ClusterWatch: Flexible, Lightweight Monitoring for High-end GPGPU Clusters

Magdalena Slawinska, Karsten Schwan, Greg Eisenhauer
College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332–0250
Email: magg@gatech.edu, karsten.schwan@cc.gatech.edu, eisen@cc.gatech.edu

***Abstract*—The ClusterWatch middleware provides runtime flexibility in what system-level metrics are monitored, how frequently such monitoring is done, and how metrics are combined to obtain reliable information about the current behavior of GPGPU clusters. Interesting attributes of ClusterWatch are (1) the ease with which different metrics can be added to the system—by simply deploying additional "cluster spies," (2) the ability to filter and process monitoring metrics at their sources, to reduce data movement overhead, (3) flexibility in the rate at which monitoring is done, (4) efficient movement of monitoring data into backend stores for long-term or historical analysis, and most importantly, (5) specific support for monitoring the behavior and use of the GPGPUs used by applications. This paper presents our initial experiences with using ClusterWatch to assess the performance behavior of the a larger-scale GPGPU-based simulation code. We report the overheads seen when using ClusterWatch, the experimental results obtained for the simulation, and the manner in which ClusterWatch will interact with infrastructures for detailed program performance monitoring and profiling such as TAU or Lynx. Experiments conducted on the NICS Keeneland Initial Delivery System (KIDS), with up to 64 nodes, demonstrate low monitoring overheads for high fidelity assessments of the simulation's performance behavior, for both its CPU and GPU components.**

I. Introduction

Modern scientific discovery requires investments in next-generation hardware and software, provoking questions by sponsoring organizations, hardware vendors, site administrators, and end-users regarding the efficient use of the computing resources acquired by the simulation codes being run. Of particular interest in this space is the effectiveness and utility of accelerators, such as GPGPUs, which are becoming increasingly important for obtaining high performance for scientific simulations.

Realizing the importance of performance monitoring and profiling large-scale parallel codes and machines, tools like Ganglia [1] and MRNet [2] have sought scalability in operation and low overhead in use through distributed data structures efficiently mapped to parallel machines. Program profiling tools such as TAU [3] have created rich infrastructures for monitoring desired program attributes, while seeking to minimize monitoring overheads and resulting program perturbation.

We extract from extensive prior work like the efforts cited above the following characteristics critical for the scalability of monitoring large-scale systems.

- Monitoring should be *selective*, meaning that it must be possible to select precisely what program or machine attributes should be tracked, and with what accuracy. This makes it possible to control the monitoring overheads experienced by applications and systems.
- To reduce the volume of monitoring data and following an approach similar to what is now being pursued for high-performance I/O in exascale systems [4], the capture of monitoring data (i.e., data logging) should be combined with its *dynamic analysis*, termed *monalytics* in recent monitoring research [5].
- Monitoring selectivity and analytics should be dynamic, leading to *interactive models* of system and program monitoring. An example are the "zoom in" and "zoom out" capabilities described in [6]. This is particularly useful for troubleshooting long-running applications [7], [8], [9], [10], [11], [12], [13], where it is difficult to know what metrics to capture and how to process metric data, a fact that is causing current enterprises such as Amazon to simply dump all monitoring logs to disk or other offline storage [14], in order to later analyze it in the various ways needed for performance or program debugging and understanding. Unfortunately, with machine sizes growing toward the exascale, it is no longer possible or desirable to capture the potential petabytes of potentially useful log data (and write it to offline storage) just because such data might be needed for future analysis. There is a consequent need for the *online management of monitoring* and monitoring processes.
- Raw or analyzed monitoring data may require preservation for subsequent analysis, which implies the need to *efficiently store monitoring outputs* on long-term storage, and to effectively manage such historical data, including to analyze offline jointly with online data. One way to address this need is to use time series databases, like OpenTSDB [15], to jointly maintain select data about the current vs. long-term data captured and analyzed [16].

This paper describes the *ClusterWatch* monitoring system, designed for monitoring large-scale parallel applications and the systems on which they run. ClusterWatch is a lightweight, pluggable monitoring middleware designed for (i) selective monitoring, (ii) online analytics for monitoring data, along with the (iii) online management of the monitoring and analytics actions being used, and (iv) efficient interactions with offline storage via a TSDB, using the OpenTSDB framework [15]. ClusterWatch fills a gap between either system-level or application-level monitoring, making it possible to simultaneously do both, in order to offer end users rich monitoring experiences, including those needed to obtain insight into how GPGPUs are utilized from the system's as well as from the particular application's standpoints. ClusterWatch uses the notion of "spies" to obtain such functionality, and for detailed program profiling, it has the unique ability to co-run with sophisticated program profiling tools like TAU [3].

ClusterWatch is constructed with the EVPath [17] middleware, using it both as an efficient transport and as a way to add analytics to the log data stream. EVPath is an event processing middleware whose features of interest for ClusterWatch include its evDFG (EVPath Data Flow Graphs) facility for dynamically deploying entire monitoring/analytics – monalytics – graphs when and if needed – to obtain dynamic selectivity – and its capability of runtime binary code generation for dynamically pushing analytics actions to where they are needed.

This paper's experiments with the current implementation of ClusterWatch serve to gain useful insights into the execution of a GPGPU-based simulation, using the CUDA version of the LAMMPS materials modeling code [18] as a driving example. The paper presents results of validation experiments demonstrating the reliability of obtained monitoring data both with smaller scale experiments and within larger scale GPGPUs settings, all run on the XSEDE NICS Keeneland Initial Delivery System (up to 64 12-core nodes and total 192 GPGPUs). Experimental evaluations demonstrate low ClusterWatch overheads, in part due to monitoring selectivity, and additional design discussions explain how ClusterWatch will obtain the other desirable properties of large-scale monitoring systems.

II. Related Work

Current distributed monitoring systems such as Ganglia [1], Nagios [19], Munin [20], Host sFlow [21], Heka [22], MRNet [2] make it possible to capture a rich set of monitoring data. Ganglia scales up to 2000-node clusters and it targets monitoring federations of HPC clusters. It uses XML to represent data, XDR as a data transport, and RRDtool (Round Robin Database Tool) [23] to store and visualize data. Nagios offers comprehensive monitoring solutions for the entire IT infrastructure, allowing for monitoring applications, services, operating systems, network protocols, system metrics, etc [19]. It can be tailored to user's needs with a rich selection of plugins (50 basic plugins and nearly 2000 additional plugins). Munin [20] aims to aid users in diagnosing performance issues, emphasizing its plug-and-play capabilities and ease of developing new plugins tailored to the user's specific needs. It focuses on monitoring resource performance, and, similarly to Ganglia, it exploits RRDtool. The Host sFlow [21] aims at providing an open source monitoring solution to capture various server-related performance metrics. It implements the sFlow standard [24], an industry specification for monitoring high-speed switched networks. sFlow specifies the format of the exported data and it can collect data of an sFlow compliant device. The sFlow data can be handled by many popular monitoring systems, including Ganglia, Wireshark [25], and the Linux ntop utility. There is also an sFlow specification that describes NVML GPU structures [26]. Mozilla Heka [22] has been designed to be pluggable and simplify the collection, and analysis of data at the cluster level from multiple data sources. Finally, MRNet [2] is focused on monitoring scalability, for single, large-scale parallel machines, offering efficient methods for data collection and aggregation.

Examples of popular tools targeting program-level monitoring are Vampir [27], TAU [3], Scalasca [28], Periscope [29]. Vampir is a complex tool enabling scalable trace analysis of gigabytes of monitoring data, interactive visualization of collected data, and it also offers accelerator monitoring (CUDA and OpenCL). Data analysis is based on Score-P [30], which supports runtime data collection and code instrumentation. TAU [31] is a profiling and tracing toolkit aimed at the performance analysis of parallel programs. It instruments the application source code and provides useful performance visualization analyses and displays, the latter via ParaProf [32]. It supports monitoring GPGPUs via the NVIDIA CUDA Profiling Tools Interface (CUPTI) [33]. Scalasca [28] allows for measuring and analyzing the runtime behavior of parallel programs, specifically targeting communication and synchronization bottlenecks. Periscope [29] offers the application-level monitoring via Monitoring Request Interface (MRI); the application processes need to be linked with (MRI) in order to enable monitoring. Monitoring is provided by a hierarchy of communication and analysis agents.

A plethora of node-level tools make it possible to monitor various system and hardware resources, including the popular Linux tool top and its variants (ntop, htop), sar (activity node-level collection system), PAPI (Performance Application Programming Interface) [34], etc.

Numerous research efforts have addressed program troubleshooting, summarized in [7], [8], [9], [10], [11], [12], [13]. One recent effort also using EVPath [17], a messaging constructor middleware used by ClusterWatch, and its evDFGs (EVPath Data Flow Graphs) is VScope [6], which is a middleware that can capture various metrics and dynamically deploy relevant analysis actions to understand the collected metric data and accordingly, troubleshoot the application. A similar approach to metric deployment flexibility is present in SDIMS [35] and Moara [36] that are aggregation systems exploiting Distributed Hash Tables (DHT) to construct dynamic aggregation trees.

In the current ecosystem of monitoring tools, ClusterWatch aims to contribute to several research topics. First, ClusterWatch will explore the collection and analysis of combined application (TAU, Lynx [37]–a GPGPU code profiling tool) and system level data to provide robust analytics of the collected data at the cluster, node, and application level. In contrast to popular monitoring systems such as Ganglia and Nagios that merely offer a selection of various plugins to configure the infrastructure to capture relevant data, ClusterWatch goes beyond that. Its goal is to provide to the user a comprehensive and extensive monitoring status perspective based on all (or select) available levels of monitoring granularity. Second, ClusterWatch will offer flexible metric deployment mechanisms. Similarly to VScope, it will use EVPath's Data Flow Graphs to deploy automatically relevant metrics tailored to current monitoring requirements. Third, ClusterWatch will investigate what, when, and how monitoring can best be performed, e.g., via a "zoom in capability" to provide more or less monitoring data on demand, turning on/off metrics at disparate granularity levels. Finally, ClusterWatch targets specifically large-scale accelerated systems, i.e., HPC GPGPU clusters. We believe that combining advanced GPGPU monitoring solutions such as Lynx [37] with cluster-level knowledge can provide more insights into utilization of GPGPU resources by modern, highly parallel applications at leadership HPC facilities.

