee, Dr. Zhu, Dr. Abedi and Dr. Weaver for teaching such wonderful classes and providing countless help whenever it is needed. I am also grateful to Ms. Cindy Plourde and Lynn Hathaway for taking care of my administrative-related business.

Being a foreign student, I want to thank the staff from the Office of International Program, especially Ms. Mireille Le Gal and Ms. Sarah Joughin. Without them, my study in Maine was not possible.

I acknowledge the financial support from teaching assistance program of the department and the National Science Foundation under Grant No. SSI-1450122.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS....................ii

LIST OF TABLES....................v

LIST OF FIGURES....................vi

1 INTRODUCTION AND MOTIVATION....................1

1.1 Motivation....................2

1.2 Background....................3

1.2.1 Hardware counters....................3

1.2.2 PAPI Library....................4

2 RELATED WORK....................5

2.1 PAPI Overhead....................5

2.2 Lower-Level Interface Overhead....................5

2.3 Other Performance Counter Tools....................6

2.4 Non-Temporal Overhead....................7

3 SPEEDING UP PAPI WITH THE RDPMC INSTRUCTION....................9

3.1 Proposal for using *rdpmc* in PAPI code....................9

3.2 *RDPMC* instruction....................9

3.3 PAPI *rdpmc* code....................10

4 EXPERIMENTAL SETUP....................14

4.1 PAPI_Cost Benchmark....................15

4.1.1 PAPI benchmark setup....................16

4.2 Sobel Edge Detector benchmark....................19

4.2.1 Sobel Edge Detector benchmark setup......19
4.3 High-Performance Linpack (HPL) Benchmark......21
4.3.1 HPL benchmark installation and setup [30]......21
4.4 STREAM benchmark......29
4.4.1 STREAM benchmark setup......29
5 RESULTS......32
5.1 *read ()* vs *rdpmc* overhead......32
5.2 Additional Event Overhead......37
5.3 Read overhead by PAPI version......39
5.4 Kernel update influence......39
5.5 PAPI_start/stop overhead......39
6 CONCLUSION AND FUTURE WORK......43
REFERENCES......44
APPENDICES......48
Appendix A: Code patch for papi_cost benchmark......48
Appendix B: Sobel code......67
Appendix C: Code patch for HPL benchmark......82
Appendix D: Code patch for STREAM benchmark......97
BIOGRAPHY OF THE AUTHOR......109

LIST OF TABLES

Table 4.1 Machines used in this study....15

Table 5.1 Median *read* vs *rdpmc* speedup from *papi_cost* running the *PAPI_read* 1 million times....33

Table 5.2 Sobel comparing with papi_cost on Haswell machine....34

Table 5.3 HPL comparing with *papi_cost* on Haswell machine....34

Table 5.4 Event value for each HPL function....35

Table 5.5 User and kernel space event cost by *PAPI_read ()* with *rdpmc* instruction....35

Table 5.6 User and kernel space event value cost by *PAPI_read ()* with *read ()* system call....36

Table 5.7 STREAM comparing with papi_cost....36

Table 5.8 Level 1 cache miss and TLB misses for each vector operation in STREAM....37

Table 5.9 Total Level 1 cache miss and TLB miss for STREAM when *PAPI_read()* supported by *rdpmc* and *read()* separately....37

Table 5.10 *PAPI_start ()/PAPI_stop ()* speedup on Haswell machine....40

LIST OF FIGURES

Figure 3.1 Instrument user code with PAPI....12

Figure 3.2 RDPMC operation on Pentium MMX processor....12

Figure 3.3 Code required for a perf_event rdpmc read....13

Figure 4.1 *PAPI_read ()* test code from papi_cost utility....16

Figure 4.2 Sample for *papi_cos*t output....16

Figure 4.3 PAPI_read in Sobel Convolution....20

Figure 4.4 Sobel output....21

Figure 4.5 Part of Makefile for HPL benchmark....22

Figure 4.6 HPL.dat file....25

Figure 4.7 HPL output....26

Figure 4.8 Free memory for Haswell machine....27

Figure 4.9 Main calculation code of HPL_pdgesv0....27

Figure 4.10 Part of HPL code with *PAPI_read ()*....28

Figure 4.11 Measuring user and kernel space event for *PAPI_read()*....29

Figure 4.12 STREAM output....30

Figure 4.13 STREAM code with PAPI instrumented (*PAPI_read ()* after long vector operation)....31

Figure 4.14 STREAM code with PAPI instrumented (*PAPI_read ()* inside long vector operation)....31

Figure 5.1 *PAPI_read ()* cost with *rdpmc* instruction varies with event number....38

Figure 5.2 *PAPI_read ()* cost with *read ()* system call varies with event number....38

Figure 5.3 *papi_cost* varies with PAPI version....40

Figure 5.4 *rdpmc* read code with loopCount checking..41

Figure 5.5 Output for *strace -c ./papi_cost*..42

1 INTRODUCTION AND MOTIVATION

The number of transistors that can be fit on a silicon chip doubles about every 18 months. However, despite the remarkable development in CPUs, software performance improvement based on these ever-faster machines is not in the same step. One of the reasons for this phenomenon could be that DRAM bandwidth increases quite slowly compared with the dramatic growing speed of CPU performance [1], while the performance of most applications depends on both core speed of the processor and bandwidth of the memory system.

Software optimization is quite critical no matter the application type: embedded application, desktop application or science computing. For better application optimization, performance measurement is an essential step. Reading data from the performance counters contained in most modern microprocessors can help with performance measurement. However, gathering this kind of data requires understanding the interaction of computer architecture, operating system, compiler and the running application [2]. The information about accessing these counters is usually poorly documented or unavailable to application developers and performance tool designers.

However, there are various tools available that can read the values of these hardware counters, such as PAPI [4], LIKWID [5] and perf [6]. Usually there are 3 ways for these tools to get hardware counter values: aggregate measurement, statistical sampling and self-monitoring [7]. Aggregate measurement is the kind of method that gathers total counts of specific events occurrences from the start to the end of the workload. This is the easiest and lowest overhead method to get hardware counter values. Statistical sampling obtains profiling data via counter overflows on timer interrupts. Using this method usually can get an imprecise result with low overhead, or fine-grained resolution values with high operating system overhead because of the increased

interrupts. Self-monitoring, on the other hand, allows developers to gather exact hardware performance readings for any specific blocks of code, rather than the entire program.

Though all the performance measurement tools can get performance data either by aggregate measurement or statistical sampling, Performance Application Programming Interface (PAPI) is one of the only tools that support self-monitoring. PAPI allows users to instrument a simple routine including start, read event, and stop to any block of code, that helps gather fine-grain and exact "caliper" measurement solely around the code of interest. Instrumenting PAPI into user's code involves adding extra code to the program. This extra code will perturb the program's execution and cause overhead. A goal with measurement software is to make this perturbation as small as possible.

1.1 Motivation

An overhead refers to a benchmark invariant difference between reported and actual count for a given event such as cache misses, branch instructions, total cycles and so on. [9]. Mytkowicz et al. [10] found that an overhead as low as a 2.5% increase in instruction count could interfere with properly correlating results. Mytkowicz et al. [11] also later shows that simply adding an additional PAPI counter could be enough to cause noticeable perturbations. Low overhead is critical for accurate performance measurements.

There is a long list of application performance analysis third-party tools that use or incorporate support for PAPI to gather, display and analyze performance information [11]. For instance, TAU (Tuning and Analysis Utilities) developed at the University of Oregon uses PAPI to support hardware counter reading and maintain portability. HPCToolkit can collect measurements based on hardware performance counters with installation of PAPI; otherwise, it can only collect measurements based on the operating system timer. There are various other tools that have

combined with PAPI to enhance their performance analysis module, such as KOJAK, PerfSuite, Titanium, SCALEA, OpenlSpeedshop, SvPablo and so on [12]. The wide use of PAPI as a performance tool illustrates the importance of reducing the PAPI overhead of collecting performance data, which is very important for application developers to tune and optimize the performance of their programs.

1.2 Background

1.2.1 Hardware counters

Hardware performance counters, or hardware counters, are a set of registers contained in most modern microprocessors. These counters can provide critical hardware event information for hardware verification/debugging and software related tasks such as performance monitoring, analysis or tuning. For most modern microprocessors, a small number of counter registers (4-7 for recent x86 processors) support measurement of a subset of hundreds of events. These events are usually about cycle count, instruction count, branch taken and prediction accuracy, TLB misses and invalidations, pipeline stalls, and memory access behaviors like miss rate at each memory hierarchy level.

There are usually two types of performance counter registers: configuration registers and counting registers. To monitor a performance event, the user has to access the configuration registers to select an event, start or stop counting, or enable interrupts. The operating system can then send low-level CPU calls to carry out the user's request. The user can than read from the counting registers that holds the current counts. Extra permission may be needed for users to access the counting registers, although they can also read the event value directly on some processors by special instructions (like *rdpmc* on x86) [6].

1.2.2 PAPI Library

The information about accessing hardware counters is usually poorly documented or unavailable to application developers and performance tool designers [3]. PAPI [13], initiated at the University of Tennessee at Knoxville, is a consistent, portable, cross-platform easy-use interface to the hardware counters on a wide variety of machines, providing developers information for performance analysis and tuning.

PAPI has two interfaces for the users to get access to hardware counters: the high level interface provides users simple and straightforward start, stop and read routines so that they can get specified event information from hardware counters without changing code across different platforms; the low level interface is fully programmable, thread safe, and allow users to define their own event sets.

On Linux, PAPI uses the perf_event interface to get access to performance counters. The perf_event subsystem was added to the Linux kernel in version 2.6.31 with the name of "Performance Counters for Linux" and was renamed to perf_event in Linux 2.6.32 [14]. To enable/disable counters, the *ioctl ()* system call is called. The *read ()* system call is used to read counter values. When sampling, a ring buffer that contains results can be gained by calling *mmap ()* and users can check to see if the result is available by a signal or the *poll ()* call[14]. Before perf_event was standard, PAPI used the perfctr [15] and perfmon2 [16] Linux interface, but they needed extra custom patching of the Linux kernel. Perfctr reads the values by accessing the mmap page and using the x86 *rdpmc* instruction without calling the *read ()* system call. Once perf_event was merged into the mainline Linux kernel, PAPI stopped the use of perfctr and perfmon2. However, after changing to perf_event, PAPI read met with a sharp increase of read overhead mainly because perf_event uses the *read ()* instead of *rdpmc* instruction to read counter values.

2 RELATED WORK

Low-overhead counter access is important, and there are various previous works on the topic, such as PAPI overhead, low-level interface overhead, and other performance tool overhead. PAPI is widely used and is often the comparison point for such studies.

2.1 PAPI Overhead

As mentioned in the last chapter, PAPI is now implemented via the *perf_event* interface. However, most previous PAPI comparisons use *perfctr* or *perfmon2* because they predate the introduction of *perf_event*. This makes direct comparison to our work difficult.

The overhead of PAPI including *PAPI_start () / PAPI_stop ()* and *PAPI_read ()* was measured by Moore et al. [18] [19] on several architectures and it seems that Linux / x86 had the least overhead. But the Linux version and x86 machine information is not given.

Weaver [20] instruments several PAPI overhead measurements [3] on various kernel interfaces for multiple PAPI versions (PAPI 4.0.0-4.1.2). The results show that the Linux 2.6.32 kernel version with perfctr generated the least overhead, while perf_events has the most overhead. His work gives a clue that the use of the *read ()* system call introduces many extra clock cycles.

2.2 Lower-Level Interface Overhead

Weaver [7] investigates the overhead of perf_event in terms of start/stop/read/overall on various x86_64 machines with a full range of Linux kernels. The measurements are at the raw system call level, one level lower than the PAPI interface we investigate. He also conducts the comparison against perfctr and perfmon2, and shows that perf_event has larger overhead in some implementations. Several methods, including using *rdpmc* to reduce overhead of

userspace to kernel space, are used to minimize the overhead and results are comparable with the perfctr and perfmon2 interfaces after the use of *rdpmc*.

Zaparanuks, Jovic and Hauswirth [28] investigate measurement overhead of both user and user+kernel counters using PAPI on top of perfmon2 and perfctr, as well as using perfmon2 and perfctr directly. It is a detailed investigation into obtaining minimum overhead on these interfaces, but predates the introduction of perf_event.

2.3 Other Performance Counter Tools

Roehl et al. [20] study the performance of likwid-perfctr and the LIKWID Marker API under the Linux OS on Intel IvyBridge-EP, Intel Haswell and AMD Interlagos. With the support of Marker API, likwid-perfctr can provide self-monitoring for an application. The Marker API is designed to measure code region performance by inserting API calls into user's code and do self-monitoring. However, comparing with PAPI, Marker API has to specify the events that need to be measured on the command line rather than in user's code. LIKWID relies on the MSRs using the Linux /dev/msr interface, causing high overhead because every time it reads or writes to counters it must context switch into the kernel. They mention the idea of using the *rdpmc* instruction to reduce overhead caused by system calls, but since the kernel disables rdpmc by default if not using perf_event, they cannot use it without patching the kernel. They also instrument the same STREAM test code with PAPI and compare with the LIKWID start/stop routine, however the result is not valuable and hard to know how much overhead read operations cost since they did not use a separate read routine.

Demme and Sethumadhaven propose LiMiT [21], a Linux interface to provide fast, userspace access to performance counters. Compared with PAPI, it is relatively complicated to install since it requires a user land API and a kernel patch, and a note on the project's website notes that the

patch is unstable and can cause system crashes. They claim LiMiT is 90x faster than PAPI and 23x faster than perf_event, but the results are somewhat unclear since there is no detailed description about the test or the kernel versions.

AMD proposed an advanced Lightweight Profiling [22] interface providing userspace-only access to all aspects of controlling performance counters, not just reads. This could potentially speed up much more than reads, however the Linux kernel developers have refused to add support for the interface unless it moderated by the kernel, which would defeat the entire purpose [23].

DeRose [24] describes the HPM Toolkit, which contains components such as the hpmcount utility that can measure application overall wall clock time as well as hardware performance counter information. The HPM library supports the measurement of program sections. HPM Toolkit uses the PAPI interface, so it supports the measurement of programs that run on the Intel platform under Linux in addition to IBM Power 3 with AIX. The majority of the overhead of this tool is caused by *gettimeofday ()*, which is used to measure time cost. The reason why they do not chose cycles for time measurement is recording cycles could occupy an extra hardware counter and it is not accurate for PAPI since it measures total user mode time.

2.4 Non-Temporal Overhead

Our work, and much of the previous work, is mainly about the overhead in terms of extra cycles introduced by the measurement. Performance can also affect other metrics.

Lehr [25] implements performance measurement on a subset of the SPEC CPU 2006 benchmark suite [33] and found that even with as low as 10% of running time overhead, performance event counts (including stores, branch misprediction and cache related events) can be perturbed significantly.

Huang et al. [26] studied the power and energy consumption as well as PAPI power overhead when doing the measurement on the Intel Haswell processors. They found that power monitoring by PAPI could induce large power overhead (32% more than idle). The measurement itself is using processor resources (memory, cache, cpu and so on) that can lead to a certain amount of power and energy overhead.

Babka and Tuma [27] investigate the overhead of PAPI in both cycle count and cache behavior on an AMD and Intel machine. Although the results were high, the PAPI version they were using was based on a Perfctr interface that is out of date and benchmarks they used in the experiment were not described in detail.

Maxwell et al. [17] test the PAPI overhead in terms of event counts (for example load/store counts). The overhead is decided to be significant or not by comparing with the event counts that PAPI read. If the expected value is very large then the overhead is insignificant, otherwise, calibration is needed. By reducing the instruction overhead, event counts obtained by PAPI are more accurate.

3 SPEEDING UP PAPI WITH THE RDPMC INSTRUCTION

3.1 Proposal for using *rdpmc* in PAPI code

In order to get hardware event values for any block of code of an application, one has to go through several steps. Figure 3.1 presents example code instrumented with PAPI. The first step is initialization where users have to initialize the PAPI library, create an event set, check event availability and add events. These parts may have relative higher overhead than read calls, but are of much less significance since users can separate this part of code from the object code to avoid overhead influence. *PAPI_start ()* and *PAPI_stop ()* can be put in the startup/shutdown part of the program as well to limit perturbation of the measurement. This leaves the *PAPI_read ()* routine as the most important routine that needs to be low-overhead. To get counter values on Linux, *PAPI_read ()* calls the r*ead ()* system call, which takes a relatively long time (hundreds to thousands of cycles) to enter into the Linux kernel and executes a lot of code before returning to the user. This overhead can be avoided by using the *rdpmc* instruction, which can read the counters directly from userspace, without involving the operating system. In the following work, we will extend PAPI to use this low-overhead *rdpmc* instruction instead of using the default *read ()* system call.

3.2 *RDPMC* instruction

RDPMC (Read Performance Monitor Counter) is a native instruction from the x86 instruction set. Performance counters were first introduced with the Pentium MMX and rdpmc was introduced at the same time. While at first only two counters were available, now all of the core CPU counters support them (up to 8 general purpose plus three fixed on Intel processors).

Figure 3.1 illustrates rdpmc operation on the Pentium MMX processor. Two 40-bit performance-monitoring counters (0 and 1) are specified in the ECX register. By calling the *rdpmc* instruction, the value contained in performance-monitoring counter will be loaded into EDX: EAX, with the high-order 8 bits going to EDX register and low-order 32 bits going to EAX register.

The purpose of the instruction is to allow application code to access performance counters directly to avoid the overhead of calling into the operating system. The application code can read the performance-monitoring counters at a privilege level of 1, 2, or 3 with the instruction enabled (setting the PEC flag in the CR4 register). The instruction itself only takes a short amount of time to run, on the order of a few tens of cycles [29].

3.3 PAPI *rdpmc* code

Before *perf_event* was merged into Linux, users could set performance counters to be free-running and access them using an assembly-language *rdpmc* call. However, with *perf_event* using *rdpmc* is not that simple any more, since Linux is a multi-tasking operating system and there might be multiple users of *perf_event*. The kernel may rearrange event counters at any point, which could happen due to context switching or multiplexing (when the number of events that the kernel is measuring is more than the number of available counters). The counter value might have changed since the last read because of multiplexing. Also, due to context switching or counter overflowing, the kernel may have read and restarted the counters.

