Power Monitoring with PAPI for Extreme Scale Architectures and Dataflow-based Programming Models

Heike McCraw*, James Ralph*, Anthony Danalis*, Jack Dongarra*†‡
*Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, TN, USA
†Oak Ridge National Laboratory, Oak Ridge, TN, USA
‡University of Manchester, Manchester, UK

***Abstract*—For more than a decade, the PAPI performance-monitoring library has provided a clear, portable interface to the hardware performance counters available on all modern CPUs and other components of interest (e.g., GPUs, network, and I/O systems). Most major end-user tools that application developers use to analyze the performance of their applications rely on PAPI to gain access to these performance counters.**

One of the critical roadblocks on the way to larger, more complex high performance systems, has been widely identified as being the energy efficiency constraints. With modern extreme scale machines having hundreds of thousands of cores, the ability to reduce power consumption for each CPU at the software level becomes critically important, both for economic and environmental reasons. In order for PAPI to continue playing its well established role in HPC, it is pressing to provide valuable performance data that not only originates from within the processing cores but also delivers insight into the power consumption of the system as a whole.

An extensive effort has been made to extend the Performance API to support power monitoring capabilities for various platforms. This paper provides detailed information about three components that allow power monitoring on the Intel Xeon Phi and Blue Gene/Q. Furthermore, we discuss the integration of PAPI in PARSEC – a task-based dataflow-driven execution engine – enabling hardware performance counter and power monitoring at true task granularity.

I. INTRODUCTION

Parallel application performance analysis tools on large scale computing systems typically rely on hardware counters to gather performance data. For more than a decade, the PAPI performance-monitoring library has provided consistent platform and operating system independent access to the hardware performance counters available on different CPU families and other components of interest (e.g., GPUs, network, and I/O systems) [1], [2], [3], [4].

With the increase in scale, complexity, and heterogeneity of modern extreme scale systems, it is anticipated that the design of future HPC machines will be driven by energy efficiency constraints. With supercomputers having hundreds of thousands of cores, the ability to reduce power consumption by just a couple of Watts per CPU quickly adds up to major power, cooling, and monetary savings [5].

In order for PAPI to continue playing its well established role in HPC performance optimization, it is crucial to provide valuable performance data that not only originates from within the processing cores but also pays more attention to the power consumption of the system as a whole. We have already demonstrated the merit of transparent access to power and energy measurements via PAPI for Intel Sandy Bridge (and its successors) and for NVIDIA GPUs in [5]; the viability of PAPI RAPL energy consumption and power profiles for studying advanced dense numerical linear algebra in [6]; and the relevance of PAPI providing power and energy measurement abilities for virtualized cloud environments in [7]. Recently, an additional effort has been made to extend the Performance API with new components supporting transparent power monitoring capabilities for Intel Xeon Phi co-processors and the IBM Blue Gene/Q system. This paper provides detailed information describing these new components as well as exploring the usefulness of each of them with different case studies.

Exposing performance counter and energy readings for post-mortem analysis is only a partial target for PAPI. A tight integration with an execution environment that can take advantage of this information online – specifically, during the execution of the monitored application – is of high interest. Ideally, such a programming environment will amend either its execution pattern and behavior to match specific goals (e.g., lowering energy consumption), or adjust the frequency, and thus the power usage, based on the known application behaviors. We discuss therefore the integration of PAPI in PARSEC, which is a task-based dataflow-driven execution engine that enables efficient task scheduling on distributed systems, providing a desirable portability layer for application developers. Dataflow-based programming models, in contrast to the control flow model (e.g., as implemented in languages such as C), have become increasingly popular, especially on distributed heterogeneous architectures. Consequently, performance measurement tools for task-based dataflow-driven runtimes, like the Parallel Runtime Scheduling and Execution Controller (PARSEC) [8], [9], have become increasingly important. Our early prototyping work of the integration of PAPI into PARSEC has proven to be valuable as it allows hardware performance counter measurements at a finer granularity – more precisely, at true task granularity as opposed to thread/process granularity – providing a richer and more precise mapping between PAPI measurements and application behavior.

In this paper we outline two new PAPI components that support power and energy monitoring for the Intel Xeon Phi co-processors and the IBM Blue Gene/Q system. A detailed

description of their monitoring capabilities is provided in addition to case studies that validate and verify the measurements. Furthermore, we briefly describe PaRSEC, and then focus on the integration of PAPI into PaRSEC, exploring the benefits of measuring hardware counter data at task granularity.