III. ClusterWatch

The purpose of ClusterWatch is to provide a flexible and lightweight framework for capturing arbitrary system- and application-level metrics in high-end clusters, while offering straightforward ways to add to that metric set. ClusterWatch aims at enabling filtering and processing monitoring metrics on the entire path from their sources to eventual metric storage, to reduce data movement overhead and to permit rapid metric evaluation for use in real-time monitoring [6], [5].

Monitoring data is represented with efficient self-describing data formats, called Fast Flexible Serialization (FFS) [38], to permit its rapid analysis via data filters and/or aggregators placed dynamically into the data path, but using standard formats for monitoring logs placed into backend storage. The FFS description of the C structure shown in Listing 1 that defines the monitoring record for GPU related data is presented in Listing 2.

The captured data, "raw" or processed are stored in a persistent backend storage. There is an ongoing effort to implement an advanced analytics module based on OpenTSDB, a distributed, and scalable Time Series Database on top of HBase, that will allow for performing complex analysis on collected monitoring data.

Although ClusterWatch supports capturing metrics from CPUs such as cpu utilization, memory utilization, or network load, its primary focus is on GPUs present on the nodes of current high-end cluster machines. The current ClusterWatch prototype allows for capturing system-level data; support for obtaining application-level (both CPU and GPGPU) data is to some extent implemented (TAU based) and partially is a work in progress (Lynx based).

Conceptually, the ClusterWatch implementation consists of transporting and monitoring layers, as depicted in Figure 1. The transporting layer is responsible for enabling communication between ClusterWatch components whereas the monitoring layer comprises modules for capturing, processing, storing, and analyzing monitoring data.

A. Monitoring Data Format

One of the ClusterWatch design goals is to use as little bandwidth as possible whereas providing enough flexibility to allow for fast data analysis and standard log formats on the backend storage. To reconcile those two oppose aims we leverage the FFS self-describing data format [38]. FFS was implemented to support efficient marshaling of data across a heterogeneous computing environment. FFS is record-oriented: data writers must provide a description of the records

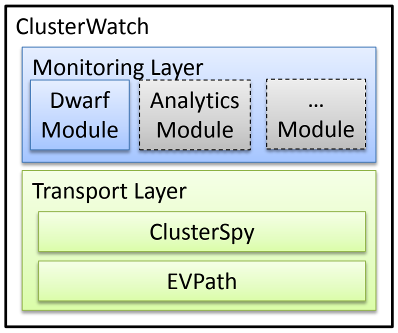


Fig. 1. The ClusterWatch architectural overview. Gray dashed items indicate work-in-progress components.

written out and the data readers must provide a description of the records that they are interested in reading. For instance, the binary structure such as nvml_mon (Listing 1) has to be described by providing information such as the name of the field, the type of the field, the size, and the offset relative to the beginning of the structure, and next registered with the FFS format server. Correspondingly, the receiver side needs to register with the FFS format server a similar description of the record it intends to read, so data marshaling between a sender and a receiver can be appropriately performed.

In order to gauge the efficiency of the FFS self-describing data format we compared sizes of our binary format to a hypothetical, XML-based format of our monitoring data. To obtain the XML description of structures sent to the ClusterWatch aggregator by ClusterWatch collectors, we wrote a simple program that uses FFS API to wrap C structures with a hypothetical XML tags. Listing 3 presents the example, suppositive, and automatically generated XML description for the GPU monitoring record showed in Listing 1. The sizes of binary C structures (i.e., sizeof(·)) vs. speculative XML-based textual format (i.e., the number of characters) describing the ClusterWatch monitoring data are shown in Listing 3. Unsurprisingly, they demonstrate that the FFS binary format performs $\sim$3-9 better in terms of data compactness than the hypothetical, textual-based description. Having efficient data format in the context of marshaling is particularly important for frequent and periodic monitoring records (vs. capability records sent only once during the resource registration in ClusterWatch).

B. Transport Layer

The current implementation of the transport layer, concretized as ClusterSpy, is built on top of EVPath [17], [39], the event-based middleware. EVPath is designed to enable an easy implementation of overlay networks supporting active data processing, routing, and management at all points in the overlay. One of the central ideas built into the library is the concept of "stones" that correspond to processing entities in dataflow diagrams. EVPath supports creation, monitoring, management, modification and destruction of the stones. It provides several types of stones that enable data filtering, data transformation, data multiplexing and demultiplexing, and data transfer between processes over the network interconnect.

Listing 1. The GPGPU monitoring data structure sent periodically by the NVML spy to the aggregator.

```
// monitoring structure for sending GPU related monitoring data
struct nvml_mon {
        struct timeval ts;     // the timestamp of the measurement
        int id;                // the id of the monitoring node
        int gpu_count;         // the number of devices attached to a node

        char** performance_state;

        int *mem_used_MB;
        float *util_gpu;        // utilization GPU in percentage
        float *util_mem;        // utilization memory in percentage
        float *power_draw;      // power draw limit
        float *graphics_clock; // clock_graphics / Max Clock Graphics
        float *sm_clock;        // clock_graphics / Max Clock Graphics
        float *mem_clock;       // clock_graphics / Max Clock Graphics
};
```

Listing 2. FFS descriptions of GPGPU monitoring data sent by NVML spies to the aggregator.

```
FMField nvml_mon_field_list[] = {
 {"ts", "struct timeval", sizeof(struct timeval), FMOffset(struct nvml_mon *, ts)},
 {"id", "integer", sizeof(int), FMOffset(struct nvml_mon *, id)},
 {"gpu_count", "integer", sizeof(int), FMOffset(struct nvml_mon *, gpu_count)},
 {"performance_state", "string[gpu_count]", sizeof(char*), FMOffset(struct nvml_mon *,
    performance_state)},
 {"mem_used_MB", "float[gpu_count]", sizeof(float), FMOffset(struct nvml_mon *, mem_used_MB)},
 {"util_gpu", "float[gpu_count]", sizeof(float), FMOffset(struct nvml_mon *, util_gpu)},
 {"util_mem", "float[gpu_count]", sizeof(float), FMOffset(struct nvml_mon *, util_mem)},
 {"power_draw", "float[gpu_count]", sizeof(float), FMOffset(struct nvml_mon *, power_draw)},
 {"graphics_clock", "float[gpu_count]", sizeof(float), FMOffset(struct nvml_mon *,
    graphics_clock)},
 {"sm_clock", "float[gpu_count]", sizeof(float), FMOffset(struct nvml_mon *, sm_clock)},
 {"mem_clock", "float[gpu_count]", sizeof(float), FMOffset(struct nvml_mon *, mem_clock)},
 {NULL, NULL, 0, 0}
};
```

Listing 3. Automatically generated, hypothetical, XML counterpart of the description of the GPGPU monitoring data sent by NVML spies to the aggregator.

```
<struct_nvml_mon>
<ts>
<tv_sec>140309924857192</tv_sec>
<tv_usec>140309922810500</tv_usec>
</ts>
<id>1644578478</id>
<gpu_count>3</gpu_count>
<performance_state>
P0 P0 P12</performance_state>
<mem_used_MB>
204 205 200 </mem_used_MB>
<util_gpu>
100 30 89.03 </util_gpu>
<util_mem>
0.2 0.21 3.56 </util_mem>
<power_draw>
33.76 34.83 35.21 </power_draw>
<graphics_clock>
100 0 100 </graphics_clock>
<sm_clock>
0 0 0 </sm_clock>
<mem_clock>
0 0 0 </mem_clock>
</struct_nvml_mon>
```

TABLE I
BINARY VS. HYPOTHETICAL TEXTUAL FORMAT OVERHEAD FOR MONITORING DATA IN CLUSTERWATCH. THE TABLE PRESENTS A ROUGH COMPARISON BETWEEN SIZES OF RECORDS SENT TO THE CLUSTERWATCH AGGREGATOR BY COLLECTORS FOR THE HYPOTHETICAL XML FORMAT AND FOR THE BINARY FFS FORMAT EXPLOITED BY CLUSTERWATCH. THE ROWS SHOW THE SIZE OF TWO TYPES OF RECORDS: THE CAPABILITY RECORD (E.G., CPU-CAP), AND THE MONITORING DATA RECORD (E.G., CPU-MON). THE CAPABILITY RECORDS ARE SENT ONLY ONCE WHEN THE RESOURCE REGISTERS ITS PRESENCE IN CLUSTERWATCH. THE MONITORING RECORDS ARE SENT PERIODICALLY TO THE CLUSTERWATCH AGGREGATOR.

Data	XML size [bytes]	Binary size [bytes]	Ratio [XML/Binary]	Data	XML size [bytes]	Binary size [bytes]	Ratio [XML/Binary]
cpu-cap	231	49	4.71	cpu-mon	291	84	3.46
mem-cap	261	57	4.58	mem-mon	513	56	9.16
net-cap	302	73	4.14	net-mon	410	48	8.54
nvml-cap	1,797	549	3.27	nvml-mon	451	146	3.09
tau-cap	220	49	4.49	tau-mon	1,005	192	5.23
Sum	2,811	777	3.62	Sum	2,670	526	5.08

The ClusterSpy transport implementation leverages EVPath, yet provides higher level interfaces suitable for monitoring. For instance, it provides an API call `arrange_node()` to plug a monitoring entity into its underlying, EVPath-based messaging infrastructure that executes a sequence of EVPath primitives to establish a connection (`CMlisten()`, `EValloc_stone()`, `create_attr_list()`, `add_xxx_attr()`, `EVassoc_bridge_action()`, etc).

ClusterSpy offers a centralized many-to-one communication topology, i.e., there are many local data collectors and one data aggregator. Supporting more complex data flow graphs is a work in progress, and is briefly described in Section III-D.

C. Monitoring Layer

The monitoring layer aims at providing modules for actual monitoring and analysis of captured monitoring data. At present, the Dwarf module is implemented and used, and the development of the analytics OpenTSDB module is a work in progress.

