Chapter 4
Power Management and Event Verification in PAPI

Heike Jagode, Asim YarKhan, Anthony Danalis and Jack Dongarra

Abstract For more than a decade, the PAPI performance monitoring library has helped to implement the familiar maxim attributed to Lord Kelvin: "If you cannot measure it, you cannot improve it." Widely deployed and widely used, PAPI provides a generic, portable interface for the hardware performance counters available on all modern CPUs and some other components of interest that are scattered across the chip and system. Recent and radical changes in processor and system design—systems that combine multicore CPUs and accelerators, shared and distributed memory, PCI-express and other interconnects—as well as the emergence of power efficiency as a primary design constraint, and reduced data movement as a primary programming goal, pose new challenges and bring new opportunities to PAPI. We discuss new developments of PAPI that allow for multiple sources of performance data to be measured simultaneously via a common software interface. Specifically, a new PAPI component that controls power is discussed. We explore the challenges of shared hardware counters that include system-wide measurements in existing multicore architectures. We conclude with an exploration of future directions for the PAPI interface.

4.1 Introduction

Most of the major tools that high-performance computing (HPC) application developers use to conduct low-level performance analysis and tuning of their applications typically rely on hardware performance counters to monitor hardware-related activities. The kind of counters that are available is highly hardware dependent; even across the CPUs of a single vendor, each CPU generation has its own implementation. The PAPI performance-monitoring library provides a clear, portable interface to the hardware performance counters available on all modern CPUs, as well as some GPUs, networks, and I/O systems [1, 8, 9, 13]. Additionally, PAPI supports trans-

H. Jagode (✉) · A. YarKhan · A. Danalis · J. Dongarra
Innovative Computing Laboratory, University of Tennessee, Knoxville,
Knoxville, TN 37996, USA
e-mail: jagode@icl.utk.edu

A. Knüpfer et al. (eds.), *Tools for High Performance Computing 2015*,
DOI 10.1007/978-3-319-39589-0_4

parent power monitoring capabilities for various platforms, including Intel Xeon Phi and Blue Gene/Q [10]—enabling PAPI users to monitor power in addition to traditional hardware performance counter data, without modifying their applications or learning a new set of library and instrumentation primitives.

With the increase in scale, complexity, and heterogeneity of modern extreme scale systems, the design of future HPC machines will be driven by energy efficiency constraints. Hence, the ability to control power and energy consumption has become one of the critical features for future development. To allow the HPC community to not only use PAPI to monitor but also manage power consumption, PAPI has been extended with a component supporting power writing capabilities. This paper provides detailed information describing this new component and explores its usefulness with different case studies.

At the same time, the rapid changes and increased complexity we have witnessed in processor and system design—with systems that combine multicore CPUs and accelerators, shared and distributed memory, PCI-express and other interconnects—require a continuous series of updates and enhancements to PAPI with richer and more capable methods that are needed to accommodate these new innovations. Extending PAPI to monitor performance-critical resources that are shared by the cores of multicore and hybrid processors, including on-chip communication networks, memory hierarchy, I/O interfaces, and power management logic, will enable tuning for more efficient use of these resources. Failure to manage the usage and, more importantly, contention for these "inter-core" resources has already become a major drag on overall application performance. We discuss one of PAPI's new features: the Counter Inspection Toolkit (CIT), which is designed to improve the understanding of these inter-core events. Specifically, the CIT integrates micro-benchmarking based methods to gain a better handle on off-core/un-core/NorthBridge related events. We aim to define and verify accurate mappings between particular high-level concepts of performance metrics and underlying low-level hardware events. This extension of PAPI engages novel expertise in low-level and kernel-benchmarks for the explicit purpose of collecting meaningful performance data of shared hardware resources.

In this paper we outline a new PAPI component that supports power and energy controlling through the Intel RAPL interface. A detailed description of the power writing capabilities is provided in addition to case studies that validate the usage of this component. Further, we briefly describe the objectives of the PAPI Counter Inspection Toolkit, and then focus on the micro-kernels that will be used to measure and correlate different native events.

