Dynamic Model Specific Register (MSR) Data Collection as a System Service

Gregory H. Bauer*, J. Brandt†, A. Gentile†, A. Kot*, and M. Showerman*
*National Center for Supercomputing Applications
Champaign, IL
Email: (gbauer|kot|mshow)@ncsa.illinois.edu
*Sandia National Laboratories
Albuquerque, NM.
Email: (brandt|gentile)@sandia.gov

***Abstract*—The typical use case for Model Specific Register (MSR) data is to provide application profiling tools with hardware performance counter data (e.g., cache misses, flops, instructions executed). This enables the user/developer to gain understanding about relative performance/efficiencies of the code overall as well as smaller code sections. Due to the overhead of collecting data at sufficient fidelity for the required resolution, these tools are typically only run while tuning a code.**

In this work we present a substantially different use case for MSR data, namely system wide synchronized and relatively low fidelity collection as a system service on NCSAs 27,648 node Blue Waters platform. We present which counters we collect, the motivation for this particular data, and associated overhead. Additionally we present some associated pitfalls, how we address them, and the effects. We finally present some analysis results and the insight they provide about applications, system resources, and their interactions.

I. Introduction

NCSA's Blue Waters machine is a 27,648 node Cray XE/XK compute platform with a Gemini interconnect. Application execution on a platform of this size, especially in the face of contention with other applications for shared resources, such as the Gemini High Speed Network (HSN) and Lustre parallel file system, can be subject to relatively wide performance variation. It can be difficult for the application developers and users to unravel the reasons behind such variation given all the background interaction. Additionally it can be difficult from an administrative perspective to assess how well the platform resources are being utilized beyond gross measures of node allocations which may be high because users over allocate and under utilize the resources.

In order to better understand, from an operational perspective, how users and applications are utilizing the underlying computational resources and how well matched those resources are to the applications being run on them, the Blue Waters staff has been increasing their level of system wide monitoring. In addition to monitoring the shared resources mentioned above, we have begun monitoring node level resource utilization by periodically reading the Interlagos processor's Model Specific Registers (MSRs) [1].

Collection and processing of this data, even at the relatively coarse granularity of once per minute, enables us to gain insight into application level processor and memory level behavioral characteristics. This paper presents details on data collection methodologies, impediments and pitfalls, and counters of interest and is organized as follows: Section II describes the motivation for this work. In Section III we present the counters and data that are of interest in our system. Details of the architecture and implementation of our MSR collection as a system service are presented in Section IV. Data validation and collection overhead are discussed in Section V. In Section VI we describe support for analysis. In Section VII we present use cases for the data through analysis of targeted applications run in production on Blue Waters. Finally, we conclude in Section VIII.

II. Motivation

NCSA's decision to collect hardware performance counter data was primarily driven by the following factors: system wide reporting demands from interested parties such as funders and investors, the need for application performance analysis by HPC center support staff, and the need for performance information by the Blue Waters Science and Engineering partners. The funders and investors typically want to know about system performance aggregates such as floating point or aggregate memory bandwidth rates in order to ascertain how well the various computing resources, as a whole, are being utilized. The center staff want to know which applications and which users are not effectively utilizing the resources allocated for the workloads running on the system in order to help them run more effectively and increase overall system throughput. Easy identification of offending application performance can facilitate working on the so called low hanging fruit in terms of performance

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number ACI 1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

improvement. The Science and Engineering partners in some cases have similar demands as center staff but additionally need to assess the aggregate values for use in performance projection and usage reporting to various agencies. Lastly, access to this type of data would enable the users to extract high level performance metrics in order to assess performance of runs they did not profile or to determine how tuning of an application using low level, high overhead, profiling tools has changed the gross execution efficiency. While resolution is not high enough for fine grained code tuning, it can be very useful for coarse grained comparisons to understand the evolution of utilization characteristics due to code and platform changes.

In particular, collection of system wide counter data by core, floating point unit, and NUMA domain enables the center staff to investigate possible misconfigured batch jobs where users are unintentionally not using all of the resources available within the nodes being allocated. The Cray XE/XK systems use an application launch utility called *aprun* to place applications on the compute nodes allocated to a job. The complexity of the dual-socket nodes in conjunction with the two NUMA domains per CPU, the four floating point units per domain, and the two integer cores per floating point unit can lead to less than optimal task and thread placement due to user mistakes and/or lack of a complete understanding of this hierarchy and associated performance implications. A wrapper to *aprun* is one way to catch incorrectly configured application launches but affinitization can be implemented from within an application that makes parsing impractical. Significant insight can be gained through analysis and visualization of per core, per floating point unit, and per NUMA domain performance counter data grouped according to jobs/applications over their execution times. Time series plots of relevant counter data for a particular application run should show where placement of tasks and threads only utilize cores on one socket per node, floating point units on one NUMA domain per CPU, etc.

Collection of hardware performance counters has also been of interest at other HPC sites. In particular at TACC, such information is collected via TACCStats [2]. The data is not available for processing at run-time, but rather 24 hours worth of data is downloaded off the nodes once every 24 hours. The collection period for the actual data on the TACC computational resources is 10 minutes, which also impacts the resolution of analysis.

III. Counters of Interest

The Blue Waters project has had a goal of employing performance modeling, using hardware performance counters, as part of its support goals since its initial year of operation. During the initial deployment phase of Blue Waters, representative applications were identified for the first year workload. These applications were analyzed using a process developed by Hoefler [3]. In this process, the on-node performance of an application is characterized by a method similar to the roofline modeling method of Williams [4] through use of floating point instructions completed, total instructions completed, memory accesses, total loads and stores, and cycles stalled. As shown in [3], this type of analysis can help in determination of the potential for performance improvement in an application and in identification of the limiters of application performance. For the initial analysis work, a variety of custom performance analysis tools were used to collect detailed performance counter information across the processes of the target applications. While we were able to perform this work as a dedicated study, the real target has continued to be continuous system wide collection and analysis in order to characterize the behaviors and interactions of all applications as they are run natively. Additionally we want to track how these characteristics change due to both application and platform changes.

There are a limited number of registers available for counting performance related hardware events. Our current set of performance counters used in system wide collection on Blue Waters, as of the time of this publication, is described in Table I. This set uses the entire set of registers available for both core and uncore counters. Thus adding another counter implies dropping one of those listed. Alternatively we could move to a more statistical approach where we sample more counters for shorter intervals and make assumptions about how well these intervals represent the whole run time. Currently we have chosen to use a constant set that fits the number of available registers. The names in the *Counter* column of the table are user setable and were chosen to match a set of PAPI [5] events with the same definitions. The related PAPI counters of interest, ordered in the same order as Table I, are `PAPI_TOT_CYC`, `PAPI_TOT_INS`, `PAPI_FP_OPS`, `PAPI_VEC_INS`, `PAPI_L1_DCM`, `PAPI_TLB_DM`. Higher resolution profiling using statistical sampling of a larger set of counters for specific application runs can still be performed by users on a case by case basis using the Cray Performance Measurement & Analysis Tools (CPMAT) [6] or any of a number of other performance profiling and analysis tools.

IV. MSR Counter Sampling Architecture

Our goals in collecting hardware performance counters as a system service are to provide: 1) continuous and uniform system-wide counter collection, 2) analysis results during run-time and within the sampling interval (currently 60 seconds) and 3) a long term repository of this information for historic comparison.