II. Overview

A. Performance API (PAPI)

While PAPI can be used as a stand-alone tool, it is more commonly applied as a middleware by third-party profiling, tracing, sampling, even auto-tuning tools – such as TAU [10], Scalasca [11], Vampir [12], HPCToolkit [13], CrayPat [14], Active Harmony [15], among others – to provide coherent access to performance counters that can be found on a wide variety of architectures.

The events that can be monitored involve a wide range of performance-relevant architectural features: cache misses, floating point operations, retired instructions, executed cycles, and many others. Over time, other system components, beyond the processor, have gained performance interfaces (e.g., GPUs, network and I/O interfaces). To address this change, PAPI was redesigned to have a component-based architecture that applies a modular concept for accessing these new sources of performance data [2]. With this redesign, additional PAPI components have been developed to also address subsets of data and communication domains – enabling users to measure I/O performance, and to monitor synchronization and data exchange between computing elements.

Furthermore, more attention has been paid to the monitoring of power usage and energy consumption of systems. A number of PAPI components have been publicly available since the PAPI 5.0.0 release, allowing for transparent power and energy readings via (a) the Intel RAPL ("Running Average Power Limit") interface [16] for Intel Sandy Bridge chips and its successors, and (b) the NVML ("NVIDIA Management Library") interface [17] for NVIDIA GPUs. More details on this work is described in [5], [6], [7]. In recent work, we build on these results and extended PAPI's current power monitoring features to other architecture; specifically Intel Xeon Phi co-processors and the IBM Blue Gene/Q architecture.

B. PaRSEC

A new direction that we have been exploring in PAPI is the support for task-based counter measurements. This is important because task-based runtime systems, such as PaRSEC (Parallel Runtime Scheduling and Execution Controller) [8], [9], are become increasingly popular in the high performance computing community. The PaRSEC framework is a task-based dataflow-driven system designed as a dynamic platform that can address the challenges posed by distributed heterogeneous hardware resources. PaRSEC contains schedulers that orchestrate the execution of the tasks on the available hardware, based on different criteria, such as data locality, task priority, resource contention, etc. In addition, task scheduling is affected by the dataflow between the tasks that comprise the user application. To store the information regarding the tasks and their dependencies PaRSEC uses a compact, symbolic representation known as a Parameterized Task Graph (PTG) [18]. The runtime combines the information contained in the PTG with supplementary information provided by the user – such as the distribution of data onto nodes, or hints about the relative importance of different tasks – in order to make efficient scheduling decisions.

PaRSEC is an event driven system. Events are triggered when tasks complete their execution, or data exchanges between tasks on different nodes occur. When an event occurs, the runtime examines the dataflow of the tasks involved to discover the future tasks that can be executed based on the data generated by the completed task. Since the tasks and their dataflow are described in the PTG, discovering the future tasks, given the task that just completed, does not involve expensive traversals of DAG structures stored in program memory. This contrasts to other task scheduling systems which rely on building the whole DAG of execution in memory at run-time and traversing it in order to make scheduling decisions.

In PaRSEC the runtime performs all necessary data exchanges without user intervention. This is possible because the PTG provides the necessary information regarding the data that each task needs in order to execute, and the runtime is aware of the mapping of tasks onto compute nodes.

Optimizing a task-based application can be achieved by optimizing the dataflow between tasks, and/or optimizing the tasks themselves. Acquiring sufficient information about the behavior of individual tasks through measurement tools such as PAPI, is critical for performing such optimizations. To this end, PaRSEC includes a modular instrumentation system with modules that can be selectively loaded at initialization by the user. In Section V we demonstrate how PAPI measurements can provide insight about the execution of a linear algebra code when using a task-based implementation over PaRSEC versus a legacy alternative.

