Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 2096–2104

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

# User-defined events for hardware performance monitoring

Shirley Moore* and James Ralph

*Innovative Computing Laboratory, EECS Department, University of Tennessee, Knoxville, Tennsee, USA 37996*

**Abstract**

PAPI is a widely used cross-platform interface to hardware performance counters. PAPI currently supports native events, which are those provided by a given platform, and preset events, which are pre-defined events thought to be common across platforms. Presets are currently mapped and defined at the time that PAPI is compiled and installed. The idea of user-defined events is to allow users to define their own metrics and to have those metrics mapped to events on a platform without the need to re-install PAPI. User-defined events can be defined in terms of native, preset, and previously defined user-defined events. The user can combine events and constants in an arbitrary expression to define a new metric and give a name to the new metric. This name can then be specified as a PAPI event in a PAPI library call the same way as native and preset events. End-user tools such as TAU and Scalasca that use PAPI can also use the user-defined metrics. Users can publish their metric definitions so that other users can use them as well. We present several examples of how user-defined events can be used for performance analysis and modeling.

*Keywords*: hardware counters; performance modeling; performance metrics; power consumption

## 1. Introduction

PAPI is a widely used cross-platform interface to hardware performance counters [1]. PAPI currently supports native events, which are those provided by a given platform, and preset events, which are pre-defined events thought to be common across platforms. Although the PAPI project has until now focused primarily on providing a portable interface to hardware performance counters and left selection and interpretation of hardware events up to higher-level tools and to users, questions about selection and interpretation are frequently posted to the PAPI support email address and answered by the PAPI developers. The PAPI team is familiar with the native events available on various platforms and works on testing and validation of those events [2]. They also keep abreast of the literature on using hardware events for performance analysis and modeling. The metrics used for analysis and modeling are mostly derived events, such as sums and ratios of native and preset events and combinations of events with system constants, and they vary too much, depending on the context, to be defined as preset events. Consequently, the team saw a need for a mechanism for dynamic definition of derived events to suit the performance analysis or modeling context. Higher-level tools such as TAU [3] and Scalasca [4] typically do not assist with selection of events, but rather leave it up to the user to select a relevant set of events. In the case of TAU, the user can define derived

* Corresponding author. Tel.: 865-974-3547; fax: 865-974-8296.
E-mail address: shirley@eecs.utk.edu .

1877–0509 © 2011 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.229

metrics and have them computed from the Paraprof analysis interface, but the user is responsible for having the necessary events collected ahead of time and there is no provision for inclusion of system constants in the derived event definitions. Tools that do select and define derived metrics for the user, such as PerfExpert [5] and PTU [6], are often specific to a given platform and the metrics are not portable.

It is desirable to separate collection of performance data from analysis of the data. Separation of these tasks allows performance analysis and modeling experts to focus on developing a high-quality end-user tool, rather than on the low-level details of instrumenting code and carrying out performance measurements. One of the major benefits of PAPI has been to free developers of performance analysis tools from the low-level details of implementing access to hardware counters on various platforms. In order for the analysis task to drive the data collection, however (and not have the analysis that can be done be dictated by what data have been collected), the high-level tool must have a way of specifying what data need to be collected. If the higher-level tool can specify its metrics in a form that is understood by PAPI, then any data measurement tool that uses PAPI for access to hardware counter data can collect the necessary data.

Performance modelers who define derived metrics and publish modeling results often do not give explicit definitions of their metrics that would make their models usable by others. Having a mechanism available that would allow performance modelers to publish their metric definitions in a well-defined way, and in a form that would allow the necessary performance data to be collected by any tool that supports PAPI, would allow other users to repeat the modeling experiments and to use the models for their own codes.

For the above reasons, the PAPI project has developed a mechanism for user-defined events. This extension allows users to define their own metrics and to have those metrics mapped to events on a platform without the need to re-install PAPI. User-defined events can be defined in terms of native, preset, and previously defined user-defined events. The user can combine events and constants in an arbitrary expression to define a new metric and give a name to the new metric. This name can then be specified as a PAPI event in a PAPI library call the same way as native and preset events. End-user tools such as TAU and Scalasca that use PAPI can also use the user-defined metrics. Users can publish their metric definitions so that other users can use them as well.

The remainder of the paper is organized as follows. Section 2 explains how user-defined events may be specified by the user and how they have been implemented in the PAPI library. Section 3 gives examples of how user-defined events can be used in performance analysis and modeling. Section 4 contains related work, and section 5 gives conclusions and discusses future work.