To accomplish these goals we chose to utilize the Model Specific Register (MSR) interface [7] for access to hardware performance counter data. We further chose to utilize the

Counter	Definition	Note
TOT_CYC	Unhalted clock cycles	Per core
TOT_INS	Retired instructions	Per core
RETIRED_FLOPS	Retired floating point instructions	Per core
VEC_INS	Retired vector instructions	Per core
L1_DCM	L1 data cache misses	Per core
TLB_DM	Data translation lookaside buffer misses	Per core
L3_CACHE_MISSES	L3 data cache misses	Per L3
DCT_PREFETCH	Memory (DRAM) prefetch requests	Per DRAM controller (DCT)
DCT_RD_TOT	Memory (DRAM) read requests including prefetches	Per DCT
DCT_WRT	Memory (DRAM) write requests	Per DCT

Table I
SELECTION OF COUNTERS CURRENTLY BEING COLLECTED SYSTEM WIDE ON BLUE WATERS

OVIS/LDMS [8] infrastructure for collection, transport, rate analysis, and storage of the counter data. The reason for using the MSR interface is its simplicity and access to process independent data. The reason for choosing the OVIS/LDMS infrastructure is that it was already being used for collection and transport of other system level data and the only additional work was writing a MSR specific sampler plugin. Unlike other LDMS data sampler plugins (or *samplers*), which read a fixed data source, the LDMS MSR *sampler* must operate in an environment where the data sources (MSR counters) may be dynamically reset both by system and user processes. In order to enable users to utilize other tools for performing their own performance counter profiling we must also ensure that our system wide monitoring detects such use and does not interfere with the related measurements. Additionally, once a user is finished profiling, we want to resume our MSR sampling on the released resources and ensure we are still sampling the original list of counters. These differences put additional design requirements on the LDMS MSR *samplers*. The resulting implementation details are discussed in this section.

A. Background on LDMS

The Lightweight Distributed Metric Service (LDMS) [8] has been in use, for more than 2 years, on Blue Waters for collection of system resource utilization information such as the Gemini network performance counters, Lustre counters (e.g., reads, writes, opens, closes), and memory and CPU utilization. LDMS was chosen for its low host overhead and large fan-in characteristics. The MSR counter data are largely per-core data and significantly increase our number of metrics collected (roughly doubling from 200 metrics/host to 400 metrics/host). Low-overhead collection continues to be important as we don't want to adversely impact application performance through collection of this additional data. Large fan-in ratios continue to be a requirement of the transport as we are not procuring additional resources to accommodate the MSR data collection.

LDMS uses a plugin architecture, with the same *ldms* daemons running on each node. Functionalities are differentiated by the plugins and configuration files used. Daemons running sampling plugins run on the nodes; daemons that *aggregate* the data from the *samplers* via RDMA over Gemini run on service nodes; finally the daemons that *aggregate* also run *store* plugins that write the data to named pipes for off-platform analysis and storage. Any daemon in the hierarchy can be queried for its current data. Data is stored in a *metric set* data structure which includes the data values and a single timestamp to be associated with those values. Only the most recent *metric set* data values for any given component is retained in the daemon; this minimizes the memory footprint.

Prior to initial OVIS/LDMS deployment on Blue Waters we performed a variety of tests to ensure minimal impact on application performance [8]. While we have not yet performed the same level of testing with the addition of the MSR *sampler*, the testing we have done indicates it should be minimal. We have also established that our current fan-in ratio of around 7,000:1 for normal operation and 14,000:1 in the case of fail-over is sufficient to handle this additional load at our current collection interval of 60 seconds.

B. Design Requirements

There are a number of practical considerations that must be addressed in enabling MSR counter collection as a system service. From the perspective of data collection and interaction:

- there are fewer address spaces available for specifying and collecting counters in comparison with the full body of counters available
- the desired counters may dynamically change at the system and per-application level, and even over time within the same application
- the user tools may use the available address space for the same or conflicting counters
- illogically assigned values to the counter address space may result in problems to the node, necessitating node reboot
- the user has pre-expectations of counter names and values based on experiences with application profiling tools, such as PAPI (e.g., [5]).

In general, these are rather unique constraints for system software with respect to support for dynamic reconfiguration and discovery and handling of externally-invoked conflicting states.

In addition, an application user or system administrator should not have to make a code call, link, or change to obtain the default set of collected counters. In order to otherwise take advantage of the service, changing the counters on-demand can be accomplished via a command line utility which can be called dynamically. For example a script can be triggered to automatically invoke a change of counters or frequency of collection, based on conditions of interest.

There are system-support constraints as well. In general, LDMS is installed in the image and frequent updates to the image are impractical. Therefore enabling changes to the collectors and the counters must be handled other than in code.

Finally, from a data processing perspective, we must be able to recognize and handle when the counter identities have changed, in order to make valid inferences from the data.

C. Configuration and Collection

While using the MSR interface is conceptually simple (i.e., write a control register and read a result register in `/dev/cpu/CPU#/msr`), understanding what to write, knowing what is being read, and ensuring non-interference can be quite involved and is described here, at a high level, for the AMD Family 15h Models 00h-0Fh processors. The information presented was obtained from the BIOS and Kernel Developer's Guide (BKDG) [1].

Performance Event Select registers are defined to be: `MSRC001_020[A,8,6,4,2,0]`. As an example, in order to count an event of interest for core 0 in the register controlled by control register `0xc001020` one would write the 64 bit value defining the counter of interest to `/dev/cpu/0/msr/0xc001020` and read the corresponding values from `/dev/cpu/0/msr/0xc001021`. In order to define the counter of interest, the bit fields defined below must be populated for the particular event of interest:

- Bits 63:42 Reserved
- Bits 41:40 HostGuestOnly: count only host/guest events. Read-write
- Bits 39:36 Reserved
- Bits 35:32 EventSelect[11:18]: performance event select
- Bits 31:24 CntMask: counter mask. Read-write. Controls the number of events counted per clock cycle.
- Bits 23 Inv: invert counter mask.
- Bits 22 En: enable performance counter.
- Bits 21 Reserved
- Bits 20 Int: enable APIC interrupt.
- Bits 19 Reserved
- Bits 18 Edge: edge detect
- Bits 17:16 OsUserMode: OS and user mode
- Bits 15:8 UnitMask: event qualification
- Bits 7:0 EventSelect[7:0]: event select

An event is defined by an *event code* and associated unit mask. As an example, `RETIRED_FLOPS` in Table I refers to information obtained using event code `0xc003` (Retired Floating Point Ops or FLOPS) along with the appropriate *UnitMask* value. *UnitMask* bit positions for this event are defined as follows:

- Bit 7 Double precision multiply-add FLOPS. Multiply-add counts as 2 FLOPS.
- Bit 6 Double precision divide/square root FLOPS.
- Bit 5 Double precision multiply FLOPS.
- Bit 4 Double precision add/subtract FLOPS.
- Bit 3 Single precision multiply-add FLOPS. Multiply-add counts as 2 FLOPS.
- Bit 2 Single-precision divide/square root FLOPS.
- Bit 1 Single-precision multiply FLOPS.
- Bit 0 Single-precision add/subtract FLOPS.

For this example configuration if we wanted all (both single and double) precision FLOPS counts the UnitMask value would be `0xFF` and we would write the control word `0x43ff03` into `/dev/cpu/0/msr/0xc001020`. We would then periodically read the corresponding counts from `/dev/cpu/0/msr/0xc001021`. Likewise if we wanted to read the same counters for all cores we would write the same control word to an appropriate `msr` register for each core (e.g. for core 1 we would write to `/dev/cpu/1/msr/0xc001020`). It is important to note that not all core counters can be counted into all core registers. Which ones can be used is defined on a per *event code* basis in the BKDG. Note that the acronym FLOPS here refers to a count of floating point operations and *not* a rate.

In order to support the operational constraint of requiring changes in the image, as well as enabling the flexibility of investigating different options of counters and the registers, the options for the counters are defined in a configuration file which is loaded as part of the MSR sampler configuration. Based on system administrator and user input, we have down-selected the counters to a well-defined set, listed in Section III whose number matches the number of registers available for counters.