III. Power monitoring on Xeon Phi co-processors

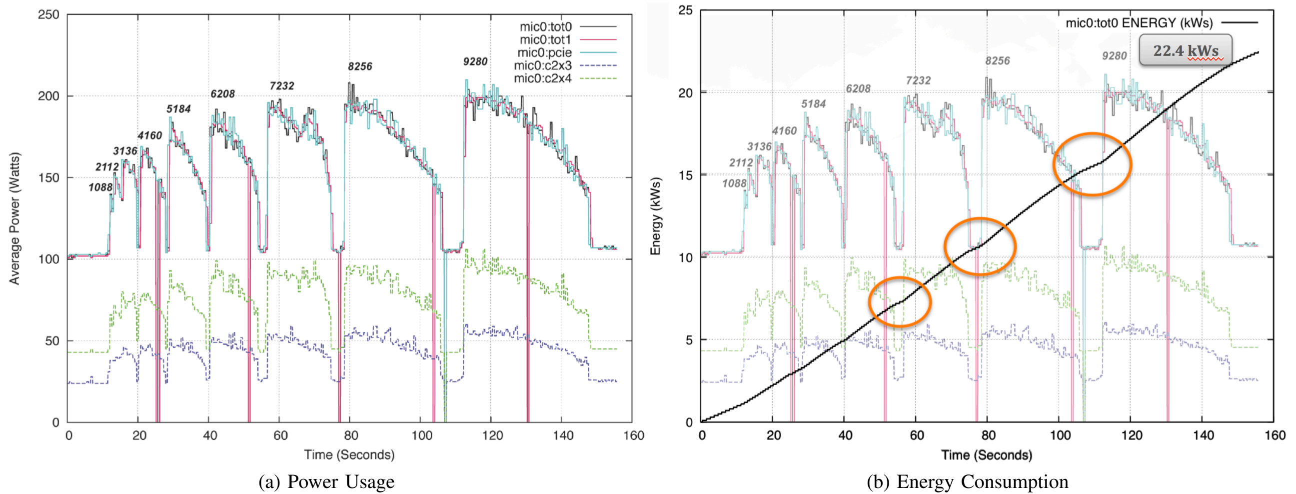
Power and energy measurement activities continue to be of importance for both PAPI and the larger HPC community. The latest PAPI release (5.3) offers two components that allow for monitoring of power usage and energy consumption on the Intel Xeon Phi (a.k.a. MIC) architecture.

A. Direct Power Reading

The "`micpower`" component runs in native mode, meaning, both the actual application as well as PAPI are running natively on the co-processor and its operating system, without being offloaded from a host system. This component provides access to an on-board power sensor on the Xeon Phi which allows measurement of current and voltage (and computed power) for various subsystems on the MIC card at roughly 50 milliseconds resolution. The power values are periodically read from the contents of the file `/sys/class/micras/power`. Table I provides a list of events that a user can choose from to obtain power readings for an application running on the Xeon Phi co-processor.

B. Offloaded Power Reading

The second component, called "`host_micpower`", appears to be more convenient for users as PAPI is offloaded from the host system, and only the application is running on the



(a) Power Usage

(b) Energy Consumption

Fig. 1: PAPI Power measurements of 9 Hessenberg reductions (each with a different matrix size) performed on Xeon Phi.

Event	Description
tot0, tot1	Total (average) power consumption over two different time windows (uW)
pcie	power measured at the PCI-express input (connecting CPU with the Phi) (uW)
inst imax	Instantaneous power consumption reading (uW) Maximum instantaneous power consumption observed (uW)
c2x3, c2x4	power measured at the input of the two power connectors located on the card (uW)
vccp vddg vddq	Power supply to the cores (core rail) (Current (uA), Voltage (uV), and Power reading (uW)) Power supply to everything but the cores and memory (uncore rail) (Current (uA), Voltage (uV), and Power reading (uW)) Power supply to memory subsystem (memory rail) (Current (uA), Voltage (uV), and Power reading (uW))
The vccp, vddg, and vddq rails are powered from the PCI Express connector and the supplementary 12V inputs [19].	

TABLE I: Power events on the Intel Xeon Phi co-processor

Xeon Phi. In this case, the power data is exported through the MicAccess API, which is distributed with the Intel Manycore Platform Software Stack (MPSS) [20]. The additional support for reading Phi power from the host system, makes it much easier to measure power consumption of MIC code at fairly high resolution without actually instrumenting the MIC code directly. The events that can be monitored are the same as for the `micpower` component, which are listed in Table I.

As part of the `host_micpower` component – and mainly as convenience for the users – PAPI ships a utility that can be used to gather power (and voltage) measurements. The tool works by using PAPI to poll the MIC power statistics every 100 milliseconds and dumps each statistic to different files, which then can be use for post-mortem analysis.

