2015 IEEE International Conference on Cluster Computing

Comparison of Vendor Supplied Environmental Data Collection Mechanisms

Sean Wallace*[†], Venkatram Vishwanath[†], Susan Coghlan[†], Zhiling Lan*, and Michael E. Papka[†][‡]

*Illinois Institute of Technology, Chicago, IL, USA
[†]Argonne National Laboratory, Argonne, IL, USA
[‡]Northern Illinois University, DeKalb, IL, USA
swallac6@iit.edu, venkat@anl.gov, smc@anl.gov, lan@iit.edu, and papka@anl.gov

***Abstract*—The high performance computing landscape is filled with diverse hardware components. A large part of understanding how these components compare to others is by looking at the various environmental aspects of these devices such as power consumption, temperature, etc. Thankfully, vendors of these various pieces of hardware have supported this by providing mechanisms to obtain this data. However, differences not only in the way this data is obtained but also the data which is provided is common between products.**

In this paper, we take a comprehensive look at the data which is available for the most common pieces of today's HPC landscape, as well as how this data is obtained and how accurate it is. Having surveyed these components, we compare and contrast them noting key differences as well as providing insight into what features future components should have.

***Index Terms*—Environmental Data, Power Profiling, Blue Gene/Q, Intel Xeon Phi, NVML, RAPL**

I. Introduction

In recent years, supercomputers have become more heterogeneous and now commonly employ acceleration devices to help with several aspects of the computation. As systems become larger and more complex, it becomes increasingly difficult to get an accurate picture of what the "environmental" aspects (e.g., motherboard, CPU, GPU, hard disk and other peripherals' temperature, voltage, current, fan speed, etc.) of the system are like with any decent accuracy. Putting accurate sensors in hardware is an expensive proposition, therefore hardware manufacturers do so sparingly and only where really necessary primarily for diagnostic purposes. What's more, there is a distinct lack of tools available to access and interpret this data across a variety of systems. As a result, data like power consumption, temperature, etc. are some metrics which are largely not understood on a system level.

A Supercomputing 2011 State of the Practice Report [1] highlighted a number of difficulties in the monitoring of large systems and provided some insight into what would alleviate those difficulties. Two of these suggestions are of particular interest to this paper. First, better power monitoring and control is going to be critical as we move to exascale. This includes accurate power consumption and control at the sub-system level (CPU, RAM, NIC, etc.). Secondly, there needs to be standard interfaces to monitoring data. If there were better out of the box monitoring and a standard that vendors could work against for exposing the monitoring data, we could largely eliminate effort expended in that area and focus on higher level tools that turn that into useful, actionable information.

Once obtained, this information is useful in a number of ways which have already shown promising results. In our own previous work previous work [2] we proposed a power aware scheduling design which using power data from IBM Blue Gene/Q resulted in savings of up to 23% on the electricity bill.

Clearly then, it is important not only that this data be available, but also relatively easy to gather. To this end, we seek to investigate just what data is currently available and what the process of collecting it is like. Further, we seek to provide insight into what we think future generations of hardware should look like in terms of environmental data collection.

The majority of this paper will deal with discussions about "environmental" data. As will be discussed, most platforms today support power collection at some level, however we do not want to focus on power collection alone. As such, we will also discuss what mechanisms are in place for collection of data other than power consumption. More specifically, we provide the following contributions:

- We discuss and analyze the obtainable data from four major hardware platforms common in HPC today. With each we discuss what the procedure to obtain the data are, how reliable the data are, what frequency the data can be reliably obtained, and show what this data looks like for some benchmarks.
- We discuss our power profiling library, MonEQ, which we extended in this work to support all of the data access mechanisms discussed throughout this paper. We show that with as few as two lines of code on any of the hardware platforms mentioned in this paper one can easily obtain environmental data for analysis.

The remainder of this paper is as follows: we will introduce the problem of environmental data collection in Section II which will include detailed analysis of four popular platforms; the Blue Gene/Q, Intel's RAPL interface, NVIDIA GPUs via the NVIDIA Management Library, and finally the Intel Xeon Phi. We will discuss the power profiling library, MonEQ,

978-1-4673-6598-7/15 $31.00 © 2015 IEEE
DOI 10.1109/CLUSTER.2015.120

which we developed to obtain these results in Section III. Finally, we will provide our conclusions and discussion of future systems in Section IV.

II. Vendor Supplied APIs

Power measurement and therefore analysis would not be possible without sensors deployed in hardware. From the CPU in a node of a supercomputer to the memory on an accelerator, there must be sensors present to gather meaningful data. The presence of these sensors alone however is not enough, hardware manufacturers must provide end users the ability to gather the information which these sensors gather. Fortunately, every major hardware manufacturer does provide access to this data in one way or another. Most commonly, this is done through access to an exposed low-level API. However, this is certainly not the only way. Some systems, such as the Intel Xeon Phi, employ a daemon approach where a process takes care of the data gathering and the actual collection is done by reading a pseudo-file mounted on a virtual file system. Other systems, such as Intel processors, have neither a daemon nor an API and instead expose direct access to the registers which hold environmental data through kernel drivers. Clearly then, there is hardly a uniform method of access.