A representative and functional configuration file is provided as part of the code release. The fields include the *event code*, UnitMask, address of the control register, and the address of the corresponding counter register. These provide sufficient information for setting the control registers for the counters and if selection is made among them at *sampler* configuration time, conflicts, if any, are identified in the log file but not used. The configuration file also provides the flexibility for the user to define their own names and corresponding information. Checking for conflicts and sanity settings is not performed in this case and the user could

potentially cause instability or crash the host through use of erroneous control registers and values.

The system service provides a command line utility to specify the counters to collect, to halt and continue collection, and to reassign a counter to a new or previous collected option. Counter specification in the configuration file, and thus in the command line utility, is in terms of counter names which we have defined to be as identical as possible to the PAPI naming convention. The administrator/user is only responsible for using the naming convention, not the raw addresses, eliminating the potential for harmful assignment (given a valid configuration file). System administrators, and other system services and users to whom the utility is exposed, can also dynamically change which counters are being collected. Details of this are given in the next subsection.

D. Assigning and Reassigning Counters

The LDMS command line interface, *ldmsctl*, accepts the commands below. It can be used in the init script as well as at runtime, for example, invoked in the prolog/epilog scripts.

Functions:

- `initialize` - initializes the plugin
- `add` - add a counter metric to the set.
- `finalize` - creates the set after the adds.
- `halt` - halts collection for this counter or all counters. Values of zero will be reported for all metrics for this counter.
- `continue` - continues collection for a counter or all counters after a halt.
- `rewrite` - rewrites a counter's register variable or all counter's variables.
- `reassign` - replace a metric in the set with another one (via reassignment of a counter's register variable). Must be same size (numcores vs. single valued).
- `ls` - writes information about the intended counters to the log file.

The commands are designed to enable setting the initial counters, changing counters, and resetting them if they have been changed underneath. The expected usage of the commands are described below.

At the time of LDMS configuration, the initial set of plugins are identified. This would be done by the system administrator. First, the `initialize` command is used to specify the configuration file for the counters; information on the system relevant to the counters, such as cores per numa node. This initialization is done per node, so that at this time a *Component Id* that can be used at the output data store to associate the values with a particular source node is also specified. This is followed by a series of `add` calls where counters of interest are specified. For any given counter added, a metric in the set is added for both the counter register and for the values. The `finalize` call is used to indicate that all the counters of interest have been specified. Within the `finalize` call, the appropriate control registers are written to enable collection of the counters of interest. This includes checks for where multiple counters have been assigned to the same control register (via specifications in the configuration file). The counters are added to the set in the order they are specified. This results in a metric set like that shown in Figure 1.

The identities of the counters are not necessarily static. This is because they can be defined in a configuration file where the label to *Event Select* value mapping can be arbitrary and the counter number depends on loading order. Because of this we generically identify metric names using the labels "Ctr0, Ctr1, ...". These labels refer to the decimal value written to the corresponding *Event Select* register. The individual components for which data is being collected use labels "Ctr0_c00, Ctr0_c01, ..." to refer to the decimal values being read from their respective *Event Counter* registers. In the Comma Separated Value (CSV) output, the value of the first entry is the identity of the counter in the form of the *Event Select* register value and the subsequent entries are the N values (depending on if it is a per-core metric and the number of cores) for the metric as read from their *Event Counter* registers. As an example, if "Ctr0" referenced the counter value corresponding to the *Event Select* register programmed for the `RETIRED_FLOPS` described in Section IV-C, its value would be the decimal equivalent of 0x43ff03 or 4456195 and all counters with the same "Ctr0" prefix would be the counts associated with the referenced cores. This format also enables the current identity of the counter to be found out both in queries to any *ldms* daemon and in any entry in the stored data. It does however, lead to some complexities in analysis, as described in Section VI.

```
nid00010: consistent, last update: Wed Jun 17 12:09:38 2015 [4108us]
U64 4456195         Ctr0
U64 3262245033      Ctr0_00
U64 3043909658      Ctr0_01
U64 2890296409      Ctr0_02
U64 2820872074      Ctr0_03
U64 2875395942      Ctr0_04
U64 2799488093      Ctr0_05
U64 2863281127      Ctr0_06
U64 2807005447      Ctr0_07
U64 2797754409      Ctr0_08
U64 2921013146      Ctr0_09
U64 2778675164      Ctr0_10
U64 2710421253      Ctr0_11
U64 2665393997      Ctr0_12
U64 2698160014      Ctr0_13
U64 2929310685      Ctr0_14
U64 3466780295      Ctr0_15
```

Figure 1. Metric set contains both the control register value (Ctr0) and the data values (Ctr0_00 - Ctr0_15) for this 16 core processor.

At this point, the collection of counters via LDMS can be started. The remaining commands pertain to counter updates during the collection.

For each counter, the control register is checked before

the counter is read. This minimizes the chance that the counter has been changed by an external process. More details are given on this in Section IV-E. If this has occurred, the `rewrite` command can be called to rewrite a control register(s) to the expected value and values of zero are reported as long as the counter register is not the expected value. This is typically done in the prolog/epilog scripts in order to reset the counters to the defaults for the next user.

The `reassign` command can be invoked during run time to change a counter being collected. This enables users to take advantage of the system service to collect counters of interest. The service can also be suspended using the `halt` command. The `halt` command bypasses the check for the register as well as the data. Collection is restarted upon issue of `continue`. `halt` is a deliberate suspension of the checks; as opposed to when the check occurs and fails on the counter's register value. The `halt`, `continue`, and `rewrite` can work on a single counter or all as indicated by the counter name or the `ALL` keyword.

The `ls` command can be used to dump the current identity of the counters to a log file.

E. Non-Interference with User Performance Analysis

Non-interference with user-invoked profiling tools that use the same MSR registers is ensured in the following way. Our procedure for reading the counters includes a check of the *Event Select* register before the *Event Counter* register is read. This enables the service to identify if the event parameters for a register have been modified. If they have been modified, the corresponding counter register and counter values in the *metric set* are set to zero. Non-zero values will be reported until the counter register value matches the currently configured value. Thus, external user modification of the *Event Select* registers overrides any of the system services functionality.

Since the read of the *Event Select* and *Event Counter* registers is not atomic, there is a slight chance that the *Event Select* register will have been changed between checking its value and reading the *Event Counter* register. This cannot be obviated, by rechecking the *Event Select* register after the *Event Counter* read, for the same reason. We do not expect this race condition to occur frequently and it should cause at most a single incorrect value that should appear as a counter wrap and should be generally detectable in the data processing step.

The change of the counter assignment is generally detectable in the data because both the counter name and the counter values are reported as zero until the counter is reset again. The command line utility to reassign the counter to the desired option and to continue collection is integrated into the init scripts for the daemon as a function, called *reload*, that `rewrites` the counter registers to match the configuration. The system resource manager calls the *reload* function on each compute node from the job during its epilogue script in case the user has modified it as part of their job run.

V. Data Validation

We tested the validity of the MSR data being collected and the streaming calculations (Section VI-A) performed on the data in the following manners. *Memory Bandwidth*: To check our memory bandwidth measurements we used the STREAM sustainable memory bandwidth benchmark to generate a known amount of memory read traffic over a known time interval and compared the average calculated memory bandwidth as well as the total traffic streamed, as measured by our LDMS MSR sampler. The benchmark was run with 4 OpenMP threads that were pinned to a single NUMA domain. It used a memory size of 9155.3 MB, and the delivered bandwidth was calculated as the average of 200 iterations of the Copy+Scale+Add+Triad loops. While not a highly optimized build, the benchmark achieved nearly 50% of theoretical peak as measured by both STREAM and LDMS with a small margin of error. Results are shown in Table II.

