2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing

Cross-Layer Self-Adaptive/Self-Aware System Software for Exascale Systems

Roberto Gioiosa, Gokcen Kestor, Darren J. Kerbyson, Adolfy Hoisie
High Performance Computing
Pacific Northwest National Laboratory
Richland, WA
{roberto.gioiosa, gokcen.kestor, darren.kerbyson, adolfy.hoisie}@pnnl.gov

***Abstract*—The extreme level of parallelism coupled with the limited available power budget expected in the exascale era brings unprecedented challenges that demand optimization of performance, power and resiliency in unison. Scalability on such systems is of paramount importance, while power and reliability issues may change the execution environment in which a parallel application runs. To solve these challenges exascale systems will require an introspective system software that combines system and application observations across all system stack layers with online feedback and adaptation mechanisms.**

In this paper we propose the design of a novel self-aware, self-adaptive system software in which a kernel-level Monitor, which continuously inspects the evolution of the target system through observation of Sensors, is combined with a user-level Controller, which reacts to changes in the execution environment, explores opportunities to increase performance, save power and adapts applications to new execution scenarios.

We show that the monitoring system accurately monitors the evolution of parallel applications with a runtime overhead below 1-2%. As a test case, we design and implement a runtime system that aims at optimizing application's performance and system power consumption on complex hierarchical architectures. Our results show that our adaptive system reaches 98% of performance efficiency of manually-tuned applications.

I. Introduction

Despite the high level of parallelism, the design of petascale systems was mainly driven by performance metrics, such as operations/sec or number of traversed edges/sec. Given the expected limitations on system power consumption (20-25 MW) [1], future systems will achieve exascale performance mainly through massive parallelism and processor and memory modules that operate at low or near-threshold voltage. The consequences of this new design will impact not only performance but also system power consumption and resiliency. These new and unique challenges of the exascale era need to be addressed by novel research at all level of the system stack and to be coordinated toward a holistic solution. Exascale systems will have to manage and coordinate multiple, possibly conflicting goals, such as performance, power efficiency and correctness, dynamically during the execution of a parallel application. Current petascale solutions typically address performance (e.g., HPCSched [2], Active Harmony [3]) or power consumption with fixed performance level (e.g., Adagio [4]). None seems to meet the expected level of concurrency and dynamism and to address changes in the execution environment caused by the expected power constraints or hardware faults. Several studies [5], [6] point out that the system software stack needs to be completely re-designed to meet the exascale requirements. These studies stress the need of an introspective system that combines performance, power and resilience observations, in-situ analysis and dynamic generation of applications' performance and power models across all system stack layers with online feedback and adaptation mechanisms. The overhead of such system must be minimal as not to impact on the application's performance during production runs while still collecting information that can be used both online to initiate corrective actions, and offline for debugging and performance analysis. Adaptive actions must be coordinated to avoid that independent operations take opposite directions and invalidate the overall benefits. This is particularly important, as performance, power and resilience are often conflicting with each other.

Recent DOE ASCR projects have started investigating possible solutions for exascale operating systems and adaptive runtimes [7], [8]. It seems reasonable to expect that exascale system will be required to dynamically and autonomously adapt to changing execution environments and/or applications' characteristics. In this paper we investigate a novel self-aware, self-adaptive system software design that meets the exascale requirements and has the potential of solving some of the unique challenges of the exascale era. Our self-aware/self-adaptive system software works as the closed-loop system: a Monitor continuously inspects the evolution of the target system (hardware and application) through observation of Sensors. The Monitor provides feedback information that is compared to reference values: If there is a significant difference between the *observed* and the *reference* values (*self-awareness*), the Monitor notifies a Controller which will apply corrective actions to stabilize the system (*self-adaptiveness*).