Aside from the collection process, the exact location of these sensors is what determines what data can be gathered. Obviously, going so far as to put sensors in the processor registers would be too costly and likely overkill. On the other hand, putting only a single sensor on the CPU die would be inexpensive, however only represent one small portion of the system on the whole. As far as devices currently available, there is a wide variety in location, count, sampling frequency, as well as other aspects. Said another way, in certain cases, it's not possible to gather the exact same type of data between two devices, or, it is possible to collect the same data, but the collection frequency is different. While it's clear that the differences in devices today warrants speculation of a head-to-head comparison, there are certain situations where it would be beneficial to look at two devices in terms of their environmental data. An overview of the various sensors present on the devices being discussed is presented in Table I.

This section then seeks to serve a number of purposes. First, we will take an in-depth look at the most common devices found in high performance computing systems. This discussion will include what type of data exists, how this data is accessed, the accuracy of this data, how frequently it can be collected, as well as how useful it is. Secondly, we will show some examples of what this data looks like at scale with a variety of applications written specifically for these devices. Finally, we will have an informal discussion of what we believe future systems should look like in terms of environmental data.

A. Blue Gene/Q

We have looked extensively at the IBM Blue Gene/Q (BG/Q) in our previous research [3], [4]. The Blue Gene/Q architecture is described in detail in [5]. Our analysis of power usage on BG/Q is based on Argonne National Laboratory's 48-rack BG/Q system, Mira. A rack of a BG/Q system consists of two midplanes, eight link cards, and two service cards. A midplane contains 16 node boards. Each node board holds 32 compute cards, for a total of 1,024 nodes per rack. Each compute card has a single 18-core PowerPC A2 processor [6] (16 cores for applications, one core for system software, and one core inactive) with four hardware threads per core, with DDR3 memory. BG/Q thus has 16,384 cores per rack.

TABLE I
Comparison of environmental data available for the Intel Xeon Phi, NVIDIA GPUs, Blue Gene/Q, and RAPL.

	Xeon Phi	**NVML**	**Blue Gene/Q**	**RAPL**
Total Power				
Consumption (Watts)	✓	✓	✓	✓
Voltage	✗	✓	✓	✓
Current	✗	✓	✓	✗
PCI Express	✓	✗	✓	N/A
Main Memory	✗	✗	✓	✓
Temperature				
Die	✓	✓	✗	✗
DDR/GDDR	✓	✗	✗	✗
Device	✗	✓	✗	✓
Intake (Fan-In)	✓	✓	N/A	N/A
Exhaust (Fan-Out)	✓	✓	N/A	N/A
Main Memory				
Used	✓	✓	✓	✗
Free	✓	✓	✓	✗
Speed (kT/sec)	✓	✗	✓	✗
Frequency	✓	✗	✓	✗
Voltage	✓	✗	✓	✗
Clock Rate	✓	✓	✓	✗
Processor				
Voltage	✓	✗	✓	✓
Frequency	✓	✗	✓	✓
Clock Rate	✓	✓	✓	✓
Fans				
Speed (In RPM)	✓	✓	N/A	N/A
Limits				
Get/Set Power Limit	✓	✓	✗	✓

In each BG/Q rack, bulk power modules (BPMs) convert AC power to 48 V DC power, which is then distributed to the two midplanes. Blue Gene systems have environmental monitoring capabilities that periodically sample and gather environmental data from various sensors and store this collected information together with the timestamp and location information in an IBM DB2 relational database – commonly referred to as the environmental database [7]. These sensors are found in locations such as service cards, node boards, compute nodes, link chips, bulk power modules (BPMs), and the coolant environment. Depending on the sensor, the information collected ranges from various physical attributes such as temperature, coolant flow and pressure, fan speed, voltage, and current. This sensor data is collected at relatively long polling intervals (about 4 minutes on average but can be configured anywhere within a range of 60-1,800 seconds), and while a shorter polling interval would be ideal, the resulting volume of data alone would exceed the server's processing capacity.

The Blue Gene environmental database stores power consumption information (in watts and amperes) in both the input and output directions of the BPM. An example of the power data collected is presented in Figure 1. In this instance, the job running was the million messages per second (MMPS) benchmark [8]. The MMPS benchmark helps us understand *how many messages can be issued per unit time*. It measures the interconnect messaging rate, which is the number of messages that can be communicated to and from a node within unit of time.