The Dwarf module offers the concept of "collectors" executed locally on monitored nodes and an "aggregator" that aggregates data sent by collectors, as depicted in Figure 2. Collectors, called "spies," gather metrics such as memory, cpu, gpu utilization, network load. The aggregator analyzes the monitoring data obtained from collectors and then prompts actions that respond to the recognition of certain monitoring states; it can also write obtained monitoring data to storage for postmortem data analysis. A memory mapped file is used as an intermediate buffer for such output actions, to enable periodic readers that place data into backend stores such as Sqlite3 or OpenTSDB.

The concept of "spies" aims at providing a convenient and uniform abstraction that enables attaching ClusterWatch to an arbitrary source of monitoring data. The primary task of the spy is to get the data out of the source and present them to ClusterWatch. The actual implementation of taking into possession the monitoring data varies from spy to spy and is data source specific. For instance, it might be as simple as opening the file if the monitoring data are exposed via a standard file interface; it might require usage of an API provided by a library or, in certain cases, it might be necessary to modify the source code of the monitoring data source in order to embed the spy and allow for getting the data out to ClusterWatch.

Technically, all spies are executed in a single process space (a patrol process). Currently, the patrol program is configurable via a simple key-value file that tells the patrol program to turn on/off a particular spy. We are currently adding functionality that can take such actions at runtime, to enable the "zoom in" functionality explained in [6]. Basically, the idea is to enable detailed monitoring if needed and disable monitoring or limit the monitoring details if such information is not required at the moment.

The aggregator process (a ranger process) is configured via a similar key-value file, and it is executed on an arbitrary cluster node, provided there are communication paths enabled between all collector processes and the aggregator. The memory mapped file readers are implemented as a Python script that can be configured to run a specific reader via command-line parameters.

We have implemented six spies, namely cpu, memory, network, nvidia_smi, NVML, and TAU spies. The former three spies utilize the information obtained from the pseudo-file system (/proc); the nvidia_smi spy uses the NVIDIA System Management Interface (nvidia-smi) utility—a command line utility intended to aid in the management and monitoring of NVIDIA GPU devices [40]; the NVML spy uses NVIDIA Management Library (NVML) [41] that provides API to gather information about NVIDIA GPU devices (the nvidia_smi utility exploits this library as well); and the TAU spy exploits TAU [3]. There is an ongoing effort to implement the Lynx spy that will enable detailed, kernel-level and program-level GPGPU monitoring.

In order to implement a new spy it is required to describe the data sent by a spy to the aggregator and routines to enable reading the data. Since ClusterWatch relies on the EVPath-based transport, the data need to be described in the Fast Flexible Serialization (FFS) format [38]. Listing 2 presents the example of the description of the monitoring data sent by the NVML spy to the collector. Specifically, ClusterWatch assumes that there are two types of data transferred from a spy to the aggregator: the description of the resource's capabilities and the actual monitoring data. For instance, the description of a GPGPU resource sent by the spy to the aggregator includes information about the number of GPGPU devices attached

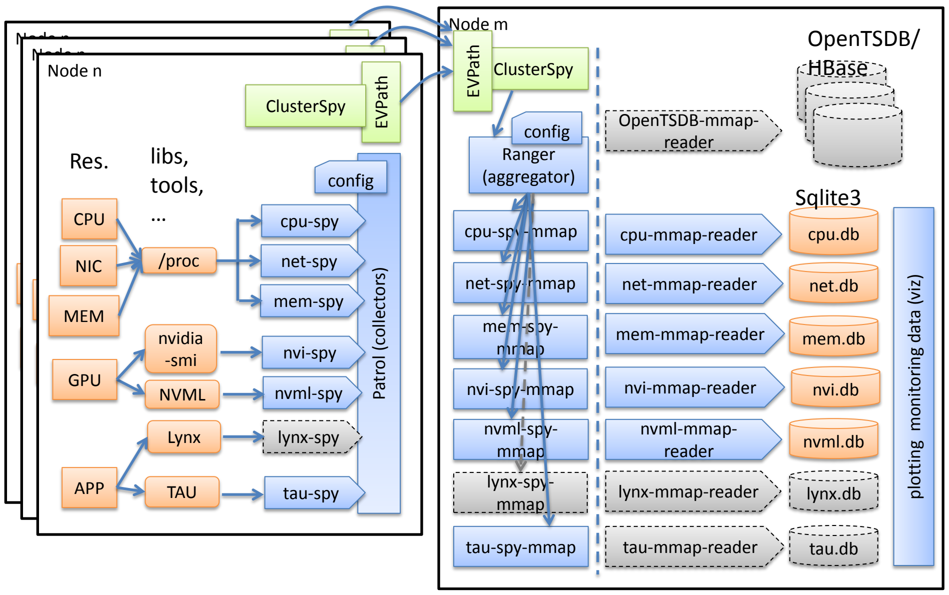


Fig. 2. Monitoring in ClusterWatch. Gray dashed items indicate work-in-progress components. Blue and green components correspond to layers depicted in Figure 1 and illustrate which components have been already implemented.

to the node, total amount of available memory per GPGPU device, the power limit, maximum values of graphics, memory and SM clocks, the compute mode, the serial number, and the product name. The resource monitoring data include the current "raw" values of the resource's attributes (e.g., power draw, used memory in MB) or basic metrics such as utilization (e.g., GPGPU utilization in percentage, network load).

In addition to the description of the data, adding the new spy requires implementation of a few functions to provide the initialization of the spy and aggregator, the actual read routines to obtain data from the resource, and optional processing routines (e.g., to compute metrics); the example read routine for the NVML spy is shown in Listing 4.

In the current implementation, spies may (1) collect raw data and/or (2) preprocess some of the data. Preprocessing is important for performance reasons, especially if large-scale runs are taken into account that produce enormous amount of raw data and the user is interested only in obtaining certain metrics. The other reason for data preprocessing might be ensuring the quality of monitoring data (e.g., dealing with out-of-range values).

As a result, the data sent to the aggregator can be a mixture of raw data and/or calculated metrics. In particular, the cpu spy allows to obtain information about aggregated cpu utilization (per node) or individual core usage within a node; the memory spy calculates memory utilization as well as buffers, cached, active, inactive, slab, mapped, swap cached, and swap utilization; the net spy reports about network load in terms of bytes transmitted, received or aggregated (both transmitted and received) per network interface; the GPGPU spy (both nvidia_smi and NVML) sends GPGPU utilization, memory utilization and raw memory usage in MB, power draw, performance state, and clocks related data (graphics clock, memory clock, SM clock); the TAU spy logs occurrence of specified user's events such as MPI calls with parameters; and Lynx spy will enable obtaining kernel-level metrics such as the degree of branch divergence, memory efficiency, as well as GPGPU application-level metrics, e.g., CPU-GPGPU data movement costs.

The monitoring module provides also a simple graphical tool, specifically a Python script exploiting matplotlib.pyplot (a MATLAB-like plotting framework), that allows for basic, yet useful visualization of monitoring data stored in Sqlite3 databases by the aggregator. The diagrams can be exported to several popular graphics formats (e.g., png, jpg).

The implementation of a spy may be more or less challenging, depending on the degree of the exposure of monitoring data by a monitoring data source. Some monitoring data are exposed in a well-defined manner, e.g., through an API (a file API for accessing the pseudo-file system /proc), or a library (e.g., NVML). In other cases such as TAU [3], the appropriate monitoring data exposure by a data source has to be implemented in addition. For instance, in order to implement the TAU spy we had to modify TAU sources to enable the TAU events to be written out to a message queue so the TAU spy can collect the data and send them to the aggregator. The TAU spy also implements a prototype selective monitoring functionality, i.e., the user can enforce starting or stopping a collection of a specified TAU event.

The other interesting work-in-progress is the implementation of the Lynx spy. Similarly to the TAU spy, the Lynx spy will offer the program-level monitoring; specifically, it will enable reporting detailed GPGPU kernel-level metrics such as the branch divergence, memory efficiency, CPU-GPGPU data movement costs, obtained from Lynx [37]—the GPGPU code profiling tool.

Listing 4. The monitoring data reading routine for the NVML spy.

```
int read_mon_nvml(void *data){
  ...
  struct nvml_mon *p_rec = (struct nvml_mon*) data;
  ...
  gettimeofday(&p_rec->ts, NULL);        // get the timestamp

  for (i = 0; i < p_rec->gpu_count; i++){
    nvmlDevice_t p_device = NULL;
    ...
    nvmlMemory_t mem;
    if ( nvmlDeviceGetMemoryInfo(p_device, &mem) == NVML_SUCCESS){
      p_rec->mem_used_MB[i] = mem.used / 1000000;
      p_rec->util_mem[i] = (float) mem.used / (float) mem.total * 100.0;
    }
    nvmlUtilization_t util_rates;
    if ( nvmlDeviceGetUtilizationRates(p_device, &util_rates) == NVML_SUCCESS){
      p_rec->util_gpu[i] = util_rates.gpu;
    }
    ...
  }
  return ret;
}
```

D. Advanced Features

The primary use case for ClusterWatch is the online capture and analysis of both node- and machine-level information about the current state of GPGPU-based cluster machines. There are several ongoing efforts to enhance ClusterWatch with advanced monitoring features including (1) providing scalable and flexible spy topologies based on EVPath Data Flow Graphs (evDFGs), and (2) implementing the analytics OpenTSDB module to enable long-term or historical analysis of the application's behavior.

- Scalable and Flexible Communication Topologies—ClusterWatch relies on the centralized, many-to-one communication topology. Although it is acceptable at our current GPGPU cluster testbed, which has a relatively small number of nodes (120 nodes on the Keeneland Initial Delivery System and 264 nodes on the Keeneland Full-Scale System), the centralized topology will obviously not scale if the number of nodes increases. We have started a development effort to leverage EVPath DFGs in ClusterWatch. We have implemented the tool to easily generate and deploy an arbitrary DFG and we have started integration of the tool with ClusterWatch.
- OpenTSDB analytics—the current readers store the monitoring data obtained by the aggregator from collectors in the Sqlite3 database. ClusterWatch aims at providing a complex analytics backend for the collected data. There is an ongoing effort to develop an OpenTSDB-based analytics engine to store the data and calculate interesting metrics and relationships at runtime as well as provide postmortem analysis.

IV. ClusterWatch Experimental Evaluation

We performed two types of experiments: (1) fidelity tests on a local workstation and a single KIDS node, and (2) larger scale measurements to observe the ClusterWatch overhead. The evaluation of the TAU spy is a work in progress.