Extra PAPI code (show in Figure 3.3) is needed to ensure that the kernel has not changed the event configuration since last time the event was read. The overhead of the code is on the order of a few hundred cycles, which is double the read overhead. However, it is still much faster than using the *read ()* system call, which is slow, disturbs to the CPU, and involves running an unpredictable amount of kernel code.

To conduct a *rdpmc* read, first the *pc->lock* variable is read in the beginning of the code and checked at the end to ensure it stays the same since it may been be incremented by the kernel any time the *perf_event_update_userpage ()* is called. The calling of *barrier ()* ensures that the *pc->lock* value is synchronized with the kernel. To check whether or not multiplexing is happening, *pc->time_running* and *pc->time* enabled is checked. There is multiplexing if those values are not equal to each other, and more accurate multiplex calculation follows (The results need to be scaled appropriately to account for the time where the event was not running). The count value stores the most recent value read by the kernel. Then the *rdpmc* instruction is called and the sign extended result returned from it is added to the count value. Variable *pc->lock* is read again in the end and if it is not the same as the beginning, the whole process will run again until the value is not changed.

The code listed in Figure 3.3 is based on sample code provided by the kernel developers. It works in PAPI but may need some additional error handling. If the kernel is busy updating the page, the sequence checking could never make progress and a livelock may happen, but we have not met this problem in practice. Extra code is needed to help break out the loop and go back to a *read ()* if this happens.

One should *set --enable-perfevent-rdpmc = yes* in configure when building and installing since it currently is not enabled by default in PAPI. This feature will be enabled by default in the soon to be released PAPI 5.6.

Figure 3.2 Instrument user code with PAPI

```
  /* user start up code */
  retval = PAPI_library_init( PAPI_VER_CURRENT ); //initlize PAPI library
  if (retval != PAPI_VER_CURRENT ) {
          fprintf(stderr,"PAPI_library_init\n");
          exit(retval);
  }
  retval = PAPI_create_eventset( &EventSet ); // create eventset
  if (retval != PAPI_OK ) {
          fprintf(stderr,"PAPI_create_eventset\n");
          exit(retval);
  }
  retval = PAPI_add_event( EventSet, PAPI_TOT_CYC ); // add PAPI_TOT_CYC(total cycle count) event
  if (retval != PAPI_OK ) {
          fprintf(stderr,"PAPI_add_event\n");
          exit(retval);
  }
  retval = PAPI_add_event( EventSet, PAPI_TOT_INS ); // add PAPI_TOT_INS(total instruction count) event
  if (retval != PAPI_OK ) {
          retval = PAPI_add_event( EventSet, PAPI_TOT_IIS );
          if (retval != PAPI_OK ) {
                  fprintf(stderr,"PAPI_add_event\n");
                  exit(retval);
          }
  }
/* user code*/

  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK ) { // start event measurement
          fprintf(stderr,"PAPI_start");
          exit(retval);
  }

  /* user code */

  PAPI_read(EvenSet,values); // read current event value and store into values array

 /* user code */

  if ( ( retval = PAPI_stop( EventSet, NULL ) ) != PAPI_OK ) { //stop event measurement
          fprintf(stderr,"PAPI_stop");
          exit(retval);
  }

 /* user clean up code*/

PAPI_shutdown();
```

Figure 3.3 RDPMC operation on Pentium MMX processor.

```
1  RDPMC {
2      IF (CPL != 0) && (CR4.PCE == 0) {
3          #GP(0);
4      }
5      IF (ECX = 0) {
6          EDX:EAX = Performance_Counter0;
7      } ELSE IF (ECX = 1) {
8          EDX:EAX = Performance_Counter1;
9      } ELSE #GP(0);
10 }
11
```

Figure 3.4 Code required for a perf_event rdpmc read

```
do {
        /* The kernel increments pc->lock any time */
        /* perf_event_update_userpage() is called */
        /* So by checking now, and the end, we */
        /* can see if an update happened while we */
        /* were trying to read things, and re-try */
        /* if something changed */
        /* The barrier ensures we get the most up to date */
        /* version of the pc->lock variable */
        seq=pc->lock;
        barrier();

        /* For multiplexing */
        /* time_enabled is time the event was enabled */
        enabled = pc->time_enabled;
        /* time_running is time the event was actually running */
        running = pc->time_running;

        /* if cap_user_time is set, we can use rdtsc */
        /* to calculate more exact enabled/running time */
        /* for more accurate multiplex calculations */
        if ( (pc->cap_user_time) && (enabled != running)) {
                cyc = rdtsc();
                time_offset = pc->time_offset;
                time_mult = pc->time_mult;
                time_shift = pc->time_shift;

                quot=(cyc>>time_shift);
                rem = cyc & (((uint64_t)1 << time_shift) - 1);
                delta = time_offset + (quot * time_mult) +
                        ((rem * time_mult) >> time_shift);
        }
        enabled+=delta;

        /* actually do the measurement */

        /* Index of register to read */
        /* 0 means stopped/not-active */
        /* Need to subtract 1 to get actual index to rdpmc() */
        index = pc->index;

        /* count is the value of the counter the last time */
        /* the kernel read it */
        /* If we don't sign extend it, we get large negative */
        /* numbers which break if an IOC_RESET is done */
        width = pc->pmc_width;
        count = pc->offset;
        count<<=(64-width);
        count>>=(64-width);

        /* Ugh, libpfm4 perf_event.h has cap_usr_rdpmc */
        /* while actual perf_event.h has cap_user_rdpmc */

        /* Only read if rdpmc enabled and event index valid */
        /* Otherwise return the older (out of date?) count value */
        if (pc->cap_usr_rdpmc && index) {

                /* Read counter value */
                pmc = rdpmc(index-1);

                /* sign extend result */
                pmc<<=(64-width);
                pmc>>=(64-width);

                /* add current count into the existing kernel count */
                count+=pmc;

                /* Only adjust if index is valid */
                running+=delta;
        }

        barrier();

} while (pc->lock != seq);

if (en) *en=enabled;
if (ru) *ru=running;
return count;
```

4 EXPERIMENTAL SETUP

We test on fourteen different machines as shown in Table 4.1. These machines cover multiple generations of Intel and AMD processors, from a 20-year-old Pentium II machine up to and including more modern machines.

Most of the machines tested ran the Linux 4.9 kernel provided with the Sid release of Debian Linux. A few of the machines are running the 3.16 kernel provided with Jessie Debian Linux.

We first compare the overhead of *PAPI_read ()* and *PAPI_start () / PAPI_stop ()* routines in the *papi_cost* tool before and after being enhanced with *rdpmc* instructions across the 14 machines mentioned above. On the Haswell machine we also test and compare the before and after overhead by instrumenting PAPI code into several benchmarks like Sobel Edge Detector, HPL and STREAM. Next, we run several benchmarks at the same time to create a situation of context switching and multiplexing, aiming to catch event counter rearrangement. STREAM is a heavy memory utilizing benchmark. Each event you want to read via *rdpmc* must have an associated mmap page. This could potentially have overhead issues. We ran experiments on STREAM with PAPI instrumented in to see if there are any page faults or significant numbers of extra TLB accesses or misses.

Most of our experiments are against a development git snapshot a21e3da5096bb410eaf6 from March 2017, as at the time of this writing no full PAPI release contains the *rdpmc* code. For comparison we also look at the 5.4.0, 5.4.1, 5.4.3, 5.5.0, and 5.5.1 official PAPI releases.

Table 4.1 Machines used in this study

Processor	Counters Available	Processor	Counters Available
Intel Pentium II	2 general	Intel Broadwell i7-5557U	4 general 3 fixed
Intel Pentium 4	18 general	Intel Broadwell-EP E5-2620	4 general 3 fixed
Intel Core2 P8700	2 general 3 fixed	Intel Skylake i7-6700	4 general 3 fixed
Intel Atom Cedarview D2550	2 general 3 fixed	AMD fam10h Phenom II	4 general
Intel IvyBridge i5-3210M	4 general 3 fixed	AMD fam15h A10-6800B	6 general
Intel Haswell i7-4770	4 general 3 fixed	AMD fam15h Opteron 6376	6 general
Intel Haswell-EP E5-2640	4 general 3 fixed	AMD fam16h A8-6410	4 general

4.1 PAPI_Cost Benchmark

We use the *papi_cost* utility provided by PAPI as the reference benchmark to measure *PAPI_read ()* overhead. A segment of code in this utility for read testing is shown in Figure 4.1. The code is wrapped with *PAPI_start ()* and *PAPI_stop* () with the *EventSet* as parameter that was defined in the set up part of the *papi_cost*. After one call to *PAPI_read ()*, it loops *num_iters* (initially 1 million) times calling the *PAPI_read ()* function, and the value of events is saved in array parameter *values*. To measure cycle cost for each *PAPI_read ()* run, it calls the PAPI library function *PAPI_get_real_cyc( )* that uses *rdtsc* read timestamp instruction on x86 systems.

We extend *papi_cost* code to return the median, 25th and 75th percentile values so that we can use those to make boxplots. The patch of the modified code is contained in Appendix A. A sample of output about how many cycles *PAPI_read()* cost after running *papi_cost* utility is shown in Figure 4.2. The 99th percentile value is used to show that the max cycles value is an

outlier. We also modified *papi_cost* further to log in which loop cycle the outliers happen to help with outlier analysis.

Figure 4.5 *PAPI_read ()* test code from papi_cost utility

```
/* Start the read eval */
printf( "\nPerforming read test...\n" );

if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK ) {
        fprintf(stderr,"PAPI_start");
        exit(retval);
}
PAPI_read( EventSet, values );

for ( i = 0; i < num_iters; i++ ) {
        totcyc = PAPI_get_real_cyc(  );
        PAPI_read( EventSet, values );
        totcyc = PAPI_get_real_cyc(  ) - totcyc;
        array[i] = totcyc;
}
if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK ) {
        fprintf(stderr,"PAPI_stop");
        exit(retval);
}

do_output( 2, array, bins, show_std_dev, show_dist );
```

Figure 4.6 Sample for *papi_cost* output

```
Total cost for PAPI_read (2 counters)
over 1000000 iterations
min cycles    : 139
max cycles    : 13382
mean cycles   : 143.275044
std deviation: 23.831947
25% cycles    : 142
50% cycles    : 142
75% cycles    : 142
99% cycles    : 161
```

4.1.1 PAPI benchmark setup

To measure *PAPI_read ()* overhead with the *papi_cost* utility, one needs to go through the following steps to download PAPI, configure it properly and run the *papi_cost* program found in the *src/utils* directory.

(1) Get the latest version of the PAPI repository.

If it is the first time, clone it with the following command:

git clone https://bitbucket.org/icl/papi.git

The command above creates a *papi* folder on the local computer that contains a complete copy of the PAPI git repository. If one is using an existing checkout of PAPI, be sure it is a version from April 2017 or newer. One can update the current git tree via the git pull command.

(2) Configure papi with *rdpmc* enabled

cd src

./configure --enable-perfevent-rdpmc=yes

(3) Run make.

When make is running, among the options passed there should be something like:

-DUSEPERFEVENTRDPMC=1 in the command line options scrolling by just before the always there message:

-DPEINCLUDE = libpfm4/include/perfmon/perfevent

(4) Once things are compiled, one can check if perf_event rdpmc is enabled by running

./papi_component_avail -d

and it should list Fast counter read: 1

(5) Run papi_cost

utils/papi_cost

(6) A subset of the results looks something like this:

Total cost for PAPI_read (2 counters)over 1000000 iterations

min cycles : 139
max cycles : 34190
mean cycles : 148.594704
std deviation: 67.128018

(7) Modify PAPI code to get percentile values. Patch the *papi_cost* with code in Appendix A. This will report the boxplot information similar to the following:

Total cost for PAPI_read (2 counters)
over 1000000 iterations
min cycles : 139
max cycles : 34190
mean cycles : 148.594704
std deviation: 67.128018
25th percentile :145
50th percentile : 148
75th percentile : 148
99th percentile: 167

(8) Run through different number of event set from 1 event to 4 events. The default number of events is 2, with *PAPI_TOT_CYC* and *PAPI_TOT_INS*. To delete or add events, modify the papi/src/utils/cost.c file by the use of the *PAPI_add_event ()* function. Be sure to change the length of the *value []* array appropriately.

(9) We run the comparative test on the latest PAPI git repository and released versions from 5.4 to 5.5.

One can use the *wget* command to download a specific released version, for example, if you want to download papi 5.5.1, run the following command:

wget http://icl.utk.edu/projects/papi/downloads/papi-5.5.1.tar.gz tar -zxvf papi-5.5.1.tar.gz

Configure PAPI as before, though it is not possible to enable *rdpmc* support as it does not exist in older versions.

(10) The test to generate the outlier plots modified *papi_cost* to print out all 1-million timing results and counter results to disk, which is a minor change to *papi_cost*

(11) The historical PAPI versions are trickier, as they run on old versions of Linux that are not necessarily easy to obtain or set up.

To get *perfctr* running, it is necessary to get a 2.6.32 Linux kernel source, and patch with the most recent (confusingly named) 2.6.40 *perfctr* patch. Then boot into a version of Linux that can handle old kernels (something newer like Debian Jessie might have problems, but booting with

init=/bin/sh and manually mounting the file system should let you run the tests). When configuring and building PAPI should automatically detect you are using *rdpmc*. If it does not, it may be necessary to manually make the permissions on /dev/perfctr be 666 Be sure to use a current git snapshot of PAPI as there was a long-standing bug breaking older perfctr and perfmon2 support that was not fixed as of 5.5.1.

Perfmon2 is more difficult because the newest supported version is 2.6.30. Checkout the perfmon2 git tree and build the kernel, and boot into it. It is necessary to have an older version of Linux, such as Debian Squeeze (most newer Linux distributions require 2.6.32 kernels at least). It is necessary to be running on an older processor supported. For our measurements we created a Debian squeeze chroot, booted with init=/bin/sh, and manually switched to and entered the chroot. This worked, but can be a bit of a complex task if unless one is a Linux expert. Again building PAPI is the same as it is on other platforms and should auto detect you are on perfmon2.

4.2 Sobel Edge Detector benchmark

The Sobel edge detector, also called Sobel operator, usually is used to perform a 2-D gradient measurement on an image and create an image emphasizing edges. The operator contains two 3*3 kernels that are convolved with the original image to get the approximate value of the image derivatives. The kernels are applied to the image separately, producing Gx and Gy, which are the horizontal and vertical magnitude. Combining the above two images together, can get the gradient magnitude, using:

$$G=\sqrt{G_X^2+G_Y^2}$$

4.2.1 Sobel Edge Detector benchmark setup

As shown in 4.3, we instrumented the *PAPI_read ()* function in the convolution section of the sobel code (code is listed in Appendix B). The center of the matrix (3*3 kernel) is applied to the pixel of interest from the input matrix and the surrounding pixels are added together and used to calculate the value for the pixel in the output matrix. For example, with a convolution matrix

$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$ of and pixel values $\begin{matrix} a & b & c \\ d & e & f \\ g & h & i \end{matrix}$, if we want to apply the convolution to pixel f, the resulting output pixel would end up being

out [1][1] = (0 * a) + (−1* b) + (0* c) + (−1 * d) + (5* e) + (−1*f) + (0 * g) + (−1* h) + (0* i) .

After each iteration of the convolution, the *PAPI_read (Eventset,values)*function is called and the number of cycles is read by *PAPI_get_real_cyc ()* function. Events PAPI_TOT_CYC and PAPI_TOT_INS are added in the *Eventset*. By running command *./sobel test.jpg*, we can get output similar with Figure 4.4. Test.jpg is a jpg format image with the size of 1920*1200.

Figure 4.7 PAPI_read in Sobel Convolution

```
//PAPI start
if((result=PAPI_start(eventset))!=PAPI_OK)
        printf("Error PAPI start:%s\n",PAPI_strerror(result));
PAPI_read(eventset,values);
for(d=0;d<3;d++) {  //depth
   for(x=0;x<old->x;x++) { // x direction
     for(y=0;y<old->y;y++) { // y direction
        sum=0;
        for(k=-1;k<2;k++) {
           for(l=-1;l<2;l++) {
                color=old->pixels[((y+l)*width)+(x*depth+d+k*depth)]; // convolution
                sum+=color * (*filter)[k+1][l+1];
           }
        }

        if (sum<0) sum=0;
        if (sum>255) sum=255;

        new->pixels[(y*width)+x*depth+d]=sum;
        if(num_iters<=MAX)
        {
                totcyc=PAPI_get_real_cyc();
                PAPI_read(eventset,values);//perfom once PAPI_read in every convolution iteration
                totcyc=PAPI_get_real_cyc()-totcyc; // get total cycle for PAPI_read
                array[num_iters]=totcyc;
                num_iters++;
        }
     }
   }
}

if((result=PAPI_stop(eventset,values))!=PAPI_OK) //papi stop
        printf("Error PAPI stop:%s\n",PAPI_strerror(result));
cal_stats(num_iters,array,&min,&max,&avag,&std);
printf("Total cost for PAPI_read over %d iterations \n", --num_iters);
printf("min cycles : %lld\nmax cycles : %lld\naverage cycles : %lf\nstandard deviation: %lf\n",
        min,max,avag,std);
```

Figure 4.8 Sobel output

```
yanliu@haswell:~/research/papioverhead/sobel$ ./sobel test.jpg
output_width=1920, output_height=1200, output_components=3
Total cost for PAPI_read over 1000000 iterations
25% cycles   : 154
50% cycles   : 163
75% cycles   : 172
99% cycles   : 209
min cycles : 142
max cycles : 12763
average cycles : 165.710793
standard deviation: 27.853726
```

4.3 High-Performance Linpack (HPL) Benchmark

HPL is a software package that solves a dense linear equation Ax=b, where A is an N*N matrix in double precision (64 bits). The main aim of the benchmark is to measure the floating point

computing power of a large distributed-memory computer, the result of which can be submitted to the TOP500 supercomputer list [34].

The first step of solving the order-n linear system is computing LU factorization with row partial pivoting of the n-by-n+1 coefficient matrix that will be partitioned into nb-by-nb blocks. The data is distributed onto a P-by-Q grid of processes for cluster computing, however, in this experiment, we only measure *PAPI_read () i*n one process.