4.2 Power Management Using the Intel RAPL Interface

As processors have grown more complex, power consumption has also increased. This increase in power consumption not only results in increased power costs, but also increased costs for managing the waste heat that is generated by the components. This power and cooling situation is exacerbated by the sizes of data centers and supercomputers. For some time now, interest in managing this power consumption has been increasing, and hardware manufacturers have been introducing low level controls that allow users to make trade-offs between power and computational performance.

Starting with the Sandy Bridge architecture, Intel has included a RAPL (Running Average Power Limit) interface for accessing and managing power features on the processor [5]. This interface exposes features designed for thermal management to a more general user-space. Much of the RAPL interface is accessed by reading and writing information to MSRs (Model Specific Registers).

Current directions for PAPI development include providing applications the ability to trade-off power for performance. To this end, we are experimenting with a PAPI component that will include control aspects, i.e., this component has an active interface that **writes** values to the RAPL/MSR interface. This is a significant change from all prior PAPI components which have had an entirely passive measurement interface to **read** information and events.

This new RAPL/MSR component is expected to, eventually, actively handle interrupts (thermal limits) and change system states (set per-core clock gating, set per-node power cap, enable/disable turbo mode). The following section discusses our early prototype version of the RAPL/MSR component, which has not yet been released.

Providing users with unrestricted access to write data to MSRs (Model Specific Registers) can have many significant performance and security implications. In order to encourage system administrators to give wider access to the MSRs on a machine, LLNL has released a Linux kernel module (`msr_safe`) which provides safer, white-listed access to the MSRs that can be tuned by the site administrator [11]. Lawrence Livermore National Laboratory scientist Barry Rountree has released a library (libmsr) to provide a simple, safe, consistent interface to several of the model-specific registers (MSRs) in Intel processors via the `msr_safe` kernel module [15].

PAPI has created a component that can provide read and write access to the information and controls exposed via libmsr. A scientist can use the well-known, standardized PAPI API to read the state of power consumption on a CPU socket and the current performance of the code, make determinations about the desired CPU performance, and adjust and cap the power consumption as desired. RAPL allows users to set power limits over two specific time windows—meaning, one can have local power spikes, while still keeping the power low over a larger time window. One example of a situation where this might be of interest is when a scientist is aware that the computation requirements will decrease due to communication (I/O bound) and that the overall execution time will not suffer if the CPU power is capped temporarily.

Figure 4.1 shows a simple example using PAPI to measure and adjust the power consumption of an iterative program where each iteration does one consistent unit of work. Initially the power is at a default high level so the performance of the unit task is high. Then we attempt to cap the power at a very low level, below the minimum allowed for the CPU. The power drops to the allowed minimum and the computation time for a unit of work increases. At each 10th iteration the power cap is increased, and the time taken for a unit of work decreases. Finally, we attempt to increase the

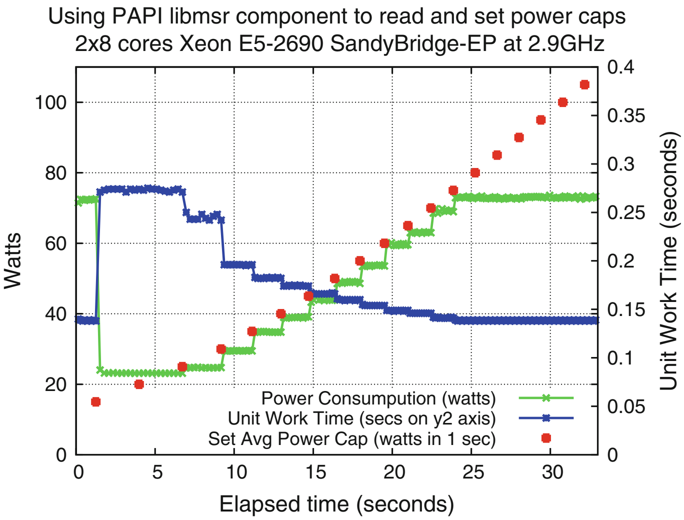

Fig. 4.1 Power controlling with PAPI example

power cap higher than the allowed maximum for the machine. At this point, time taken for a unit of work stays consistent. This simple demonstration shows PAPI's new ability to write information to a counter as well as reading data from a counter.