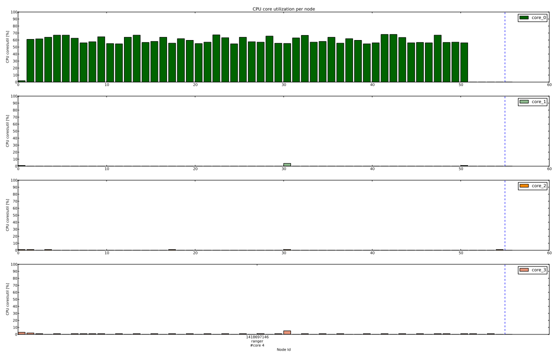
A. Fidelity Tests on a Single Node

In order to validate the fidelity of results reported by ClusterWatch, we have performed validation experiments on our local workstation for cpu, mem, and network spies. The results are presented in Figure 3. The cpu spy was tested by limiting the workload on workstation's individual cores (a workstation hosts Intel(R) Xeon(R) CPU X5365, 3.00GHz) to 30% and 60% with a cpulimit program, which is included in the ClusterWatch distribution. Figure 3(a), presents the expected workload under 60% workload on core 0. The performed experiments showed that ClusterWatch correctly reported cpu utilization under various workloads.

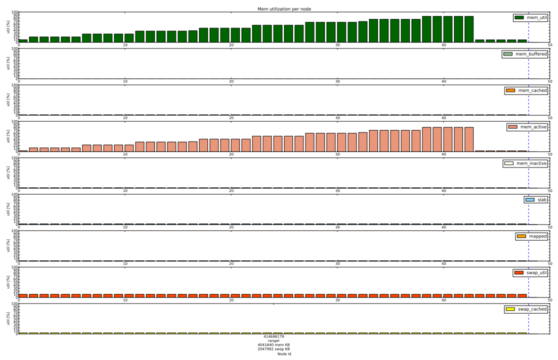
The mem spy was tested by allocating certain amount of memory over short periods of time. Figure 3(b) shows the results reported by ClusterWatch where 400MB of memory has been malloc'ed eight times every five seconds. Again, our experiments demonstrated that ClusterWatch reported expected memory utilization.

Figure 3(c) and Figure 3(d) show the net spy tests—the scp test and the ping test, respectively. Figure 3(c) shows the scp test that contains two consecutive loops—the first loop sends a 16MB file from the local workstation to the other local machine ten times, and the following second loop sends a 16MB file in the opposite direction ten times. The total load presents the sum of bytes transmitted and received. The ping test sends the 65KB packets to and from the workstation every one second. In the scp test, we can observe that ClusterWatch reports as anticipated constant outgoing network load for the first loop, and next constant incoming network load for the second loop. The ping test shows that ClusterWatch recorded as expected ∼65-70KB receive network load, ∼65-70KB transmitted network load, and ∼140KB cumulative network load.

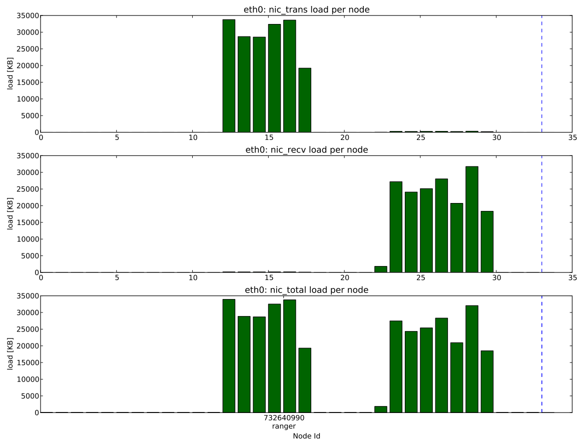
The validation of NVML spy was performed on a single KIDS node by enabling both GPGPU spies, NVML spy and nvidia_smi spy, and carrying out the comparison of their



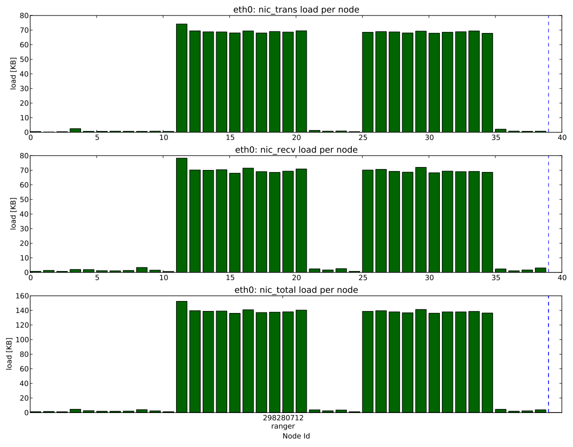
(a) Cpu spy validation on a local workstation. The cpu workload has been limited to 60% on core 0 by the cpulimit program.



(b) Mem spy validation on a local workstation. An in-house program allocated 400MB eight times every five seconds.



(c) Net spy validation on local workstations—an scp test: an in-house script loops 10 times and sends a 16384 KiB file first from the workstation to the other workstation, and next in the opposite direction (the send loops were continually "scp-ing" a file).



(d) Net spy validation on local workstations—a ping test in two directions, i.e., *from* the workstation and *to* the workstation; the test sent ten 65507-byte packets; an interval between consecutive sends was one second.

Fig. 3. ClusterWatch validation experimental results for cpu, memory, and network spies.

outputs. The results are presented in Figure 4. We have observed minor discrepancies between reported values by the NVML spy and nvidia_smi spy, specifically with respect to memory readings. For instance, the total amount of memory used by a GPU workload reported by the NVML spy is 200MB, 204MB, 205MB, whereas the nvidia_smi spy reports 191MB, 195MB. The other discrepancy is in the reported total amount of available memory 5636MB, and 5375MB, for the NVML spy and nvidia_smi spy, respectively. (The NVML spy uses the NVML library, whereas nvidia_smi spy executes the nvidia_smi utility provided by NVIDIA that exploits the NVML library.) The overall outcome of this experiment demonstrated that, apart from mentioned minor discrepancies, both spies report similar values.

B. Overhead Measurements on Larger-scale Runs

We have tested the ClusterWatch prototype in larger scale setups on the NICS KIDS machine (up to 64 nodes totaling to 192 GPGPUs) with LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [18], [42] as a workload, a GPU-accelerated classical molecular dynamics code used in production runs. The KIDS node architecture is HP ProLiant SL390 G7, with two 2.8 GHz Intel Xeon X5660 (Westmere 6-core) per node, 24GB host memory per node, InfiniBand QDR as interconnect, and three NVIDIA Tesla 2090 per node. KIDS has 120 compute nodes.

LAMMPS offers potentials for a rich set of materials (biomolecules, polymers, metals, semiconductors, coarse-grained systems, mesoscopic systems). It can be used as a generic parallel (MPI-based), spatial-decomposition particle simulator at various scales (atomic, meso, or continuum) that can be parameterized with an arbitrary input script.

For ClusterWatch testing purposes, we used LAMMPS v. 5Mar2013 (compiled with OpenMPI v. 1.6.1 and Intel compiler v. 12.1.5 20120612 on Keeneland Initial Delivery System—KIDS [43], [44]) accelerated with a standard GPU package (CUDA Toolkit v. 4.2).

In general, LAMMPS execution involves several execution stages, namely, *pairing*, *neighboring*, *communication*, *outputting*, and *other*. The percentage share of those phases depends on the executed input script. For the experiments, we used the "melt" script distributed with LAMMPS sources (`in.melt.2.5.gpu`); Figure 5 presents the measured percentage time share of those phases with respect to the overall duration of the simulation in the function of the number of GPGPUs. As the number of GPGPUs increases, the communication cost starts to dominate over the pairing phase, and for 192 GPGPUs it is about 52% of the overall duration of the simulation, in comparison to about 40% spent in the pairing phase.

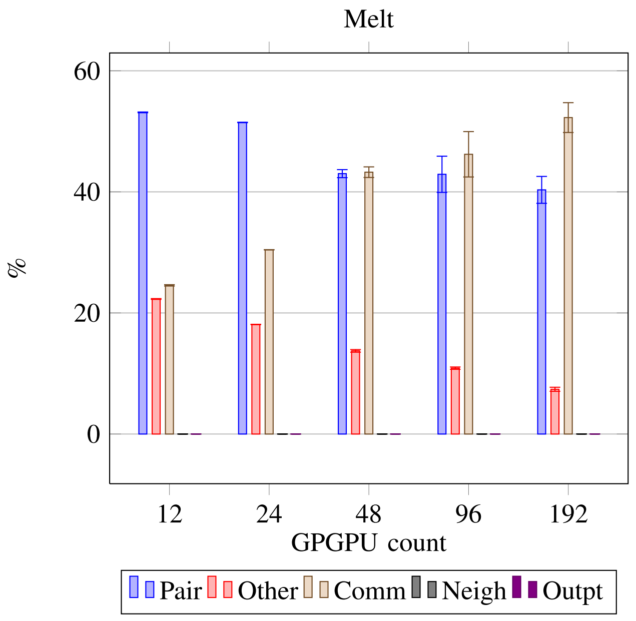


Fig. 5. LAMMPS profile (pairing, neighboring, communication, outputting, other) for the melt script configured with 256K atoms, 180K timesteps. All percentage might not sum up to 100% for a particular GPGPU count due to averaging and rounding errors. The diagram presents the average over 4 or 5 runs for a particular GPGPU count.

We ran ClusterWatch with the LAMMPS as a workload to measure the ClusterWatch overhead. We compared the baseline LAMMPS run with timings obtained from LAMMPS runs executed with ClusterWatch running. After successful execution, LAMMPS outputs a report containing various simulation data including timings for the total LAMMPS simulation duration, i.e., the loop duration. The reported loop duration was used to evaluate the ClusterWatch overhead.

The absolute LAMMPS simulation timings with and without ClusterWatch are presented in Figure 6(a); the percentage with respect to the baseline run is presented in Figure 6(b).

The measured ClusterWatch overhead for 192 GPGPUs was less than 3%.