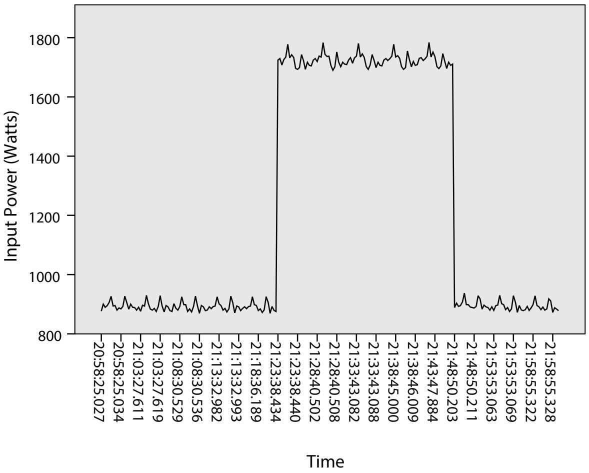

Fig. 1. Power as observed from the data collected at the bulk power supplies. The idle period before and after the job is clearly observable.

In addition to the environmental database, IBM provides interfaces in the form of an environmental monitoring API called EMON that allows one to access power consumption data from code running on compute nodes, with a relatively short response time. The power information obtained using EMON is *total* power consumption from the oldest generation of power data. Furthermore, the underlying power measurement infrastructure does not measure all domains at the exact same time. This may result in some inconsistent cases, such as the case when a piece of code begins to stress both the CPU and memory at the same time.

In the mean time, we needed a reliable and accurate way to measure this data. Out of this necessity MonEQ [9], a "power profiling library" that allows us to read the individual voltage and current data points for each of the 7 BG/Q domains, was born. Much more discussion on MonEQ and its features follows in Section III.

One limitation of the EMON API that we cannot do anything about is that it can only collect data at the node card level (every 32 nodes). This limitation is part of the design of the system and it is not possible to overcome in software.

The same MMPS benchmark as observed by MonEQ is presented in Figure 2. As can be seen, the power consumption of the node card matches that of the data collected at the BPM in terms of total power consumption and overall length, but since the data is collected by MonEQ at run time, the idle period before and after the application run is no longer visible. What's more, because of the higher sampling frequency, there are many more data points than observed from the BPM.

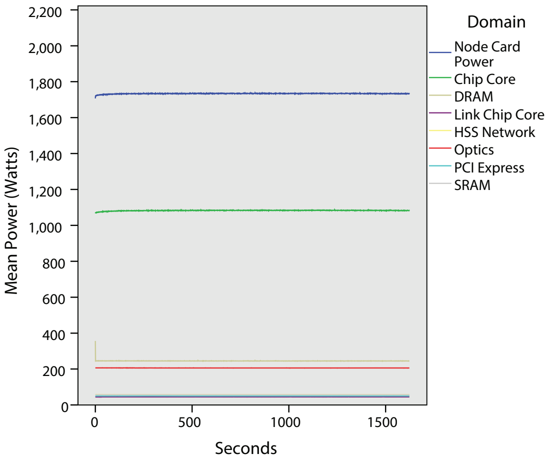

Fig. 2. Power as observed from the data collected by MonEQ across the 7 domains available captured at 560ms. The top line represented the power consumption of the node card. This data is the same as that collected from the BPMs, but at a higher sampling frequency.

As far as the collection overhead, we found that each collection takes about 1.10 ms which results in a total overhead of about 0.19%.

B. Intel RAPL

As of the Sandy Bridge architecture, Intel has provided the "Running Average Power Limit" (RAPL) interface [10]. While the original design goal of RAPL was to provide a way to keep processors inside of a given power limit over a given sliding window of time, it can also be used to calculate power consumption over time which is especially useful for applications. The circuitry of the chip is capable of providing estimated energy consumption based on hardware counters.

The Intel model-specific registers (MSRs) are implemented within the x86 instruction sets to allow for the access and modification of parameters which relate to the execution of the CPU. There are many of these registers, but most of them aren't useful in terms of environmental data collection. Table II gives an overview of the registers which are useful for environmental data collection.

TABLE II
LIST OF AVAILABLE RAPL SENSORS.

Domain	Description
Package (PGK)	Whole CPU package.
Power Plane 0 (PP0)	Processor cores.
Power Plane 1 (PP1)	The power plane of a specific device in the uncore (such as a integrated GPU–not useful in server platforms).
DRAM	Sum of socket's DIMM power(s).

Accessing these MSRs requires elevated access to the hardware which is something that typically only the kernel can do. As a result, a kernel driver is necessary to access these registers in this way. As of Linux 3.14 these kernel drivers have been included and are accessible via the perf_event (perf) interface. Unfortunately, 3.14 is a much newer version of kernel than most distributions of Linux have.