To obtain a comprehensive picture of an exascale system (hardware, software and application), the Monitor must collect information from a variety of sources, spanning from the hardware to the application, and dimensions (performance, power and reliability). The information gathered should also be correlated to the application's execution phases, which, in the exascale era, are expected to be much shorter execution than in current petascale applications. This is partially due to the asynchronous synchronization strategy used by the novel programming models proposed for exascale systems, such as Cilk [9], TBB [10], Charm++ [11] and Chapel [12], and partially because of the reduced per-process data set in strong scaling executions. Considering that some applications (e.g., NAMD [13]) already present phases in the order of micro-seconds, we anticipate that a monitoring frequency of 1kHz

1550-6533 2014

DOI 10.1109/SBAC-PAD.2014.29

as the minimum monitoring frequency suitable for exascale systems. Moreover, a high monitoring frequency guarantees that runtime issues are promptly detected and corrective actions are timely initiated. Finally, in order to adapt the system to new execution scenarios, this information must be available in-band and the system should have access to hardware and software actuators. Current performance analysis tools used for postmortem analysis do not seem adequate to this task, as they often collect information related to only one dimension and/or do not provide feedback to the other software components. Reliability, Availability and Serviceability (RAS) systems, on the other hand, collect data out-of-band, hence the information cannot be effectively used to drive the system software.

We propose an approach in which frequent monitoring activities (information gathering and comparison with reference values) are performed at kernel level while adaptive actions are initiated by the user-level runtime. More in details, the Monitor accurately and precisely collects information from the hardware sensors and event counters, kernel, resource and process state variables (*observed values*). The Controller is responsible for setting the desired performance, power and resiliency policies (*reference values*), processing the information provided by Monitor, and eventually reacting to the change in the execution environment to meet the desired level of system efficiency and resilience. Monitor and Controller communicate through a well-defined interface that effectively decouples the low-level system details from the high-level programming model runtime implementation. Our interface combines performance, power and resiliency information in one single interface, exposing user-level runtimes and applications a comprehensive view of the system and application execution. We also present the design and implementation of our self-aware/self-adaptive system in which the Monitor and the interface are implemented as Linux kernel modules and Controller is implemented as part of the MPI and OpenMP runtimes. Our decoupled approach allows the development of Controllers as part of other programming model runtimes, external threads or application modules without the need of modifying the Monitor or the interface.

We perform our evaluation on a 32-core AMD Interlagos system with a variety of benchmarks and applications from the ASCR Exascale Co-Design centers [14] and DOE Office of Science. First, we show that the overhead of our Monitor is negligible (within 1-2%) and indistinguishable from the normal execution time variation of a parallel application running on a state-of-the-art compute node, even when monitoring the system at 1kHz. As a test case, we present self-adaptive Controller that aims at optimizing performance and system power consumption by dynamically selecting the configuration of OpenMP threads and MPI processes that yield the highest performance with an acceptable power consumption. Our system achieves 98% of the performance efficiency of a manually tuned version of the tested applications.

In summary, this paper makes the following contributions:

- We present a system monitor that introduces minimal runtime overhead on HPC applications.
- We implement a Controller that automatically and dynamically adjust the number of OpenMP threads and MPI tasks to achieve maximum performance with limited power consumption.
- We show that a co-designed, cross-layer approach provides performance efficiency improvements and control with minimal runtime overheads.

The rest of this paper is organized as follows: Section II describes background and related work; Section III details the design and implementation of our system; Section IV presents our experimental results; Section V concludes this work.

II. Background and Related Work

Self-aware/self-adaptive systems [15] have been studied in the past in various contexts, from HPC [2], [4] to real-time embedded systems to autonomic and auto-tuning systems [3]. Most of the proposed solutions tackle the problem at a specific level (hardware, operating system, runtime, compiler) or focus on the optimization of a single metric, be it performance, system utilization or power. While these systems worked well in the petascale era, the unique exascale challenges will impose a tighter cooperation among the system layers to achieve the desired level of performance, power consumption and correctness. This level of interaction and co-design is, perhaps, one of the major points that marks the distinction between petascale and exascale systems. Moreover, exascale systems will require the optimization of performance, power and resilience in concert and in changing execution environments. Systems that target performance [2], [3] or power [4] optimization need to be extended to include the other dimensions.