We observed that the standard deviation between runs on a large number of nodes can be significant. Figure 7 presents the standard deviations over 5 or 8 runs for the baseline runs and two different configuration of ClusterWatch—all implemented spies enabled (apart from the TAU spy) and only cpu/mem/net/nvml spies enabled.

The observed high standard deviations was the main premise to measure overhead introduced on the local workstation. The results are presented in Table II. The overhead is 4.5-fold smaller on the isolated workstation (0.16%) in comparison to a single KIDS node (0.72%).

TABLE II
ClusterWatch overhead w.r.t. the LAMMPS reported "loop time" for the melt input script on a local workstation averaged over 10 runs each. LAMMPS (v. 5Mar2013).

Run name	AVG Loop duration in [sec]	STD DEV	Overhead [%]
M-Baseline	224.95	0.29	0.00
M-5-spies	225.32	0.40	0.16

ClusterWatch can give insights about how system resources are used at large scale. Figures 8—10 present the typical set of monitoring data that can be obtained by running workloads with ClusterWatch.

In particular, Figures 8—10 show monitoring data obtained from LAMMPS executed with ClusterWatch on 192 GPGPUs. In the experiment, LAMMPS was ran with three processes per node, since the KIDS has three GPGPUs attached to a node. The collector and aggregator nodes were configured to perform measurements periodically every one second. Figure 8(a) shows the utilization per core captured by ClusterWatch and confirms that cores 1, 3, 5 were utilized during the simulation on each of 64 compute nodes participating in the LAMMPS simulation (in the experiment LAMMPS processes were pinned to NUMA domain 1 that contains odd cores). Figure 8(b) shows that in general the simulation used 20-30% of memory per node. From Figure 9(b) and Figure 9(a), we can learn that each node used mainly eth0 with 35-40KB total network load (a sum of bytes transmitted and received), apart from the aggregator node that is depicted as a ranger; the total network load for the ranger node was ∼250KB. The transmitting network load was at about 20KB per second, whereas the receiving network load was at about 10-15KB per second. Finally, a few metrics related to GPGPUs are presented in Figure 10(a), Figure 10(b), and Figure 10(c)). We can observe that all 192 GPGPUs were utilized at c.a. 25%. They used about 80% of total memory available on a GPGPU. The GPGPU power draw was at the level 35%-40% of the maximum power draw. We noticed that the change in the GPGPU performance state was reported with a delay (the 12 value means that the GPGPU is in an *idle* state, the 0 value means that the GPGPU is in a *busy* state.)

V. Summary

In this paper, we present the design, prototype implementation, and evaluation of the ClusterWatch monitoring system that aims at providing a flexible, pluggable monitoring infrastructure capable of collecting monitoring system- and program-level data, specifically targeting high-end GPGPU clusters. We evaluate ClusterWatch by testing its fidelity and measuring the overhead that for the LAMMPS workload executed on 192 GPGPUs achieved ∼3%.

Finally, we describe our ongoing and future work that will enhance ClusterWatch with advanced capabilities such as GPGPU kernel-level monitoring with Lynx, scalable and flexible topologies for large-scale setups with EVPath's DFGs, or advanced OpenTSDB-based analytics for enabling long-term or historical analysis of the application's behavior.

ClusterWatch has potential to provide a comprehensive insight into utilization of resources at cluster, node, and application levels that will help understand how to effectively utilize computational resources.

VI. Acknowledgments

Work supported in part by NSF grant 0910735.

We thank students Alexander Merritt and Naila Farooqui for their work on the ClusterWatch transport layer, and Yanwei Zhang for the TAU spy implementation.

References

[1] "Ganglia site," http://ganglia.sourceforge.net/, 2013.

[2] M. J. Brim, L. DeRose, B. P. Miller, R. Olichandran, and P. C. Roth, "MRNet: A Scalable Infrastructure for the Development of Parallel Tools and Applications," *Cray User Group 2010*, May 2010, edinburgh, Scotland.

[3] S. S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[4] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu, "FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics," *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2013)*, May 2013, Boston, MA.

[5] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf, "Monalytics: online monitoring and analytics for managing large scale data centers," in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809073

[6] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, and C. Huneycutt, "Vscope: middleware for troubleshooting time-sensitive data center applications," in *Middleware 2012*. Springer, 2012, pp. 121–141.

[7] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson, "Lightweight, high-resolution monitoring for troubleshooting production systems," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 103–116. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855749

[8] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu, "Fay: extensible distributed tracing from kernels to clusters," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 311–326. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043585

[9] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, vol. 30, no. 4, pp. 65–79, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.68

[10] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google Inc., Tech. Rep., 2010, http://research.google.com/archive/papers/dapper-2010-1.pdf.

[11] *Flume User Guide*, Revision 0.9.4-cdh3u6 ed., Cloudera, March 2013, http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html.

[12] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, 2010, pp. 170–177.

[13] A. Rabkin and R. Katz, "Chukwa: a system for reliable large-scale log collection," in *Proceedings of the 24th international conference on Large installation system administration*, ser. LISA'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924976.1924994

[14] "Amazon cloudwatch," http://aws.amazon.com/cloudwatch/, 2013.

[15] *OpenTSDB manual*, 2013, http://opentsdb.net/manual.html.

[16] I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger, "Specialized Storage for Big Numeric Time Series," in *Proceedings of the 5th Workshop on Hot Topics in Storage and File Systems*, June 2013.

[17] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan, "Event-based systems: opportunities and challenges at exascale," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '09. New York, NY, USA: ACM, 2009, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/1619258.1619261

[18] S. J. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *J. Comp. Phys.*, vol. 117, pp. 1–19, 1995.

[19] "Nagios site," http://www.nagios.org/, 2013.

[20] "Munin site," http://munin-monitoring.org/, 2013.

[21] "Host sFlow site," http://host-sflow.sourceforge.net, 2013.

[22] "Heka–Data Acquisition and Collection Made Easy," http://heka-docs.readthedocs.org/en/latest/, 2013.

[23] "RRDtool–logging and graphing," http://oss.oetiker.ch/rrdtool/, 2013.

[24] "sFlow standard site," http://www.sflow.org, 2013.

[25] L. Chappell., *Wireshark 101: Essential Skills for Network Analysis*, ser. Chappell University. Protocol Analysis Institute, Inc., Chappell University, 2013. [Online]. Available: http://www.wiresharkbook.com

[26] R. Alexander and P. Phaal, "sFlow NVML GPU Structures," http://www.sflow.org/sflow_nvml.txt, August 2012.

[27] "Vampir site," http://www.paratools.com/Vampir, 2013.

[28] "Scalasca," http://www.scalasca.org/, 2013.

[29] "Periscope Performance Measurement Toolkit," http://www.lrr.in.tum.de/ periscop/, 2013.

[30] "Scalable Performance Measurement Infrastructure for Parallel Codes," http://www.vi-hps.org/projects/score-p/, 2013.

[31] "TAU—Tuning and Analysis Utilities," http://www.cs.uoregon.edu/research/tau/, 2013.

[32] *ParaProf—User's Manual*, University of Oregon Performance Research Lab, 2010, http://www.cs.uoregon.edu/research/tau/docs/paraprof/.

[33] "CUDA Profiling Tools Interface," https://developer.nvidia.com/cuda-profiling-tools-interface, 2013, nVIDIA. Developer Zone.

[34] "Performance Application Programming Interface," http://icl.cs.utk.edu/papi/, 2013.

[35] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 379–390. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015509

[36] S. Y. Ko, P. Yalag, I. Gupta, V. Talwar, D. Milojicic, and S. Iyer, "Moara: Flexible and scalable group-based querying system," in *In Proceedings of the 9th ACM/IFIP/USENIX Middleware*, 2008.

[37] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili, "Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures," in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software*, ser. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 58–67. [Online]. Available: http://dx.doi.org/10.1109/ISPASS.2012.6189206

[38] G. Eisenhauer, "FFS Users Guide and Reference," http://www.cc.gatech.edu/systems/projects/FFS/manual.pdf, October 2011.

[39] ——, "The EVPath library website," http://www.cc.gatech.edu/systems/projects/EVPath/doxygen/index.html, March 2013.

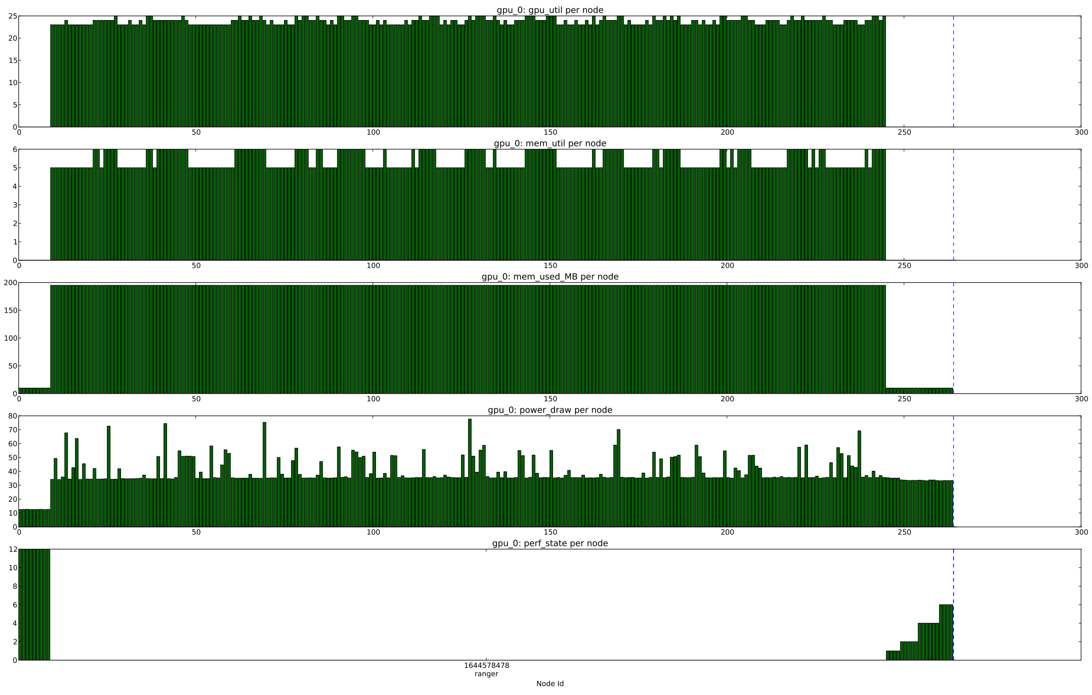
[40] "NVIDIA System Management Interface program," https://developer.nvidia.com, nvidia-smi.4.304.pdf, August 2011, NVIDIA Corporation.