Currently, short of having a supported kernel the only way to get around this problem is to use the Linux MSR driver which exports MSR access to userspace. Once the MSR driver is built and loaded, it creates a character device for each logical processor under `/dev/cpu/*/msr`. For the purposes of collecting this data, this still does not get around the root only limitation. The MSR driver must be given the correct read-only, root-only access before it is accessible by any process running on the system.

There are a number of limitations with RAPL, with the biggest being that of scope. For the CPU, the collected metrics are for the whole socket. As a result, it's not possible to collect data for individual cores. What's more, the DRAM memory measurements do not distinguish between channels. This also means that currently, it's not possible to set per-core power limits.

The subject of the accuracy of the data obtained from the RAPL interface has been looked at fairly extensively [11], [12]. It has been generally concluded that the updates are not accurate enough for short-term energy measurements with the updates happening within the range of $\pm 50,000$ cycles. However, few updates deviate beyond 100,000 cycles making the RAPL interface relatively accurate for data collection at about 60ms. On the other hand, these registers can "overfill" if they are not read frequently enough, so a sampling of more than about 60 seconds will result in erroneous data. Given most applications are likely going to want more frequent collection than that, this does not render the RAPL interface useless.

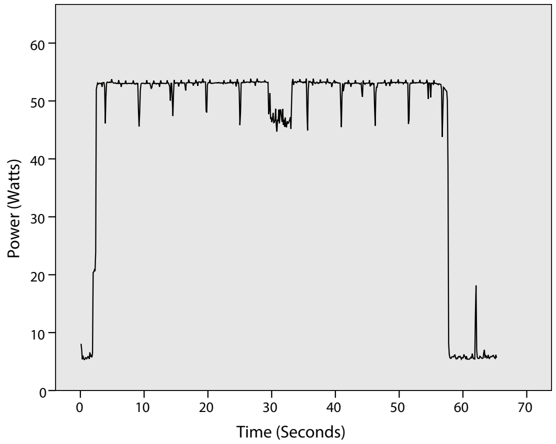

Fig. 3. Power consumption of a Gaussian Elimination workload captured at 100ms for the whole CPU package. Capture started before and terminated after program execution.

An example of the data obtained from the RAPL interface for Gaussian Elimination code is shown in Figure 3. In this instance, the capture of data was started before and terminated after the program had executed to show what the idle state of the CPU looks like. One of the more interesting things to note is the rhythmic drop of about 5 Watts in power consumption throughout the execution of the workload. What's more, between these drops there are tiny spikes in power at regular intervals. It is not known at this time why this is the case.

Overhead of data collection for RAPL will almost certainly depend on the method which is used. One would expect that using the perf interface would result in higher access times than reading the MSRs directly due to the overhead of having to go through the kernel. Unfortunately, at the time of this paper we did not have ready access to a Linux machine running a new enough kernel to test the overhead of collection using the perf interface.

The overhead of accessing the MSR however we know to be about 0.03 ms per query. This is the fastest access time that we have seen for all of the hardware discussed in this paper as polling the MSR is essentially pulling the data directly from the registers on the CPU.

C. NVIDIA Management Library

The NVIDIA Management Library (NVML) is a C-based API which allows for the monitoring and configuration of NVIDIA GPUs. The only NVIDIA GPUs which support power data collection are those based on the Kepler architecture, which at this time are only the K20 and K40 GPUs.

In terms of power consumption, the only call that exists to collect power data is `nvmlDeviceGetPowerUsage()`. On the current generation of GPUs, the reported accuracy by NVIDIA is ±5W with an update time of about 60ms. Unlike other devices discussed in this paper, the power consumption reported is for the entire board including memory.

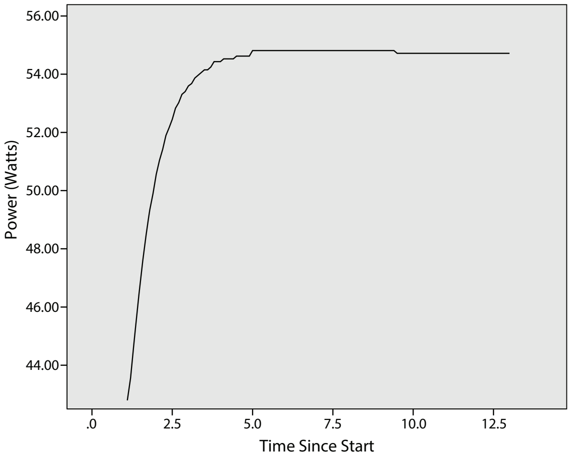

Fig. 4. Power consumption of a NOOP workload on a NVIDIA K20 GPU captured at 100ms. Shows gradual increase until finally leveling off and staying there for the rest of the time.