## 2. Specification and implementation of user-defined events

User-defined events are specified in a file that must be parsed by the PAPI library before any of the events are available. Parsing can happen at PAPI_library_init time or anytime thereafter with a PAPI_set_opt call. We also provide for static definitions at compile time, if the user is able and willing to re-compile his/her PAPI library to implement his/her events. If at init time, then the file is specified by setting the environment variable PAPI_USER_EVENTS_FILE. Otherwise, the filename may be specified as an option to the PAPI_set_opt call. The syntax for defined events is currently fairly simple. Predefined constants may be given, for example #define TWO 2 (with whitespace separation). An event is defined as

Event, OPERATION_STRING

where OPERATION_STRING is a series of preset|native|predefined events using | as a separator and using reverse Polish notation. Reverse Polish notation was chosen for its simplicity to parse and because PAPI preset event are expressed in this manner. An example event definition file is shown in Figure 1. The event definitions can also be published in an XML format that can be automatically converted to the above format for input to PAPI.

Once the user-defined events have been initialized, they can be used just like any other PAPI events, for example:

```
PAPI_eventname_to_code("my_event_name", &eventcode);
PAPI_add_event(eventset, eventcode);
```

PAPI_add_event will fail if the event cannot be counted on the underlying hardware (e.g., a native event is not available, there are not enough counters, etc.).

```
------ Example events file -------
#define BR_lat 5
#define BR_miss_lat 45
#define L1_lat 3
#define L2_lat 13
#define Mem_lat 450

Branch_cat, PAPI_BR_INS|BR_lat|*|PAPI_BR_MSP|BR_miss_lat|*|+|PAPI_TOT_INS|/
Mem_cat, PAPI_L1_DCA|L1_lat|*|PAPI_L2_DCA|L2_lat|*|+|PAPI_L2_DCM|Mem_lat|*|+
```

Figure 1. Example event definition file

One technical issue is that the native events required to construct an arbitrary user-defined event may not all be available simultaneously. In such a case, the choices are either to use multiplexing to estimate counts for all of the required events, or to do multiple runs. Our current solution to this problem is to add the event only if the user or higher-level tool has enabled multiplexing, and to return a flag indicating an error otherwise. Higher-level tools will then be able to make the decision whether to use multiplexing or multiple runs. Future work will revolve around providing estimates of error and other statistics of how the counters were multiplexed for the run.

The overheads of computing the values of user-defined events are low, on par with the overheads of derived preset events. If a higher-level tool wished to avoid this overhead, however, it could use the event definition file to parse out the required native events, collect those events, and then compute the user-defined derived events in a post-processing step. We plan to provide PAPI utilities that 1) parse a user-defined events specification file and produce a list of the native events to be collected (native events so that tool that use sampling could also use user-defined events), and 2) take measurement results for the native events and the user-defined events specification file and output the results for the user-defined metrics. We will use a "standard" format, most likely XML, for the three files involved.

Another technical issue is how to obtain the system constants, such as cache and memory latencies, that are used in event definitions. Some constants can be obtained from architecture manuals. We provide a set of benchmarks for measuring as many of these constants as possible and we maintain a database of benchmark results for various platforms. Since the values of some constants will depend on machine configuration, however, the user is encouraged to run the benchmarks on his or her system. LMbench [7] and STREAM [8] have reasonable coverage for a useful set of system values. Some of the values, such as the effective memory access latency, are not actually constant but can be somewhat variable. In these cases, we opt for using conservative values that will result in upper bounds for most applications, as in [5, 6].

## 3. Performance and power analysis examples

User-defined events are intended for use in performance analysis and modeling. Rather than having each performance modeler define events from scratch and furthermore figure out the instrumentation and often develop his/her own tool to do the measurements, PAPI user-defined events provide a high-level interface for defining the necessary metrics and allow any instrumentation tool that accesses hardware counters using PAPI to be used to instrument the code and do the measurements. Some examples of how user-defined events can be used to support performance and power analysis and modeling are given below.

### *3.1. Cycle accounting*

The methodology of cycle accounting is explained in [6]. The method is intended for analyzing performance on a target architecture that has out-of-order (OOO) execution. With OOO, after instructions are decoded into executable micro operations (uops), they are issued downstream if there are adequate resources. Necessary resources include the following:

- space in the Reservation Station (RS), where the uops wait until the inputs are available,
- space in the Reorder Buffer, where uops wait until they can be retired,
- sufficient load and store buffers in the case of memory related uops

The focus of cycle accounting is to minimize the cycles consumed to accomplish the desired work.