[41] "NVML web site," https://developer.nvidia.com/nvidia-management-library-nvml, 2013, NVIDIA Corporation.

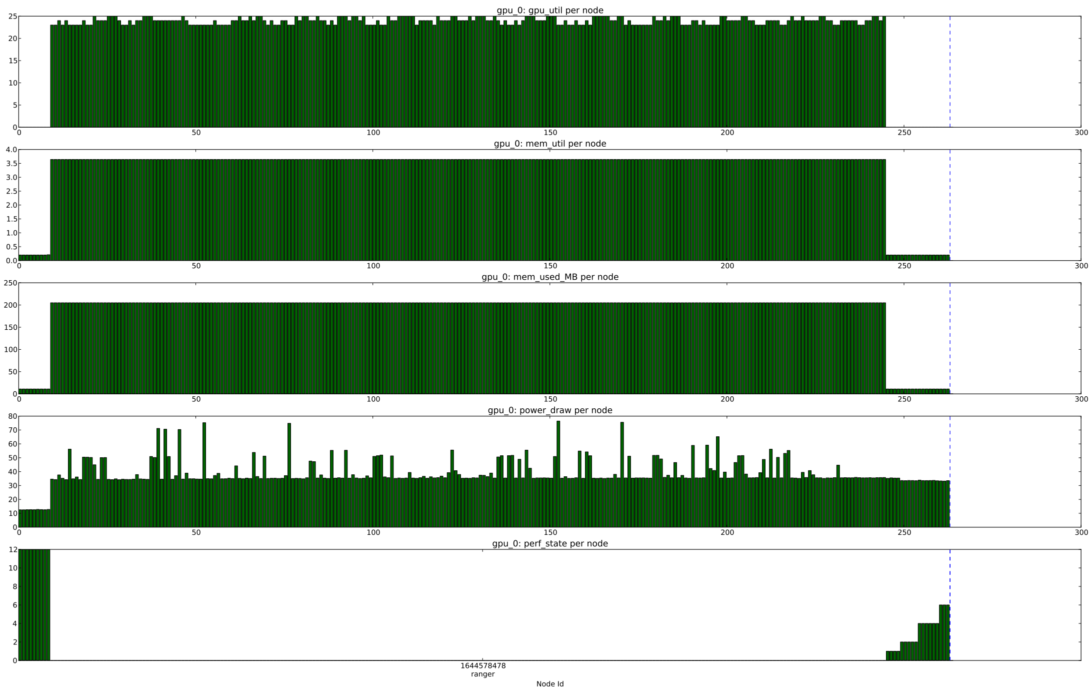
[42] "LAMMPS Molecular Dynamics Simulator – LAMMPS WWW Site," http://lammps.sandia.gov, 2013, (distributed by Sandia National Laboratories).

[43] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community," *Computing in Science Engineering*, vol. 13, no. 5, pp. 90–95, Sept-Oct 2011.

[44] "Keeneland web site," http://keeneland.gatech.edu/, 2013.

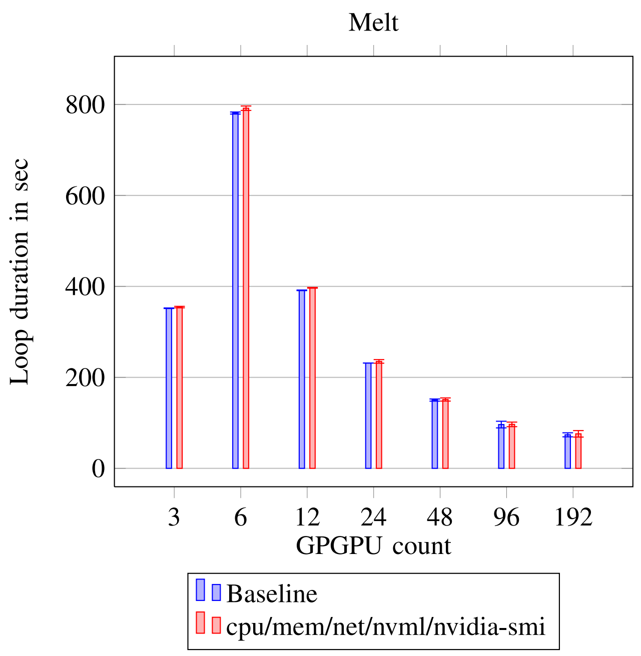


(a) Monitoring data for GPU0 and LAMMPS as a workload captured by the nvidia_smi spy on KIDS. All values are as % (y axis), apart from mem_used_MB (in MB) and GPGPU performance state (an integer number 0-12; 0 meaning the “busy” state, and 12—“idle state.”)

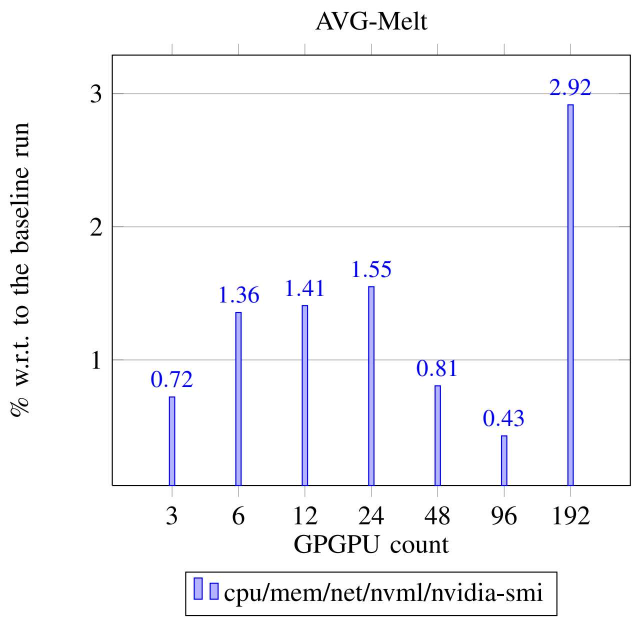


(b) Monitoring data for GPU0 and LAMMPS as a workload captured by the NVML spy on KIDS. All values are as % (y axis), apart from mem_used_MB (in MB) and GPGPU performance state (an integer number 0-12; 0 meaning the “busy” state, and 12—“idle.”

Fig. 4. ClusterWatch validation experimental results for GPU spies.



(a) LAMMPS/melt: 256K atoms; 36K timesteps for 3 GPGPU count, 180K timesteps for 6,12,24,48,96,192 GPGPU count.



(b) LAMMPS/melt: 256K atoms, 3 GPGPUs—36K timesteps and 6, 12, 24, 48, 96, 192—180K timesteps timesteps each.

Fig. 6. ClusterWatch overhead w.r.t. the LAMMPS reported loop time for the melt input script Figure 6(a), and corresponding overhead percentage Figure 6(b). The diagrams present average over 4 or 5 runs for a particular GPGPU count.