C. Case Study: Hessenberg Reduction

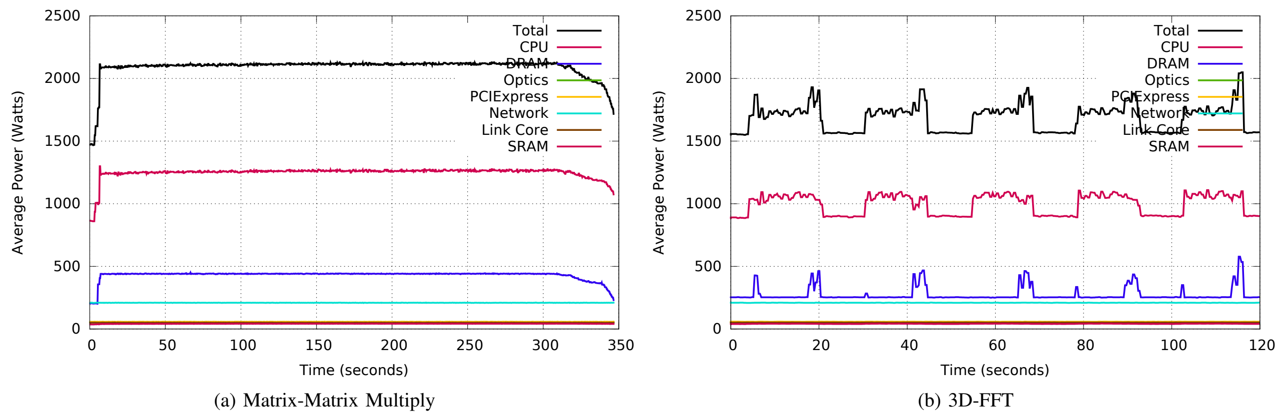
For our case study we are running a Hessenberg reduction kernel from the MAGMA library [21] computed on the Xeon Phi coprocessor utilizing all 244 cores. For the power readings, we are using the `host_micpower` component and the measurement utility with a sampling rate of 100 milliseconds. We have validated the power measurements obtained from this instrumentation to ensure that it correlates with software activity on the hardware. Figure 1a shows the total power data over two time windows (black solid line for window 0; magenta solid line for window 1). The light-blue line shows power data measured at the PCI-Express bus that connects the CPU host with the co-processor. Furthermore, the two dotted lines show power measured at the input of the two power connectors that are located on the card.

The Hessenberg reduction is computed nine times, each with a different matrix size. The numbers on top of each curve show the chosen matrix size. The measured power data clearly mimics the computational intensity of the Hessenberg computations. The factorization starts off on the entire matrix – in this case consuming most of the power – and as the factorization progresses, it operates on smaller and smaller matrices, resulting in less and less power usage.

Additionally, the energy consumption can be computed from the power numbers measured by PAPI. In Figure 1b, the energy consumption curve is placed on top of the power usage graph. The power plot clearly demonstrates a declining power usage for the window between the Hessenberg computations, when data is exchanged with the CPU host. As expected, this is also reflected on the energy curve in a form of a decreased slope for this time window. The energy slope peaks again as soon as the next computation starts. The total energy consumption until completion is 22.4 kWs.

IV. Power Monitoring on Blue Gene/Q

The Blue Gene/Q (BG/Q) system is the third generation in the IBM Blue Gene line of massively parallel, energy efficient supercomputers that increases not only in size but also in complexity compared to its Blue Gene predecessors. Recent studies ([22], [23]) show that gaining insight into the power usage and energy consumption of applications running on the BG/Q system is of high interest to the HPC community. Since



(a) Matrix-Matrix Multiply

(b) 3D-FFT

Fig. 2: PAPI BG/Q Power Usage sampled at 300ms

PAPI is a portable cross-platform interface that allows users on many types of machines to access performance information, we build on this model and extended PAPI's previous BG/Q support (described in [4]) with power monitoring capabilities. We have developed a PAPI component for a high speed power measurement API for Blue Gene/Q, called EMON, to provide access to power and energy data on BG/Q in a transparent fashion. This additional PAPI support enables users and tool developers to use their PAPI instrumented code as is on BG/Q without having to learn a new set of library and instrumentation primitives.

A Blue Gene/Q rack contains two mid-planes, each consisting of 16 node-boards. With each node-board comprising 32 compute cards, the card itself holds an 18-core PowerPC A2 processor and 16 GB of memory. Out of the 18 cores, only 16 are used to perform mathematical calculations.[1] Each processor operates at a moderate clock frequency of 1.6 GHz, consuming a modest 55 Watts at peak [24]. Also, each node-board contains a field programmable gate array (FPGA) that captures all temperature and voltage data [25]. More details on the IBM Blue Gene/Q architecture are discussed in [25], [26].