STREAM results	MSR counts from LDMS	Difference
11625 MB/s	11356 MB/sec	2.3 %

Table II
STREAM Memory Bandwidth as measured

Floating Point Operations (`FLOP`): The HPL benchmark was chosen as the validation test for this metric because it performs its own `FLOP` accounting and provides this as output upon completion. We ran multiple HPL jobs though the batch system and summed the per minute `FLOP` rates across the nodes used. These per minute sums were graphed to confirm the average `FLOP` rate recorded throughout the job matched the performance reported by HPL. A simple single node example of this process is demonstrated in Figure 2. This validation was also used to confirm the correctness of the realtime processing of the data files and database insertion.

The run of HPL, shown in the figure, reported a performance of 12.82 GFlops and the sustained rate measured was 12.9 GFlops. Higher resolution runs show that the FLOP rate of HPL runs decrease slightly near the end of the calculation and would account for the small difference between the measured one minute averages and the HPL reported value.

Results are shown in Figure 2.

VI. Analysis Support

A. Desired Functional Forms

In order to minimize impact on compute nodes, LDMS typically only samples and transmits raw data. While we want to preserve data in its raw form for future analysis, there are many analyses and related visualizations of interest

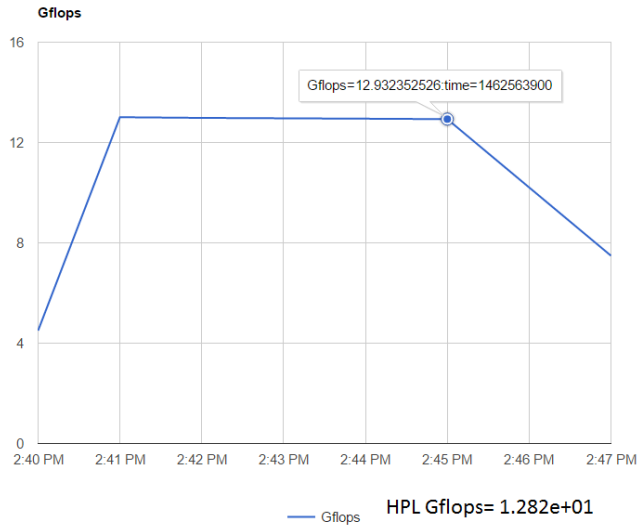

Figure 2. *Flop Validation*: Single process HPL

that require processing and/or functional combinations of the raw data. For instance, time series analysis of the data requires the calculation of counter value differentials for each successive pair of data points. Such computations can be cumbersome to do entirely as a post-processing step on the large CSV data files being generated. Currently in CSV format, the Blue Waters non-MSR raw data amounts to ~55 GB/day and the raw MSR data amounts to ~93 GB/day.

In the validation and initial operation phase of the MSR samplers, described in Section V, we had a need to process per core counter values. We wrote a C++ application to process the successive per core values of the `FLOP` rates. This application can be generally applied to generate a stream of summed differentials and rates from a set of monotonically increasing fields in CSV format. It takes as input the output of the raw CSV-formatted data pipe and provides as output a stream of per node rates for the selected counter. This utility has been extended to watch the end of a file as it is being written and periodically process newly appended data. The result is a stream of per node counter rates written to an output stream or file. For the Blue Waters configuration, that output stream is fed to a named pipe and into our Integrated System Console (ISC) [9] via syslog-ng. The ISC supports a web interface by which data from the ISC can be queried for a variety of both raw and processed data. Web-based time-history plots are presented and the data can also be downloaded for further user processing. The web-based visualizations are the source of the per job `FLOPS` figures in Section VII.

Currently on Blue Waters, for our non-MSR metrics, we run two LDMS store plugins [10]: one that writes all raw data to a pipe through which it is forwarded to the Lustre filesystem for long-term storage and analyses, which we perform in parallel on Blue Waters itself; and the other that performs *in transit* raw, rate, and difference calculations on a subset of the data, that we have determined to be of immediate interest. This derived data is written to another pipe and forwarded on to our ISC for analysis in conjunction with other system data.

In addition to the processing for MSR data mentioned above, we are interested in more complex functional forms. In order to support these complex derivations, as well as the need to streamline the handling of the increased number of metrics involved with the MSR counters, we are developing more flexible *in transit* processing tools. Because all data must transit an LDMS *aggregator* in order to be stored, a LDMS store plugin is a convenient place for *in transit* processing of the data with no adverse impact on the compute nodes.

Recent enhancements to LDMS include support for vector types, which are a better intrinsic match for per core data, and a more general *function store* plugin. The functions as well as input and output variables are defined in a configuration file. The current set of functional forms supported operate on both scalar and vector data and includes addition, subtraction, multiplication, and division; deltas and rates from the previous timestep; minimums, maximums, averages, and threshold comparison. All computations include an optional scaling factor to address issues of resolution in the case of computations having integer outputs. Outputs of one function can be specified to be the inputs to a subsequent function. With our planned upgrade to the latest LDMS version on Blue Waters, some of the current interim processes for the MSR data manipulation can be alleviated.

B. Dynamically Changing Counter Analysis

Dynamic indication of the state of the MSR counters must be included with the data in order to enable meaningful analysis.

Including the *Event Select* register value as part of the reported metrics, as described in Section IV-E enables the system service to signal the counter state with respect to the expected state. When the MSR *Event Select* register is set to the configured value, using either the `initialize` or `reassign` LDMS MSR *sampler* plugin directives, the *Event Select* register value is reported for that counter metric value. As discussed in Section IV-E when the *Event Select* register has changed by other means, both the reported *Event Counter* and *Event Select* values are reported as zero. This enables distinguishing a configured change from an externally (to the LDMS *sampler*) invoked change. While this means that a second value has to be checked for each *Event Counter* read, the required comparison is trivial.

The generalized support of counters and specification of the *Event Select* register in the LDMS metric sets does have some complexities with respect to processing. Currently the metric names in the CSV header do not provide any

indication of the nature of counters being collected. Further, the *Event Select* register value, displayed as a decimal number, cannot be easily translated into the counter identity. While such a conversion can be done in a specialized store plugin that cross-references with the configuration file, this adds significant complexity. In LDMS, there is an option for assigning an additional datafield to a metric; this can be used to write out the *Event Select* value in a more usable format (e.g., use the additional datafield to hold the string name or a simpler numerical identifier); this would be at the expense of adding an additional metric for every counter collected. We are currently evaluating viable approaches and tradeoffs in handling these issues.

VII. Use Cases

This section describes our current use of `RETIRED_FLOPS` and `L3_CACHE_MISSES` for investigations into applications utilization of floating point processing and memory bandwidth resources on compute hosts. These represent 2 of the 10 MSR counters currently being collected system wide across all 27,648 hosts of Blue Waters and called out in Table I. `RETIRED_FLOPS` is collected on a per core basis and represents the sum of the following counts:

- Double precision multiply-add FLOPS (Multiply-add) counts as 2 FLOPS.
- Double precision divide/square root FLOPS.
- Double precision multiply FLOPS.
- Double precision add/subtract FLOPS.
- Single precision multiply-add FLOPS (Multiply-add) counts as 2 FLOPS.
- Single-precision divide/square root FLOPS.
- Single-precision multiply FLOPS.
- Single-precision add/subtract FLOPS.

`L3_CACHE_MISSES` is collected on a per NUMA node basis and represents the number of L3 cache misses for all cores within a NUMA node. In particular we sum counts for the following events:

- Both prefetch and non-prefetch
- Read Block Modify
- Read Block Shared (Instruction cache read)
- Read Block Exclusive (Data cache read)

Because the L3 Cache is the Last Level Cache (LLC) for the AMD Interlagos architecture, L3 Cache misses can be used in conjunction with the cache line size in order to determine the memory read bandwidth generated within a particular NUMA node. In this work we calculate memory read bandwidth for each NUMA node separately both in terms of bytes/sec and as a percentage of the maximum theoretical memory bandwidth of a NUMA domain. For our case a L3 cache line is 64 bytes; therefore the rate of L3 cache misses times 64 yields the generated read traffic in bytes per second. The DRAM memory size is 32GB per socket and 16GB per NUMA node. Four floating-point compute units or eight integer-cores share a memory controller and a 8 MB L3 data cache and comprise a NUMA node.