A very basic example of what the power data look like is presented in Figure 4. The kernel function here is a basic NOOP which is executed a certain number of times as to gather enough data to give a decent representation. Interestingly, and in contrast to the other devices discussed, the jump in power consumption once the workload is tasked to the GPU is not nearly as severe. In fact, it takes about 5 seconds before the power consumption levels off to a constant value. While the exact reason for this is unknown, the working theory we have come up with is that because of the lock-step nature of the way threads are executed on a GPU, it's possible that it takes a few seconds before they are all synchronized. The experiment was run on a NVIDIA K20 GPU which has a peak performance of 1.17 teraFLOPS at double precision, 5 GB of GDDR5 memory, and 2496 CUDA cores.

A more interesting vector add workload is presented in Figure 5. As with the NOOP workload, the first few seconds show the power consumption slowly increasing to a level value of about 55 Watts. Important to note here is this workload first generates the data on the host side and then transfers the data to the GPU for the vector addition, so for the first 10 or so seconds, the GPU hasn't been given any work to do. After the data is generated and handed off to the GPU for computation, the power consumption increases dramatically where it remains for the remainder of the computation.

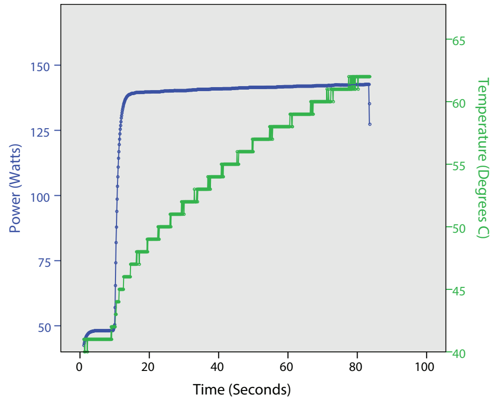

Fig. 5. Power consumption and temperature of a vector add workload. Power curve shows same gradual increase in first few seconds as sleep workload with rapid increase after data generation until workload finishes. Temperature shows steady increase.

The overhead for polling a NVIDIA GPU is higher than any of the hardware we have seen thus far. The primary reason being that any call to the GPU for data collection not only needs to go through the NVML library, it must also transfer data across the PCI bus. Each collection takes about 1.3 ms which results in an overhead of about 1.25%.

D. Intel Xeon Phi / MIC

The Intel Xeon Phi is a coprocessor which has 61 cores with each core having 4 hardware threads per core yielding a total of 244 threads with a peak performance of 1.2 teraFLOPS at double precision.

On the Intel Xeon Phi, there are two ways in which environmental information may be collected on the host side. The first is the "in-band" method which uses the symmetric communication interface (SCIF) network and the capabilities designed into the coprocessor OS and the host driver.

The SCIF enables communication between the host and the Xeon Phi as well as between Xeon Phi cards within the host. Its primary goal is to provide a uniform API for all communication across the PCI Express buses. One of the most important properties of SCIF is that all drivers should expose the same interfaces on both the host and on the Xeon Phi. This is done so that software written for SCIF can be executed wherever it is most appropriate. This implementation includes both a user mode library and a kernel mode driver to maximize portability. This is graphically illustrated in Figure 6.

The second is the "out-of-band" method which starts with the same capabilities in the coprocessors, but sends the information to the Xeon Phi's System Management Controller (SMC). The SMC can then respond to queries from the platform's Baseboard Management Controller (BMC) using the intelligent platform management bus (IPMB) protocol to pass the information upstream to the user.

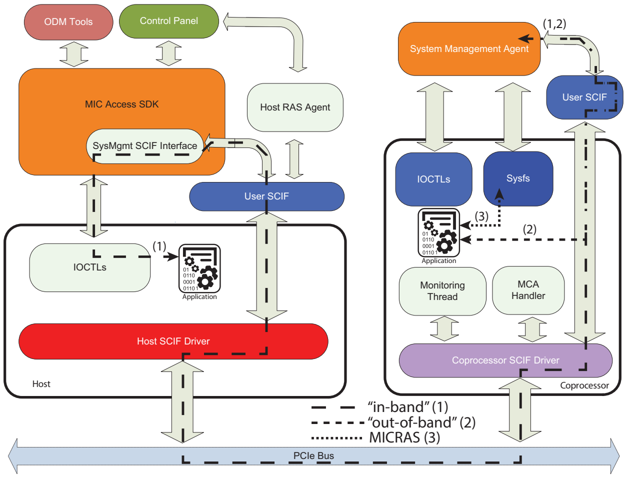

Fig. 6. Control panel software architecture. Shows the SCIF interface on the host and device side as well as the communication pattern. [13]

Both of these methods assume the user wishes to gather information on the host side. However, there is a third way in which environmental information can be obtained. The MICRAS daemon is a tool which runs on both the host and device platforms. On the host platform this daemon allows for the configuration of the device, logging of errors, and other common administrative utilities. On the device though, this daemon exposes access to environmental data through pseudo-files mounted on a virtual file system. In this way, when one wishes to collect data, it's simply a process of reading the appropriate file and parsing the data.