A. EMON Power Reading

The PAPI "`bgq-emon`" component exposes the power data through the IBM EMON API. Power is supplied to a node-board by two identical main power modules providing seven high voltage Direct Current (DC) power lines. Each power line from one main power module is paired with the equivalent line from the other module, and a step down transformer provides final voltage for the compute cards. These lines or "domains" are described in Table II. These are also the power statistics that can be monitored through the PAPI `bgq-emon` component.

Current is measured on each of the 14 domains leaving the main power modules, and voltage is measured after the final step-down transformation. The FPGA queries a micro-controller for each measurement point sequentially and is able to take a complete sample in $\sim 500\mu$s. The IBM EMON interface accesses this data from the node-board FPGA, which includes a 1-10ms blocking delay for the caller.

[1]The 17th core is used for node control tasks such as offloading I/O operations which "talk" to Linux running on the I/O node. The 18th core is a spare core which is used when there are corrupt cores on the chip.

Domain	Description
1	Chip core voltage (0.9V)
2	Chip memory interface and DRAM (1.35V)
3	Optics (2.5V)
4	Optics + PCIexpress (3.5V)
6	Chip HSS network transceiver (1.5V)
8	Link chip core (1V)
7	Chip SRAM voltage (0.9+0.15V)

TABLE II: Power Domains on BG/Q node-boards

It is important to note that we have also developed a PAPI interface for EMON2 which is a more advanced power measurement API for BG/Q. This API provides integrated power and energy measurements at the node level. Since the EMON2 functionality is not included in the supported firmware yet, nor is it distributed with the driver on the BG/Q systems at ANL (Mira/Vesta) and Juelich-Germany (JuQueen), we have not been able to test (and release) our component yet. The current EMON interface has an update latency of ~300ms, while the new EMON2 interface should allow for ~10ms sampling, and for finer granularity sampling on a subset of the domains.

B. Case Studies

In the following demonstrations PAPI was used to collect the data via the `bgq-emon` component. On one rank per node-board we register to receive a standard *SIGPROF* Unix profiling signal, which, when received, triggers a routine to sample EMON data. A *ITIMER_PROF* system timer is then requested to trigger at an interval of 300ms, calling the sampling routine.

1) Matrix-Matrix Multiply: Our first case study performs a matrix multiply calculation (`gemm`), which is a common kernel with heavy processor and memory requirements. A

benchmark was constructed to run many small gemms on all 512 cores in a node-board. The observed power graph, seen in Figure 2a, demonstrates a rapid climb to full core utilization with a reduction past five minutes as memory is freed and the program winds down. This experiment reproduces data observed in [23] where the EMON API is used directly for power readings.

2) Parallel 3D-FFT: As a second case study, we implemented a parallel 3D-FFT kernel with 2D decomposition of the data array. We start the discussion with a short overview of the kernel implementation. Consider $A_{x,y,z}$ as a three-dimensional array of $L \times M \times N$ complex numbers. The Fourier transformed array $\tilde{A}_{u,v,w}$ is computed using the following formula:

$$\tilde{A}_{u,v,w} := \underbrace{\sum_{x=0}^{L-1} \underbrace{\sum_{y=0}^{M-1} \underbrace{\sum_{z=0}^{N-1} A_{x,y,z} \exp(-2\pi \frac{wz}{N})}_{\text{1st 1D FT along } z} \exp(-2\pi \frac{vy}{M})}_{\text{2nd 1D FT along } y} \exp(-2\pi \frac{ux}{L})}_{\text{3rd 1D FT along } x} \quad (1)$$

Each of the three sums in Eq. 1 is evaluated by using the fast Fourier transform algorithm (FFT) in one dimension. To achieve a decent parallel speedup of the 3D-FFT, the array $A_{u,v,w}$ is best distributed onto a two-dimensional virtual processor grid, leaving one direction of the array local to the processor. This allows the first set of FFTs to be evaluated locally without any communication. Before further FFTs in the originally distributed directions can be evaluated, the virtual processor grid needs to be rearranged. When using a 2D processor grid, two groups of All-to-All type communications are required. Assuming the data array is already distributed onto the 2D processor grid of dimension $P_c \times P_r$, with the z-direction being local to the processors, the basic algorithm looks as follows:

- Each processor performs $L/P_r \times M/P_c$ 1D-FFTs of size N
- An all-to-all communication is performed within each of the rows of the virtual processor grid to redistribute the data. At the end of the step, each processor holds an $L/P_r \times M \times N/P_c$ sized section of A. These are P_r independent all-to-all communications.
- Each processor performs $L/P_r \times N/P_c$ 1D-FFTs of size M.
- A second set of P_c independent all-to-all communications is performed, this time within the columns of the virtual processor grid. At the end of this step, each processor holds a $L \times M/P_c \times N/P_r$ size section of A.
- Each processor performs $M/P_c \times N/P_r$ 1D-FFTs of size L

More details on the parallelization are discussed in [27], [28].

We ran our 3D-FFT kernel on a 32 node partition, utilizing an entire node-board on the BG/Q system at Argonne National Laboratory, using all 16 compute cores per node for each run. For the 32 node partition, we have a total of 512 MPI tasks, and for the virtual 2D processor grid, we chose 16×32, meaning that each communicator group has 16 MPI tasks and we have 32 of those groups.

The power data for the 3D-FFT computation – running on an entire node-board – is shown in Figure 2b for a problem size of 1024^3. The entire kernel is computed five times with a 10 second pause in-between. The power data nicely reflects this with an expected power drop between each of the five runs. Examining the DRAM power line (blue), for each FFT there is an initial spike due to the generation of synthetic data for the FFT. After the 3D data array is generated and distributed onto the 2D virtual processor grid, the FFTW plan is computed. This step takes the longest, for our experiment approx. 10 seconds. The dip that spans over about 10 seconds – immediately after the initial spike of the DRAM power line – is precisely due to the plan creation. Following the plan, the FFTs are computed according to Eq. 1 which includes data redistribution between each of the three sets of 1D-FFTs, causing the power usage to peak again. These power measurements via the PAPI bgq-emon component closely match the intuitive understanding of the behavior of our 3D-FFT implementation.

V. Power Monitoring for Dataflow-Based Execution Systems

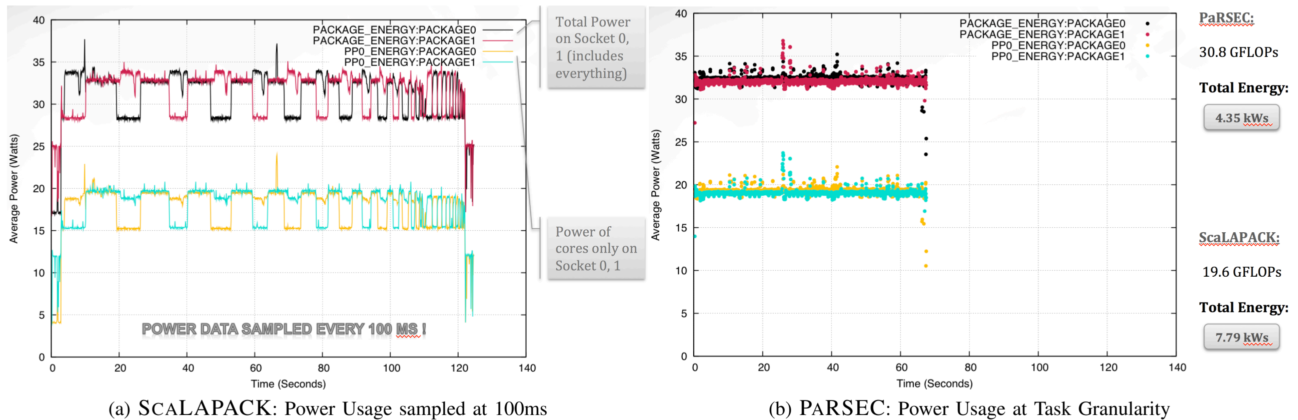
Harnessing the processing power of modern large scale computing platforms has become increasingly difficult. This is one of the reasons why dataflow-based programming models – in contrast to the control flow model (e.g., as implemented in languages such as C) – have become increasingly popular, especially on distributed heterogeneous architectures. Consequently, performance tools for task-based dataflow-driven runtimes – like PaRSEC – have become increasingly important.