There are 4 channels per socket (2 per NUMA domain). Each channel runs at 1,600 MT/s with a payload of 8 bytes per transaction. Thus the theoretical peak (or maximum) memory bandwidth per NUMA domain is 2x1600x8=25600 MB/s. This is used in our calculations to convert a bandwidth number to a percentage of the maximum theoretical bandwidth. As a point of comparison for the values presented for the various jobs here, AMD documentation shows the stream[11] TRIAD benchmark attaining a sustained rate of 75 GB/s out of 102.4 GB/s, or 73% of peak, for a 6276 dual-socket G34 [12]. This is comparable to values reported by other vendors for the TRIAD benchmark [13]

The calculations presented are not based on mapping of *L3 Cache* misses into location related reads. Rather we make the assumption that each *L3 Cache* miss from a NUMA node results in a 64 Byte load from memory within its NUMA domain. In reality, a *L3 Cache* miss that cannot be satisfied by a load from the local NUMA domain DRAM will result in a request to a remote NUMA node to be satisfied. Such a miss in our current analysis will result in incorrect assumptions due to incorrect attribution of the miss to a 64 Byte read from the local NUMA domain. If this resulted in a L3 cache miss on the remote node, this would result in double counting. If it was satisfied by the remote node's L3 it would still result in an incorrect 64 byte read being attributed to the local NUMA's DRAM. Likewise IO through links attached to a remote NUMA node will be incorrectly accounted for.

Our locality assumptions are only reasonably correct (with the exception of IO devices) if the user configured processor and memory affinity for their application runs. This was the case in all of the use case applications presented in this paper. Additional counters do exist for counting how many DRAM reads and writes are generated by cores on the local node to other nodes in the coherent fabric. Likewise there are counters for providing similar insight into cross NUMA domain IO traffic. Use of both of these counters can provide insight into the level of processor data and IO device affinity and will be incorporated into future work so the use of processor and memory affinity can be assessed.

Figures showing *FLOPS* performance (in GFlops) that are presented in the following sections are taken from web-based visualizations of the run-time processed data described in Section VI-A.

A. Job Analyses

The `L3_CACHE_MISSES` and `RETIRED_FLOPS` MSR data for several applications are discussed in the following section. These applications compose part of the suite of applications used for the Sustained Petascale Performance

benchmark suite [3] and are representative of the workload run on Blue Waters at the time of initial operations. Each application has a unique computational signature as described in [3]. These same features, reflecting how the code interacts with the architectural features of the hardware, can be seen in the MSR data presented here.

1) NWCHEM: The NWCHEM package [14], [15] provides a suite of scalable capabilities to perform mixed quantum-mechanics and molecular-mechanics simulations. The NWCHEM application used in this paper was run using coupled-cluster singles+doubles (CCSD) that are more memory bandwidth limited compared to triples T(CCSD) that are floating-point intensive, complex element matrix-matrix multiplication based. The benchmark was run on 100 Blue Waters XE nodes with 8 MPI tasks per node and 4 OpenMP threads per MPI task, and used approximately 30 GB of memory per node. Task placement and thread affinitization were used to maintain memory allocation local to NUMA domains. Each XE node has 64 GB of DRAM with about 2 GB used by the OS and tmpfs. NWCHEM uses Global Arrays for parallel data movement.

Figure 3 (top) shows a trace of FLoating point OPeration (FLOP) rates used by the whole job as presented via the Blue Waters web interface to the collected MSR data. The average value is 971 GF/s or approximately 9.7 GF/s per node. Figure 3 (bottom) shows the *L3 Cache* miss rate based used bandwidth trace for a representative node in the job. Statistics across nodes of the job are presented in Table III. The used bandwidth rate attributed to NUMA domain 3 is lower than that for the other three. The maximum memory bandwidth achieved is about 14% of the theoretical peak per NUMA domain. The saw-tooth pattern in memory bandwidth is likely due to the differences in characteristics of the single excitation (CCS) phase to the double excitation (CCD) phase. The `Instructions Per Cycle` (IPC) trace from reference [3] shows a similar variation although at a much higher frequency. Further investigation into the coincidence of the phases and the observed MSR data is needed to determine the computational phase dependence.

NUMA	Min (B/s)	Max (B/s)	Avg (B/s)
0	12.2e6 +/- 1.81e6	3580e6 +/- 107e6	1342e6 +/- 45.7e6
1	12.5e6 +/- 1.94e6	3564e6 +/- 115e6	1355e6 +/- 43.7e6
2	12.3e6 +/- 1.84e6	3552e6 +/- 101e6	1359e6 +/- 44.6e6
3	29.4e6 +/- 3.75e6	2119e6 +/- 72.9e6	885e6 +/- 46.4e6

Table III
NWCHEM SUMMARY STATISTICS ACROSS ALL NODES FOR EACH NUMA DOMAIN. AVERAGE OF THE MIN, MAX, AND AVG VALUES ACROSS ALL NODES IN (B/S).

2) NAMD: NAMD [16], [17] is a parallel molecular dynamics code designed for high performance classical simulation of large bio-molecular systems. NAMD is designed to overlap various force computations ranging from pairwise Lennard-Jones forces to short-range and long-range

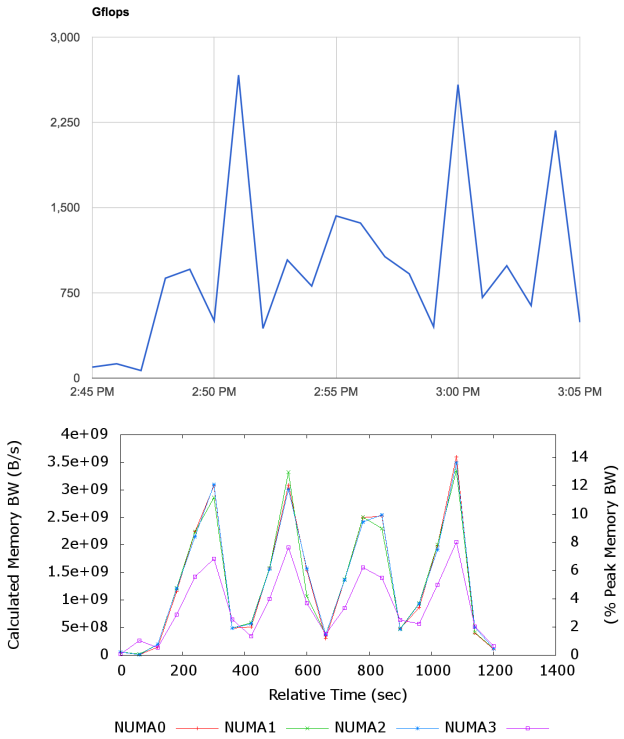

Figure 3. *NWCHEM* Job GFlops (top). Rate of *L3 Cache* misses converted to memory bandwidth in B/s and % of theoretical peak memory bandwidth (60 sec intervals) vs. time for a representative node running within the same NWCHEM job (bottom).

electrostatics forces. NAMD uses the Charm++ parallel programming model and scales to hundreds of thousands of cores though typical simulations utilize hundreds of cores. Charm++ provides a dynamic load balancer to effectively utilize all available resources.