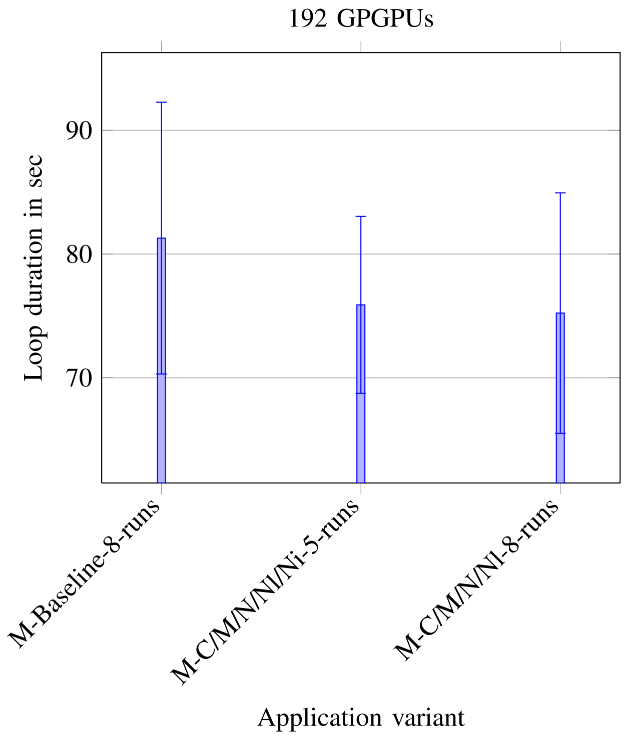
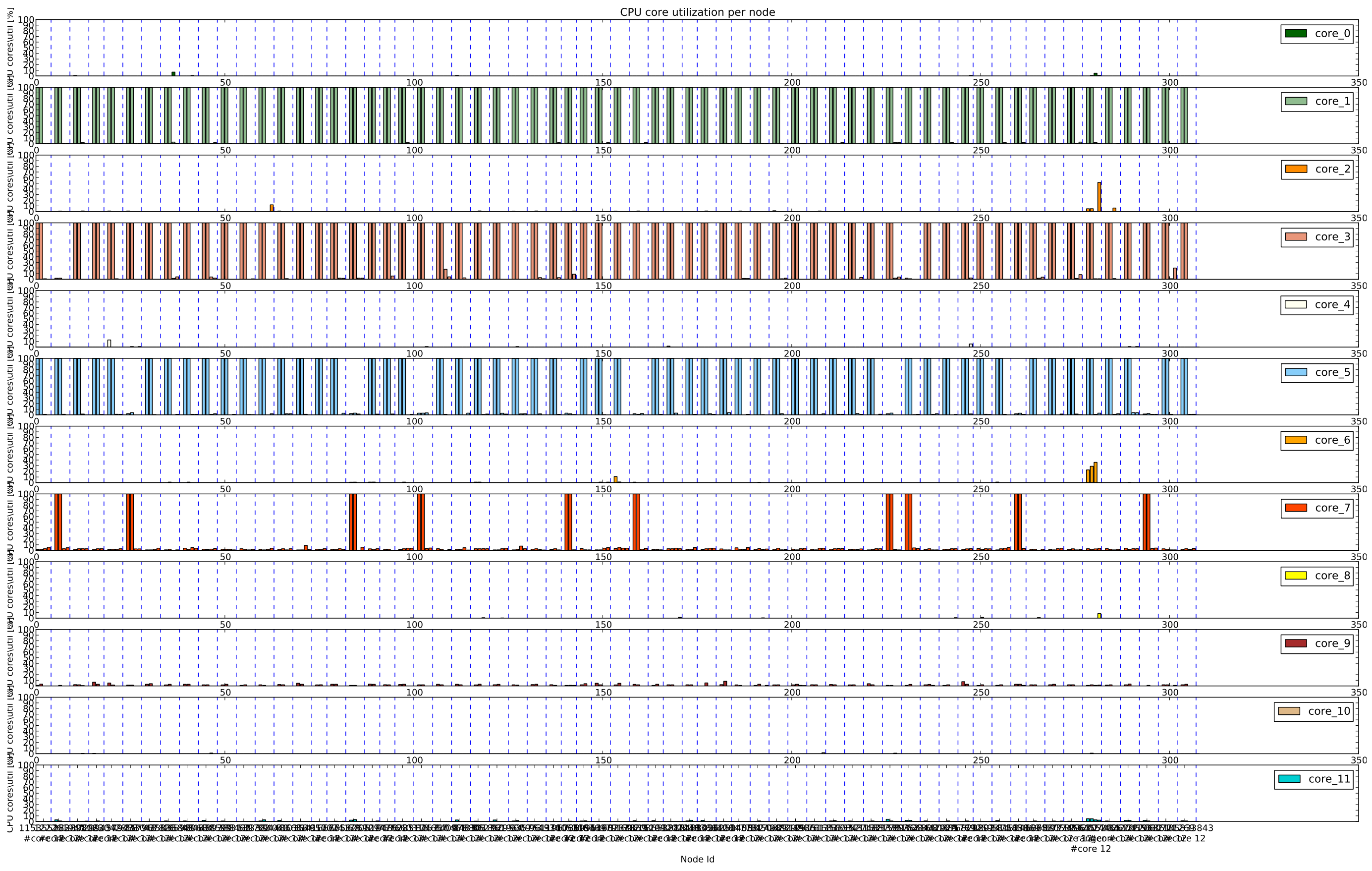
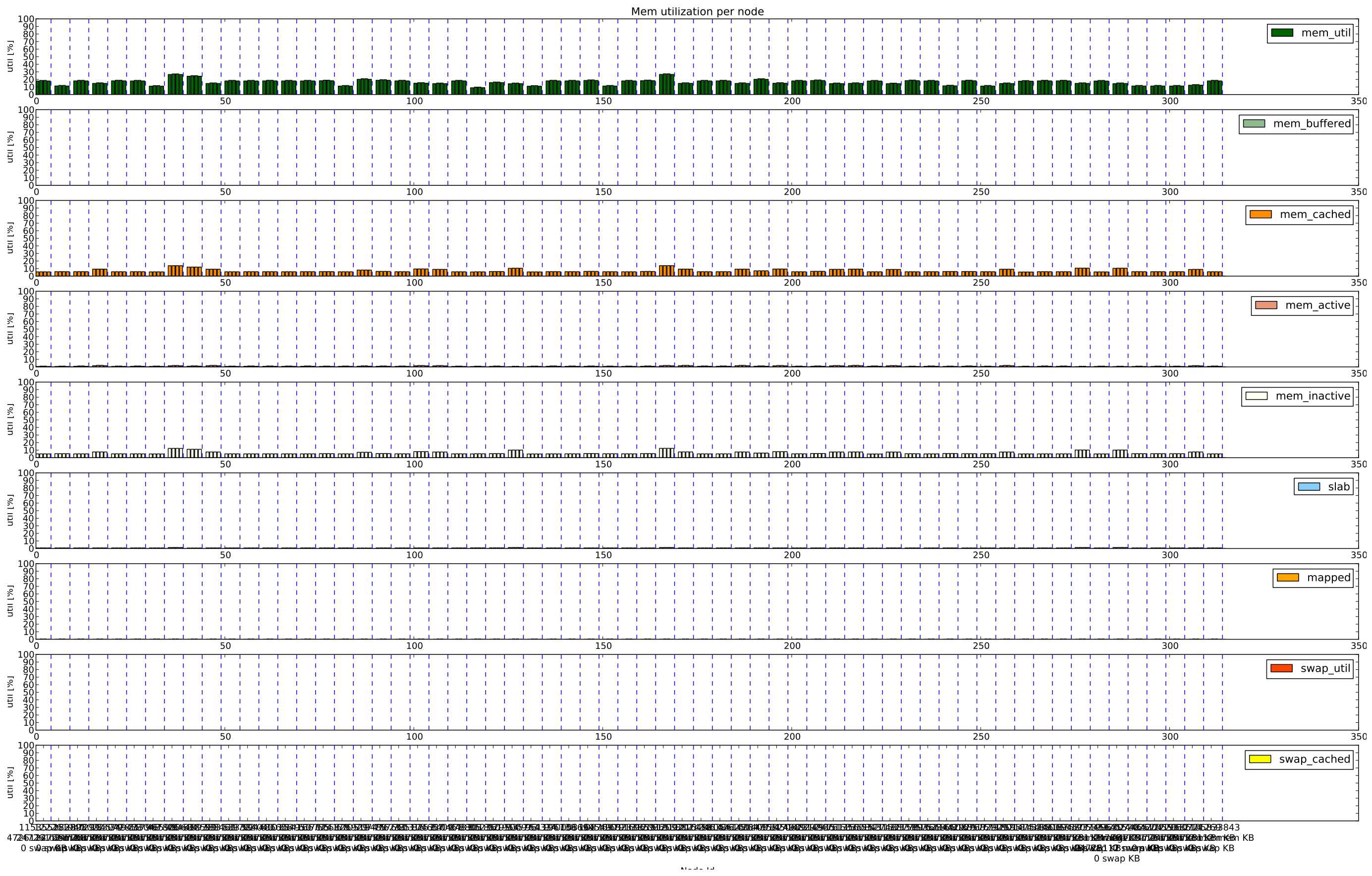


Fig. 7. Standard deviations over multiple LAMMPS runs on 192 GPGPUs for the melt baseline (8 runs), melt cpu/mem/net/nvml/nvidia-smi (5 runs), and melt cpu/mem/net/nvml (8 runs). The diagram data come from both data presented in Figure 6 and additional runs (i.e., not used for plotting in diagrams from Figure 6).

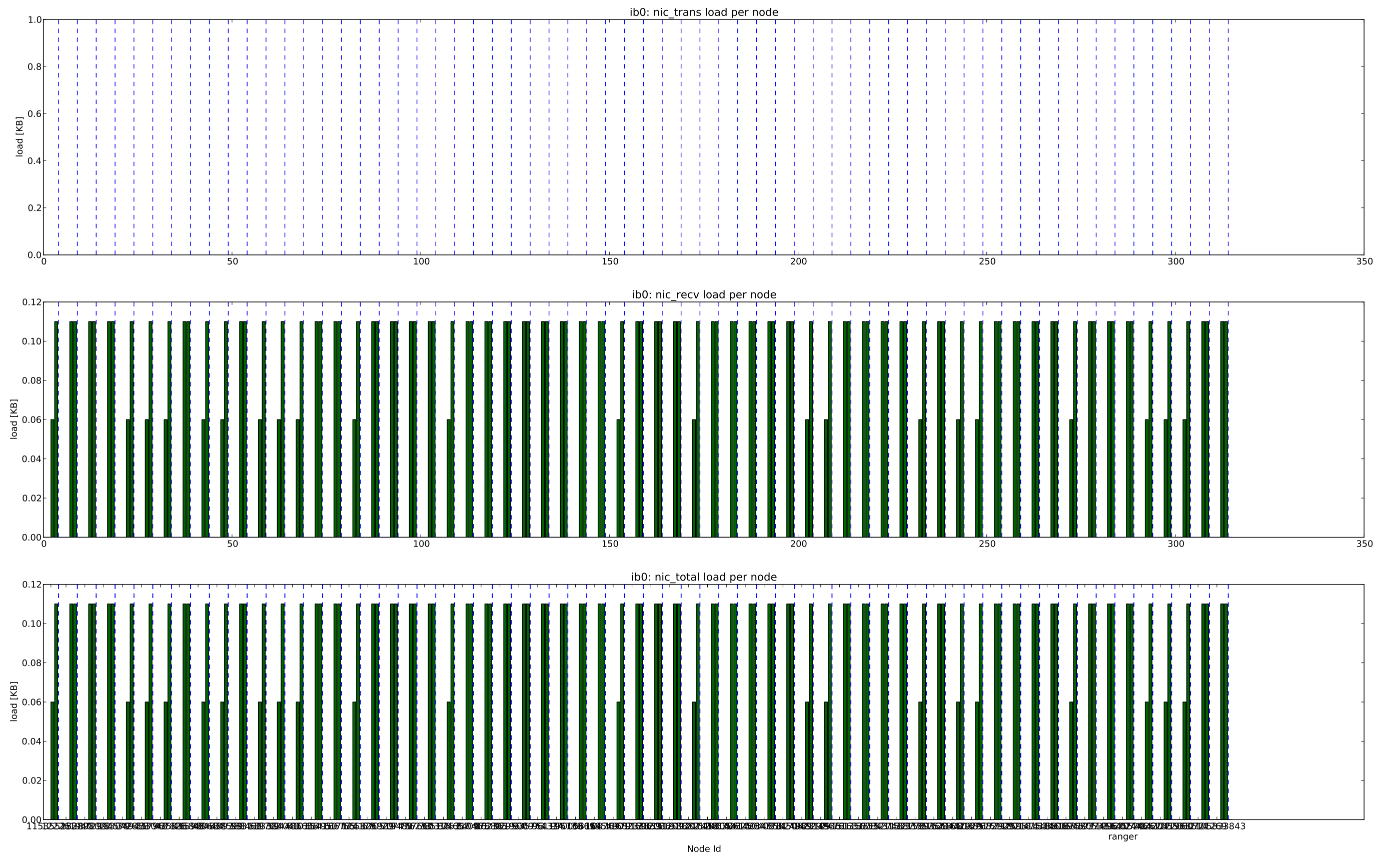


(a) Monitoring cores utilization in ClusterWatch (64 nodes, melt).