We started exploring the integration of PAPI into PaRSEC to allow hardware performance counter measurements at pure task granularity. Our early prototyping work proves to be valuable as can be observed in the example below where we demonstrate power measurements at task granularity.

Figures 3a and 3b show the power usage of a QR factorization performed by two threads running on two different sockets (packages) of a shared memory machine. Figure 3b shows the power usage for the dataflow-based QR factorization implemented over PaRSEC. Each dot in the figure illustrates the power measurements for one task – measured via the PAPI linux-rapl component from within PaRSEC. As a comparison, Figure 3a shows the power usage of the legacy QR factorization found in ScaLAPACK, using the same matrix size and block size. The power data for this run is sampled every 100 ms using the original PAPI linux-rapl component.

The linux-rapl component (featuring power and energy measurement capabilities for the Intel Sandy Bridge architecture and its successors) ensures access to the full 32-bits of dynamic range of the register. The RAPL events are reported either as scaled values with units of joules, watts, or seconds; or, with slightly different event names, they can be reported as raw binary values, suitable for doing arithmetic. More details on the linux-rapl component are discussed in [5].

Figure 3a exhibits a clear pattern in the energy usage of the two ScaLAPACK threads. This matches the intuitive understanding of the behavior of this code, since the two threads take turns performing a panel factorization (while



(a) SCALAPACK: Power Usage sampled at 100ms

(b) PARSEC: Power Usage at Task Granularity

Fig. 3: PAPI RAPL Power measurements of dgeqrf – MatrixSize = 11,584 – TileSize = 724 on a Sandy Bridge EP 2.60GHz, 2 sockets, running on 1 (out of 8) core per socket.

one thread works on the panel the other is idle) followed by a trailing matrix update performed by both. We also see that as the factorization progresses the turns become smaller since both threads work on a smaller submatrix. Figure 3b on the other hand shows a very different picture. Namely, when PARSEC is used the threads perform work opportunistically, based on the available tasks at each time as opposed to the predetermined logical steps found in SCALAPACK. Therefore, neither CPU is ever idle, which leads to a uniform power consumption and a shorter execution time.

In this experiment we chose to execute only one thread per socket. We made this choice because power is a shared resource, i.e., there is a single counter per socket that is shared between all threads. Measuring the power consumption of tasks executing on multiple threads located on the same CPU is also possible, but requires post-mortem analysis of the measurements. In such a scenario, each measurement will correspond to power consumed by different segments of tasks executing on the different cores of a socket. Thus, when all measurements have been collected, inferring the average power consumption of each thread type requires solving a system of linear equations [29].

VI. Related Work

The PAPI interface has been widely used by HPC users for many years, drawing on its strength as a cross-platform and cross-architecture API. There are many other tools for gathering performance information but they are often not as flexible as PAPI, and none of the related tools provide an API.

The *likwid* lightweight performance tools project [30] 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. 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.

The *perf tool* [31] comes with the Linux kernel. It provides performance counter measurements, both traditional and uncore. It does not present an API; it is a complex command line tool that either prints total results to the screen or else records the results to disk for later analysis.

Processor vendors supply tools for reading performance counter results. This includes *Intel VTune* [32], *Intel VTune Amplifier*, *Intel PTU* [33], and *AMD's CodeAnalyst* [34]. Like likwid, these program the CPU registers directly. Since counter state is not saved on context switch only system wide sampling is available. There is also no API for accessing the results.

VII. 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 monitor power and energy consumption on the Intel Xeon Phi co-processor and the IBM Blue Gene/Q system, PAPI has been extended with components supporting transparent power reading capabilities through existing interfaces. These additional components allow 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.

For future PAPI releases, we will build on these components and extend out work to (a) EMON2 for Blue Gene/Q, and (b) other architecture (e.g., transparent access to the Application Power Management (APM) introduced my AMD for the 15h family of processors). We have also plans to support distributed memory environments by integrating energy information from the interconnect, providing users with a comprehensive view of energy efficiency for large scale clusters.

Additionally, one of the main objectives of PAPI 6 will be to offer new levels of performance counter measurements by seamlessly incorporating technologies for monitoring at task granularity.

Acknowledgment

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". This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000.

[2] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting Performance Data with PAPI-C," *Tools for High Performance Computing 2009*, pp. pp. 157–173, 2009.

[3] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with gpus," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 176–185.