The benchmark used for this evaluation of NAMD was configured to use double precision operations for most calculations with some single precision operations in the Ewald sum phase. The job referenced here was run on 100 Blue Waters XE nodes with 2 communication tasks per node and 16 threads per task. The total per node memory requirement was approximately 5 GB. During execution, the dynamic load-balancer uses a hierarchical strategy [18] to balance the work across groups of processors after a user-configurable number of steps. The amount of computational work is sufficiently uniform across the processor groups that the load balancer does not have a visible performance impact as shown in in Figure 4 (top). The average FLOP rate in Figure 4 (top) is approximately 35 GF/s or 11% of peak. This is a typical sustained rate for NAMD.

Figure 4 (bottom) shows memory bandwidth rate derived from *L3 Cache* misses for a representative node and summarized in Table IV. The difference between memory bandwidth rates, due to *L3 Cache* misses, between consecutive domains on the same processor package (NUMA domains

1 and 3 have higher rates than 2 and 4) is possibly due to processor-memory affinitization issues within the Charm++ runtime but will need to be studied further.

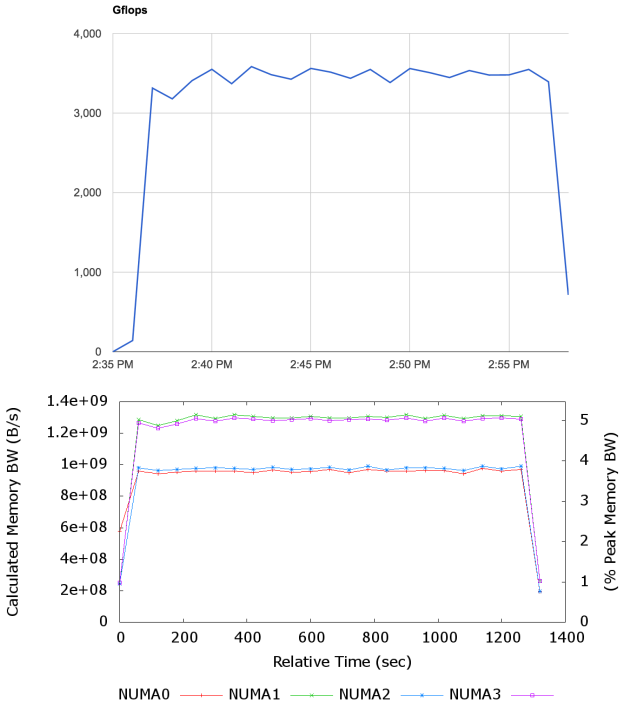

Figure 4. *NAMD* Job GFlops (top). Rate of L3 Cache misses in B/s and % Peak Memory BW (60 sec intervals) vs. time for a representative node in the job (bottom).

NUMA	Min (B/s)	Max (B/s)	Avg (B/s)
0	203e6 +/- 21.7e6	1002e6 +/- 31.3e6	920e6 +/- 31.5e6
1	246e6 +/- 15.7e6	1325e6 +/- 17.6e6	1216e6 +/- 15.9e6
2	195e6 +/- 4.57e6	997e6 +/- 15.6e6	913e6 +/- 13.8e6
3	249e6 +/- 16.2e6	1336e6 +/- 20.5e6	1226e6 +/- 18.4e6

Table IV
NAMD SUMMARY STATISTICS ACROSS ALL NODES FOR EACH NUMA DOMAIN. AVERAGE OF THE MIN, MAX, AND AVG VALUES ACROSS ALL NODES IN (B/S).

3) PPM: PPM, [19] is an astrophysical, finite difference CFD simulation code capable of investigating flash events in early generation stars which result in explosive events such as supernovae and modeling inertial confinement fusion processes. It is a hybrid MPI + OpenMP code written to be highly efficient with respect to cache utilization by use of cache blocking, cache line flushing, volatile variables, and data prefetching. The code also utilizes single precision.

The PPM benchmark was run on 132 Blue Waters XE nodes with 16 MPI tasks per node and 2 OpenMP threads per task fully using all cores on a node with appropriate processor-memory affinitization, using approximately 10 GB per node.

In Figure 5 the FLOP rate for PPM is shown. Each node is achieving approximately 75 GF/s or about 24% of peak which is typical for this application and is the highest sustained FLOP performance of the applications presented in this study.

In Figure 5 (2nd-4th) the calculated memory bandwidth for various nodes is shown. Many nodes are well-balanced in calculated memory bandwidth across time and across numa nodes; however on some nodes there is a single numa node where the rate is substantially lower than the rest. Further study is needed to understand the factors leading to that imbalance. The fraction of peak memory bandwidth achieved by this application falls somewhere in the middle compared with the other applications in the study despite having the highest percent of peak FLOP rate.

Table V shows the balanced, average memory bandwidth rates from L3 cache misses for PPM.

NUMA	Min (B/s)	Max (B/s)	Avg (B/s)
0	2.71e6 +/- 0.270e6	2695e6 +/- 424e6	2420e6 +/- 415e6
1	3.05e6 +/- 0.271e6	2700e6 +/- 420e6	2424e6 +/- 416e6
2	2.89e6 +/- 0.259e6	2701e6 +/- 420e6	2425e6 +/- 416e6
3	3.26e6 +/- 0.276e6	2699e6 +/- 421e6	2423e6 +/- 415e6

Table V
PPM SUMMARY STATISTICS OF CALCULATED MEMORY BANDWIDTH ACROSS ALL NODES FOR EACH NUMA DOMAIN. AVERAGE OF THE MIN, MAX, AND AVG VALUES ACROSS ALL NODES IN (B/S).

4) MILC: MILC [20] is a large scale suite of applications for the numerical simulation of lattice quantum chromodynamics on a wide variety of platforms. MILC based applications account for significant usage of many NSF funded HPC center systems. The applications are generally sensitive to interconnect performance variation due to the 4-dimensional halo-exchanges and the global reduction operations in the conjugate gradient phase. It is also sensitive to memory latency and bandwidth as access patterns often result in non-uniform striding into data structures. The code supports single and double precision data types.

For this use case the MILC application su3_rhmd_hisq was run on 108 Blue Waters XE nodes with 16 MPI tasks per node using less than 1 GB per node. Task affinitization was used to pin the MPI tasks to each floating point core of the processor. OpenMP was not enabled but the code base does support the hybrid MPI+OpenMP programming model. The code implements uniform domain decomposition of a 4-dimensional lattice with uniform computational work per node. Imbalances in performance can be due to communication variation and the irregular memory access patterns.

The small local lattice per MPI task results in an average of 11 GF/s per node. This is shown in the top plot of Figure 6. This 3.3% of peak FLOP rate is the lowest average seen in the applications used in this study. The low floating point rate is due in part to poor cache utilization