4.2.1 Case Study: LU Factorization

Here we consider a different usage scenario where controlling power may be appropriate. Many applications can be decomposed as dataflow DAGs (Directed Acyclic Graph), with data dependencies between the various tasks that compose the application. This approach to structuring applications as dataflow DAGs is increasing in popularity with the advent of heterogeneous hardware platforms with large numbers of computation resources, since programming and scheduling on such platforms is a challenge. Dataflow DAGs can be efficiently managed by a runtime and tend to have good load balancing and efficiency characteristics.

If we have an application that is instantiated as a dataflow DAG, then there is often a critical path of tasks that determines the overall execution time of the application. An opportunity for saving power could exist if we schedule all the tasks in the critical path on fast resources, and then schedule the remaining tasks on sockets with decreased power consumption. Under certain circumstances, it can be possible to execute the application without any loss of overall execution time while saving power. However, in most cases there will be some increase in overall execution time.

As an example of a dataflow DAG computation, we consider the tile-based implementation of the LU factorization of a matrix as described in pseudocode in Fig. 4.2a.

(a)

```
GETRF( A ) {
  for (k = 0; k < M; k++) {
     // panel factorization
     GETRF( A(k:M-1,k) );
     for (n = k+1; n < N; n++) {
        // row interchanges to the right
        LASWP  ( A(k:M-1,n) );
        // triangular solve at the top
        TRSM  ( A(k,n) );
        for (m = k+1; m < M; m++) {
           // matrix multiply to the right
           GEMM( A(m,k), A(k,n), A(m,n) );
        }
     }
  }
  for (k = 1; k < M; k++) {
     // row interchanges to the left
     for (n = 0; n < k-1; n++) {
        LASWP( A(k:M-1,n) );
     }
  }
}
```

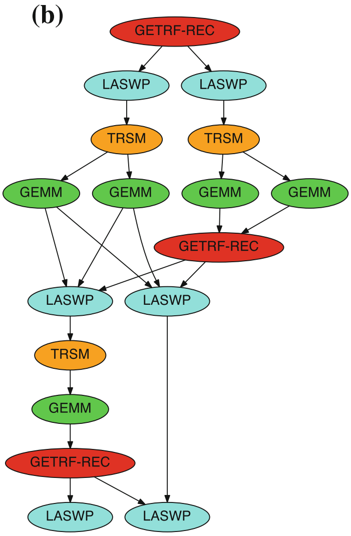

Fig. 4.2 On the *left* we have an LU factorization algorithm that is expressed as tasks (i.e., GETRF, LASWP, TRSM, GEMM) acting on data items (i.e., A(i, j)). When these tasks are executed, the execution can be viewed as a dataflow DAG, where the vertices are the tasks and the edges are data dependencies between them. **a** LU Factorization. **b** LU Factorization DAG

The details of this implementation are beyond the scope of this paper, for more information on the LU DAG implementation, the reader is referred to [2, 7]. In a nutshell, each function corresponds to a task that works on a unit of data—either a tile or a row or an entire panel of data. For example, the red GETRF tasks are large complex tasks that work on an entire panel of data, while the other tasks work on smaller chunks of data and execute much faster.

The DAG of the execution is shown in Fig. 4.2b which reveals that the red GETRF tasks are in the critical path of the graph. Since the GETRF tasks operate on an entire panel of data and are relatively inefficient, they take longer to execute than the other tasks. Our goal is to take this knowledge into consideration by scheduling these expensive GETRF tasks on faster cores because we want them to execute as fast as possible. Other tasks that are not on the critical path are allowed to execute on sockets and cores where the power usage is decreased and where they may run more slowly. Ultimately, this approach will enable us to save energy at a minimal cost to the overall execution time as determined by the critical path.

We conducted this experiment on a 2.90 GHz Intel Xeon Sandy Bridge E5-2690 system, and here we show some small scale results using two sockets (four cores per socket). In Fig. 4.3 we have a small trace from the execution of the dataflow DAG for LU factorization. We can see that the red GETRF is taking a large amount of time as expected. Since this slow task is in the critical path of the execution, there are many occasions where the other CPUs complete all other available tasks and are in an idle state waiting for the GETRF to complete.