4.3.1 HPL benchmark installation and setup [30]

(1) Install dependencies

To successfully install and use HPL, a few software dependencies must be met. They are: Gfortran (Fortran compiler), MPICH2 (an implementation of MPI), mpich2 (dev - development tools) and BLAS (Basic Linear Algebra Subprograms) [30].

(2) Download HPL and set it up

Download the HPL package, extract the tar file and create a makefile based on the given one. Run the following commands one by another.

```
tar xf hpl-2.1.tar.gz
cd hpl-2.1/setup
sh make_generic
cd ..
cp setup/Make.UNKNOWN Make.OpenBLAS
```

The last command makes a copy of Make.UNKNOWN to Make.OpenBLAS. The make file contains the details of system configuration, libraries like mpich2, atlas/blas package, home directory and so on. Next we will adjust the Make.OpenBLAS file and compile HPL with the support of BLAS.

(1) Modify make file

The changes in *makefile*areshown in Figure 4.5

Figure 4.9 Part of Makefile for HPL benchmark

```
# - Message Passing library (MPI) --------------------------------------
# ----------------------------------------------------------------------
# MPinc tells the  C  compiler where to find the Message Passing library
# header files,  MPlib  is defined  to be the name of  the library to be
# used. The variable MPdir is only used for defining MPinc and MPlib.
#
MPdir        = /usr/
MPinc        = -I$(MPdir)/include/mpi
MPlib        = -lmpich
#$(MPdir)/lib/x86_64-linux-gnu/libmpich.a
#
# ----------------------------------------------------------------------
# - Linear Algebra library (BLAS or VSIPL) -----------------------------
# ----------------------------------------------------------------------
# LAinc tells the  C  compiler where to find the Linear Algebra  library
# header files,  LAlib  is defined  to be the name of  the library to be
# used. The variable LAdir is only used for defining LAinc and LAlib.
#
LAdir        =
LAinc        =
LAlib        = /home/yanliu/research/blas/OpenBLAS-0.2.19/libopenblas.a -lpthread
#


# - HPL includes / libraries / specifics -------------------------------
# ----------------------------------------------------------------------
PAPIlib        = /home/yanliu/research/papioverhead/papi/papi/src/libpapi.a -lm
#

HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc)
HPL_LIBS     = $(HPLlib) $(LAlib) $(MPlib) $(PAPIlib)
#
```

In the last row shown in Figure 4.5, the path of PAPI library is added in the HPL_LIBS variable, so that we can instrument PAPI code into HPL benchmark.

(2) Compiling and running the HPL benchmark

Before making any changes to the code, we compiled the HPL benchmark using the following command to make sure that the makefile was working properly.

make arch=OpenBLAS

After make finished successfully, the file “xhpl” will show up in the “bin/OpenBLAS” folder. Next, it is necessary to create a “HPL.dat” input file in the same folder as shown in Figure 4.6 and run the *./xhpl* command. The output is shown in Figure 4.7.

(3) HPL parameter tuning

HPL.data is the input file for HPL. N is the problem size, which is the size of the coefficient matrix. Normally, using the largest problem size fitting in memory can help give a best performance result of the system. As shown in Figure 4.8, the free memory for this Haswell machine we are

working on is 3245M which is 405M double precision (8 bytes) elements. The size of the coefficient matrix is the square root of that number and equals 20607. Since the OS and other things need memory as well, 80%*20607=16000 should be a good setting for N.

NB is the block size and is used for data distribution and computational granularity. According to [32], a good block sizes falls in [32..256] interval. We test the performance when NB is set to 32, 64, 128, 256 respectively, and find that when NB is set to 128, it gets the best performance (Gflop). We set NB to 128 for our runs.

We are only studying the case of single processors, so we set Qs and Ps to 1.

(4) Instrument HPL with PAPI

We set Ps and Qs to 1 (we only use 1 process) and chose *HPL_pdgev0* code as the host code (Shown in Figure 4.9) of the *PAPI_read ()* measurement since it takes the most time during the whole calculation and iterates the main calculation code including LU factorization, panel broadcast and update for Ns/NBs times. In this way, we can get the results of the total cycles used for the *PAPI_read ()*. The patch for the code instrumented with PAPI is listed in Appendix C. As shown in Figure 4.10, we insert the code of *PAPI_read* () and calculate the cost of it after each HPL function in every iteration. The functions are *HPL_pdpanel_init* (re-initializes panel data structure), *HPLpdfac*t (factors current panel), *HPL_binit* (initializes the row broadcast), *HPL_bcast* (performs the row broadcast), *HPL_bwait* (finalizes row broadcast, waits for the row broadcast of current panel to terminate), *HPL_pdupdate* (broadcasts a panel and update the trailing submatrix) and *MNxtMgid* (updates message id for next factorization). The value *array [i][num_iters]* (i=0,1..6) saves the cost of *PAPI_read ()* after the *(i+1)*th fuction, *num_iters* is the index of the current iteration. For instance, if i=0, *array [0][num_iters]* saves the total cycle cost of *PAPI_read ()* after the calling of the first function *HPL_pdpanel_init ()*. Similarly, the value *eventVal[i][num_iters]* (i=0,1..6) saves the value of the event that being measured for the *(i+1)*th

function. The event value *eventVal[1][num_iters]*, for example, stores event values for the second function (*HPL_pdfact*). We first add events *PAPI_TOT_CYC* (total cycle) and *PAPI_TOT_INS* (total instruction completed) to *eventset* so that we can keep consistent with *papi_cost*. To analyze the cost of each *PAPI_read*, we then respectively set the event to *PAPI_L1_DCM* (level 1 data cache miss), *PAPI_L1_ICM* (level 1 instruction cache miss), *PAPI_BR_INS* (branch instruction), *PAPI_BR_MSP* (conditional branch instructions mispredicted), *PAPI_TLB_DM* (translation buffer data miss) and *PAPI_TLB_IM* (translation buffer instruction miss).

(5) Measuring user and kernel space event cost by *PAPI_read ()*

Our result shows that, for different subroutines where we instrumented *PAPI_read ()* function, the cycles that the *PAPI_read ()* cost vary considerably. So we use the *rdpmc* instruction to measure the user and kernel events that P*API_read ()* costs for each subroutine. Figure 4.11 demonstrates how we measure the events value for *PAPI_read ()* in HPL_pdfact section. Total user events can be calculated by *rawAfter-rawBefore1* and kernel event can be obtained by *rawAfter2-rawBefore2*. *USER_EVENT* and *KERNEL_EVENT* are used as variables for *rdpmc ()* and they are the macro definitions for the counter index that *rdpmc ()* requires. Usually they are 0 and 1 since we are measuring two events. To make sure the counters are getting desired event data, we use the *PAPI_add_named_event ()* function to help us setup counters. For example, if we add the following two lines of code before using *rdpmc (counter_index)*, the counter with index 0 would measure userspace *PAPI_TOT_CYC* value and the counter with index 1 would measure kernel space *PAPI_TOT_CYC* value.

PAPI_add_named_event (eventset," L1D:REPLACEMENT:u=1");

PAPI_add_named_event (eventset," L1D:REPLACEMENT:u=0:k=1");

We did this measurement on the Haswell machine, so the event name can be looked up from the Haswell machine section in *papi/src/papi_events_table.h* file.

Figure 4.10 HPL.dat file

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
16000           Ns
1            # of NBs
128             NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
1         Ps
1         Qs
16.0         threshold
1            # of panel fact
0          PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
2           NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
2         RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
2            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

Figure 4.11 HPL output

```
================================================================================
HPLinpack 2.1  --  High-Performance Linpack benchmark  --   October 26, 2012
Written by A. Petitet and R. Clint Whaley,  Innovative Computing Laboratory, UTK
Modified by Piotr Luszczek, Innovative Computing Laboratory, UTK
Modified by Julien Langou, University of Colorado Denver
================================================================================

An explanation of the input/output parameters follows:
T/V    : Wall time / encoded variant.
N      : The order of the coefficient matrix A.
NB     : The partitioning blocking factor.
P      : The number of process rows.
Q      : The number of process columns.
Time   : Time in seconds to solve the linear system.
Gflops : Rate of execution for solving the linear system.

The following parameter values will be used:

N      :   16000
NB     :     128
PMAP   : Row-major process mapping
P      :       1
Q      :       1
PFACT  :    Left
NBMIN  :       2
NDIV   :       2
RFACT  :   Right
BCAST  :   2ring
DEPTH  :       0
SWAP   : Mix (threshold = 64)
L1     : transposed form
U      : transposed form
EQUIL  : yes
ALIGN  : 8 double precision words

--------------------------------------------------------------------------------

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be               1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR02R2L2       16000   128     1     1              24.47              1.116e+02
HPL_pdgesv() start time Wed Sep  6 15:21:30 2017

HPL_pdgesv() end time   Wed Sep  6 15:21:55 2017

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0026303 ...... PASSED
================================================================================

Finished      1 tests with the following results:
              1 tests completed and passed residual checks,
              0 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------

End of Tests.
================================================================================
```

Figure 4.12 Free memory for Haswell machine

```
yanliu@haswell:~/research/benchmarks/hpl-2.1/bin/OpenBLAS$ free -m
              total        used        free      shared  buff/cache   available
Mem:           3750         138        3245          25         367        3326
Swap:             0           0           0
```

Figure 4.13 Main calculation code of HPL_pdgesv0

```
/*
 * Loop over the columns of A
 */
   for( j = 0; j < N; j += nb )
   {
      n = N - j; jb = Mmin( n, nb );
/*
 * Release panel resources - re-initialize panel data structure
 */
      (void) HPL_pdpanel_free( panel[0] );
      HPL_pdpanel_init( GRID, ALGO, n, n+1, jb, A, j, j, tag, panel[0] );
/*
 * Factor and broadcast current panel - update
 */
      HPL_pdfact(               panel[0] );
      (void) HPL_binit(         panel[0] );
      do
      { (void) HPL_bcast(       panel[0], &test ); }
      while( test != HPL_SUCCESS );
      (void) HPL_bwait(         panel[0] );
      HPL_pdupdate( NULL, NULL, panel[0], -1 );
/*
 * Update message id for next factorization
 */
      tag = MNxtMgid( tag, MSGID_BEGIN_FACT, MSGID_END_FACT );
   }
/*
 * Release panel resources and panel list
 */
   (void) HPL_pdpanel_disp( &panel[0] );

   if( panel ) free( panel );
/*
 * End of HPL_pdgesv0
 */
```

Figure 4.14 Part of HPL code with *PAPI_read ()*

```
/*
 * Release panel resources - re-initialize panel data structure
 */
      (void) HPL_pdpanel_free( panel[0] );
      HPL_pdpanel_init( GRID, ALGO, n, n+1, jb, A, j, j, tag, panel[0] );
/*
 /*start papi read*/
        if(num_iters<=max_iters)
       {
              totcyc=PAPI_get_real_cyc();
              PAPI_read(eventset,values);
              totcyc=PAPI_get_real_cyc()-totcyc;
              array[0][num_iters]=totcyc;
              eventVal[0][num_iters]=values[0]-lastVal;
              lastVal=values[0];
        }

    /*end papi read*/

 /* Factor and broadcast current panel - update
 */
      HPL_pdfact(                 panel[0] );
   /*start papi read*/
      if(num_iters<=max_iters)
     {
             totcyc=PAPI_get_real_cyc();
             PAPI_read(eventset,values);
             totcyc=PAPI_get_real_cyc()-totcyc;
             array[1][num_iters]=totcyc;
             eventVal[1][num_iters]=values[0]-lastVal;
             lastVal=values[0];
      }

    /*end papi read*/

      (void) HPL_binit(           panel[0] );
      /*start papi read*/
       if(num_iters<=max_iters)
      {
              totcyc=PAPI_get_real_cyc();
              PAPI_read(eventset,values);
              totcyc=PAPI_get_real_cyc()-totcyc;
              array[2][num_iters]=totcyc;
              eventVal[2][num_iters]=values[0]-lastVal;
              lastVal=values[0];
       }

    /*end papi read*/

      do
      { (void) HPL_bcast(         panel[0], &test ); }
      while( test != HPL_SUCCESS );
    /*start papi read*/
     if(num_iters<=max_iters)
     {
             totcyc=PAPI_get_real_cyc();
             PAPI_read(eventset,values);
             totcyc=PAPI_get_real_cyc()-totcyc;
             array[3][num_iters]=totcyc;
             eventVal[3][num_iters]=values[0]-lastVal;
             lastVal=values[0];
        }

    /*end papi read*/
```

Figure 4.15 Measuring user and kernel space event for *PAPI_read()*

```
 PAPI_start(eventset);

HPL_pdfact(               panel[0] );

rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read

// totcyc=PAPI_get_real_cyc();
PAPI_read(eventset,values);
//totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read

rawAfter1=rdpmc(USER_EVENT);
rawAfter2=rdpmc(KERNEL_EVENT);

PAPI_stop(eventset,values);
array[1][num_iters]=totcyc;
eventVal1[1][num_iters]=values[0]; // event value measured by papi_start/stop
eventVal2[1][num_iters]=values[1];
rawVal1[1][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
rawVal2[1][num_iters]=rawAfter2-rawBefore2;

 /*end papi read*/
```

4.4 STREAM benchmark

The STREAM benchmark [31] is used to test sustainable memory bandwidth (in MB/s) by measuring the performance of four long vector operations including COPY ($a (i) = b (i)$), SCALE ($a (i) = q*b (i)$), SUM ($a (i) = b (i) + c (i)$) and TRIAD ($a (i) = b (i) + q*c (i)$). The array size can be set to a size that is larger than the machine cache size.

4.4.1 STREAM benchmark setup

The STREAM benchmark is quite easy to setup since we run it on a uniprocessor. The first thing to do is to get the C code and the external timer code from the STREAM website [31]. Next, it is necessary to run the following compiling command to compile a standard-conforming version of STREAM.

gcc -O stream.c -o stream

After running the*./stream* command, the output will be similar to Figure 4.12.

With STREAM working properly, the *PAPI_read ()* function is added after each long vector operation in the stream code as shown in Figure 4.13. The vector operations are copy, scale, add and triad separately. We also add the *PAPI_read ()* function inside the operation of copy to create a better comparison with the *papi_cost* benchmark as shown in Figure 4.14. The patch of tuned STREAM code is listed in Appendix D. To compile the code instrumented with PAPI, run the following commands one after another:

gcc -O -c stream.c

gcc -o stream stream.o PAPIPATH/src/libpapi.a

Figure 4.16 STREAM output

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 13787 microseconds.
   (= 13787 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            7938.1     0.020170     0.020156     0.020189
Scale:           7888.4     0.020298     0.020283     0.020336
Add:             8944.9     0.026848     0.026831     0.026878
Triad:           8979.0     0.026778     0.026729     0.026809
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Figure 4.17 STREAM code with PAPI instrumented (*PAPI_read ()* after long vector operation)

```
/*start papi read*/
        if(num_iters<=max_iters)
                {
                        totcyc=PAPI_get_real_cyc();
                        PAPI_read(eventset,values);
                        totcyc=PAPI_get_real_cyc()-totcyc;
                        array[num_iters]=totcyc;
                        num_iters++;
        }

    /*end papi read*/
        times[0][k] = mysecond();
#ifdef TUNED
        tuned_STREAM_Copy();
#else
#pragma omp parallel for
        for (j=0; j<STREAM_ARRAY_SIZE; j++)
            c[j] = a[j];
#endif
        times[0][k] = mysecond() - times[0][k];

        times[1][k] = mysecond();
/*start papi read*/
        if(num_iters<=max_iters)
                {
                        totcyc=PAPI_get_real_cyc();
                        PAPI_read(eventset,values);
                        totcyc=PAPI_get_real_cyc()-totcyc;
                        array[num_iters]=totcyc;
                        num_iters++;
        }

    /*end papi read*/
```

Figure 4.18 STREAM code with PAPI instrumented (*PAPI_read ()* inside long vector operation)

```
for (j=0; j<STREAM_ARRAY_SIZE; j++)
    {
            c[j] = a[j];
           /*start papi read*/
           if(num_iters1<max_iters1)
           {
                   totcyc=PAPI_get_real_cyc();
                    PAPI_read(eventset,values);
                   totcyc=PAPI_get_real_cyc()-totcyc;
                   array1[num_iters1++]=totcyc;
        }
           /*End papi read*/
    }
```

5 RESULTS

We first compare the overhead results between existing PAPI and PAPI enhanced with our *rdpmc* changes across the fourteen x86 machines. Next we will show the results when several real world benchmarks are instrumented with PAPI, and read overhead with additional events is also described. Finally, there are the result of read overhead across several past PAPI versions and historical performance interfaces.

The results show that we can reduce overhead by at least three times (and more typically around six times) on a wide variety of x86 hardware. We instrumented the PAPI library code into several benchmarks and found that the more computer resources that the host code takes, the more overhead *PAPI_read ()* costs.

5.1 *read ()* vs *rdpmc* overhead

We test *read ()* vs *rdpmc* overhead on several benchmarks, including *papi_cost*, Sobel Edge Detector, HPL and STREAM. The results are reported in tables from Table 5.1 to Table 5.7.

Table 5.1 summarizes the *read ()* system call vs *rdpmc* instruction speedup found on the fourteen x86 machines when running *papi_cost* util. The values are cycles that are measured by the *PAPI_get_real_cyc ()* calls wrapping around the target code. In *papi_cost*, *PAPI_read ()* is called 1million times and we report the median value instead of the average as extremely large outliers may show up occasionally and skew the average and standard deviation. Among all the fourteen machines we test, the speedup is at least 2.5x, and is typically around 6x on recent Intel machines. This varies a bit more on AMD machines, and also on low-end machines such as the ATOM processors.