The overall cycle accounting equation is as follows:

$$CPU_CLK_UNHALTED.CORE = Retired + Non_Retired + Stalls \quad (1)$$

where Retired is the cycles for retiring uops and Non_Retired is the cycles for non-retiring uops. Each of the components in (1) can be measured by hardware counters on Intel Core 2 processors. The event RS_UOPS_DISPATCHED counts the number of uops dispatched from the RS on every cycle. The event UOPS_RETIRED.ANY counts uops that are retired. The event UOPS_RETIRED.FUSED counts the number of those that represent the fusion of two executed uops. Thus

$$retired_uops_executed = UOPS_RETIRED.ANY + UOPS_RETIRED.FUSED$$

is the total number of uops that are executed in the production of useful work.

$$RS_UOPS_DISPATCHED - retired_uops_executed$$

is the number of executed non-retired uops, representing non-productive work.

$$uop_dispatch_rate = RS_UOPS_DISPATCHED/RS_UOPS_DISPATCHED{:}C{=}1$$

where RS_UOPS_DISPATCHED:C=1 is the number of cycles dispatching uops. Thus, the first and second components of (1) can be measured as follows:

$$Retired = retired_uops_executed/uop_dispatch_rate$$

$$Non\ Retired = (RS_UOPS_DISPATCHED-retired_uops_executed)/uop_dispatch_rate$$

Stall cycles can be approximately decomposed into a sum of counts of events causing stalls weighted by their penalties:

$$Counted_Stall_Cycles = \Sigma P_i * N_i$$

For example, stall cycles due to L1 DTLB misses on the Core 2 is given in [6] as

$$MEM_LOAD_RETIRED.DTLB_MISS*4 + PAGE_WALKS.CYCLES$$

The objective of optimizing a program is to minimize the sum in (1) as follows:
1) Minimize the “Retired” component by minimizing the instructions generated by the compiler by vectorization and other techniques.
2) Minimize the “Stalls” by removing memory access and other bottlenecks.
3) Minimize the “Non-retired” component by reducing the branch mispredictions
Thus the sum in (1) can be viewed as an objective function that could be minimized either by hand-tuning or by a compiler or auto-tuning system.

Although the hardware counter events discussed above are specific to the Core 2 architecture, similar events are available on other Intel processors and on other architectures. For example, on AMD Family 10h processors, RETIRED_UOPS counts retired micro operations and DISPATCH_STALLS counts dispatch stall cycles. POWER7 has events for counting numbers of PowerPC instructions dispatched and completed and dispatch hold cycles. We plan to work with experts on the various architectures to implement cycle accounting for each architecture based on equation (1). The exact model will be somewhat different for each platform depending on the availability of native hardware events.

The methodology in [5] combines hardware counter measurements with architectural parameters to compute upper bounds on local cycle-per-instruction (LCPI) contributions of various instruction categories at the granularity of loops and procedures. The six categories are data memory accesses, instruction memory accesses, floating point operations, branches, data TLB accesses, and instruction TLB accesses. Based on measured hardware counter data and latency constants, PerfExpert computes an upper bound on the latency caused by each measured LCPI contribution. In the expressions below for the PerfExpert LCPI metrics, bold face indicates hardware counter measurements, and italics indicates system constants from manuals or benchmarks. The system constants should be in cycles. The expressions may need to be modified for a particular platform, depending on the availability of counters and constants.

Branch category:
(**BR_INS** * *BR_lat* + **BR_MSP** * *BR_miss_lat*)/**TOT_INS**
Data memory access category:
(**L1_DCA** * Data_*L1_lat* + **L2_DCA** * *L2_lat* + **L3_DCA** * *L3_lat* + **L3_DCM** * *Mem_lat*)/**TOT_INS**
Instruction memory access category:
(**L1_ICA** * *Instr_L1_lat* + **L2_ICA** * *L2_lat* + **L3_ICA** * *L3_lat* + **L3_DCM** * *Mem_lat*)/**TOT_INS**
Data TLB access category:
(**TLB_DM** * *Data_TLB_lat*)/**TOT_INS**
Instruction TLB access category:
(**TLB_IM** * *Instr_TLB_lat*)/**TOT_INS**
Floating-point instruction category:
(**FP_INS** * *FP_add_sub_mul_lat* + (**FPDIV** + **FPSQRT**) * *FP_div_sqrt_lat*)/**TOT_INS**

We have a preliminary implementation of the above events, with the expressions in the format shown in Figure 1, thus allowing any routine and loop-level instrumentation tool that uses PAPI to perform the measurements that can then be input to the PerfExpert analysis tool. We have used our implementation to measure the contributions of the six categories to CPI for the HPC Challenge benchmarks [9] run on an AMD Opteron 8358 SE processor. The machine constants we used are shown in Table 1. These values are based on Lmbench benchmark [19] results and on vendor documentation.