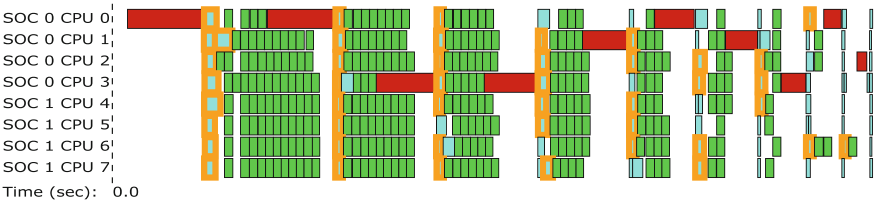

Fig. 4.3 Both socket 0 and socket 1 running at full power. Note that the panel factorization GETRF task (*red*) is long and on the critical path, so there is white space where no tasks are available to run on socket 1

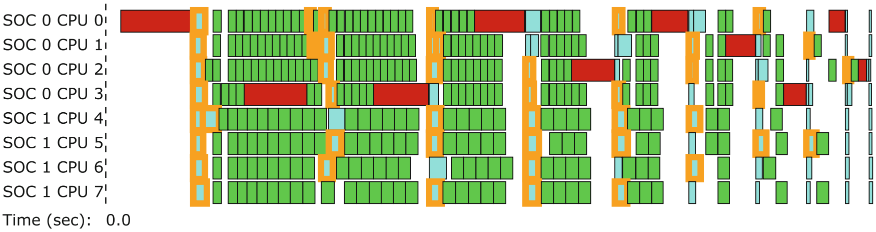

Fig. 4.4 Slow down socket 1 using RAPL and lock critical path GETRF tasks to socket 0. The GEMM tasks (*green*) take longer, filling out the white space on socket 1. This occurs without any overall loss in time for the full execution

This leads us to our opportunity to save power without affecting the overall computation time. We use the runtime environment to restrict the GETRF task to run on socket 0. We use the PAPI libmsr component to write to the RAPL MSRs in order to limit and decrease the power consumption at socket 1. This causes the tasks assigned to socket 1 to take longer, so socket 1 has a higher level of occupation (at a lower power) and does not have as much idle time. In Fig. 4.4 we observe that the individual green GEMM tasks on socket 1 take longer to execute than on socket 0, absorbing the idle time that was wasted in the previous trace. In this small example, the overall computation time was unaffected by slowly reducing the power consumption on socket 1.

We now extend to a larger scale experiment, where we run the tile LU factorization on a matrix that is of size $N = 17920$, consisting of 80×80 tiles of 224×224 double precision numbers. Once again this experiment is run on a 2.90 GHz Intel Xeon SandyBridge E5-2690 system using two sockets (four cores per socket). For this larger problem, the low power execution achieves a small decrease in total power consumed (4001 J) when compared to the high power execution (4136 J). However, the overall time is increased during the low power execution. This is because for the larger matrix size there are so many GEMM tasks generated for each slow POTRF task, that there is minimal idle time for the low power execution to absorb. For other types of dataflow DAG computation, where there is sufficient idle time, this technique may be able to save power while maintaining the overall execution time (Fig. 4.5).

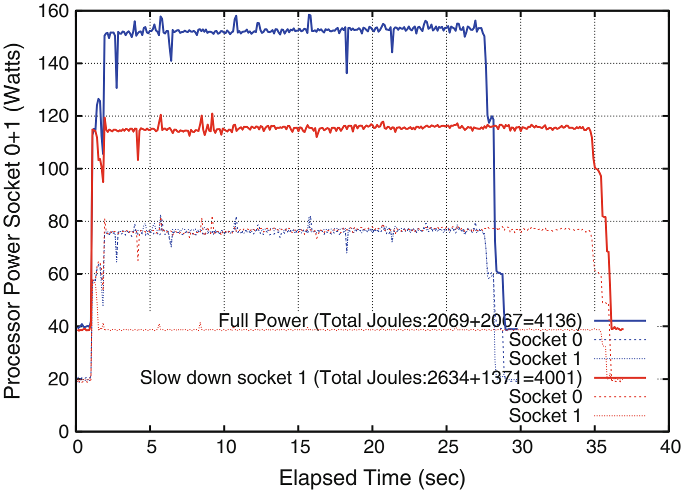

Fig. 4.5 Tiled LU (N=17920=224 × 80) using a SandyBridge EP (2 sockets, 4 cores/socket)