Table 5.2 Median *read* vs *rdpmc* speedup from *papi_cost* running the *PAPI_read* 1 million times

Vendor	Machine	*read ()* cycles	*rdpmc* cycles	Speedup
Intel	Pentium II	2533	384	6.6x
Intel	Pentium 4	3728	704	5.3x
Intel	Core 2	1634	199	8.2x
Intel	Atom	3906	392	10.0x
Intel	Ivybridge	885	149	5.9x
Intel	Haswell	1118	142	7.8x
Intel	Haswell-EP	820	125	6.6x
Intel	Broadwell	1030	145	7.1x
Intel	Broadwell-EP	750	118	6.4x
Intel	Skylake	942	144	6.5x
AMD	fam10h Phenom II	1252	205	6.1x
AMD	fam15h A10	2457	951	2.6x
AMD	fam15h Opteron	2186	644	3.4x
AMD	fam16h A8	1632	205	8.0x

Table 5.2 shows the comparison of *PAPI_read ()* cost supported by *read ()* system call and *rdpmc* instruction between *papi_cos*t and *Sobel-convolution*. We run *PAPI_read ()* after each iteration of sobel convolution operation (Figure 6) and calculate the total cost for *PAPI_read ()*for 1 million iterations. All *cycles* values are median value from the 1 million results. In the Sobel-convolution code, *PAPI_read ()* costs a bit more cycles than the plain *papi_cost* benchmark when using the *rdpmc* instruction, while the results are quite close to each other when *PAPI_read ()* is supported by the *read ()* system call. For the Sobel-convolution code, the *rdpmc* instruction shows a significant speedup (6.7x) compared to the *read ()* system call.

Table 5.3 Sobel compared with papi_cost on Haswell machine

Benchmark	*rdpmc* cycles	*read ()* cycles	speedup
papi_cost	142	1118	7.8x
Sobel-convolution	169	1139	6.7x

Table 5.4 HPL compared with *papi_cost* on Haswell machine

Benchmark	*rdpmc* cycles	*read ()* cycles	speedup
papi_cost(1million)	142	1118	7.8x
HPL_pdpanel_init	346	2277	6.6x
HPL_pdfact	3504	18076	5.2x
HPL_binit	371	2423	6.5x
HPL_bcast	383	2370	6.2x
HPL_bwait	453	2377	5.2x
HPL_pdupdate	4790	22309	4.6x
MNxtMgid	346	2603	7.5x

Table 5.3 summarizes the *PAPI_read ()* cost with the support of *rdpmc* instruction and the *read( )* system call among the *papi_cost* and several HPL functions in the HPL_pdgv0.c file. As explained in the HPL parameter tuning section, we set the problem size N to 16000 and the block to 128 so each function is called for 125 (16000/128) times. Table 4 shows the average values from 125 times, excluding *papi_cost* (1 million). For most HPL calls, *PAPI_read ()* is higher than in *papi_cost* but still in the same order (300-500 cycles). However, if one calls *PAPI_read ()* after *HPL_pdfact ()* and *HPL_pdupdate ()*, the cost is considerably higher than the plain *PAPI_read ()* call. It is likely this is due to cache misses caused by the intense workload. Table 5.4 gives average event value read by the *PAPI_read ()* for those HPL functions. We can see that for most events listed in Table 5.4, *HPL_pdfact ()* and *HPL_pdupdate ()* give much higher values than the others.

Table 5.5 shows userspace and kernel space event values for *PAPI_read ()* after each HPL function when the *PAPI_read ()* is instrumented with the *rdpmc* instruction and the *read ()* system call separately. We can see from Table 5.5 that for all three userspace events (total cycle,

L1 data cache misses and TLB data cache misses) caused by the *PAPI_read()* function that instrumented after *HPL_pdfact* and *HPL_pdupdtaed* is a relatively higher value than the others. Kernel space values are all 0. Table 5.6 shows similar values as in Table 5.5 for userspace events. However, kernel event values are not 0 anymore, which is reasonable since *PAPI_read ()* is instrumented by read system call when measuring data in Table 5.6.

Table 5.5 Event value for each HPL function

PAPI event	HPL_ pdpanel _init	HPL_pdfact	HPL_ binit	HPL_ bcast	HPL_ bwait	HPL_pdupdate	MNxtMgid
PAPI_L1_ICM	65	10721	11	8	2	17832	7
PAPI_L1_DCM	15	1753095	14	5	4	38874364	13
PAPI_TLB_IM	3	30	1	0	1	12	0
PAPI_TLB_DM	2	10590	7	3	3	299735	7
PAPI_BR_INS	162	1405652	68	71	67	5149713	65
PAPI_BR_MSP	3	5532	0	0	0	13987	0

Table 5.6 User and kernel space event cost by *PAPI_read ()* with *rdpmc* instruction

PAPI event	HPL_ pdpanel _init	HPL_pdfact	HPL_ binit	HPL_ bcast	HPL_ bwait	HPL_pdupdate	MNxtMgid
TOT_CYC(user)	512	4019	464	408	426	4813	420
TOT_CYC(kernel)	0	0	0	0	0	0	0
L1_DCM(user)	5	39	6	2	2	38	5
L1_DCM(kernel)	0	0	0	0	0	0	0
TLB_DM(user)	0	11	0	0	0	13	0
TLB_DM(kernel)	0	0	0	0	0	0	0

Table 5.7 User and kernel space event value cost by *PAPI_read ()* with *read ()* system call

PAPI event	HPL_ pdpanel _init	HPL_pdfact	HPL_ binit	HPL_ bcast	HPL_ bwait	HPL_pdupdate	MNxtMgid
TOT_CYC(user)	461	4551	383	379	396	4925	398
TOT_CYC(kernel)	1755	13545	1752	1741	1740	16024	1753
L1_DCM(user)	7	43	8	3	3	43	7
L1_DCM(kernel)	20	123	18	13	12	98	17
TLB_DM(user)	0	16	0	0	0	16	0
TLB_DM(kernel)	0	18	0	0	0	17	0

Table 5.8 STREAM comparing with papi_cost

Benchmark	*rdpmc* cycles	*read()* cycles	speedup
papi_cost(1 million)	142	1118	7.8x
element copy	142	1085	7.6x
vector copy	3974	19233	4.8x
vector scale	4191	19100	4.6x
vector add	3932	18855	4.8x
vector triad	4524	19281	4.3x

Table 5.7 describes *PAPI_read ()* cost in *papi_cost* and STREAM code. We insert a *PAPI_read ()* after each element copy operation for 1 million times as shown in Figure 4.14 and after each long vector (10M) operation for 1000 iterations as well. We can see from the table that the *read ()* system call and the *rdpmc* instructions give very similar result for both *papi_cost* and element copy. However, for vector operations, the result is much higher, but still shows significant speedup (around 4.5x). We assume there should be lots of cache misses and TLB misses for those operations and the results from Table 5.8 confirm our assumptions. Table 5.9 shows total Level 1 cache misses and TLB misses for STREAM when *PAPI_read () is* supported by

rdpmc and *read ()* system call respectively. Though the values are quite large, there is not much difference between using the *rdpmc* instruction and *read ()* system call in this case.

Table 5.9 Level 1 cache miss and TLB misses for each vector operation in STREAM

PAPI event	vector copy	vector scale	vector add	vector triad
PAPI_L1_DCM	2505104	2505348	3757675	3757657
PAPI_TLB_DM	25651	39855	27620	24158

Table 5.10 Total Level 1 cache miss and TLB miss for STREAM when *PAPI_read()* supported by *rdpmc* and *read()* separately

PAPI event	rdpmc	read
PAPI_L1_DCM	12534134630	12534099690
PAPI_TLB_DM	110624743	110819195

5.2 Additional Event Overhead

By default the *papi_cost* benchmark measures two events, *PAPI_TOT_INS*(total instructions) and *PAPI_TOT_CYC*(total cycles). They are typical events to measure, especially if you are interested in metrics such as Instruction per Cycle (IPC).

We slightly modify *papi_cost* to measure from one to four events and test the result on the Haswell machine. Figure 5.1 and Figure 5.2 show how the overhead increases with event number increases for both situations. Each event to be read via *rdpmc* must have an associated *mmap* page that can be mapped by the process. So, with *rdpmc*, each event needs to be read individually and it is reasonable to take more cycles for more events. The *read ()* interface can group multiple events that can be read with one single call. However, the kernel code still has to read the counters out one by one, so the time grows less linearly.

Figure 5.19 *PAPI_read ()* cost with *rdpmc* instruction varies with event number

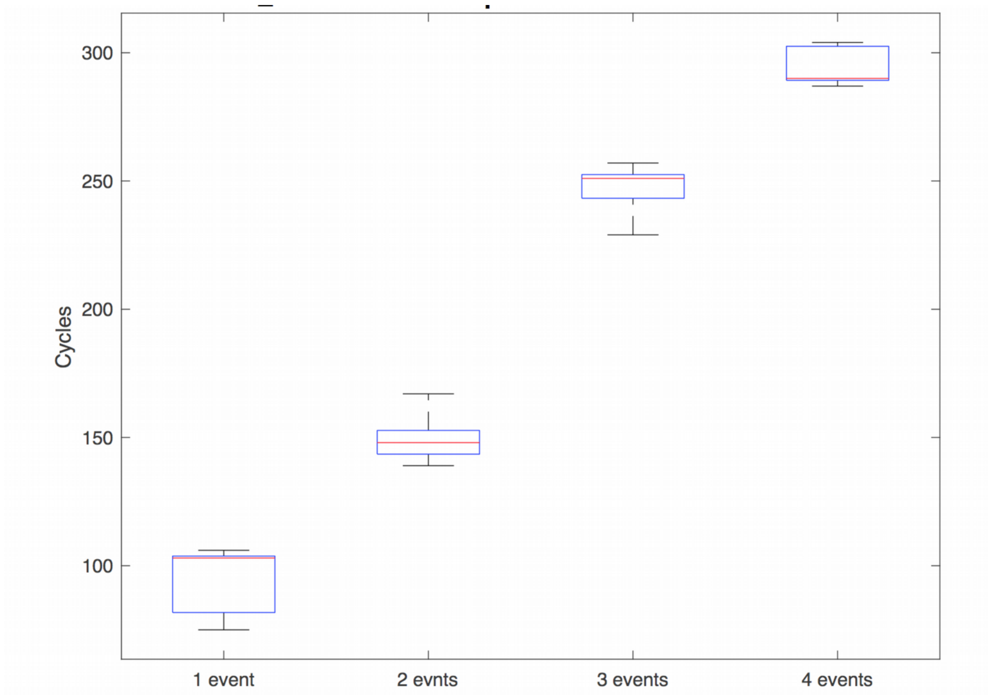
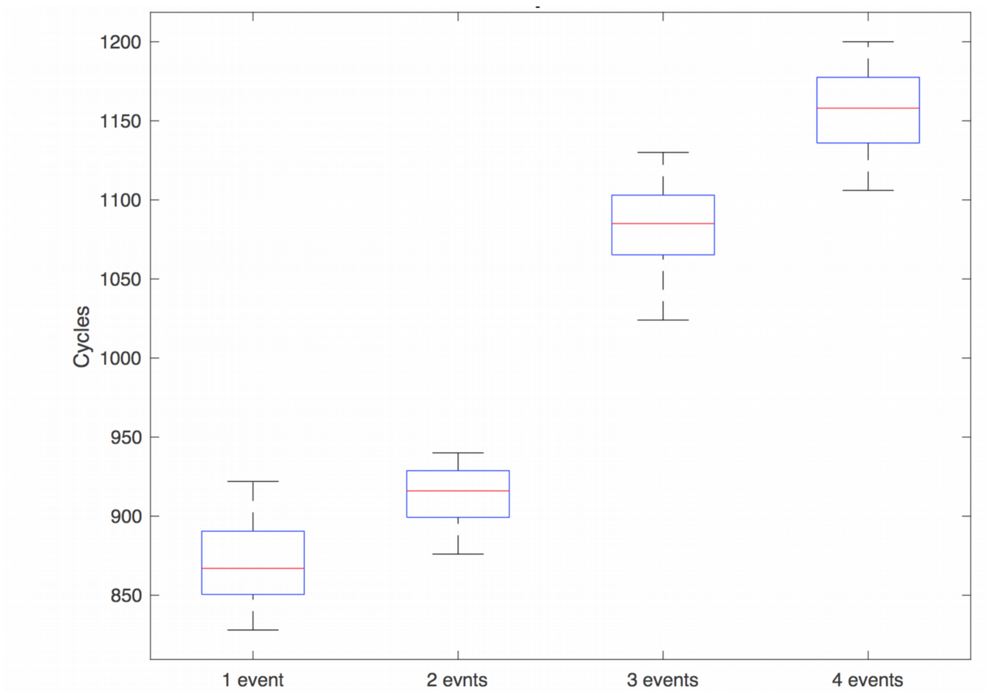


Figure 5.20 *PAPI_read ()* cost with *read ()* system call varies with event number



5.3 Read overhead by PAPI version

Figure 5.3 shows the *PAPI_read ()* overhead gathered for the past few PAPI releases. It can be seen that the overhead has not changed much in the recent past few version as they all use the *perf_event* interface. The figure reports how *PAPI_cost* varies with PAPI on Haswell machine, and results are similar on other machines.

5.4 Kernel update influence

To check if kernel changes pc->lock in rdpmc read code (see Figure 3.3 PAPI rdpmc code) when *per_event_update_userpage ()* is called, we add an index (*loopCount*) in the rdpmc code and increment it every time pc->lock changes and print a message to screen if it increases (Figure 5.4). When we run stream benchmark with the setting of *STREAM_ARRAY_SIZE=10M*, *NTIMS=1000*, the message of "*pc->lock changes*, *loopCount is 2*" happens occasionally.

5.5 PAPI_start/stop overhead

The PAPI_stop () function reads the current event value once to get a total result between the *PAPI_start ()* and the *PAPI_stop ()* section. We assume the change from the *read ()* system call to the *rdpmc* instruction could also speedup the *PAPI_start () / PAPI_stop ()* routines. Table 5.10 shows the speed up on the Haswell machine for the *papi_cost* benchmark, and results are similar. We slightly change the *papi_cost* code and make it to only perform start/stop test. By using the *strace* command, we find that the ioctl () call dominates the overhead (Figure 5.5). That could be why *rdpm*c doesn't speed up the *PAPI_start () / PAPI_stop ()* routine.

Table 5.11 *PAPI_start ()/PAPI_stop ()* speedup on Haswell machine

PAPI function	*rdpmc* cycles	*read()* cycles	speedup
PAPI_start/stop	7318	7604	1.03x

Figure 5.21 *papi_cost* varies with PAPI version

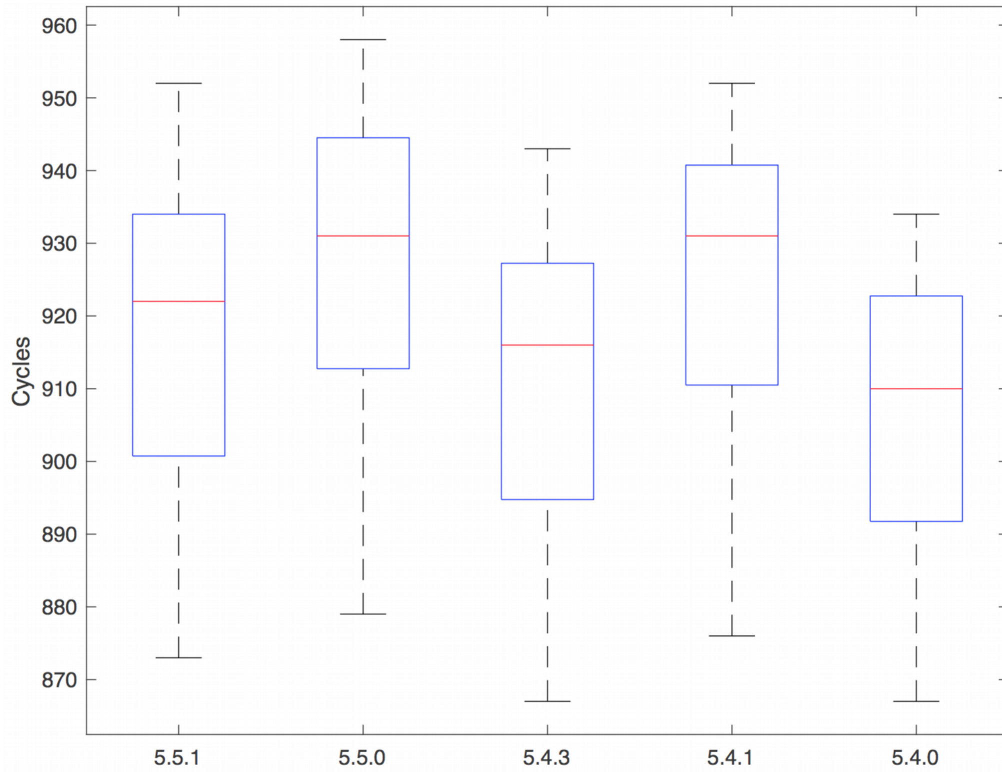


Figure 5.22 *rdpmc* read code with loopCount checking

```
uint32_t loopCount=0;
do {
        /* The kernel increments pc->lock any time */
        /* perf_event_update_userpage() is called */
        /* So by checking now, and the end, we */
        /* can see if an update happened while we */
        /* were trying to read things, and re-try */
        /* if something changed */
        /* The barrier ensures we get the most up to date */
        /* version of the pc->lock variable */
        loopCount++;
        seq=pc->lock;
        barrier();

        /* For multiplexing */
        /* time_enabled is time the event was enabled */
        enabled = pc->time_enabled;
        /* time_running is time the event was actually running */
        running = pc->time_running;

        /* if cap_user_time is set, we can use rdtsc */
        /* to calculate more exact enabled/running time */
        /* for more accurate multiplex calculations */
        if ( (pc->cap_user_time) && (enabled != running)) {
                cyc = rdtsc();
                time_offset = pc->time_offset;
                time_mult = pc->time_mult;
                time_shift = pc->time_shift;

                quot=(cyc>>time_shift);
                rem = cyc & (((uint64_t)1 << time_shift) - 1);
                delta = time_offset + (quot * time_mult) +
                        ((rem * time_mult) >> time_shift);
        }
        enabled+=delta;

        /* actually do the measurement */

        /* Index of register to read */
        /* 0 means stopped/not-active */
        /* Need to subtract 1 to get actual index to rdpmc() */
        index = pc->index;

        /* count is the value of the counter the last time */
        /* the kernel read it */
        /* If we don't sign extend it, we get large negative */
        /* numbers which break if an IOC_RESET is done */
        width = pc->pmc_width;
        count = pc->offset;
        count<<=(64-width);
        count>>=(64-width);

        /* Ugh, libpfm4 perf_event.h has cap_usr_rdpmc */
        /* while actual perf_event.h has cap_user_rdpmc */

        /* Only read if rdpmc enabled and event index valid */
        /* Otherwise return the older (out of date?) count value */
        if (pc->cap_usr_rdpmc && index) {

                /* Read counter value */
                pmc = rdpmc(index-1);

                /* sign extend result */
                pmc<<=(64-width);
                pmc>>=(64-width);

                /* add current count into the existing kernel count */
                count+=pmc;

                /* Only adjust if index is valid */
                running+=delta;
        }

        barrier();

} while (pc->lock != seq);

if (en) *en=enabled;
if (ru) *ru=running;
if(loopCount>1) printf("pc->lock changes, loopCount is %d\n",loopCount);
```

Figure 5.23 Output for *strace -c ./papi_cost*

```
yanliu@haswell:~/research/papioverhead/papi/papi/src/utils$ strace -c ./papi_cost
Cost of execution for PAPI start/stop, read and accum.
This test takes a while. Please be patient...

Performing loop latency test...

Total cost for loop latency over 1000000 iterations
min cycles   : 18
max cycles   : 32810
mean cycles  : 27.507005
std deviation: 55.076678

Performing start/stop test...

Total cost for PAPI_start/stop (2 counters) over 1000000 iterations
min cycles   : 145632
max cycles   : 1945210
mean cycles  : 171938.652270
std deviation: 7816.761506
 % time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 78.03   11.875804           3   4000011           ioctl
 21.96    3.342422           3   1000036           read
  0.00    0.000090           6        14           mmap
  0.00    0.000090          15         6           perf_event_open
  0.00    0.000087           4        23         7 access
  0.00    0.000077           5        15         1 open
  0.00    0.000075          25         3           munmap
  0.00    0.000074           4        18           close
  0.00    0.000035           2        19           write
  0.00    0.000026           3         8           mprotect
  0.00    0.000025           2        15           fstat
  0.00    0.000022          22         1           readlink
  0.00    0.000015           2         8           lseek
  0.00    0.000007           2         4           brk
  0.00    0.000005           2         3           getpid
  0.00    0.000002           2         1           uname
  0.00    0.000002           2         1           arch_prctl
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00   15.218858               5000187         8 total
```

6 CONCLUSION AND FUTURE WORK

We have added userspace (*rdpmc*) performance counter read support to the PAPI library and found that we can reduce overhead by at least three times (and more typically around six times) on a wide variety of x86 hardware. We instrument several benchmarks and found that the more computer resources that the host code takes, the more overhead *PAPI_read ()* costs. We also found that pc->lock value in *rdpmc* code may change occasionally.