Table 1. Machine constants for AMD Opteron 8358 (in cycles)

| | |
|---|---|
| L1 data cache hit latency | 3 |
| L1 instruction cache hit latency | 2 |
| L2 cache hit latency | 17 |
| L3 cache hit latency | 60 |
| Memory access latency | 540 |
| Branch latency | 2 |
| Branch misprediction penalty | 12 |
| Floating-point add/sub/mul latency | 4 |
| Floating-point div/sqrt latency | 38 |
| Data TLB miss latency | 50 |
| Instruction TLB miss latency | 50 |

The CPI results are shown in Figure 2. The PerfExpert model provides a relatively clear picture of what each of the HPC challenge programs test. STREAM, RandomAccess, PTRANS (on one node), and the Bandwidth/Latency benchmarks all show large data memory access contributions while the computationally bound tests show increased floating-point and instruction access CPI contributions. The different STREAM tests can be differentiated by their floating-point activity.

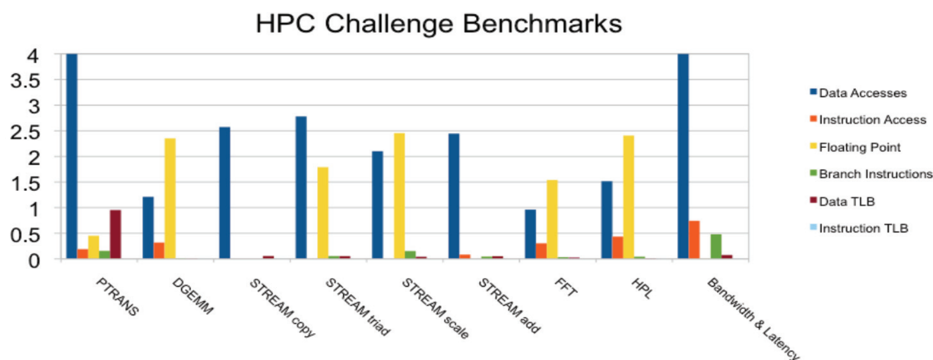


Figure 2. CPI contributions of instruction categories for HPC Challenge benchmarks on AMD Opteron

We plan to validate the above measurements and repeat the experiment on other platforms. Following that, we will repeat the experiments using the application benchmarks from [5], using a higher-level tool such as TAU to obtain LCPI metrics at routine and loop granularity.

*3.2 Roofline model*

A performance model for applications executing on multi-core architectures that ties together floating-point performance, memory bandwidth, and an application's *operational intensity* is proposed in [10]. Operational intensity is defined as operations per byte of DRAM traffic. The model is plotted on a graph that has the flops per DRAM bytes on the x axis and attainable Gflops/s on the y axis. To construct the roofline model graph, one first plots a horizontal line showing peak floating-point performance. One draws a second diagonal line representing the peak memory bandwidth. These two rooflines intersect at the point of peak computational performance and peak memory bandwidth. A series of ceilings are drawn beneath these rooflines to represent the attainable floating-point performance and memory bandwidth, respectively. For example, lower memory bandwidth ceilings may result from lack of software prefetching or lack of memory affinity. Lower floating-point performance may result from multiply-add imbalance or lack of instruction-level parallelism. Once the ceilings have been constructed, one measures the application's operational intensity to find its location on the x-axis. The point where a vertical line through that point intersects a roofline gives the application's expected performance. A methodology for using hardware counters to construct the ceilings for a runtime roofline model and to determine an application's operational intensity is explained in Appendix A of [10]. We plan to define the metrics for runtime construction of roofline models on various platforms and provide a web interface through which application developers can determine their expected performance on these platforms as well as explore optimizations that could improve performance.