4.3 Counter Inspection Toolkit

In modern architectures, native events count behaviors that relate to the specifics of a particular architecture, but do not necessarily map to higher level concepts of performance. For example, when a memory region is accessed by an application, assessing the level of cache reuse is important when studying the performance of that application. One would expect that this can be done using a native event that counts cache misses, such as the events `LLC_MISSES` and `LLC-LOAD-MISSES`—which measure the last level cache (L3) misses.

However, our experiments show that the numbers reported by PAPI do not always match expectations. Specifically, LLC measurements do not match the expected behavior of the micro-benchmarks in the BlackjackBench [3] suite, which we developed in our previous work. One of the key goals of BlackjackBench was to characterize the cache hierarchy, so special attention was given to stressing different levels of the cache hierarchy. The way this is achieved is through the use of pointer chaining.

Figure 4.6 shows an abstracted code snippet and a schematic outline of pointer chaining. The key idea is that the benchmark is split into a setup phase and a measuring phase. During the setup phase an array is allocated and each element is made to point to another element (i.e., every element stores the address of another element of the array). This creates a chain between the elements of the array. Since this is done in a setup phase—during which neither time, nor hardware events are being measured—the process can be arbitrarily expensive. As a result, every time we need to select the next element of the chain we utilize the POSIX function `random()`, which has a very large period (approximately $16 * ((2^{31}) - 1)$) and good (albeit not cryptographically strong) randomness properties. In the context of our benchmark, good randomness

```
SETUP( ) {
  p = (uintptr_t **) &array[0];
  for (i = random( ) ) {
     next = &array[i];
     *p = next;
     p = (uintptr_t **) next;
  }
}

MEASURE( ) {
  start_measurement();
  for (...) {
     p = (uintptr_t **) *p;
  }
  stop_measurement();
}
```

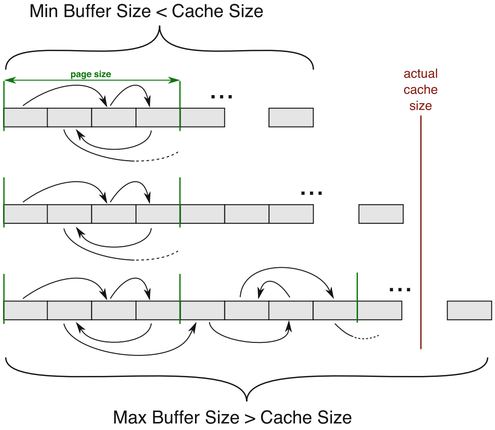

Fig. 4.6 Random pointer chaining

properties means that when we traverse the chain during the measurement phase the memory access sequence will consist of jumps that do not obey any regular patterns that the hardware prefetcher could guess.

Clearly, the actual setup code is more complex than shown in the pseudocode of the figure in order to ensure that the next element that we choose is neither an element that has already been used, nor the current element, and that each pointer is aligned properly so that we only access one element per cache line. Also, as shown in the figure, we populate the buffer one page at a time, in order to stress the cache without stressing the TLB. This is achieved by using a modulo operation (`%`) to keep only the lower bits of the numbers returned by `random()`. Finally, when we reach the last element, we make it point back to the first, so that we can traverse the chain multiple times without explicitly starting and stopping the traversal.