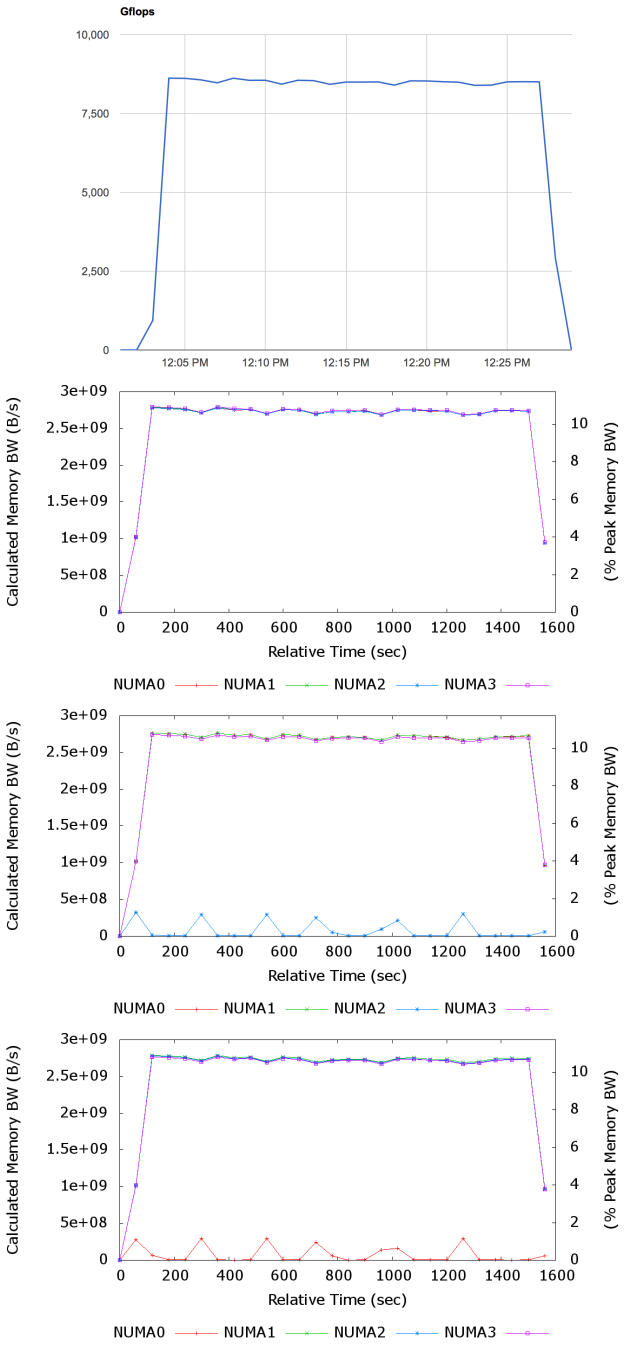

Figure 5. *PPM* GFlops for job (top). Calculated memory bandwidth in B/s and % peak memory bandwidth (60 sec intervals) vs. time. Many nodes are well-balanced in cache misses across time across numa nodes (2nd); however for some nodes there is one numa node where the rates are substantially lower (3rd & 4th).

resulting in modest pressure on the memory subsystem. Figure 6 (middle) shows calculated memory bandwidth in B/s vs. time for a representative node in the job. Statistics across nodes of the job are given in Table VI and the average memory bandwidth for MILC is the highest of the applications in this study.

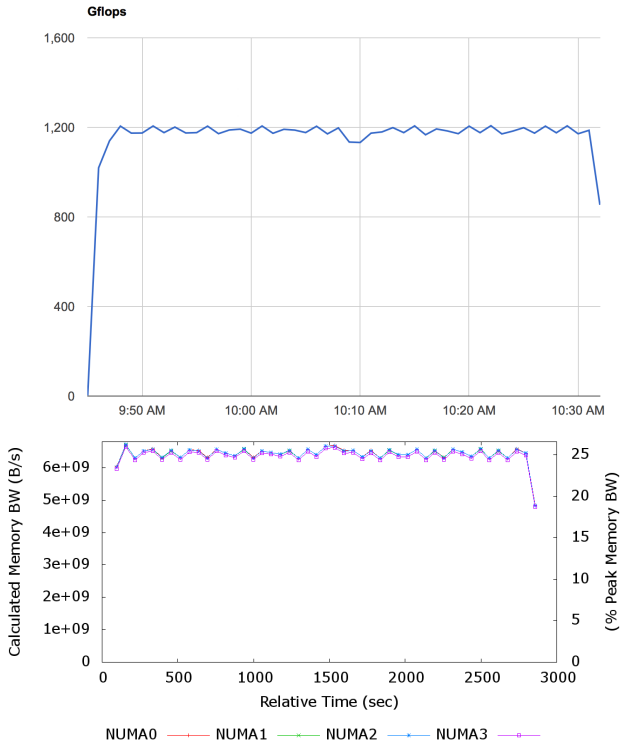

Figure 6. *MILC* GFlops for job (top). Calculated memory bandwidth in B/s and % peak memory bandwidth (60 sec intervals) vs. time for a representative node in the job (bottom).

NUMA	Min (B/s)	Max (B/s)	Avg (B/s)
0	4819e6 +/- 15.5e6	6690e6 +/- 20.5e6	6393e6 +/- 21.2e6
1	4845e6 +/- 12.3e6	6724e6 +/- 17.3e6	6427e6 +/- 16.1e6
2	4846e6 +/- 14.2e6	6725e6 +/- 17.9e6	6429e6 +/- 18.0e6
3	4828e6 +/- 16.7e6	4828e6 +/- 16.7e6	6403e6 +/- 22.9e6

Table VI
MILC SUMMARY STATISTICS FOR CALCULATED MEMORY BANDWIDTH ACROSS ALL NODES FOR EACH NUMA DOMAIN. AVERAGE OF THE MIN, MAX, AND AVG VALUES ACROSS ALL NODES IN (B/S).

5) VPIC: VPIC [21], [22] is a fully relativistic, charge-conserving, 3D explicit particle-in-cell code which simulates plasma physics. VPIC integrates the relativistic Maxwell-Boltzmann system in a linear background medium for multiple particle species, in time with an explicit-implicit mixture of velocity Verlet, leapfrog, Boris rotation and exponential differencing based on a reversible phase-space volume conserving second order Trotter factorization. VPIC is a floating point intensive code with well optimized compute kernels

that are implemented with compiler vector intrinsics where possible. It uses MPI+OpenMP for a three-dimensional, nearest-neighbor communication structure.

For the benchmark used in this study, VPIC was run on 144 Blue Waters XE nodes with 32 MPI tasks per node (no OpenMP) that required approximately 30 GB of memory per node. Task-processor affinitization was used to pin the tasks to the integer cores. Note that two integer cores share a floating point unit in the AMD Interlagos processor.

Figure 7 (top) shows the trace of FLOP rate for VPIC. The sustained average of 60 GF/s per node is 19% of peak and is second only to PPM in floating point performance. In Figure 7 the calculated memory bandwidth vs. time for a representative node in the job (bottom) is shown. Statistics across nodes of the job are given in Table VII. In general, rate is fairly consistent across time and across nodes and the average rate is second only to the MILC application. VPIC is able to achieve high floating point rates while sustaining modest pressure on the memory subsystem.

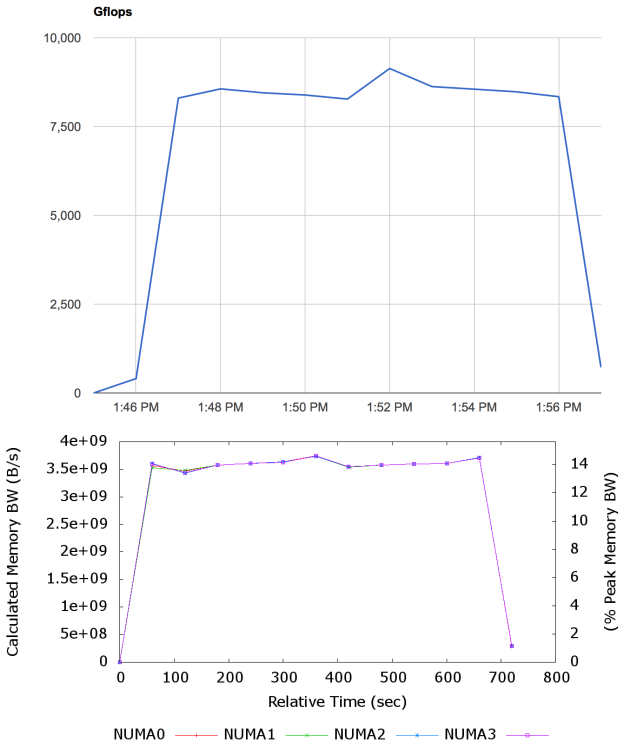

Figure 7. *VPIC* GFlops for job (top). Calculated memory bandwidth in B/s and % peak memory bandwidth (60 sec intervals) vs. time for a representative node in the job (bottom).