We gained enough confidence that now PAPI enables the feature by default on all future versions of PAPI. PAPI will once again be able to gather low-overhead performance counter data, with overhead as low as the *perfctr* interface that was used for years before the introduction of the currently used Linux *perf_event*. We plan to investigate userspace read support on other architectures that support it, most notably the ARM and ARM64 architectures. Linux *perf_event* does not properly support this yet.

In addition to the *PAPI_read ()* routine, we plan to investigate other parts of the PAPI library code with the aim to decrease the overhead. The *ioctl ()* system call is one in particular intend to investigate.

REFERENCES

[1] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23, 1 (March 1995), 20-24.

[2] Philip Mucci. 2005. Performance Monitoring with PAPI.

http://www.drdobbs.com/tools/performance-monitoring-with-papi/184406109

[3]Browne, S., Dongarra, J., Garner, N., Ho, G. and Mucci, P., 2000. A portable programming interface for performance evaluation on modern processors. The international journal of high performance computing applications, 14(3), pp.189-204.

[4] Moore, S., Terpstra, D., London, K., Mucci, P., Teller, P., Salayandia, L., Bayona, A. and Nieto, M., 2003, June. PAPI deployment, evaluation, and extensions. In User Group Conference, 2003. Proceedings (pp. 349-353). IEEE.

[5] Treibig, J., Hager, G. and Wellein, G., 2010, September. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In Parallel Processing Workshops (ICPPW), 2010 39th International Conference on (pp. 207-216). IEEE.

[6] de Melo, A.C., 2009, September. Performance counters on Linux. In Linux Plumbers Conference.

[7] Weaver, V.M., 2015, March. Self-monitoring overhead of the Linux perf_ event performance counter interface. In Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on (pp. 102-111). IEEE.

[8] Moore, S.V., 2002, April. A comparison of counting and sampling modes of using performance monitoring hardware. In International Conference on Computational Science (pp. 904-912). Springer, Berlin, Heidelberg.

[9] Salayandía, L., 2002. A study of the validity and utility of PAPI performance counter data (Doctoral dissertation, University of Texas at El Paso).

[10] Mytkowicz, T., Diwan, A., Hauswirth, M. and Sweeney, P.F., 2007, March. Understanding measurement perturbation in trace-based data. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International (pp. 1-6). IEEE.

[11] Mytkowicz, T., Diwan, A., Hauswirth, M. and Sweeney, P., 2008, April. We have it easy, but do we have it right?. In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on (pp. 1-7). IEEE.

[12] London, K.S., Dongarra, J., Moore, S., Mucci, P., Seymour, K. and Spencer, T., 2001, August. End-user Tools for Application Performance Analysis Using Hardware Counters. In ISCA PDCS(pp. 460-465).

[13] Browne, S., Dongarra, J., Garner, N., Ho, G. and Mucci, P., 2000. A portable programming interface for performance evaluation on modern processors. The international journal of high performance computing applications, 14(3), pp.189-204.

[14] Weaver, V.M., 2013, April. Linux perf_event features and overhead. In The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath (Vol. 13).

[15] M. Pettersson. 1999. The Perfctr Interface. http://user.it.uu.se/ ∼mikpe/linux/perfctr/2.6/. (1999).

[16] Eranian, S., 2006, July. Perfmon2: a flexible performance monitoring interface for Linux. In Proc. of the 2006 Ottawa Linux Symposium (pp. 269-288).

[17] Maxwell, M., Teller, P., Salayandia, L. and Moore, S., 2002, October. Accuracy of performance monitoring hardware. In Proceedings of the Los Alamos Computer Science Institute Symposium (LACSI'02).

[18] Maxwell, M., Moore, S. and Teller, P., 2002, June. Efficiency and accuracy issues for sampling vs. counting modes of performance monitoring hardware. In Proceedings of the DoD High Performance Computing Modernization Program's User Group Conference.

[19] Moore, S.V., 2002, April. A comparison of counting and sampling modes of using performance monitoring hardware. In International Conference on Computational Science (pp. 904-912). Springer, Berlin, Heidelberg.

[20] Vincent Weaver. PAPI overhead on various kernel interfaces. http://web.eece.maine.edu/~vweaver/projects/papi-cost/

[21] LiMiT-Overview. http://castl.cs.columbia.edu/limit/

[22] Advanced Micro Devices 2010. Lightweight Profiling Specification. Advanced Micro Devices.

[23] Ingo Molnar. 2010. Basic support for LWP. http://marc.info/?l=linux-kernel&m=128630554614635.

[24] DeRose, L., 2001. The hardware performance monitor toolkit. Euro-Par 2001 Parallel Processing, pp.122-132.

[25] Lehr, J.P., 2016. Counting performance: hardware performance counter and compiler instrumentation. In GI-Jahrestagung (pp. 2187-2198).

[26] Huang, S., Lang, M., Pakin, S. and Fu, S., 2015, November. Measurement and characterization of haswell power and energy consumption. In Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing (p. 7). ACM.

[27] Babka, V. and Tuma, P., Effects of Memory Sharing on Contemporary Processor Architectures. MEMICS 2007, p.3.

[28] Zaparanuks, D., Jovic, M. and Hauswirth, M., 2009, April. Accuracy of performance counter measurements. In Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on (pp. 23-32). IEEE.

[29] Weaver, V. perf event use rdpmc rather than rdmsr when possible in kernel. https://lkml.org/lkml/2012/2/20/418 (2012)

[30] HPL (High Performance Linpack): Benchmarking Raspberry. https://www.howtoforge.com/tutorial/hpl-high-performance-linpack-benchmark-raspberry-pi/#introduction