*3.2. Memory bandwidth*

Achievable memory bandwidth is used in construction of the roofline model but is a generally useful metric as well. The memory bandwidth that an application is actually achieving can be measured by performance counters on most platforms. The memory bandwidth metric for Intel Core 2 (in Bytes/s) is defined in [11] as

*64 * BUS_TRANS_BURST:SELF * core_freqency/elapsed_cycles*

The memory bandwidth for the AMD 10h Family processors is defined in [12] as

(NORTHBRIDGE_READ_RESPONSES:0x07*64+OCTWORD_WRITE_TRANSFERS:0x01*8)/elapsed_time

and also as

(DRAM_ACCESSES:0x07 + DRAM_ACCESSES:0x38) * 64 / elapsed_time

We have implemented these metrics and are currently validating them using variations of the STREAM benchmark.

*3.3. Power consumption modelling*

A number of researchers have investigated power consumption models based on hardware counters. These models estimate power consumption by fitting a statistical model to hardware counter measurements. They allow power consumption of an application to be estimated when power meters are not available or cannot be used.

A model for an Intel Core i7 system that has an absolute estimation error of 5.32 percent (median) and acceptable data collection overheads on varying workloads, CPU power states (frequency and voltage), and number of active cores is presented in [13]. Regression analysis is used to search for a small set of counters that correlate well, and independently, with power measurements. The hardware events chosen and their coefficients are shown in Table 2.

Table 2. Hardware events and regression coefficients for Core i7 power model

| **Predictors** | **Coefficients** |
|---|---|
| INSTRUCTIONS_RETIRED | -2.6158e-05 |
| UNHALTED_CORE_CYCLES | 5.6177e-05 |
| Average effective CPU Frequency ratio | 46.6053 |
| UNC_LLC_HITS:ANY | 3.6163e-05 |
| L1D_ALL_REF:ANY | 4.5867e-05 |
| RESOURCE_STALLS:ANY | -4.9286e-05 |
| UNC_QHL_REQUESTS:LOCAL_READS | 2.5473e-04 |
| Base power | 25.7593 |

On the Core i7 microprocessor, there are 12 power (voltage and frequency) states that can be set dynamically in software. The tested system in [13] also included Turbo Boost, which modulates CPU core frequency based power utilization and chip temperature. Effective CPU frequency is calculated using

*BaseOperatingFrequency x (UNHALTED_CORE_CYCLES/UNHALTED_REFERENCE_CYCLES)*

A model for the POWER7 is described in [14]. The mathematical model is given in equation (2) below.

$$PWR(fn) = A_n * GIPS(f0) + B_n * GBS(f0) + C_n \quad (2)$$

GIPS(f0) and GBS(f0) are application characteristics measured at the normal frequency (f0). $A_n$, $B_n$ and $C_n$ are determined by regression analysis for the given platform at all possible frequencies. This model hides the dependency of GIPS and GBS of a given workload with the clock frequency. GIPS(f0) and GBS(f0) are determined from hardware counter measurements. The native events used are shown in Table 3 below.

Table 3. Native POWER7 hardware counters used for power modeling

| | |
|---|---|
| PM_RUN_CYC | Non Idle Cycles |
| PM_RUN_INST_CMPL | Non Idle Instructions Completed |
| PM_MEM_RQ_DIST | Memory Read Dispatches at memory controller |
| PM_MEM_WR_DIST | Memory Write Dispatches at memory controller |

The GIPS and GBS metrics are derived as follows:

*CPI = PM_RUN_CYC/PM_RUN_INST_CMPL*

*Read GB/s = (64*PM_MEM0_RQ_DISP)/(PM_CYC/Frequency)*

*Write GB/s = (64*PM_MEM0_SR_DISP)/(PM_CYC/Frequency)*

*GBS = Read GB/s + Write GB/s*

*GIPS = 32 * Frequency / CPI*

where Frequency is in GHz. On the POWER7 (p750), normal frequency is 3.55GHz and power save frequency is 2.5 GHz. The coefficients reported in [14] for these frequencies are shown in Table 4.

We are currently implementing power models for these platforms and validating them with actual power measurements.

Table 4. Power model coefficients for POWER7.

| Platform | Frequency | $A_n$ | $B_n$ | $C_n$ |
|---|---|---|---|---|
| p750 | 2.5 | 8.4 | 4.3 | 666.3 |
| p750 | 3.55 | 22.3 | 4.3 | 877.1 |

## 4. Related work

The OpenMP profiler ompP supports "evaluators", arbitrary arithmetic expressions involving constants and hardware counters [15]. ompP extracts the counter names automatically, sets up PAPI to measure the counters and evaluates the expression for each function/region.