B. Multi-run Analysis

MSR data can be helpful in understanding application behavior when supporting our partners. In this example, it was reported that some runs of the same application were not completing in the time allocated for the workload. It was difficult to determine if and when the jobs were experiencing unexpected behavior. Through analysis of floating point execution rates across nodes, it was determined that successful jobs exhibited uniform FLOP rates across all nodes within the job and failed jobs began to show load imbalance across the processes. By monitoring the standard deviation of FLOP rates across nodes, variations can be clearly detected and the divergent values can be clearly highlighted as in Figure 8. This plot shows the standard deviation across the 20 nodes of the job using the one minute FLOP average throughout the job runtime. Each line represents a separate job instance with the same application, input and node count. Excluding the first and last minutes of startup and shutdown, each of the four successful runs show a consistent and low standard deviation of FLOP rates across nodes of approximately 0.1. In each of the 3 failed runs, the failure can be observed visually via the graph or numerically by measuring this single value. The behavior can be monitored in realtime graphically or scripted and monitored to identify errant runs.

NUMA	Min (B/s)	Max (B/s)	Avg (B/s)
0	2.63e6 +/- 0.398e6	3716e6 +/- 26.0e6	3056e6 +/- 15.1e6
1	2.96e6 +/- 0.455e6	3714e6 +/- 25.5e6	3054e6 +/- 14.9e6
2	2.79e6 +/- 0.423e6	3715e6 +/- 25.6e6	3055e6 +/- 15.0e6
3	3.10e6 +/- 0.426e6	3714e6 +/- 25.8e6	3055e6 +/- 15.1e6

Table VII
VPIC SUMMARY STATISTICS FOR CALCULATED MEMORY BANDWIDTH ACROSS ALL NODES FOR EACH NUMA DOMAIN. AVERAGE OF THE MIN, MAX, AND AVG VALUES ACROSS ALL NODES IN (B/S).

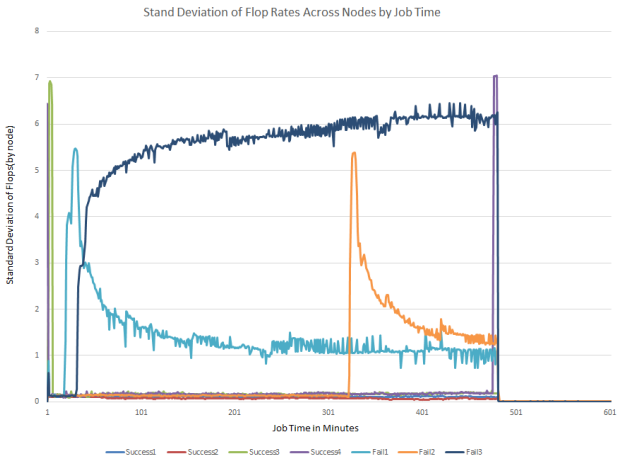

Figure 8. *Multi-run Analysis* Flop Variation: Standard deviation of FLOP rates across nodes at one minute intervals. Failing runs had inconsistent FLOP rates across nodes across the job run time.

VIII. CONCLUSIONS

In this work we have presented our implementation for collection of MSR counters as a system service. This enables us to obtain system-wide insight into overall, and job based,

utilization of resources, including compute processing and memory bandwidth. Unlike other system monitoring data sources, the MSR counters are subject to external reassignment due to other user tools which may be setting the counters for other purposes. This results in unique design requirements, which we have handled, for the discovery of counter change, recovery of counter settings for the next user, and reporting of values upon change in a manner that is easily discoverable during subsequent data analysis. In addition, we provide a user-invokable interface for interacting with the service that can enable users to take advantage of the service without requiring any changes to their code. Current state is made available during run time to analysis tools.

We have presented use cases of analyses from the system data collection for applications representative of the Blue Waters initial workload. In these use cases we have shown that useful information can be obtained from continuous system-level monitoring data. Higher fidelity profiling with typical profiling tools can be performed for further investigation.

Our on-going work centers on support for low-latency analysis of the run-time data. This includes increased support for *in transit* processing, easing the interpretation of the dynamic data, and enabling user visualization of significant data that can give insight into their applications resource utilization. We are also investigating inclusion of counters necessary for assessing processor and memory affinity.

ACKNOWLEDGMENT

The authors would like to thank Stephen Olivier of SNL for useful discussions on MSR counters and Jeremy Enos and Joshi Fullop of NCSA for useful discussions on implementation within the Blue Waters environment.

REFERENCES

[1] Advanced Micro Devices (AMD), "BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors," Technical report 42301 Rev 3.14, January 2013.

[2] T. Evans, W. Barth, J. Browne, R. DeLeon, T. Furlani, S. Gallo, M. Jones, and A. Patra, "Comprehensive Resource Use Monitoring for HPC Systems with TACC Stats," in *Proc. of the First International Workshop on HPC User Support Tools*, 2014, p. 1321.

[3] G. Bauer, T. Hoefler, W. Kramer, and B. Fiedler, "Analyses and Modeling of Applications Used to Demonstrate Sustained Petascale Performance on Blue Waters," May 2012, Cray User's Group.

[4] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[5] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proc. of the Department of Defense HPCMP User's Group Conf.*, 1999, pp. 7–10.

[6] Cray Inc., "Using Cray Performance Analysis Tools," Cray Doc S-2376-52, 2011.

[7] *msr(4) Linux Programmer's Manual*, March 2009.

[8] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *Proc. Int'l Conference for High Performance Storage, Networking, and Analysis (SC14)*, 2014.

[9] B. Semeraro, R. Sisneros, J. Fullop, and G. Bauer, "It Takes a Village: Monitoring the Blue Waters Supercomputer," in *1st Wrk. on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA) Proc. IEEE Int'l Conf. on Cluster Computing (CLUSTER)*, 2014.

[10] J. Brandt, A. Gentile, M. Showerman, J. Enos, J. Fullop, and G. Bauer, "Large-scale Persistent Numerical Data Source Monitoring System Experiences," in *Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA) at IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[11] J. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers." [Online]. Available: https://www.cs.virginia.edu/stream/

[12] "Memory Bandwidth (STREAM) - Two-Socket Servers (including AMD Opteron 6300 Series Processors." [Online]. Available: http://www.amd.com/en-us/products/server/benchmarks/memory-bandwidth-stream-two-socket-servers-6300

[13] A. Supalov, A. Semin, C. Dahnken, and M. Klemm, *Optimizing HPC Applications with Intel Cluster Tools: Hunting Petaflops*. Berkeley, CA, USA: Apress, 2014, pp. 68–69.

[14] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations," *Comput. Phys. Commun.*, vol. 181, no. 1477, 2010.

[15] "NWChem: Open Source High-Performance Computational Chemistry." [Online]. Available: http://www.nwchem-sw.org/index.php/Main_Page

[16] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, 2005.

[17] "NAMD Scalar Molecular Dynamics." [Online]. Available: http://www.ks.uiuc.edu/Research/namd/

[18] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical load balancing for charm++ applications on large supercomputers," in *2010 39th International Conference on Parallel Processing Workshops*, Sept 2010, pp. 436–444.

[19] P. Colella and P. R. Woodward, "The piecewise parabolic method (ppm) for gas-dynamical simulations," *Journal of Computational Physics*, vol. 54, no. 1, pp. 174 – 201, 1984. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0021999184901438

[20] G. Bauer, S. Gottlieb, and T. Hoefler, "Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3_rmd," in *Proc. 12th Int'l. IEEE/ACM Symp. on Cluster, Cloud, and Grid Computing*, 2012.

[21] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, 2008.

[22] "Vector Particle-in-Cell (VPIC) Project." [Online]. Available: https://github.com/losalamos/vpic