An holistic approach in which all layers of the system stack cooperate toward the optimization of (possibly conflicting) goals has the potential of satisfying exascale requirements and providing sustainable performance and programmability [1], [6]. This direction is currently pursued by several DOE ASCR exascale projects, such as Argo [7] and Hobbes [8]. In particular, Huck et al. [16] describe the needs of an online Autonomic Performance Environment for eXascale systems (APEX) in the XPRESS project for the ParalleX execution model. The effectiveness of such online introspective system requires 1) a complete set of information gathered from all system layers, from hardware and OS to application, that provides a comprehensive view of the application and system status, and 2) a minimal runtime overhead that introduces extremely low system noise [17]–[19] and does not affect the scalability of applications. In this respect, monitoring systems based on dynamic instrumentation, such as Dyninst and Adaptive Harmony [3], [20], might not be adequate to be used as online introspective exascale monitor because of the excessive binary instrumentation overhead [21]. Instead, gathering information at the proper level and passing it to adjacent levels on-demand according to a publish-subscribe model, has the advantage of drastically reducing runtime overhead while still providing a comprehensive view of the system and application status.

In the context of HPC, several proposals looked at self-adaptive algorithms to minimize power consumption or improve performance autonomously. Adagio [4] extends the MPI runtime to automatically determine the application's critical path and reduce the frequency of non-critical *tasks*, thereby reducing power consumption while maintaining constant performance. CPU MISER [22] attempts to predict performance of

parallel application with different processor frequencies. CPU MISER consists of three components: a performance monitor that periodically collects performance information (only four performance counters are collected); a workload predictor and a DVFS scheduler. Similarly to Adagio, CPU MISER attempts to determine the best CPU frequency to save power without performance degradation (i.e., the performance metrics is fixed). However, results show performance degradation between 3% and 10% with energy savings between -3% and 20% for NPB applications. As opposed to Adagio and CPU MISER, in which one of the metrics (performance) is fixed while the other (power) can vary, our solution maximizes performance with acceptable power consumption. Adagio has been tested with single-core compute nodes while we tested our solution on complex, hierarchical multi-core systems. HPCSched [2] is a scheduler that dynamically detects load imbalance in scientific applications and automatically biases the allocation of hardware resources in favor of the most computing intensive tasks. The scheduler is completely implemented at kernel level, which reduces the runtime overhead and allows direct access to the processor resource allocator. On the other hand, HPCSched does not have access to application and programming model runtime information and does not take power consumption into account. Thermal-aware load balancing [23], [24] has been proposed as a way to reduce the number of soft-errors in Charm++ applications. While temperature affects the static leakage power, these works do not directly consider dynamic power and performance. An exascale solution requires a combination of these solutions in the same framework to address performance, power in unison.

In the context of data centers/multi-programmed workloads, the SElf-awarE Computing model (SEEC) [25] monitors and allocates resources in multi-core systems to independent application. SEEC uses Application Heartbeats [26], an API that allows the user to specify a desired level of QoS in terms of *heartbeats* between any two points of a program. With SEEC the user has to modify the program control state variables (knobs) and insert Application Heartbeats API calls manually. A following work [27] proposes PowerDial, a system that automatically identifies control variables and insert Application Heartbeats API. PowerDial adapts the running applications so that their execution respect the established QoS within a given power budget. PowerDial profiles the application to identify and calibrate control variables, establish their influence, and insert the application's knobs into the source code. This process requires multiple runs of the same application with different set of application-specific parameters. SEEC and PowerDial focus on shared memory applications, whereas our system focus on distributed/shared memory applications. Moreover, HPC systems usually runs only one application on a compute node, thus shifting cores between applications, as in the data-center workloads examined by SEEC, would not be an effective actuator.

III. Design and Implementation

The design of the proposed self-aware/self-adaptive system follows the classical closed-loop flow used in control theory and depicted in Figure 1. In the Figure, *System* represents the monitored exascale system and running application; The *Sensors* are all sources of information that can be observed while the application is running. Examples include processor and

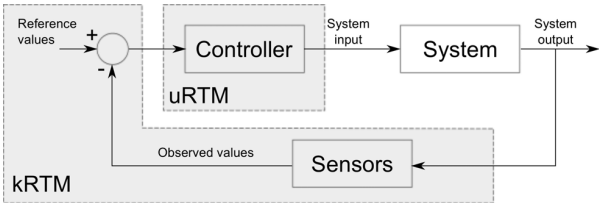


Fig. 1: Closed-loop control system.

network performance counters, power and temperature sensors, resource usage counters (e.g., PURR register in IBM POWER processors [28]), kernel and runtime state variables and algorithmic information provided by the application. During the execution of the applications, the Sensors are continuously inspected to monitor the application's evolution and detect changes in the execution environment or the application's characteristics. This information (*observed values*) is compared to the *reference values* provided as input. We envision that reference values could be obtained from previous runs of the application, static or dynamic analytical models [29] or simply set by the programmer as QoS requirements. If the values observed through the sensors do not match the reference values, the *Controller* takes actions that aim at restoring the desired QoS and stabilizing the system. For example, if load imbalance is detected as the result of lowering the frequency of a core in order to maintain the system power consumption under the defined power budget, the Controller may re-distribute internal resources and/or re-schedule threads to reduce the load imbalance [30].