Curiously, there are trade-offs between these two collection methods. Figure 7 shows a boxplot of the power consumption as measured between the SysMgmt API and MICRAS daemon for a no-op workload. As can be seen, while slight, there is a statistically significant difference between the two collection methods. The reason behind this will be explained later.

Another trade-off is the data collected by the daemon is only accessible by the portion of code which is running on the

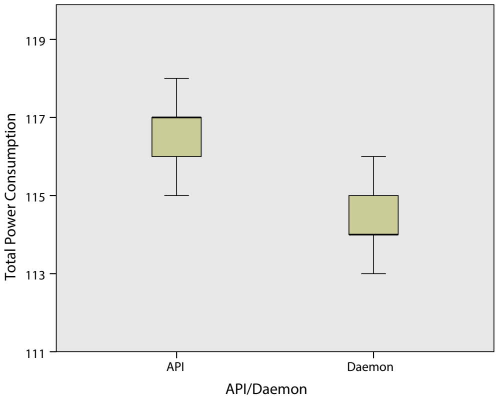

Fig. 7. Boxplot of power data for both the SysMgmt API ("in-band") and daemon capture methods.

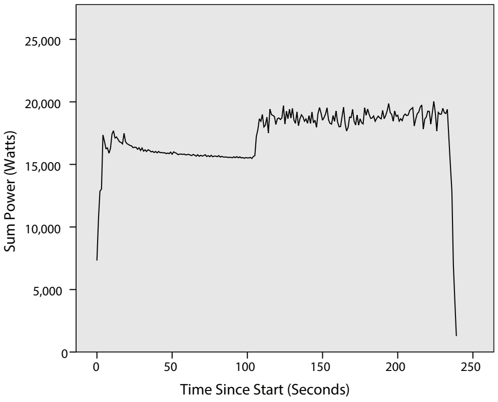

Fig. 8. Sum of power consumption for a Gaussian Elimination workload running on 128 Xeon Phi cards on Stampede. Data generation takes place for about the first 100 seconds. After which, data is transferred to the cards and computation begins.

device. As a result, there is an unavoidable overhead associated with any data collection which is performed in this mode. In other words, any collection performed must occur during the execution of the application which is running thus causing contention between the application and the data collection process.

However, as previously mentioned, there is a difference in baseline power consumption between these collection methods, and, despite the data collection code which utilizes the API executing on the host, it actually results in greater power consumption over idle. As SCIF implementations have both a user library and kernel driver, when an API call is made to the lower-level library to gather environmental data, it must travel across the SCIF to the card where user libraries call kernel functions which allow for access of the registers which contain the pertinent data. This explains the rise in power consumption as a result of using the API; code that wasn't already executing on the device before the call was made must run, collect, and return.

Further complicating the issue is the overhead associated with both of the collection methods. When accessing the data through the API, each collection takes a staggering 14.2 ms which results in an overhead of about 14%. Polling the data provided by the MICRAS daemon however results in nearly the same overhead as RAPL, about 0.04 ms per query. These results are almost the same because the implementation on both is essentially the same; the Xeon Phi actually uses RAPL internally for power consumption limitation.

To show what data looks like at a bit larger scale we profiled a Gaussian elimination code running on 128 Xeon Phi's on the Stampede supercomputer. It should be noted that MonEQ can easily scale to a full system run on Stampede just as it has been shown to on other supercomputers. This experiment was run on 16 Xeon Phi's in the interest of preserving allocation. Stampede is a Dell Linux Cluster at the University of Texas at Austin. It is based on 6,400+ Dell PowerEdge server nodes, each outfitted with 2 Intel Xeon E5 (Sandy Bridge) processors and an Intel Xeon Phi Coprocessor. The results of this are shown in Figure 8. Clearly shown is the point where data generation stops and computation starts.

III. Discussion of Power Profiling Tools

While we have focused on the results obtained from our profiling tool MonEQ, it's worth mentioning there are tools other than MonEQ which allow for the collection of power data. One such tool is PAPI [14], [15]. PAPI is traditionally known for its ability to gather performance data, however the authors have recently begun including the ability to collect power data. PAPI supports collecting power consumption information for Intel RAPL, NVML, and the Xeon Phi. PAPI allows for monitoring at designated intervals (similar to MonEQ) for a given set of data.

Another such tool is TAU [16]. Like PAPI, TAU is mostly know for its profiling and tracing toolkit for performance analysis. However, as of version 2.23, TAU also supports power profiling collection of RAPL through the MSR drivers. To the best of our knowledge this is the only system that TAU supports for power profiling.