[31] STREAM reference. http://www.cs.virginia.edu/stream/ref.html

[32] HPL reference. http://www.netlib.org/benchmark/hpl/faqs.html

[33] Standard Performance Evaluation Corporation. https://www.spec.org/cpu2006/

[34] Top500 Supercomputer Sites. https://www.top500.org/

APPENDICES

Appendix A: Code patch for papi_cost benchmark

```
--- papi_cost.c 2017-10-26 15:39:50.247793044 -0400
+++ cost1.c 2017-10-26 15:37:50.910831915 -0400
@@ -1,44 +1,39 @@
-/** file papi_cost.c
+/** file cost.c
  * @brief papi_cost utility.
  *    @page papi_cost
  * @section  NAME
- *       papi_cost - computes execution time costs for basic PAPI operations.
+ *       papi_cost - computes execution time costs for basic PAPI operations.
  *
  *    @section Synopsis
  *       papi_cost [-dhs] [-b bins] [-t threshold]
  *
  *    @section Description
- *       papi_cost is a PAPI utility program that computes the min / max / mean / std. deviation
- *       of execution times for PAPI start/stop pairs and for PAPI reads.
- *       This information provides the basic operating cost to a user's program
- *       for collecting hardware counter data.
+ *       papi_cost is a PAPI utility program that computes the min / max / mean / std. deviation
+ *       of execution times for PAPI start/stop pairs and for PAPI reads.
+ *       This information provides the basic operating cost to a user's program
+ *       for collecting hardware counter data.
  *       Command line options control display capabilities.
  *
```

```
  *    @section Options
  *    <ul>
- *       <li>-b < bins > Define the number of bins into which the results are
+ *       <li>-b < bins > Define the number of bins into which the results are
  *          partitioned for display. The default is 100.
  *       <li>-d  Display a graphical distribution of costs in a vertical histogram.
  *       <li>-h  Display help information about this utility.
- *       <li>-s  Show the number of iterations in each of the first 10
+ *       <li>-s  Show the number of iterations in each of the first 10
  *          standard deviations above the mean.
- *       <li>-t < threshold >    Set the threshold for the number of iterations to
+ *       <li>-t < threshold >    Set the threshold for the number of iterations to
  *          measure costs. The default is 100,000.
  *    </ul>
  *
  *    @section Bugs
- *       There are no known bugs in this utility. If you find a bug,
- *       it should be reported to the PAPI Mailing List at <ptools-perfapi@icl.utk.edu>.
+ *       There are no known bugs in this utility. If you find a bug,
+ *       it should be reported to the PAPI Mailing List at <ptools-perfapi@ptools.org>.
  */
-
-#include <stdio.h>
-#include <stdlib.h>
-#include <string.h>
-
-#include "papi.h"
+#include "papi_test.h"
 #include "cost_utils.h"
```

```
-int
+int
 find_derived( int i , char *type)
 {
   PAPI_event_info_t info;
@@ -106,7 +101,7 @@
    };
    printf( "\nTotal cost for %s over %d iterations\n", test[i], num_iters );
    printf
-      ( "min cycles   : %lld\nmax cycles   : %lld\nmean cycles  : %lf\nstd deviation: %lf\n ",
+      ( "min cycles   : %lld\nmax cycles   : %lld\nmean cycles  : %lf\nstd deviation: %lf\n",
         min, max, average, std );
 }

@@ -145,19 +140,59 @@
       printf( "\n" );
    }
 }
-
+static void
+print_percentile(long long percent25, long long percent50, long long percent75,long long percent99)
+{
+  printf
+     ("25the percentile    : %lld\n50th percentile    : %lld\n75th percentile    : %lld\n99th percentile  : %lld\n",
+  percent25,percent50,percent75,percent99);
+}
 static void
 do_output( int test_type, long long *array, int bins, int show_std_dev,
```

```
-         int show_dist )
+                 int show_dist )
+{
+        int s[11];
+        long long min, max;
+        double average, std;
+        long long percent25,percent50,percent75,percent99;
+        std = do_stats( array, &min, &max, &average);
+        print_stats( test_type, min, max, average, std );
+
+        do_percentile(array,&percent25,&percent50,&percent75,&percent99);
+
+        print_percentile(percent25,percent50,percent75,percent99);
+        if ( show_std_dev ) {
+                do_std_dev( array, s, std, average );
+                print_std_dev( s );
+        }
+
+        if ( show_dist ) {
+                int *d;
+                d = calloc( bins , sizeof ( int ) );
+                do_dist( array, min, max, bins, d );
+                print_dist( min, max, bins, d );
+                free( d );
+        }
+}
+
+static void // add by YAN Liu for rdpmc outliers
+do_output_eventvalue( int test_type, long long *array, int bins, int show_std_dev,
+          int show_dist,long long*eventarray )
```

```
{
   int s[11];
   long long min, max;
   double average, std;
+  long long percent25,percent50,percent75,percent99;
+  long long min_i,max_i;
+  std = do_stats_eventvalue( array, &min, &max, &average,&min_i,&max_i);
+  print_stats( test_type, min, max, average, std );

-  std = do_stats( array, &min, &max, &average );
+  do_percentile(array,&percent25,&percent50,&percent75,&percent99);

-  print_stats( test_type, min, max, average, std );
+  print_percentile(percent25,percent50,percent75,percent99);

+          printf("\n    eventvalue    for    min(%lld)    which    located    at    array[%d]:
%lld",min,min_i,eventarray[min_i]);
+        printf("\n    eventvalue    for    max(%lld)    which    located    at    array[%d]:      %lld
\n",max,max_i,eventarray[max_i]);// add by YAN LIU for rdpmc outliers
   if ( show_std_dev ) {
      do_std_dev( array, s, std, average );
      print_std_dev( s );
@@ -179,11 +214,17 @@
   int i, retval, EventSet = PAPI_NULL;
   int retval_start,retval_stop;
   int bins = 100;
-  int show_dist = 0, show_std_dev = 0;
-  long long totcyc, values[2];
+  int show_dist = 0, show_std_dev = 0;
+  long long totcyc, values[4];
```

```
   long long *array;
   int event;
   PAPI_event_info_t info;
+  long long *eventarray;
+      /*add by yanliu for event recording*/
+  eventarray =( long long * ) malloc( ( size_t ) num_iters * sizeof ( long long ) );
+  if ( eventarray == NULL ){printf("eventarray malloc failed\n"); return -1;};
+
+  tests_quiet( argc, argv );  /* Set TESTS_QUIET variable */

   for ( i = 1; i < argc; i++ ) {
      if ( !strcmp( argv[i], "-b" ) ) {
@@ -218,63 +259,41 @@
   printf( "Cost of execution for PAPI start/stop, read and accum.\n" );
   printf( "This test takes a while. Please be patient...\n" );

-  retval = PAPI_library_init( PAPI_VER_CURRENT );
-  if (retval != PAPI_VER_CURRENT ) {
-     fprintf(stderr,"PAPI_library_init\n");
-     exit(retval);
-  }
-  retval = PAPI_set_debug( PAPI_VERB_ECONT );
-  if (retval != PAPI_OK ) {
-     fprintf(stderr,"PAPI_set_debug\n");
-     exit(retval);
-  }
-  retval = PAPI_query_event( PAPI_TOT_CYC );
-  if (retval != PAPI_OK ) {
-     fprintf(stderr,"PAPI_query_event\n");
-     exit(retval);
```

```
- }
- retval = PAPI_query_event( PAPI_TOT_INS );
- if (retval != PAPI_OK ) {
-     fprintf(stderr,"PAPI_query_event\n");
-     exit(retval);
- }
- retval = PAPI_create_eventset( &EventSet );
- if (retval != PAPI_OK ) {
-     fprintf(stderr,"PAPI_create_eventset\n");
-     exit(retval);
- }
- retval = PAPI_add_event( EventSet, PAPI_TOT_CYC );
- if (retval != PAPI_OK ) {
-     fprintf(stderr,"PAPI_add_event\n");
-     exit(retval);
- }
- retval = PAPI_add_event( EventSet, PAPI_TOT_INS );
- if (retval != PAPI_OK ) {
-     retval = PAPI_add_event( EventSet, PAPI_TOT_IIS );
-     if (retval != PAPI_OK ) {
-         fprintf(stderr,"PAPI_add_event\n");
-         exit(retval);
-     }
- }
+ if ( ( retval =
+         PAPI_library_init( PAPI_VER_CURRENT ) ) != PAPI_VER_CURRENT )
+     test_fail( __FILE__, __LINE__, "PAPI_library_init", retval );
+ if ( ( retval = PAPI_set_debug( PAPI_VERB_ECONT ) ) != PAPI_OK )
+     test_fail( __FILE__, __LINE__, "PAPI_set_debug", retval );
+ if ( ( retval = PAPI_query_event( PAPI_TOT_CYC ) ) != PAPI_OK )
```

```
+      test_fail( __FILE__, __LINE__, "PAPI_query_event", retval );
+   if ( ( retval = PAPI_query_event( PAPI_TOT_INS ) ) != PAPI_OK )
+      test_fail( __FILE__, __LINE__, "PAPI_query_event", retval );
+   if ( ( retval = PAPI_create_eventset( &EventSet ) ) != PAPI_OK )
+      test_fail( __FILE__, __LINE__, "PAPI_create_eventset", retval );
+
+// if ( ( retval = PAPI_add_event( EventSet, PAPI_L3_TCM ) ) != PAPI_OK ) //add by YAN Liu for
rdpmc outliers
+//     test_fail( __FILE__, __LINE__, "PAPI_add_event", retval );
+
+   if ( ( retval = PAPI_add_event( EventSet, PAPI_TOT_CYC ) ) != PAPI_OK )
+      test_fail( __FILE__, __LINE__, "PAPI_add_event", retval );
+
+   if ( ( retval = PAPI_add_event( EventSet, PAPI_TOT_INS ) ) != PAPI_OK )
+      if ( ( retval = PAPI_add_event( EventSet, PAPI_TOT_IIS ) ) != PAPI_OK )
+         test_fail( __FILE__, __LINE__, "PAPI_add_event", retval );
+

    /* Make sure no errors and warm up */

    totcyc = PAPI_get_real_cyc(  );
-   if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK ) {
-      fprintf(stderr,"PAPI_start");
-      exit(retval);
-   }
-   if ( ( retval = PAPI_stop( EventSet, NULL ) ) != PAPI_OK ) {
-      fprintf(stderr,"PAPI_stop");
-      exit(retval);
-   }
+   if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK )
```

```
+     test_fail( __FILE__, __LINE__, "PAPI_start", retval );
+  if ( ( retval = PAPI_stop( EventSet, NULL ) ) != PAPI_OK )
+     test_fail( __FILE__, __LINE__, "PAPI_stop", retval );

   array =
      ( long long * ) malloc( ( size_t ) num_iters * sizeof ( long long ) );
-  if ( array == NULL ) {
-     fprintf(stderr,"PAPI_stop");
-     exit(retval);
-  }
+  if ( array == NULL )
+     test_fail( __FILE__, __LINE__, "PAPI_stop", retval );

   /* Determine clock latency */

@@ -299,8 +318,7 @@
      totcyc = PAPI_get_real_cyc(  ) - totcyc;
      array[i] = totcyc;
      if (retval_start || retval_stop) {
-        fprintf(stderr,"PAPI start/stop\n");
-         exit(retval_start );
+        test_fail( __FILE__, __LINE__, "PAPI start/stop", retval_start );
      }
   }

@@ -309,10 +327,8 @@
   /* Start the read eval */
   printf( "\nPerforming read test...\n" );

-  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK ) {
```

```
-      fprintf(stderr,"PAPI_start");
-      exit(retval);
-  }
+  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK )
+      test_fail( __FILE__, __LINE__, "PAPI_start", retval );
   PAPI_read( EventSet, values );

   for ( i = 0; i < num_iters; i++ ) {
@@ -320,30 +336,27 @@
       PAPI_read( EventSet, values );
       totcyc = PAPI_get_real_cyc(  ) - totcyc;
       array[i] = totcyc;
+      eventarray[i]=values[0];// add by YAN LIU for rdpmc outliers
+
   }
-  if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK ) {
-      fprintf(stderr,"PAPI_stop");
-      exit(retval);
-  }
+  if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK )
+      test_fail( __FILE__, __LINE__, "PAPI_stop", retval );

-  do_output( 2, array, bins, show_std_dev, show_dist );
+   do_output_eventvalue( 2, array, bins, show_std_dev, show_dist,eventarray );// add by YAN
LIU for rdpmc outliers
+// do_output( 2, array, bins, show_std_dev, show_dist );

   /* Start the read with timestamp eval */
   printf( "\nPerforming read with timestamp test...\n" );
```

```
-  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK ) {
-     fprintf(stderr,"PAPI_start");
-     exit(retval);
-  }
+  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK )
+     test_fail( __FILE__, __LINE__, "PAPI_start", retval );
   PAPI_read_ts( EventSet, values, &totcyc );

   for ( i = 0; i < num_iters; i++ ) {
      PAPI_read_ts( EventSet, values, &array[i] );
   }
-  if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK ) {
-     fprintf(stderr,"PAPI_stop");
-     exit(retval);
-  }
+  if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK )
+     test_fail( __FILE__, __LINE__, "PAPI_stop", retval );

   /* post-process the timing array */
   for ( i = num_iters - 1; i > 0; i-- ) {
@@ -356,10 +369,8 @@
   /* Start the accum eval */
   printf( "\nPerforming accum test...\n" );

-  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK ) {
-     fprintf(stderr,"PAPI_start");
-     exit(retval);
-  }
+  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK )
+     test_fail( __FILE__, __LINE__, "PAPI_start", retval );
```

```
   PAPI_accum( EventSet, values );

   for ( i = 0; i < num_iters; i++ ) {
@@ -368,20 +379,16 @@
      totcyc = PAPI_get_real_cyc(  ) - totcyc;
      array[i] = totcyc;
   }
-  if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK ) {
-     fprintf(stderr,"PAPI_stop");
-     exit(retval);
-  }
+  if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK )
+     test_fail( __FILE__, __LINE__, "PAPI_stop", retval );

   do_output( 4, array, bins, show_std_dev, show_dist );

   /* Start the reset eval */
   printf( "\nPerforming reset test...\n" );

-  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK ) {
-     fprintf(stderr,"PAPI_start");
-     exit(retval);
-  }
+  if ( ( retval = PAPI_start( EventSet ) ) != PAPI_OK )
+     test_fail( __FILE__, __LINE__, "PAPI_start", retval );

   for ( i = 0; i < num_iters; i++ ) {
      totcyc = PAPI_get_real_cyc(  );
@@ -389,10 +396,8 @@
      totcyc = PAPI_get_real_cyc(  ) - totcyc;
```

```
        array[i] = totcyc;
    }
-   if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK ) {
-       fprintf(stderr,"PAPI_stop");
-       exit(retval);
-   }
+   if ( ( retval = PAPI_stop( EventSet, values ) ) != PAPI_OK )
+       test_fail( __FILE__, __LINE__, "PAPI_stop", retval );

    do_output( 5, array, bins, show_std_dev, show_dist );

@@ -411,14 +416,12 @@

        retval = PAPI_add_event( EventSet, event);
        if ( retval != PAPI_OK ) {
-           fprintf(stderr,"PAPI_add_event");
-           exit(retval);
+           test_fail(__FILE__, __LINE__, "PAPI_add_event", retval);
        }

        retval = PAPI_start( EventSet );
        if ( retval != PAPI_OK ) {
-           fprintf(stderr,"PAPI_start");
-           exit(retval);
+           test_fail( __FILE__, __LINE__, "PAPI_start", retval );
        }

        PAPI_read( EventSet, values );
@@ -432,8 +435,7 @@
```

```
        retval = PAPI_stop( EventSet, values );
        if ( retval != PAPI_OK ) {
-           fprintf(stderr,"PAPI_stop");
-           exit(retval);
+           test_fail( __FILE__, __LINE__, "PAPI_stop", retval );
        }

        do_output( 6, array, bins, show_std_dev, show_dist );
@@ -458,14 +460,12 @@

        retval = PAPI_add_event( EventSet, event);
        if ( retval != PAPI_OK ) {
-           fprintf(stderr,"PAPI_add_event\n");
-           exit(retval);
+           test_fail(__FILE__, __LINE__, "PAPI_add_event", retval);
        }

        retval = PAPI_start( EventSet );
        if ( retval != PAPI_OK ) {
-           fprintf(stderr,"PAPI_start");
-           exit(retval);
+           test_fail( __FILE__, __LINE__, "PAPI_start", retval );
        }

        PAPI_read( EventSet, values );
@@ -479,8 +479,7 @@

        retval = PAPI_stop( EventSet, values );
        if ( retval != PAPI_OK ) {
-           fprintf(stderr,"PAPI_stop");
```

```
-         exit(retval);
+         test_fail( __FILE__, __LINE__, "PAPI_stop", retval );
      }

      do_output( 7, array, bins, show_std_dev, show_dist );
@@ -490,6 +489,6 @@
   }

   free( array );
-
-  return 0;
+  test_pass( __FILE__, NULL, 0 );
+  exit( 1 );
 }



--- cost_utils.c   2017-10-26 15:39:11.775482486 -0400
+++ cost_utils1.c  2017-10-26 15:38:34.083178869 -0400
@@ -1,10 +1,6 @@
-#include <stdio.h>
-#include <math.h>
-
-#define NUM_ITERS  1000000
+#include "papi_test.h"

 int num_iters = NUM_ITERS;
-
 /* computes min, max, and mean for an array; returns std deviation */
 double
```

```
 do_stats( long long *array, long long *min, long long *max, double *average )
@@ -17,7 +13,7 @@
   for ( i = 0; i < num_iters; i++ ) {
       *average += ( double ) array[i];
       if ( *min > array[i] )
-          *min = array[i];
+            *min = array[i];
       if ( *max < array[i] )
           *max = array[i];
   }
@@ -76,3 +72,84 @@
   }
 }

+int partition( long long a[], int l, int r) {
+  long  pivot, i, j, t;
+  pivot = a[l];
+  i = l; j = r+1;
+
+  while( 1)
+  {
+      do ++i; while( a[i] < pivot && i < r );
+      do --j; while( a[j] > pivot );
+      if( i >= j ) break;
+      t = a[i]; a[i] = a[j]; a[j] = t;
+  }
+  t = a[l]; a[l] = a[j]; a[j] = t;
+  return j;
+}
+void quickSort( long long a[], int l, int r)
```

```
+{
+  int j;
+
+  if( l < r )
+  {
+     // divide and conquer
+      j = partition( a, l, r);
+     quickSort( a, l, j-1);
+     quickSort( a, j+1, r);
+  }
+
+}
+
+void do_percentile(long long *a, long long *percent25, long long *percent50, long long
*percent75,long long *percent99)
+{
+  long long *a_sort;
+  a_sort = calloc(num_iters,sizeof(long long));
+  memcpy(a_sort,a,num_iters*sizeof(long long));
+  int i_25=(int)num_iters/4;
+  int i_50=(int)num_iters/2;
+   int i_75=(int)num_iters/4*3; // index for  75%, not quite accurate cause didn't take even or
odd in consideratio
+  int i_99=(int)num_iters/10*9.9;
+  quickSort(a_sort,0,num_iters-1);
+
+
+  *percent25=a_sort[i_25];
+  *percent50=a_sort[i_50];
+  *percent75=a_sort[i_75];
```

```
+  *percent99=a_sort[i_99];
+  free(a_sort);
+  a_sort=NULL;
+
+}
+double // add by YAN Liu for rdpmc outliers
+do_stats_eventvalue( long long *array, long long *min, long long *max, double *average,long
long *min_i,long long *max_i )
+{
+       int i;
+       double std, tmp;
+
+       *min = *max = array[0];
+       *average = 0;
+       for ( i = 0; i < num_iters; i++ ) {
+               *average += ( double ) array[i];
+               if ( *min > array[i] )
+               {
+                       *min = array[i];
+                       *min_i=i;
+               }
+               if ( *max < array[i] )
+               {
+                       *max = array[i];
+                       *max_i=i;
+               }
+       }
+       *average = *average / ( double ) num_iters;
+       std = 0;
+       for ( i = 0; i < num_iters; i++ ) {
```

```
+           tmp = ( double ) array[i] - ( *average );
+           std += tmp * tmp;
+       }
+       std = sqrt( std / ( num_iters - 1 ) );
+       return ( std );
+}
+
+

--- cost_utils.h    2017-10-26 15:39:19.167542110 -0400
+++ cost_utils1.h   2017-10-26 15:38:23.867096692 -0400
@@ -5,5 +5,6 @@
 extern double  do_stats(long long*, long long*, long long *, double *);
 extern void       do_std_dev( long long*, int*, double, double );
 extern void       do_dist( long long*, long long, long long, int, int*);
+extern double    do_stats_eventvalue(long long*, long long*, long long *, double *,long long*,long long*);

 #endif /* __PAPI_COST_UTILS_H__ */
```

Appendix B: Sobel code

```
/* sobel benchmark for Papi overhead measurement */
/* modifed based on ece574 example code  */
/*Yan Liu*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <errno.h>
#include <math.h>

#include <jpeglib.h>
//#include "../jpeg-6b/jpeglib.h"
#include  "../papi/papi/src/papi.h"
#define  MAX 1000000

/* Filters */
staticint sobel_x_filter[3][3]={{-1,0,+1},{-2,0,+2},{-1,0,+1}};
staticint sobel_y_filter[3][3]={{-1,-2,-1},{0,0,0},{1,2,+1}};

/* Structure describing the image */
struct image_t {
int x;
int y;
int depth;/* bytes */
unsignedchar*pixels;
};
```

```
struct convolve_data_t {
struct image_t *old;
struct image_t *new;
int(*filter)[3][3];
int papiEventset;
};
int partition(longlong a[],int l,int r){
long  pivot, i, j, t;
  pivot = a[l];
  i = l; j = r+1;

while(1)
{
do++i;while( a[i]< pivot && i < r );
do--j;while( a[j]> pivot );
if( i >= j )break;
       t = a[i]; a[i]= a[j]; a[j]= t;
}
  t = a[l]; a[l]= a[j]; a[j]= t;
return j;
}
void quickSort(longlong a[],int l,int r)
{
int j;

if( l < r )
{
// divide and conquer
       j = partition( a, l, r);
      quickSort( a, l, j-1);
```

```
    quickSort( a, j+1, r);
}

}
void do_percentile(int num_iters,longlong*a,longlong*percent25,longlong*percent50,longlong*percent75,longlong*percent99)
{
longlong*a_sort;
    a_sort = calloc(num_iters,sizeof(longlong));
    memcpy(a_sort,a,num_iters*sizeof(longlong));
int i_25=(int)num_iters/4;
int i_50=(int)num_iters/2;
int i_75=(int)num_iters/4*3;// index for  75%, not quite accurate cause didn't take even or odd in consideratio
int i_99=(int)num_iters/10*9.9;
    quickSort(a_sort,0,num_iters-1);


*percent25=a_sort[i_25];
*percent50=a_sort[i_50];
*percent75=a_sort[i_75];
*percent99=a_sort[i_99];
    free(a_sort);
    a_sort=NULL;

}
staticvoid print_percentile(longlong percent25,longlong percent50,longlong percent75,longlong percent99)
```

```
{
        printf
("25%% cycles   : %lld\n50%% cycles   : %lld\n75%% cycles   : %lld\n99%% cycles   : %lld\n",
        percent25,percent50,percent75,percent99);
}

staticvoid cal_stats(int num_iters,longlong*array,longlong*min,
longlong*max,double*avag,double*std){
int i;double tmp;
longlong percent25, percent50, percent75, percent99;
*max=*min=array[0];
for(i=0;i<num_iters;i++){
if(*max<array[i])*max=array[i];
if(*min>array[i])*min=array[i];
*avag+=(double)array[i];
//      printf("%d-%lld \n",i,array[i]);

}
*avag=*avag/(double)num_iters;
for( i =0; i < num_iters; i++){
                tmp =(double)array[i]-(*avag );
*std += tmp * tmp;
}
*std=sqrt(*std/num_iters);
    do_percentile(num_iters,array,&percent25,&percent50,&percent75,&percent99);

        print_percentile(percent25,percent50,percent75,percent99);
}
```

```
/* very inefficient convolve code */
staticvoid*generic_convolve(void*argument){

int x,y,k,l,d;
int32_t color;
int sum,depth,width,height;
longlong totcyc,*array,values[3];
struct image_t *old;
struct image_t *new;
int(*filter)[3][3];
struct convolve_data_t *data;
int eventset;
int result;
longlong min,max;
double avag,std;
int num_iters;
/* Convert from void pointer to the actual data type */
  data=(struct convolve_data_t *)argument;
  old=data->old;
  new=data->new;
  filter=data->filter;
    eventset=data->papiEventset;
  depth=old->depth;
  width=old->x*old->depth;
  height=old->y;
  min=max=0;
  avag=std=0;
    array=(longlong*)malloc(width*height*sizeof(longlong));
    num_iters=0;
//PAPI start
```

```
//if((result=PAPI_start(eventset))!=PAPI_OK)
//  printf("Error PAPI start:%s\n",PAPI_strerror(result));
//  PAPI_read(eventset,values);
for(d=0;d<3;d++){
for(x=0;x<old->x;x++){
for(y=0;y<old->y;y++){

    sum=0;
for(k=-1;k<2;k++){
for(l=-1;l<2;l++){
      color=old->pixels[((y+l)*width)+(x*depth+d+k*depth)];
      sum+=color *(*filter)[k+1][l+1];
}
}

if(sum<0) sum=0;
if(sum>255) sum=255;

    new->pixels[(y*width)+x*depth+d]=sum;
if(num_iters<=MAX)
{
      totcyc=PAPI_get_real_cyc();
      PAPI_read(eventset,values);
        totcyc=PAPI_get_real_cyc()-totcyc;
      array[num_iters]=totcyc;
      num_iters++;
}
}
}
}
```

```
if((result=PAPI_stop(eventset,values))!=PAPI_OK)
    printf("Error PAPI stop:%s\n",PAPI_strerror(result));
  printf("Total cost for PAPI_read over %d iterations \n",--num_iters);
  cal_stats(num_iters,array,&min,&max,&avag,&std);
  printf("min cycles : %lld\nmax cycles : %lld\naverage cycles : %lf\nstandard deviation: %lf\n",
    min,max,avag,std);

returnNULL;
}



staticint load_jpeg(char*filename,struct image_t *image){

FILE*fff;
struct jpeg_decompress_struct cinfo;
struct jpeg_error_mgr jerr;
  JSAMPROW output_data;
unsignedint scanline_len;
int scanline_count=0;

  fff=fopen(filename,"rb");
if(fff==NULL){
    fprintf(stderr,"Could not load %s: %s\n",
      filename, strerror(errno));
return-1;
}

/* set up jpeg error routines */
  cinfo.err = jpeg_std_error(&jerr);
```

```
/* Initialize cinfo */
   jpeg_create_decompress(&cinfo);

/* Set input file */
   jpeg_stdio_src(&cinfo, fff);

/* read header */
   jpeg_read_header(&cinfo, TRUE);

/* Start decompressor */
   jpeg_start_decompress(&cinfo);

   printf("output_width=%d, output_height=%d, output_components=%d\n",
      cinfo.output_width,
      cinfo.output_height,
      cinfo.output_components);

   image->x=cinfo.output_width;
   image->y=cinfo.output_height;
   image->depth=cinfo.output_components;

   scanline_len = cinfo.output_width * cinfo.output_components;
   image->pixels=malloc(cinfo.output_width * cinfo.output_height * cinfo.output_components);

while(scanline_count < cinfo.output_height){
      output_data =(image->pixels +(scanline_count * scanline_len));
      jpeg_read_scanlines(&cinfo,&output_data,1);
      scanline_count++;
}
```

```
/* Finish decompressing */
   jpeg_finish_decompress(&cinfo);

   jpeg_destroy_decompress(&cinfo);

   fclose(fff);

return0;
}

staticint store_jpeg(char*filename,struct image_t *image){

struct jpeg_compress_struct cinfo;
struct jpeg_error_mgr jerr;
int quality=90;/* % */
int i;

FILE*fff;

   JSAMPROW row_pointer[1];
int row_stride;

/* setup error handler */
   cinfo.err = jpeg_std_error(&jerr);

/* initialize jpeg compression object */
   jpeg_create_compress(&cinfo);
```

```
/* Open file */
  fff = fopen(filename,"wb");
if(fff==NULL){
    fprintf(stderr,"can't open %s: %s\n",
      filename,strerror(errno));
return-1;
}

  jpeg_stdio_dest(&cinfo, fff);

/* Set compression parameters */
  cinfo.image_width = image->x;
  cinfo.image_height = image->y;
  cinfo.input_components = image->depth;
  cinfo.in_color_space = JCS_RGB;
  jpeg_set_defaults(&cinfo);
  jpeg_set_quality(&cinfo, quality, TRUE);

/* start compressing */
  jpeg_start_compress(&cinfo, TRUE);

  row_stride=image->x*image->depth;

for(i=0;i<image->y;i++){
    row_pointer[0]=& image->pixels[i * row_stride];
    jpeg_write_scanlines(&cinfo, row_pointer,1);
}

/* finish compressing */
  jpeg_finish_compress(&cinfo);
```

```
/* close file */
   fclose(fff);

/* clean up */
   jpeg_destroy_compress(&cinfo);

return0;
}

staticint combine(struct image_t *s_x,
struct image_t *s_y,
struct image_t *new){
int i;
int out;

for(i=0;i<( s_x->depth * s_x->x * s_x->y );i++){

      out=sqrt(
(s_x->pixels[i]*s_x->pixels[i])+
(s_y->pixels[i]*s_y->pixels[i])
);
if(out>255) out=255;
if(out<0) out=0;
      new->pixels[i]=out;
}

return0;
}
```

```
int main(int argc,char**argv){

struct image_t image,sobel_x,sobel_y,new_image;
struct convolve_data_t sobel_data[2];
longlong time_start,time_end,time_total,values[3];//for PAPI time measuring

/* Check command line usage */
if(argc<2){
      fprintf(stderr,"Usage: %s image_file\n",argv[0]);
return-1;
}
int eventset=PAPI_NULL;
int  result;

if((result=PAPI_library_init(PAPI_VER_CURRENT))!=PAPI_VER_CURRENT)
       printf("Error PAPI library init:%s\n",PAPI_strerror(result));
/*create papi eventset and add events*/
if((result=PAPI_create_eventset(&eventset))!=PAPI_OK)
       printf("Error PAPI create event:%s\n",PAPI_strerror(result));

/*  if((result=PAPI_add_named_event(eventset,"PAPI_L1_TCM"))!=PAPI_OK)
   printf("Error PAPI add event:%s\n",PAPI_strerror(result));
   if((result=PAPI_add_named_event(eventset,"PAPI_L2_TCM"))!=PAPI_OK)
   printf("Error PAPI add event:%s\n",PAPI_strerror(result));
*/if((result=PAPI_add_named_event(eventset,"BR_MISP_RETIRED:CONDITIONAL:k=1"))!
=PAPI_OK)
      printf("Error PAPI add event1:%s\n",PAPI_strerror(result));
if((result=PAPI_add_named_event(eventset,"BR_MISP_RETIRED:CONDITIONAL:u=1"))!
=PAPI_OK)
      printf("Error PAPI add event2:%s\n",PAPI_strerror(result));
```

```
//if((result=PAPI_add_named_event(eventset,"INSTRUCTIONS_RETIRED:k=1:u=1"))!=PAPI_OK)
//  printf("Error PAPI add event3:%s\n",PAPI_strerror(result));
if((result=PAPI_add_named_event(eventset,"PAPI_BR_MSP"))!=PAPI_OK)
        printf("Error PAPI add event4:%s\n",PAPI_strerror(result));



   PAPI_start(eventset);
//  time_start=PAPI_get_real_usec();



/* Load an image */
   load_jpeg(argv[1],&image);

   PAPI_stop(eventset,values);
//printf("L1 load jpeg %lld\n",values[0]);
//printf("L2 load jpeg%lld\n",values[1]);

   PAPI_start(eventset);
/* Allocate space for output image */
   new_image.x=image.x;
   new_image.y=image.y;
   new_image.depth=image.depth;
   new_image.pixels=malloc(image.x*image.y*image.depth*sizeof(char));

/* Allocate space for output image */
   sobel_x.x=image.x;
   sobel_x.y=image.y;
   sobel_x.depth=image.depth;
   sobel_x.pixels=malloc(image.x*image.y*image.depth*sizeof(char));
```

```
/* Allocate space for output image */
   sobel_y.x=image.x;
   sobel_y.y=image.y;
   sobel_y.depth=image.depth;
   sobel_y.pixels=malloc(image.x*image.y*image.depth*sizeof(char));

/* convolution */
   sobel_data[0].old=&image;
   sobel_data[0].new=&sobel_x;
   sobel_data[0].filter=&sobel_x_filter;
   sobel_data[0].papiEventset=eventset;
   generic_convolve((void*)&sobel_data[0]);

   sobel_data[1].old=&image;
   sobel_data[1].new=&sobel_y;
   sobel_data[1].filter=&sobel_y_filter;
   sobel_data[1].papiEventset=eventset;
//  generic_convolve((void *)&sobel_data[1]);

/* Combine to form output */
   combine(&sobel_x,&sobel_y,&new_image);

/* Write data back out to disk */
   store_jpeg("out.jpg",&new_image);
   PAPI_stop(eventset,values);
//printf("L1 rest %lld\n",values[0]);
//printf("L2 rest %lld\n",values[1]);
   printf("kernel      %lld\n",values[0]);
   printf("user %lld\n",values[1]);
   printf("k+u %lld\n",values[2]);
```

```
    printf("papi event %lld\n",values[2]);
//  time_end=PAPI_get_real_usec();
//   time_total=time_end-time_start;
//  printf("PAPI_time_measure for main func: %lld us\n",time_total);


    PAPI_shutdown();

return0;
}
```