The TAU Paraprof profile viewer supports the definition of derived events from already collected profile data [16]. The interface allows the user to specify two previously defined events, one of which may be a scalar constant, and to apply an arithmetic operation.

We expect the new PAPI user-defined events facility to be useful to these and other tools that wish to make use of dynamic derived events.

## 5. Conclusions and future work

User-defined events have been implemented and are currently available in the development branch of PAPI. This feature will be included in the next official minor release scheduled for first quarter of 2011. Users will be able to contribute metric definitions using a web interface on the PAPI website [17]. Our goal is to improve the usability of hardware counter measurements for performance analysis and modeling by application developers.

User-defined events have been developed as part of the National Science Foundation funded Multicore application Modeling Infrastructure (MuMI) project [18], which is a project to facilitate systematic measurement, modeling, and prediction of performance, power consumption and performance-power tradeoffs for multicore systems. As part of that project, we plan to define and validate metrics for power estimation broken down by core and system component. We are also working on defining prefetching metrics, such as prefetch precision and coverage, in order to model how prefetching affects performance on multicore systems. As part of MuMI, we are also working on a set of system characterization benchmarks that will provide the machine constants needed for many user-defined events.

## Acknowledgements

This work was supported in part by the U.S. Department of Energy Office of Science under contract DE-FC02-06ER25761 and by the National Science Foundation under Grant No. 0910899 and Grant No. NSF OCI-0722072 Subcontract No. 207401

## References

1. Browne, S., et al., *A Portable Programming Interface for Performance Evaluation on Modern Processors.* International Journal of High-Performance Computing and Applications, 2000. **14**(3): p. 189-204.
2. Weaver, V. and J. Dongarra, *Can Hardware Performance Counters Produce Expected, Deterministic Results?*, in *3rd Workshop on Functionality of Hardware Performance Monitoring (FHPM 2010), December* 2010: Atlanta, GA.
3. Shende, S. and A.D. Malony, *The TAU Parallel Performance System.* International Journal of High Performance Computing Applications, 2006. **20**(2): p. 287-311.

4. Wolf, F., et al., *Automatic analysis of inefficiency patterns in parallel applications.* Concurrency and Computation: Practice and Experience, 2007. **19**(11): p. 1481-1496.
5. Burtscher, M., et al., *PerfExpert: An easy-to-user performance diagnosis tool for HPC applications*, in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC10), November* 2010, ACM: New Orleans, Louisiana, USA.
6. Levinthal, D. *Cycle Accounting Analysis on Intel Core 2 Processors*.
7. *LMbench*. Available from: http://lmbench.sourceforge.net/.
8. McCalpin, J.D. *STREAM Sustainable Memory Bandwidth in High Performance Computers*. Available from: http://www.cs.virginia.edu/stream/.
9. *HPC Challenge Benchmarks*. Available from: http://icl.cs.utk.edu/hpcc/.
10. Williams, S., A. Waterman, and D. Patterson, *Roofline: An Insightful Visual Performance Model for Multicore Architectures.* Communication of the ACM, 2009. **52**(5): p. 65-76.
11. Tuduce, I., et al., *Asymmetries in Multi-Core Systems -- Or Why We Need Better Performance Measurement Unites*, in *Exascale Evaluation and Research Techniques Workshop (EXERT) at ASPLOS 2010,* March 2010, ACM: Pittsburgh, PA.
12. Drongowski, P.J., *Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenom Processors*, September 2008, AMD.
13. Lim, M.Y., A. Porterfield, and R. Fowler, *SoftPower: Fine-Grain Power Estimations Using Performance Counters*, in *International Conference on High Performance Distributed Computing (HPDC'10),* June 2010, ACM: Chicago, IL. p. 308-311.
14. Brochard, L., R. Panda, and S. Vemuganti, *Optimizing performance and energy of HPC applications on POWER7.* Computer Science - Research and Development, 2010. **25** (3-4): p. 135-140.
15. Fürlinger, K., *OpenMP Application Profiling - State of the Art and Directions for the Future*, in *2010 International Conference on Computational Science (ICCS 2010),* May 2010: Amsterdam, Netherlands.
16. *ParaProf User's Manual*, 2010, University of Oregon Performance Research Lab, http://www.cs.uoregon.edu/research/tau/docs/paraprof/
17. PAPI project website, http://icl.eecs.utk.edu/papi/
18. MuMI project website, http://www.mumi-tool.org/
19. Lmbench website, http://lmbench.sourceforge.net/