After the setup phase has completed, the measuring phase starts by initiating the desired counters. Then we traverse the pointer chain using a simple loop that dereferences each element to find the next, and when the traversal is over we read the values of the counters. As we discussed earlier, the non-trivial randomness of the chosen setup method ensures that there will be few (if any) regular patterns in the memory traversal, or at least not enough to affect the overall execution time due to hardware prefetching. Furthermore, compilers are incapable of optimizing the code that traverses the chain, since the location of each memory access depends on program data (i.e., the next address is always the value read in the previous array access). As a result, all elements will be explicitly accessed by the code and the probability of any of them having been prefetched by the hardware is low.

In BlackjackBench, we used this technique to setup our arrays and then measured the time it took to traverse such an array as a function of the array size. As shown in Fig. 4.7 the access latency per element jumps every time the array size exceeds the size of a different level of cache, and stays constant while the array fits in a given level of the cache hierarchy. The accuracy of these results make us fairly confident

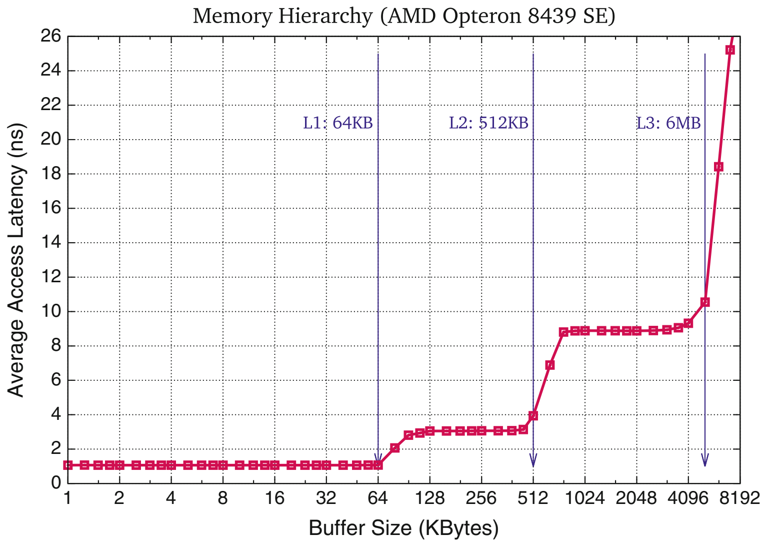

Fig. 4.7 Benchmark timing

in the behavior of the benchmark. However, this same benchmark, when used in conjunction with PAPI native events to measure LLC misses, gives us inconsistent results. Depending on the exact native event used and the architecture on which the experiment takes place, the resulting measurement can be more than expected, less than expected, or even zero!

Given this mismatch, there is a need for a way to *validate* native events, or in other words, assess whether the value a native event measures matches what a human developer thinks it is measuring. Furthermore, there is a need to *define* high level "predefined" events that combine the values of native events in order to provide measurements that match developer intuition. The need for validation and assisted definition of high level events is becoming increasingly urgent in the context of inter-core resource counters, as understanding and utilizing those is particularly challenging.

The Counter Inspection Toolkit, which we are developing, will provide kernels that perform well-defined operations and will use them to measure native events. Subsequently, automatic analyses will attempt to correlate different native events, or combinations of native events to the high level operations. We expect the outcome of this research to be threefold. First, it will increase the portability of PAPI into new hardware with new native events. Second, it will assist PAPI developers in combining native events to define predefined events. Finally, it will provide customization by enabling the user of PAPI (whether that is a human developer of an application, or an additional layer of performance tools) to define custom combinations of events to fit whatever parameters interest a particular user.

4.4 Related Work

Although PAPI has been widely utilized by HPC users for many years, drawing on its strength as a cross-platform and cross-architecture API, there are other tools for gathering performance information like hardware counter data, profiling and tracing data, and MPI library state data.

The *perf tool* [12] makes use of the *perf_event* API which is part of the Linux kernel. Although perf_event attempts to provide a generic interface for Linux platforms, it is still very low-level and the information returned requires considerable interpretation to be useful to tool developers or end users.

Processor vendors supply tools for reading performance counter results. This includes *Intel VTune* [16], *Intel VTune Amplifier*, *Intel PTU* [6], and *AMD's CodeAnalyst* [4]. These program the CPU registers directly, avoiding the Linux kernel. Since the counter state is not saved on the context switch, only system-wide sampling is available, and there is also no API for accessing the results.