Appendix C: Code patch for HPL benchmark

```
--- 1HPL_pdgev0.c   2017-08-30 15:42:32.546218175 -0400
+++ HPL_pdgesv0.c   2017-10-25 15:59:21.716712958 -0400
@@ -48,6 +48,100 @@
  * Include files
  */
 #include "hpl.h"
+/*Add for  papi test stats*/
+#include "/home/yanliu/research/papioverhead/papi/papi/src/papi.h"
+#include "stdio.h"
+#include "math.h"
+// rdpmc code
+static inline unsigned long long rdpmc(unsigned int counter) {
+
+  unsigned int low, high;
+
+  __asm__ volatile("rdpmc" : "=a" (low), "=d" (high) : "c" (counter));
+
+  return (unsigned long long)low | ((unsigned long long)high) <<32;
+}
+
+int partition( long long a[], int l, int r) {
+  long  pivot, i, j, t;
+  pivot = a[l];
+  i = l; j = r+1;
+
+  while( 1)
+  {
```

```
+       do ++i; while( a[i] < pivot && i < r );
+       do --j; while( a[j] > pivot );
+       if( i >= j ) break;
+       t = a[i]; a[i] = a[j]; a[j] = t;
+   }
+   t = a[l]; a[l] = a[j]; a[j] = t;
+   return j;
+}
+void quickSort( long long a[], int l, int r)
+{
+   int j;
+
+   if( l < r )
+   {
+       // divide and conquer
+       j = partition( a, l, r);
+      quickSort( a, l, j-1);
+      quickSort( a, j+1, r);
+   }
+
+}
+void do_percentile(int num_iters,long long *a, long long *percent25, long long *percent50,
long long *percent75,long long *percent99)
+{
+       long long *a_sort;
+       a_sort = calloc(num_iters,sizeof(long long));
+       memcpy(a_sort,a,num_iters*sizeof(long long));
+       int i_25=(int)num_iters/4;
+       int i_50=(int)num_iters/2;
+       int i_75=(int)num_iters/4*3; // index for  75%, not quite accurate cause didn't take even or
```

```
odd in consideratio
+       int i_99=(int)num_iters/10*9.9;
+       quickSort(a_sort,0,num_iters-1);
+
+
+       *percent25=a_sort[i_25];
+       *percent50=a_sort[i_50];
+       *percent75=a_sort[i_75];
+       *percent99=a_sort[i_99];
+       free(a_sort);
+       a_sort=NULL;
+
+}
+static void
+print_percentile(long long percent25, long long percent50, long long percent75,long long
percent99)
+{
+       printf
+       ("25%% cycles   : %lld\n50%% cycles   : %lld\n75%% cycles   : %lld\n99%% cycles   : %lld\n",
+       percent25,percent50,percent75,percent99);
+}
+static void cal_stats(int num_iters,long long *array, long long *min,
+                long long *max, double *avag,double *std){
+       int i; double tmp;
+       long long percent25, percent50, percent75, percent99;
+       *max=*min=array[0];
+   *avag=0; *std=0;
+       for(i=0;i<num_iters;i++){
+           if(*max<array[i]) *max=array[i];
+           if(*min>array[i]) *min=array[i];
```

```
+        *avag+=(double)array[i];
+      //           printf("%d-%lld \n",i,array[i]);
+
+      }
+     *avag=(double)*avag/num_iters;
+      for ( i = 0; i < num_iters; i++ ) {
+             tmp =  (double)array[i] - ( *avag );
+             *std += tmp * tmp;
+      }
+      *std=sqrt(*std/num_iters);
+      do_percentile(num_iters,array,&percent25,&percent50,&percent75,&percent99);
+
+      print_percentile(percent25,percent50,percent75,percent99);
+}
+
+
 #ifdef STDC_HEADERS
 void HPL_pdgesv0
 (
@@ -95,8 +189,48 @@
  */
   HPL_T_panel             * * panel = NULL;
   HPL_T_UPD_FUN           HPL_pdupdate;
-  int                     N, j, jb, n, nb, tag=MSGID_BEGIN_FACT,
-                          test=HPL_KEEP_TESTING;
+  int                     N, j, jb, n, nb, tag=MSGID_BEGIN_FACT, test=HPL_KEEP_TESTING;
+
+/*start PAPI setup*/
+  int max_iters=1000;
+  int readNum=7;
```

```
+       long long totcyc,array[readNum][max_iters],values[3],lastVal1,lastVal2,
+         eventVal1[readNum][max_iters],eventVal2[readNum][max_iters],
+             rawVal1[readNum][max_iters],rawVal2[readNum][max_iters],
+         rawBefore1,rawBefore2,rawAfter1,rawAfter2;
+       long long min[readNum],max[readNum];
+       double avag[readNum],std[readNum];
+       int num_iters=0;
+       int eventset=PAPI_NULL;
+       int  result;
+   #define USER_EVENT 0 //1073741825
+   #define KERNEL_EVENT 1
+   /*
+   haswell machine event name table
+   "PRESET,PAPI_TOT_CYC,NOT_DERIVED,CPU_CLK_THREAD_UNHALTED:THREAD_P\n"
+   "PRESET,PAPI_TOT_INS,NOT_DERIVED,INST_RETIRED:ANY_P\n"  (user 1073741825 kernel 1)
+   "PRESET,PAPI_L1_DCM,NOT_DERIVED,L1D:REPLACEMENT\n"
+   "PRESET,PAPI_L1_TCM,DERIVED_ADD,L1D:REPLACEMENT,L2_RQSTS:ALL_CODE_RD\n"
+   PRESET,PAPI_BR_MSP,NOT_DERIVED,BR_MISP_RETIRED:CONDITIONAL\n"
+
PRESET,PAPI_TLB_DM,DERIVED_ADD,DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK,DTLB_STORE_
MISSES:MISS_CAUSES_A_WALK\n"
+   "PRESET,PAPI_TLB_IM,NOT_DERIVED,ITLB_MISSES:MISS_CAUSES_A_WALK\n"
+   */
+      // array=(long long*)malloc(max_iters*sizeof(long long));
+       if((result=PAPI_library_init(PAPI_VER_CURRENT))!=PAPI_VER_CURRENT)
+           printf("Error PAPI library init:%s\n",PAPI_strerror(result));
+       /*create papi eventset and add events*/
+       if((result=PAPI_create_eventset(&eventset))!=PAPI_OK)
+           printf("Error PAPI create event:%s\n",PAPI_strerror(result));
+
```

```
+
if((result=PAPI_add_named_event(eventset,"L1D:REPLACEMENT,L2_RQSTS:ALL_CODE_RD:u=1"))
!=PAPI_OK)
+              printf("Error PAPI add event1:%s\n",PAPI_strerror(result));
+
if((result=PAPI_add_named_event(eventset,"L1D:REPLACEMENT,L2_RQSTS:ALL_CODE_RD:u=0:k
=1"))!=PAPI_OK)
+             printf("Error PAPI add event2:%s\n",PAPI_strerror(result));
+       if((result=PAPI_start(eventset))!=PAPI_OK)
+              printf("Error PAPI start:%s\n",PAPI_strerror(result));
+       PAPI_read(eventset,values);
+
+/*end PAPI setup*/
 /* ..
  * .. Executable Statements ..
  */
@@ -115,29 +249,205 @@
 /*
  * Loop over the columns of A
  */
-   for( j = 0; j < N; j += nb )
-   {
-      n = N - j; jb = Mmin( n, nb );
+   for( j = 0; j < N; j += nb )
+   {
+           n = N - j; jb = Mmin( n, nb );
 /*
  * Release panel resources - re-initialize panel data structure
  */
-      (void) HPL_pdpanel_free( panel[0] );
```

```
-     HPL_pdpanel_init( GRID, ALGO, n, n+1, jb, A, j, j, tag, panel[0] );
-/*
- * Factor and broadcast current panel - update
+  (void) HPL_pdpanel_free( panel[0] );
+ /*start papi read*/
+
+
+
+      PAPI_start(eventset);
+
+  HPL_pdpanel_init( GRID, ALGO, n, n+1, jb, A, j, j, tag, panel[0] );
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+  rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
+    //  totcyc=PAPI_get_real_cyc();
+      PAPI_read(eventset,values);
+      //totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+  rawAfter1=rdpmc(USER_EVENT);
+  rawAfter2=rdpmc(KERNEL_EVENT);
+
+  PAPI_stop(eventset,values);
+
+  array[0][num_iters]=totcyc;
+      eventVal1[0][num_iters]=values[0]; // event value measured by papi_start/stop
+  eventVal2[0][num_iters]=values[1];
+  rawVal1[0][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+  rawVal2[0][num_iters]=rawAfter2-rawBefore2;
+
```

```
+/*end papi read*/
+
+ /* Factor and broadcast current panel - update
 */
-      HPL_pdfact(             panel[0] );
-      (void) HPL_binit(        panel[0] );
+  /*start papi read*/
+
+   PAPI_start(eventset);
+
+  HPL_pdfact(             panel[0] );
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+  rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
+     // totcyc=PAPI_get_real_cyc();
+      PAPI_read(eventset,values);
+     // totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+  rawAfter1=rdpmc(USER_EVENT);
+  rawAfter2=rdpmc(KERNEL_EVENT);
+
+  PAPI_stop(eventset,values);
+
+  array[1][num_iters]=totcyc;
+      eventVal1[1][num_iters]=values[0]; // event value measured by papi_start/stop
+  eventVal2[1][num_iters]=values[1];
+  rawVal1[1][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+  rawVal2[1][num_iters]=rawAfter2-rawBefore2;
+   /*end papi read*/
```

```
+
+/*initilize panel*/
+  /*start papi read*/
+
+     PAPI_start(eventset);
+
+      (void) HPL_binit(        panel[0] );
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+  rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
+     // totcyc=PAPI_get_real_cyc();
+      PAPI_read(eventset,values);
+      //totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+  rawAfter1=rdpmc(USER_EVENT);
+  rawAfter2=rdpmc(KERNEL_EVENT);
+
+  PAPI_stop(eventset,values);
+
+  array[2][num_iters]=totcyc;
+      eventVal1[2][num_iters]=values[0]; // event value measured by papi_start/stop
+  eventVal2[2][num_iters]=values[1];
+  rawVal1[2][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+  rawVal2[2][num_iters]=rawAfter2-rawBefore2;
+
+ /*end papi read*/
+
+/*broadcast*/
+
```

```
+   /*start papi read*/
+
+  PAPI_start(eventset);
     do
     { (void) HPL_bcast(      panel[0], &test ); }
     while( test != HPL_SUCCESS );
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+  rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
+     // totcyc=PAPI_get_real_cyc();
+      PAPI_read(eventset,values);
+     // totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+  rawAfter1=rdpmc(USER_EVENT);
+  rawAfter2=rdpmc(KERNEL_EVENT);
+
+  PAPI_stop(eventset,values);
+
+  array[3][num_iters]=totcyc;
+      eventVal1[3][num_iters]=values[0]; // event value measured by papi_start/stop
+  eventVal2[3][num_iters]=values[1];
+  rawVal1[3][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+  rawVal2[3][num_iters]=rawAfter2-rawBefore2;
+
+ /*end papi read*/
+
+/*HPL_bwait*/
+   /*start papi read*/
+
```

```
+    PAPI_start(eventset);
      (void) HPL_bwait(         panel[0] );
-    HPL_pdupdate( NULL, NULL, panel[0], -1 );
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+  rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
+      // totcyc=PAPI_get_real_cyc();
+       PAPI_read(eventset,values);
+      // totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+  rawAfter1=rdpmc(USER_EVENT);
+  rawAfter2=rdpmc(KERNEL_EVENT);
+
+  PAPI_stop(eventset,values);
+
+  array[4][num_iters]=totcyc;
+       eventVal1[4][num_iters]=values[0]; // event value measured by papi_start/stop
+  eventVal2[4][num_iters]=values[1];
+  rawVal1[4][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+  rawVal2[4][num_iters]=rawAfter2-rawBefore2;
+
+   /*end papi read*/
+
+    /*pannel update*/
+
+   /*start papi read*/
+
+       PAPI_start(eventset);
+  HPL_pdupdate( NULL, NULL, panel[0], -1 );
```

```
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+  rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
+     // totcyc=PAPI_get_real_cyc();
+      PAPI_read(eventset,values);
+     // totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+  rawAfter1=rdpmc(USER_EVENT);
+  rawAfter2=rdpmc(KERNEL_EVENT);
+
+  PAPI_stop(eventset,values);
+
+  array[5][num_iters]=totcyc;
+      eventVal1[5][num_iters]=values[0]; // event value measured by papi_start/stop
+  eventVal2[5][num_iters]=values[1];
+  rawVal1[5][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+  rawVal2[5][num_iters]=rawAfter2-rawBefore2;
+
+   /*end papi read*/
+
 /*
  * Update message id for next factorization
  */
+   /*start papi read*/
+
+   PAPI_start(eventset);
     tag = MNxtMgid( tag, MSGID_BEGIN_FACT, MSGID_END_FACT );
-  }
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+  rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
```

```
+
+       //totcyc=PAPI_get_real_cyc();
+       PAPI_read(eventset,values);
+       //totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+   rawAfter1=rdpmc(USER_EVENT);
+   rawAfter2=rdpmc(KERNEL_EVENT);
+
+   PAPI_stop(eventset,values);
+
+   array[6][num_iters]=totcyc;
+       eventVal1[6][num_iters]=values[0]; // event value measured by papi_start/stop
+   eventVal2[6][num_iters]=values[1];
+   rawVal1[6][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+   rawVal2[6][num_iters]=rawAfter2-rawBefore2;
+   //printf("%lld %lld\n",rawAfter2,rawBefore2);
+    /*end papi read*/
+     num_iters++;
+
+    }
 /*
  * Release panel resources and panel list
  */
@@ -147,4 +457,57 @@
 /*
  * End of HPL_pdgesv0
  */
+/*start papi message*/
+
+if((result=PAPI_stop(eventset,values))!=PAPI_OK)
```

```
+    printf("Error PAPI stop:%s\n",PAPI_strerror(result));
+/*   for(int i=0;i<num_iters;i++)
+   {
+       printf("%d 1-panelInit %d %lld %lld  ",i,array[0][i],eventVal1[0][i],eventVal2[0][i]);
+       printf("2-pdfact %d %lld %lld  ",array[1][i],eventVal1[1][i],eventVal2[1][i]);
+       printf("3-binit %d  %lld %lld  ",array[2][i],eventVal1[2][i],eventVal2[2][i]);
+       printf("4-bcast %d  %lld %lld  ",array[3][i],eventVal1[3][i],eventVal2[3][i]);
+       printf("5-bwait %d  %lld %lld  ",array[4][i],eventVal1[4][i],eventVal2[4][i]);
+       printf("6-update %d  %lld %lld  ",array[5][i],eventVal1[5][i],eventVal2[5][i]);
+       printf("7-mnxtmgrid %d  %lld %lld  \n",array[6][i],eventVal1[6][i],eventVal2[6][i]);
+   }
+  */
+ for(int i=0;i<7;i++)
+    {
+/*                                          printf("\n****************the        %dth        read
cycle***********************************\n",i+1);
+  cal_stats(num_iters,array[i],&min[i],&max[i],&avag[i],&std[i]);
+      printf("Total cost for PAPI_read over %d iterations \n", num_iters);
+           printf("min cycles : %lld\nmax cycles : %lld\naverage cycles : %lf\nstandard deviation:
%lf\n",
+            min[i],max[i],avag[i],std[i]);
+
+     printf("-------the %dth l1-icm event---------\n",i+1);
+  cal_stats(num_iters,eventVal1[i],&min[i],&max[i],&avag[i],&std[i]);
+        printf("min val : %lld\nmax val : %lld\naverage val : %lf\nstandard deviation: %lf\n",
+            min[i],max[i],avag[i],std[i]);
+     printf("-------the %dth l1-dcm event---------\n",i+1);
+  cal_stats(num_iters,eventVal2[i],&min[i],&max[i],&avag[i],&std[i]);
+        printf("min val : %lld\nmax val : %lld\naverage val : %lf\nstandard deviation: %lf\n",
+            min[i],max[i],avag[i],std[i]);
```

```
+*/
+                                   printf("\n****************the          %dth          read
cycle***********************************\n",i+1);
+  cal_stats(num_iters,array[i],&min[i],&max[i],&avag[i],&std[i]);
+      printf("Total cost for PAPI_read over %d iterations \n", num_iters);
+        printf("\naverage cycles : %lf\n",avag[i]);
+
+     printf("\n-------the %dth  papi_start/stop user event ---------\n",i+1);
+  cal_stats(num_iters,eventVal1[i],&min[i],&max[i],&avag[i],&std[i]);
+        printf("\naverage val : %lf\\n",avag[i]);
+     printf("\n-------the %dth  papi_start/stop kernel event ---------\n",i+1);
+  cal_stats(num_iters,eventVal2[i],&min[i],&max[i],&avag[i],&std[i]);
+        printf("\naverage val : %lf\n",avag[i]);
+  printf("\n-------the %dth  raw user_event for papi_read ---------\n",i+1);
+  cal_stats(num_iters,rawVal1[i],&min[i],&max[i],&avag[i],&std[i]);
+        printf("\naverage val : %lf\\n",avag[i]);
+     printf("\n-------the %dth  raw kernel_event for papi_read ---------\n",i+1);
+  cal_stats(num_iters,rawVal2[i],&min[i],&max[i],&avag[i],&std[i]);
+        printf("\naverage val : %lf\n",avag[i]);
+    }
+
+/*end papi message*/
 }
+
```