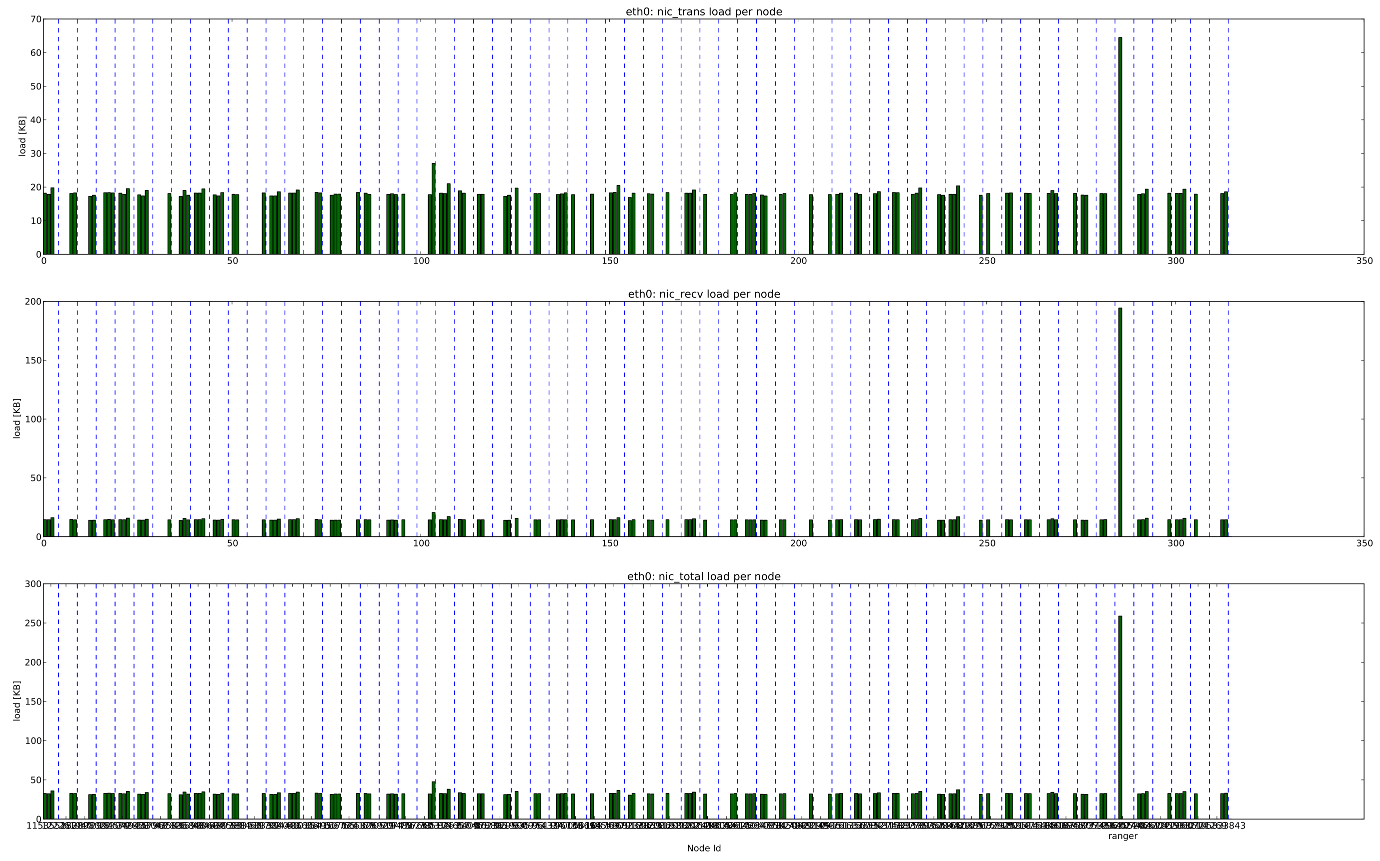


(b) Monitoring memory utilization in ClusterWatch (64 nodes, melt).

Fig. 8. The ClusterWatch experimental results with LAMMPS as a workload on 64 nodes (192 GPGPUs)–the cpu spy and memory spy on KIDS. There were 3 LAMMPS processes per node, each used one GPGPU; the LAMMPS processes were pinned to NUMA domain 1 (odd cores).

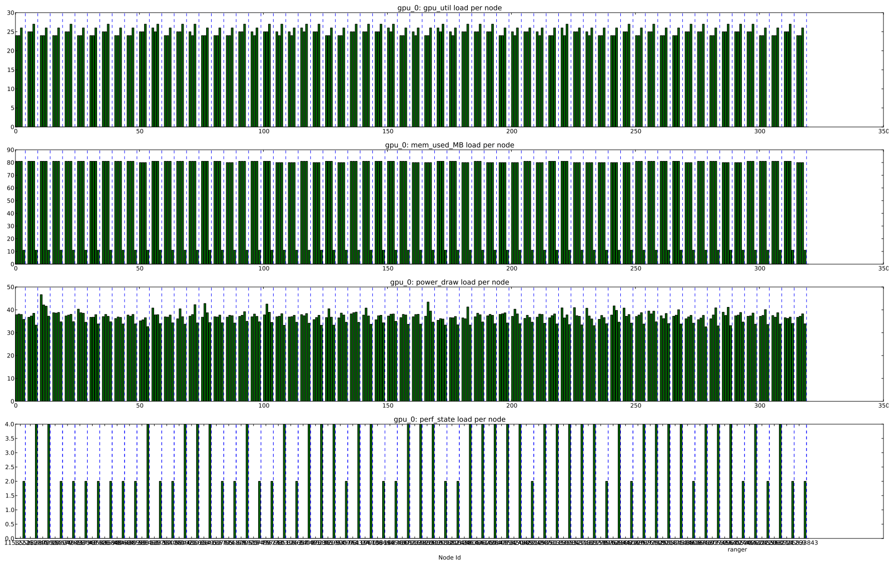


(a) Monitoring network load (ib0) in ClusterWatch (64 nodes, melt).

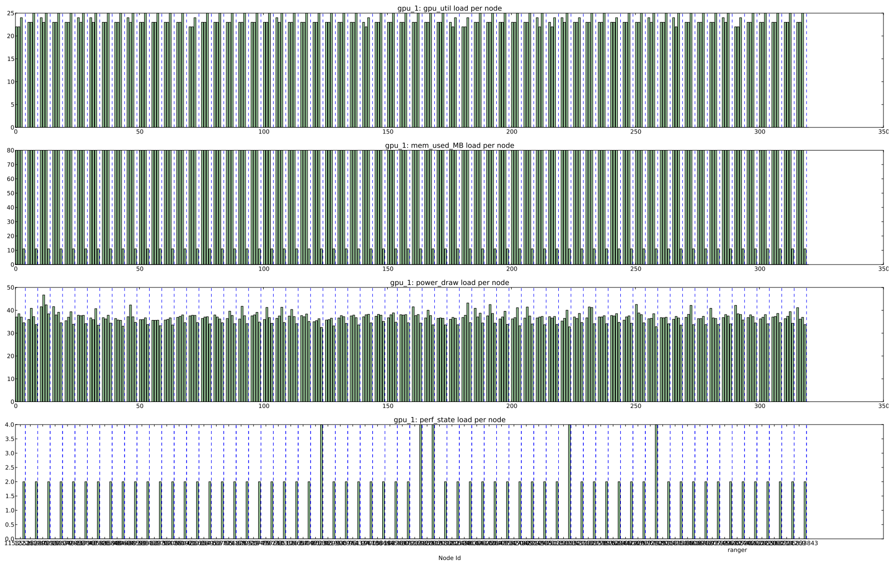


(b) Monitoring network load (eth0) in ClusterWatch (64 nodes, melt).

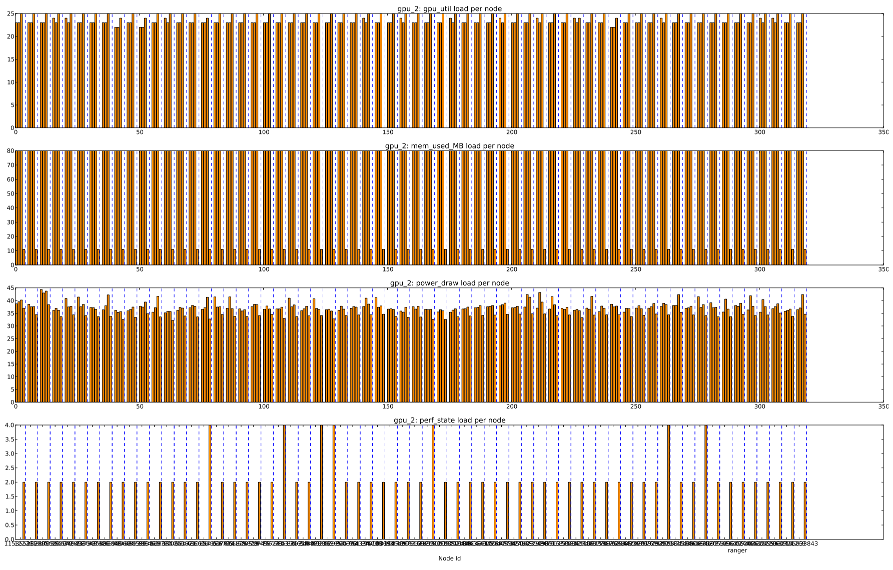
Fig. 9. The ClusterWatch experimental results with LAMMPS as a workload on 64 nodes (192 GPGPUs) on KIDS–the net spy.



(a) Monitoring GPGPU utilization (GPU0) in ClusterWatch (64 nodes, 192 GPGPUs, melt).



(b) Monitoring GPGPU utilization (GPU1) in ClusterWatch (64 nodes, 192 GPGPUs, melt).



(c) Monitoring GPGPU utilization (GPU2) in ClusterWatch (64 nodes, 192 GPGPUs, melt).

Fig. 10. The ClusterWatch experimental results with LAMMPS as a workload on 64 nodes (192 GPGPUs) on KIDS–the nvml spy.