PowerPack [17] is a well-known power profiling tool which historically gathered data from hardware tools such as a WattsUp Pro meter connected to the power supply and a NI meter connected to the CPU/memory/motherboard/etc. Recent development (PowerPack 3.0) has allowed for the collection of software accessible power data. However, even as of this latest version PowerPack does not allow for the collection of power data from newer generation hardware such as Intel RAPL, NVML, or the Xeon Phi.

Wanting to address these limitations as well as others, we designed MonEQ. In our previous work, MonEQ was only able to gather power data from the BG/Q supercomputer. In this work however, we have extended it to support the most common of devices now found in supercomputers with the same feature set and ease of use as before.

In its default mode, MonEQ will pull data from the selected environmental collection interface at the lowest polling interval possible for the given hardware. However, users have the ability to set this interval to whatever valid value is desired. With the value of the polling interval set, MonEQ then registers to receive a SIGALRM signal at that polling interval. When the signal is delivered, MonEQ calls down to the appropriate interface and records the latest generation of environmental data available in an array local to the finest granularity possible on the system. For example, on a BG/Q, this is the local agent rank on a node card, but for other systems this could be a single node. If a node has several accelerators installed locally, each of these is accounted for individually within the file produced for the node.

Listing 1. Simple MonEQ Example

```
int status, myrank, numtasks, itr;

status = MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

status = MonEQ_Initialize();    // Setup Power

/* User code */

status = MonEQ_Finalize();      // Finalize Power

MPI_Finalize();
```

Since the interface for MonEQ was already well defined from our experiences with BG/Q (an example is shown in Listing 1), we kept that the same while adding the necessary functionality for other pieces of hardware internally. As a result, one wishing to profile data with MonEQ simply needs to link with the appropriate libraries for the hardware which they are running on. This of course means that if a system has both a NVIDIA GPU as well as an Intel Xeon Phi, profiling is possible for both of these devices at the same time.

Oftentimes application developers have logically and functionally distinct portions of their software which is of primary interest for profiling. To address this, we have implemented a tagging feature. This feature allows for sections of code to be wrapped in start/end tags which inject special markers in the output files for later processing. In this way, if an application had three "work loops" and a user wanted to have separate profiles for each, all that is necessary is a total of 6 lines of code. Better yet, because the injection happens after the program has completed, the overhead of tagging is almost negligible.

In terms of overhead, we've made sure to design MonEQ so that it is as robust as possible without weighing down the application to be profiled. For each of the major pieces of hardware discussed in this paper we mentioned the overhead for the profiling call to whatever API is to be used for collection. As one would expect, this profiling call is just one part of a profiling library and we have shown that it varies from system to system. However, in general the overhead is mostly dependent on the number of devices that are being profiled. The reasoning for this is simple, the more nodes the more data points. For this reason we've designed MonEQ to perform its most costly operations when the application isn't running (i.e, before and after execution). The only unavoidable overhead to a running program is the periodic call to record data. Here again, the method which records data does so as quickly and efficiently as possible for all types of hardware.

Making it an ideal case to study the overhead of MonEQ, Table III shows a toy application profiled at the most frequent interval possible at three different scales on BG/Q. The application is designed to run for exactly the same amount of time regardless of the number of processors making it an ideal case to study the overhead of MonEQ.

From the data we can see that the time spent during initialization and collection is the same in all three cases with only the time spent during finalization having any real variability. This follows intuition as regardless of hardware or scale the initialization functions only needs to setup data structures and register timers and the collection method only needs to collect data. It's important to note that the design of MonEQ is such that collection of data is the same for all nodes assuming they are homogeneous among themselves. That is, if every node in a system has two GPUs, then every node will spend the same amount of time collecting data.

The finalization method really has the most to do in terms of actually writing the collected data to disk and therefore does depend on the scale the application was run at. While every system will certainly result in different times for this method (subject to network speeds, disk speeds, etc.), we see that on BG/Q at the 1K scale MonEQ has a total time overhead of about 0.4% including the unavoidable collection.

Memory overhead is essentially a constant with respect to scale regardless of hardware. The initialization stage of MonEQ allocates an array of a custom C struct with fields that correspond to all possible data points which can be collected for the given hardware. The size of the array is allocated to a reasonably large number such that even on one of the biggest production machines in the world it would be able to collect data for quite some time while not consuming an excess of memory. Of course, this number isn't set in stone and can be modified if desired to decrease the memory overhead of MonEQ or to support a longer execution time. Thus, in a way memory overhead is essentially up to the person who is integrating MonEQ into their application.

Overall this makes MonEQ easy to use, lightweight, and extremely scalable. Our experiences with MonEQ show that it can easily scale to a full system run on Mira (49,152 compute nodes).