[4] H. McCraw, D. Terpstra, J. Dongarra, K. Davis, and M. R., "Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q," in *Proceedings of the International Supercomputing Conference 2013*, ser. ISC'13. Springer, Heidelberg, June 2013, pp. 213–225.

[5] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring Energy and Power with PAPI," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, Sept 2012, pp. 262–268.

[6] J. Dongarra, H. Ltaief, P. Luszczek, and V. Weaver, "Energy Footprint of Advanced Dense Numerical Linear Algebra Using Tile Algorithms on Multicore Architectures," in *Cloud and Green Computing (CGC), 2012 Second International Conference on*, Nov 2012, pp. 274–281.

[7] V. Weaver, D. Terpstra, H. McCraw, M. Johnson, K. Kasichayanula, J. Ralph, J. Nelson, P. Mucci, T. Mohan, and S. Moore, "PAPI 5: Measuring Power, Energy, and the Cloud," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, April 2013, pp. 124–125.

[8] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *IEEE Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, nov 2013.

[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing." *Parallel Computing*, vol. 38, no. 1-2, pp. 27–51, 2012.

[10] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006.

[11] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[12] H. Brunst and A. Knpfer, "Vampir," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 2125–2129.

[13] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[14] "CrayPat – User's Manual," http://docs.cray.com/books/S-2315-50/html-S-2315-50/z1055157958smg.html.

[15] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, ser. SC'02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–11.

[16] "Intel, Intel Architecture Software Developers Manual, Volume 3: System Programming Guide, 2009."

[17] "NVML Reference Manual, NVIDIA, 2012," http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/NVML/nvml.pdf.

[18] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*. Washington, DC: IEEE Computer Society, 1995.

[19] "Intel, Intel Xeon Phi Coprocessor Datasheet, April, 2014."

[20] "Intel, Intel Manycore Platform Software Stack (MPSS)," https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[21] "MAGMA: Matrix Algebra on GPU and Multicore Architectures," http://icl.cs.utk.edu/magma/index.html.

[22] K. Yoshii, K. Iskra, R. Gupta, P. H. Beckman, V. Vishwanath, C. Yu, and S. M. Coghlan, "Evaluating power monitoring capabilities on ibm blue gene/p and blue gene/q," in *CLUSTER '12*, IEEE. Beijing, China: IEEE, 09/2012 2012.

[23] S. Wallace, V. Vishwanath, S. Coghlan, J. Tramm, Z. Lan, and M. Papkay, "Application power profiling on ibm blue gene/q," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, Sept 2013, pp. 1–8.

[24] M. Feldman, "IBM Specs Out Blue Gene/Q Chip," 2011, http://www.hpcwire.com/hpcwire/2011-08-22/ibm_specs_out_blue_gene_q_chip.html.

[25] J. Milano and P. Lembke, "IBM System Blue Gene Solution: Blue Gene/Q Hardware Overview and Installation Planning," *IBM Redbook SG24-7872-01*, 2013.

[26] M. Gilge, "IBM system Blue Gene solution: Blue Gene/Q application development," *IBM Redbook Draft SG24-7948-00*, 2012.

[27] M. Eleftheriou, J. E. Moreira, B. G. Fitch, and R. S. Germain, "A Volumetric FFT for BlueGene/L," *Lecture Notes in Computer Science*, vol. 2913/2003, pp. 194–203, 2003.

[28] H. Jagode, "Fourier Transforms for the BlueGene/L Communication Network," Master's thesis, EPCC, The University of Edinburgh, 2006, http://www.epcc.ed.ac.uk/msc/dissertations/2005-2006/.

[29] Q. Liu, M. Moreto, V. Jimenez, J. Abella, F. J. Cazorla, and M. Valero, "Hardware support for accurate per-task energy metering in multicore systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, pp. 34:1–34:27, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2541228.2555291

[30] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. of the First International Workshop on Parallel Software Tools and Tool Infrastructures*, Sep. 2010.

[31] I. Molnar, "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org/, 2009.

[32] J. Wolf, *Programming Methods for the Pentium™ III Processor's Streaming SIMD Extensions Using the VTune™ Performance Enhancement Environment*, Intel Corporation, 1999.

[33] "Intel™ Performance Tuning Utility," http://software.intel.com/en-us/articles/intel-performance-tuning-utility/.

[34] P. Drongowski, *An introduction to analysis and optimization with AMD CodeAnalyst™ Performance Analyzer*, Advanced Micro Devices, Inc., 2008.