To achieve the desired accuracy (high monitoring frequency) with minimal impact on the application's performance (low-overhead), we propose a decoupled solution in which high-frequency, low-overhead operations are performed at kernel-level, while low-frequency operations that may eventually introduce higher overhead are performed at user-level on-demand. Figure 2 depicts the software structure of the proposed self-aware/self-adaptive system:

> **kRTM** (Monitor) This low-overhead, kernel-level component continuously inspects the system status and notifies the Controller if a change in the execution environment or a fault is detected.
>
> **SMI** This interface abstracts the low-level architectural and operating system details to the user-level runtime library and the programmer.
>
> **uRTM** (Controller) This component is implemented in the user-level runtime and is responsible to set the performance, power and resilience policies and to react to changes in the execution environment.

This decoupled approach provides several advantages over a single-layer implementation: First, there is no additional context-switch overhead to access architectural or OS information. Second, it provides access to information that is not available at user-level, such as virtual-to-physical mappings, page faults or the dynamic scheduling priority of a task. This information could be useful to disambiguate false positives (e.g., performance degradation caused by a transient event) from real problems. Finally, the control mechanism that decides which actions need to be taken in response to an event

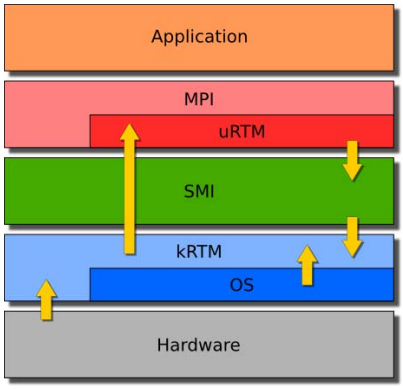


Fig. 2: System monitor software architecture.

is still implemented in user-mode, for example, as part of the programming model runtime. uRTM has, thus, access to the messaging and programming model data structures commonly managed by runtime systems. Figure 1 also shows how the various components in our system monitor map to a classical closed-loop system. Note that the "common case" in which the application performs as expected and no changes occur in the execution environment is completely handled at kernel-level with minimal overhead. The following Sections describe in details the various components.

A. Monitor

In order to obtain a comprehensive and accurate view of the application and system status, a self-aware system must collect information from a variety of sources, including hardware sensors and counters, OS state variables and statistics, and runtime statistics. Access to this information is usually either restricted to the stack level that generates the information itself or is considerably more expensive to access from other levels. For example, Intel SandyBridge power sensors can be quickly accessed at hardware level, can be accessed with contained overhead from the OS kernel through Machine State Registers (MSRs) or, with a higher overhead, from the user-level package RAPL [31] (which also requires superuser privileges). In our design the system Monitor (kRTM) is implemented at kernel level, mainly as a Linux kernel module. Although the Monitor could also be implemented as a user-level runtime library, we chose to implement the Monitor at kernel level because this design provides several important advantages. First, the Monitor can access OS- and resource-state variables that are not exported at user-level, such as the virtual-to-physical mapping of a page. Second, access to restricted hardware information, such as performance counters or power and temperature sensors, is usually faster than accessing from a user-level library. For example, reading performance counters through PAPI/PfMon libraries [32] requires issuing six systems calls (1 `read()` and 5 `ioctl()` to reset the counters) in PAPI 5.2.0.[1] Third, there is no dependency on a particular

[1]Although both Intel and AMD x86 architectures support `rdpmc` to read performance counters directly from user-level, on current Linux systems, `rdpmc` requires accessing information stored in a memory-mapped page that may not be mapped in the TLB. To avoid the overhead of a TLB miss (about 2000 cycles [33]), currently PAPI developers prefer to use the `read()` system call, as the overhead of `rdpmc` is higher. Moreover, system calls must still be issued to reset the performance counter values (`ioctl()`).