Appendix D: Code patch for STREAM benchmark

```
--- stream.c    2017-10-26 15:33:09.632593045 -0400
+++ stream1.c   2017-10-26 15:32:20.980177391 -0400
@@ -46,6 +46,10 @@
 # include <float.h>
 # include <limits.h>
 # include <sys/time.h>
+# include "../papi/papi/src/papi.h"
+#include "stdio.h"
+#include "stdlib.h"
+#include "string.h"

 /*-----------------------------------------------------------------------
  * INSTRUCTIONS:
@@ -109,7 +113,7 @@
 #endif
 #endif
 #ifndef NTIMES
-#   define NTIMES  10
+#   define NTIMES  1000
 #endif

 /*  Users are allowed to modify the "OFFSET" variable, which *may* change the
@@ -204,6 +208,97 @@
 #ifdef _OPENMP
 extern int omp_get_num_threads();
 #endif
+/*add for papi test stats */
```

```
+// rdpmc code
+static inline unsigned long long rdpmc(unsigned int counter) {
+
+       unsigned int low, high;
+
+       __asm__ volatile("rdpmc" : "=a" (low), "=d" (high) : "c" (counter));
+
+       return (unsigned long long)low | ((unsigned long long)high) <<32;
+}
+
+int partition( long long a[], int l, int r) {
+   long  pivot, i, j, t;
+   pivot = a[l];
+   i = l; j = r+1;
+
+   while( 1)
+   {
+       do ++i; while( a[i] < pivot && i < r );
+       do --j; while( a[j] > pivot );
+       if( i >= j ) break;
+       t = a[i]; a[i] = a[j]; a[j] = t;
+   }
+   t = a[l]; a[l] = a[j]; a[j] = t;
+   return j;
+}
+void quickSort( long long a[], int l, int r)
+{
+   int j;
+
+   if( l < r )
```

```
+  {
+      // divide and conquer
+      j = partition( a, l, r);
+     quickSort( a, l, j-1);
+     quickSort( a, j+1, r);
+  }
+
+}
+void do_percentile(int num_iters,long long *a, long long *percent25, long long *percent50,
long long *percent75,long long *percent99)
+{
+      long long *a_sort;
+      a_sort = calloc(num_iters,sizeof(long long));
+      memcpy(a_sort,a,num_iters*sizeof(long long));
+      int i_25=(int)num_iters/4;
+      int i_50=(int)num_iters/2;
+      int i_75=(int)num_iters/4*3; // index for  75%, not quite accurate cause didn't take even or
odd in consideratio
+      int i_99=(int)num_iters/10*9.9;
+      quickSort(a_sort,0,num_iters-1);
+
+
+      *percent25=a_sort[i_25];
+      *percent50=a_sort[i_50];
+      *percent75=a_sort[i_75];
+      *percent99=a_sort[i_99];
+      free(a_sort);
+      a_sort=NULL;
+
+}
```

```
+static void
+print_percentile(long long percent25, long long percent50, long long percent75,long long
percent99)
+{
+       printf ("25%% cycles   : %lld\n50%% cycles   : %lld\n75%% cycles   : %lld\n99%% cycles   :
%lld\n",
+       percent25,percent50,percent75,percent99);
+}
+static void cal_stats(int num_iters,long long *array, long long *min,
+                       long long *max, double *avag,double *std){
+       int i; double tmp;
+       long long percent25, percent50, percent75, percent99;
+       *max=*min=array[0];
+   *avag=0;*std=0;
+       for(i=0;i<num_iters;i++){
+           if(*max<array[i]) *max=array[i];
+           if(*min>array[i]) *min=array[i];
+         *avag+=(double)array[i];
+       //              printf("%d-%lld \n",i,array[i]);
+
+       }
+      *avag=*avag/(double)num_iters;
+       for ( i = 0; i < num_iters; i++ ) {
+               tmp =  (double)array[i] - ( *avag );
+               *std += tmp * tmp;
+       }
+       *std=sqrt(*std/num_iters);
+       do_percentile(num_iters,array,&percent25,&percent50,&percent75,&percent99);
+
+       print_percentile(percent25,percent50,percent75,percent99);
```

```
+}
+
+
+
 int
 main()
   {
@@ -213,7 +308,44 @@
   ssize_t      j;
   STREAM_TYPE      scalar;
   double    t, times[4][NTIMES];
-
+/*-- setup -- papi read TLB miss*/
+  int eventset=PAPI_NULL;
+      int  result;
+      int readNum=4;
+  int num_iters=0;
+  int max_iters=NTIMES;
+  long long totcyc,array[readNum][max_iters],values[3],lastVal1,lastVal2,
+              eventVal1[readNum][max_iters],eventVal2[readNum][max_iters],
+              rawVal1[readNum][max_iters],rawVal2[readNum][max_iters],
+              rawBefore1,rawBefore2,rawAfter1,rawAfter2;
+      long long min[readNum],max[readNum];
+      double avag[readNum],std[readNum];
+
+  #define USER_EVENT 0//1073741825
+      #define KERNEL_EVENT 1
+      /*
+      haswell machine event name table
+      "PRESET,PAPI_TOT_CYC,NOT_DERIVED,CPU_CLK_THREAD_UNHALTED:THREAD_P\n"
```

```
+        "PRESET,PAPI_TOT_INS,NOT_DERIVED,INST_RETIRED:ANY_P\n"   (user 1073741825 kernel
1)
+       "PRESET,PAPI_L1_DCM,NOT_DERIVED,L1D:REPLACEMENT\n"
+       "PRESET,PAPI_L1_TCM,DERIVED_ADD,L1D:REPLACEMENT,L2_RQSTS:ALL_CODE_RD\n"
+       PRESET,PAPI_BR_MSP,NOT_DERIVED,BR_MISP_RETIRED:CONDITIONAL\n"
+
PRESET,PAPI_TLB_DM,DERIVED_ADD,DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK,DTLB_STORE_
MISSES:MISS_CAUSES_A_WALK\n"
+       "PRESET,PAPI_TLB_IM,NOT_DERIVED,ITLB_MISSES:MISS_CAUSES_A_WALK\n"
+       */
+
+   if((result=PAPI_library_init(PAPI_VER_CURRENT))!=PAPI_VER_CURRENT)
+           printf("Error PAPI library init:%s\n",PAPI_strerror(result));
+       /*create papi eventset and add events*/
+       if((result=PAPI_create_eventset(&eventset))!=PAPI_OK)
+           printf("Error PAPI create event:%s\n",PAPI_strerror(result));
+
+
if((result=PAPI_add_named_event(eventset,"DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK,DTLB_
STORE_MISSES:MISS_CAUSES_A_WALK:u=1"))!=PAPI_OK)
+       printf("Error PAPI add event:%s\n",PAPI_strerror(result));
+
if((result=PAPI_add_named_event(eventset,"DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK,DTLB_
STORE_MISSES:MISS_CAUSES_A_WALK:k=1:u=0"))!=PAPI_OK)
+       printf("Error PAPI add event:%s\n",PAPI_strerror(result));
+       PAPI_start(eventset);
+  PAPI_read(eventset,values);
    /* --- SETUP --- determine precision and check timing --- */

    printf(HLINE);
```

```
@@ -306,17 +438,49 @@
     scalar = 3.0;
     for (k=0; k<NTIMES; k++)
    {
+
    times[0][k] = mysecond();
+/*STREAM_COPY*/
+/*papi start*/
+  PAPI_start(eventset);
+
 #ifdef TUNED
        tuned_STREAM_Copy();
 #else
 #pragma omp parallel for
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
-      c[j] = a[j];
+     {
+       c[j] = a[j];
+     }
 #endif
+
    times[0][k] = mysecond() - times[0][k];
-
+
    times[1][k] = mysecond();
+
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+      rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
```

```
+       totcyc=PAPI_get_real_cyc();
+       PAPI_read(eventset,values);
+       totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+       rawAfter1=rdpmc(USER_EVENT);
+       rawAfter2=rdpmc(KERNEL_EVENT);
+
+       PAPI_stop(eventset,values);
+
+       array[0][num_iters]=totcyc;
+       eventVal1[0][num_iters]=values[0]; // event value measured by papi_start/stop
+       eventVal2[0][num_iters]=values[1];
+       rawVal1[0][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+       rawVal2[0][num_iters]=rawAfter2-rawBefore2;
+/*papi stop*/
+
+/*STREAM_scale*/
+/*papi start*/
+       PAPI_start(eventset);
 #ifdef TUNED
        tuned_STREAM_Scale(scalar);
 #else
@@ -327,6 +491,29 @@
   times[1][k] = mysecond() - times[1][k];

   times[2][k] = mysecond();
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+       rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
```

```
+      totcyc=PAPI_get_real_cyc();
+      PAPI_read(eventset,values);
+      totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+      rawAfter1=rdpmc(USER_EVENT);
+      rawAfter2=rdpmc(KERNEL_EVENT);
+
+      PAPI_stop(eventset,values);
+
+      array[1][num_iters]=totcyc;
+      eventVal1[1][num_iters]=values[0]; // event value measured by papi_start/stop
+      eventVal2[1][num_iters]=values[1];
+      rawVal1[1][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+      rawVal2[1][num_iters]=rawAfter2-rawBefore2;
+
+
+/*STREAM_add*/
+/*papi start*/
+      PAPI_start(eventset);
 #ifdef TUNED
        tuned_STREAM_Add();
 #else
@@ -337,6 +524,29 @@
   times[2][k] = mysecond() - times[2][k];

   times[3][k] = mysecond();
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+      rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
```

```
+       totcyc=PAPI_get_real_cyc();
+       PAPI_read(eventset,values);
+       totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+       rawAfter1=rdpmc(USER_EVENT);
+       rawAfter2=rdpmc(KERNEL_EVENT);
+
+       PAPI_stop(eventset,values);
+
+       array[2][num_iters]=totcyc;
+       eventVal1[2][num_iters]=values[0]; // event value measured by papi_start/stop
+       eventVal2[2][num_iters]=values[1];
+       rawVal1[2][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+       rawVal2[2][num_iters]=rawAfter2-rawBefore2;
+/*papi stop*/
+
+/*STREAM_triad*/
+/*papi start*/
+       PAPI_start(eventset);
 #ifdef TUNED
        tuned_STREAM_Triad(scalar);
 #else
@@ -345,6 +555,27 @@
        a[j] = b[j]+scalar*c[j];
 #endif
   times[3][k] = mysecond() - times[3][k];
+
+  rawBefore1=rdpmc(USER_EVENT);  //raw rdpmc measure USER_EVENT for PAPI_read
+       rawBefore2=rdpmc(KERNEL_EVENT); // raw rdpmc measure KERNEL_EVENT for PAPI_read
+
```

```
+       totcyc=PAPI_get_real_cyc();
+       PAPI_read(eventset,values);
+       totcyc=PAPI_get_real_cyc()-totcyc; // measure total cycle cost by PAPI_read
+
+       rawAfter1=rdpmc(USER_EVENT);
+       rawAfter2=rdpmc(KERNEL_EVENT);
+
+       PAPI_stop(eventset,values);
+
+       array[3][num_iters]=totcyc;
+       eventVal1[3][num_iters]=values[0]; // event value measured by papi_start/stop
+       eventVal2[3][num_iters]=values[1];
+       rawVal1[3][num_iters]=rawAfter1-rawBefore1; // event value measured by rdpmc
+       rawVal2[3][num_iters]=rawAfter2-rawBefore2;
+/*papi stop*/
+
+  num_iters++;
   }

    /* --- SUMMARY --- */
@@ -374,8 +605,30 @@
    /* --- Check Results --- */
    checkSTREAMresults();
    printf(HLINE);
-
+/* print papi event*/
+  PAPI_stop(eventset,values);
+/*papi message start*/
+  for(int i=0;i<4;i++){
+                        printf("\n****************the        %dth        read
```

```
cycle************************************\n",i+1);
+       cal_stats(num_iters,array[i],&min[i],&max[i],&avag[i],&std[i]);
+       printf("Total cost for PAPI_read over %d iterations \n", num_iters);
+       printf("\naverage cycles : %lf\n",avag[i]);
+
+      printf("\n-------the %dth  papi_start/stop user event ---------\n",i+1);
+       cal_stats(num_iters,eventVal1[i],&min[i],&max[i],&avag[i],&std[i]);
+       printf("\naverage val : %lf\\n",avag[i]);
+      printf("\n-------the %dth  papi_start/stop kernel event ---------\n",i+1);
+       cal_stats(num_iters,eventVal2[i],&min[i],&max[i],&avag[i],&std[i]);
+       printf("\naverage val : %lf\n",avag[i]);
+       printf("\n-------the %dth  raw user_event for papi_read ---------\n",i+1);
+       cal_stats(num_iters,rawVal1[i],&min[i],&max[i],&avag[i],&std[i]);
+       printf("\naverage val : %lf\\n",avag[i]);
+      printf("\n-------the %dth  raw kernel_event for papi_read ---------\n",i+1);
+       cal_stats(num_iters,rawVal2[i],&min[i],&max[i],&avag[i],&std[i]);
+       printf("\naverage val : %lf\n",avag[i]);
+  }
    return 0;
+
 }
```

```
 # define   M   20
```

BIOGRAPHY OF THE AUTHOR

Yan Liu was born in September 24th of 1986 in China. She was raised in Shijiazhuang, Hebei, until 2006. She attended Central South University of Forestry & Technology in China and received a Bachelor of Science degree in Geographic Information System in 2010. After graduation, Yan continued her study at the same school as a master student. She achieved her Master of Science degree in Computer Science in June 2012. She worked as a software engineer for a year in NanJing Tornado Technology Company for a year after graduation. Yan started her new adventure in the department of Electrical & Computer Engineering at the University of Maine as a master graduate research assistant. She is a candidate for the master degree in computer engineering from the University of Maine in December 2017.