TABLE III
TIME OVERHEAD FOR MONEQ IN SECONDS ON MIRA

	32 Nodes	512 Nodes	1024 Nodes
Application Runtime	202.78	202.73	202.74
Time for Initialization	0.0027	0.0032	0.0033
Time for Finalize	0.1510	0.1550	0.3347
Time for Collection	0.3871	0.3871	0.3871
Total Time for MonEQ	0.5409	0.5455	0.7251

IV. Conclusions and Looking Forward

In this paper we have presented a comprehensive overview of the environmental monitoring capabilities which are present on the current set of the most common hardware found in supercomputers. We have shown that in many cases the same environmental data isn't available between two different devices. What's more, the method by which this data is accessed varies substantially as well. Just about the only data point which is collectible on all of these platforms is total power consumption at some granularity. For accelerators, this is the power consumption of the entire device, for a Blue Gene/Q, this is a node card (32 nodes).

Looking forward, the single largest issue which is practically impossible to eliminate is that of collection overhead. Whether it's overhead as a result of having to run the collection code along with the application being profiled or overhead as a result of an API call, there will also be some cost associated with gathering this data. However, there are some improvements that could be made to the current generation of hardware discussed in this work which would make the data they provide more beneficial.

The first and perhaps most important is stated limitations of the data and the collection of this data. For many of the devices discussed, the limitations in collection had to be deduced from careful experimentation. Especially in the case of the Xeon Phi; it's not necessarily intuitive that the API would have a greater base overhead than collecting the data directly from the daemon running on the card. Another example, when collecting data from RAPL, if the frequency goes above 60 seconds then it's possible the register will overflow causing the next collection to produce incorrect results.

Secondly, unification of available data is of the utmost of importance if this data is to be used for comparison of platforms. The Blue Gene/Q with its 7 domains provides a very good representation of the compute node on the whole as it covers the necessary data for the major components of the node card. In the case of NVIDIA GPUs on the other hand, one must settle for total power consumption of the whole card when clearly the power consumption of both the GPU and memory would be more beneficial. On the other hand, NVIDIA GPUs support temperature data whereas this data is only accessible in the environmental data for a Blue Gene/Q and only at the rack level.

Acknowledgments

This research has been funded in part and 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. The research has been funded in part by the Center for Exascale Simulation of Advanced Reactors (CESAR) Co-Design center, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Zhiling Lan is supported in part by US National Science Foundation grants CNS-1320125 and CCF-1422009.

The authors thank the ALCF staff for their help. They also gratefully acknowledge the help provided by the application teams whose codes are used herein.

References

[1] W. B. Allcock, E. Felix, M. Lowe, R. Rheinheimer, and J. Fullop, "Challenges of hpc monitoring," in *State of the Practice Reports*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 22:1–22:6. [Online]. Available: http://doi.acm.org/10.1145/2063348.2063378

[2] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka, "Integrating dynamic pricing of electricity into energy aware scheduling for hpc systems," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 60:1–60:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503264

[3] S. Wallace, V. Vishwanath, S. Coghlan, Z. Lan, and M. Papka, "Measuring power consumption on IBM Blue Gene/Q," in *The Ninth Workshop on High-Performance, Power-Aware Computing, 2013 (HPPAC'13)*, Boston, USA, May 2013.

[4] S. Wallace, V. Vishwanath, S. Coghlan, J. Tramm, Z. Lan, and M. E. Papkay, "Application power profiling on IBM Blue Gene/Q," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, 2013, pp. 1–8.

[5] IBM, "Introduction to Blue Gene/Q," 2011. [Online]. Available: http://public.dhe.ibm.com/common/ssi/ecm/en/dcl12345usen/DCL12345USEN.PDF

[6] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *Micro, IEEE*, vol. 32, no. 2, pp. 48 –60, march-april 2012.

[7] G. Lakner and B. Knudson, *IBM System Blue Gene Solution: Blue Gene/Q System Administration*. IBM Redbooks, June 2012. [Online]. Available: http://www.redbooks.ibm.com/abstracts/sg247869.html

[8] V. Morozov, J. Meng, V. Vishwanath, J. Hammond, K. Kumaran, and M. Papka, "ALCF MPI benchmarks: Understanding machine-specific communication behavior," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, sept. 2012, pp. 19 –28.

[9] "MonEQ: Power monitoring library for Blue Gene/Q," https://repo.anl-external.org/repos/PowerMonitoring/trunk/bgq/.

[10] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developers Manual*, February 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[11] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Micro, IEEE*, vol. 32, no. 2, pp. 20–27, March 2012.

[12] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, pp. 13–17, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2425248.2425252

[13] Intel Corporation, *Intel Xeon Phi Coprocessor System Software Developers Guide*, March 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf

[14] 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.

[15] 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.

[16] 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. [Online]. Available: http://dx.doi.org/10.1177/1094342006064482

[17] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron, "PowerPack: Energy profiling and analysis of high-performance systems and applications," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 5, pp. 658 –671, may 2010.