The *likwid* lightweight performance tools project [14] allows accessing performance counters by bypassing the Linux kernel and directly accessing hardware. This can have low overhead but can conflict with concurrent use of other tools accessing the counters. It can also expose security issues, as it requires elevated privileges to access the hardware registers and this can lead to crashes or system compromises. likwid provides access to traditional performance counters and also RAPL energy readings. Unlike PAPI, likwid is not cross-platform, only x86 processors are supported under Linux, and only system-wide measurements are available (counters are not saved on context-switch). Currently there is no API for accessing values gathered with likwid; a separate tool gathers the results and stores them in a file for later analysis.

4.5 Conclusion and Future Work

With larger and more complex high performance systems on the horizon, energy efficiency has become one of the critical constraints. To allow the HPC community to "control" power, in addition to the traditional hardware performance counter "monitoring" approach, PAPI has been extended with a component that supports power writing capabilities through the Intel RAPL interface. Whether PAPI is applied as a stand-alone tool or as a middleware by third-party performance analysis tools, the new PAPI component for power controlling can be used without the need for application developers to modify their applications or learn new library primitives.

Furthermore, we introduced PAPI's new Counter Inspection Toolkit, which will be fully integrated for future PAPI releases. It establishes methods to automatically determine which hardware event combinations map to particular high level concepts of performance.

Acknowledgments We thank the anonymous reviewers for their improvement suggestions.

This material is based upon work supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award No. DE-SC0006733 "SUPER—Institute for Sustained Performance, Energy and Resilience," and by the National Science Foundation under award No. 1450429 "PAPI-EX."

References

1. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
2. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)
3. Danalis, A., Luszczek, P., Marin, G., Vetter, J.S., Dongarra, J.: BlackjackBench: Portable hardware characterization with automated results' analysis. Comput. J. **57**(7), 1002–1016 (2013)
4. Drongowski, P.: An introduction to analysis and optimization with AMD CodeAnalyst™Performance Analyzer. Advanced Micro Devices, Inc. (2008)
5. Intel, I.: Intel 64 and IA-32 Architectures Software Developer's Manual - Systems Programming Guide, vol. 3, chap. 14 (2015)
6. Intel™ Performance Tuning Utility. http://software.intel.com/en-us/articles/intel-performance-tuning-utility/
7. Kurzak, J., Luszczek, P., YarKhan, A., Faverge, M., Langou, J., Bouwmeester, H., Dongarra, J.: Multithreading in the PLASMA Library. Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications. Computer and Information Science Series. Chapman and Hall/CRC, Boca Raton (2013)
8. Malony, A.D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., Lamb, C.: Parallel performance measurement of heterogeneous parallel systems with gpus. In: Proceedings of the 2011 International Conference on Parallel Processing, ICPP '11, pp. 176–185. IEEE Computer Society, Washington, DC, USA (2011)
9. McCraw, H., Terpstra, D., Dongarra, J., Davis, K., R., M.: Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q. In: Proceedings of the International Supercomputing Conference 2013, ISC'13, pp. 213–225. Springer, Heidelberg, June (2013)
10. McCraw, H., Ralph, J., Danalis, A., Dongarra, J.: Power Monitoring with PAPI for Extreme Scale Architectures and Dataflow-based Programming Models, pp. 385–391 (2014)
11. McFadden, M., Shoga, K., Rountree, B.: Msr-safe (2015). https://github.com/scalability-llnl/msr-safe
12. Molnar, I.: perf: Linux profiling with performance counters (2009). https://perf.wiki.kernel.org/
13. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with PAPI-C. Tools for High Performance Computing 2009, pp. 157–173 (2009)
14. Treibig, J., Hager, G., Wellein, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures September (2010)
15. Walker, S., Shoga, K., Rountree, B., Morita, L.: Libmsr (2015). https://github.com/scalability-llnl/libmsr
16. Wolf, J.: Programming Methods for the Pentium™ III Processor's Streaming SIMD Extensions Using the VTune™Performance Enhancement Environment. Intel Corporation (1999)