TABLE I: Performance counters used to model per-core power consumption for AMD Opteron 6227.

Counter code	Description
LLC	Last Level Cache Misses
INST	Number of Retired Instructions
STALL	Number of Back-end Stalled Cycles
FPs	Number of Retired Floating Point Ops

performance analysis or programming model runtime library. As we will explain in the next section, the System Monitor Interface simplifies the use of different implementations of Controllers and the use of the Monitor in conjunction with performance analysis tools. Finally, a kernel-level monitor can be extended to collect information about the health of the system and detect transient/permanent errors (such information may not be exposed to the user-level runtime).

While hardware performance counters are widely available on modern processor architectures, power and temperature sensors have only recently become exposed to software. Intel introduced power sensors with SandyBridge [34]: the processor collects power information for the entire socket at 1kHz resolution and stores it into a Model-Specific Register (MSR). Intel RAPL package can be used to access the MSR but it requires superuser privilege, which is not usually granted to user software on production systems. AMD Bulldozer [35] provides similar package-level sensors that can be accessed through MSR register or a `sysfs` interface. These power sensors, however, only provide the power consumption of the entire processor chip which might be be too coarse-grained considering the cardinality of modern multi-core processors. To overcome these limitations, we use proxy power sensors that estimate dynamic per-core power consumption based on core activity and regression models [36]. Our model has been validated in our previous work [37], where we show a good correlation with the real power consumption. Table I shows the performance counters used to obtain the power consumption of each core. Current systems do not usually provide in-band resiliency information: most of the Reliability, Availability and Serviceability (RAS) systems only provide out-of-band information that are difficult to use in an online feedback system. We envision that a fault model similar to the one we use to estimate per-core power consumption could be used to emulate processor fault [38] but we leave this as future work.

The monitoring frequency is an important parameter of a closed-loop system: on one side, a high monitoring frequency reduces the response time and thus allows the system to reacts quickly to changes in the execution environment. On the other side, a high monitoring frequency may introduce excessive run-time overhead. The monitoring frequency should be adequate for the desired average system response time and comparable to the expected frequency of changes in the execution environment and in the application's behavior. As several applications, such as NAMD [13], already shown micro-second computation phase granularity, we expect 1kHz to be the minimum monitoring frequency. The kRTM Linux kernel module takes as input parameter the monitoring frequency f, which is used to set up a high resolution kernel timer during the module initialization. For performance reasons, if $f = \texttt{HZ}$, where `HZ` is the kernel tick frequency, the monitoring activity is triggered by the normal timer interrupt rather than by a specific high resolution

timer. Once loaded in memory, kRTM constantly monitors the system execution. The actual list of parameters depends on the system and can easily and dynamically be adapted to the user-level requirements. However, the uRTM is notified of an unexpected event only if it has previously registered a process/thread to receive the notification (publish/subscribe model) and at least one of the observed value o_i is out of the expected range. There can be a list of processes for each CPU registered to receive notifications of events from kRTM. This design choice provides maximum flexibility for the design of uRTM: for example, uRTM can be implemented as a dynamic library loaded or as an independent monitor process.

B. System Monitor Interface

Each system exposes state information through architecture-specific interfaces, e.g., state or performance counter registers or processor's sensors. Several efforts attempted to rise the level of abstraction through a high-level interface that abstracts the architecture-specific details. For example, PAPI [32] provides an architecture-independent view of the hardware performance counters while RAPL [31] hides the low-level details of interfacing MSRs. Most of these interfaces, however, only target a specific dimension. We believe that there is need of a common and comprehensive interface where performance, power and resilience information are exported to user-level runtimes and applications. The System Monitor Interface (SMI) abstracts the architecture-dependent details through a well-defined interface between hardware/OS and runtime/application (Figure 2), effectively decoupling kRTM and uRTM.

SMI is implemented as a standard `sysfs` pseudo-file system, which is a common and architecture-independent interface. The `/sys/smi` entry is created when our Linux kernel module is loaded in memory. SMI has the following structure: each hardware resource category is described by a separate directory (`cpu`, `memory`, `network`). Within each category, the system features the available resource, such as multiple cores, memory modules or network cards. Each hardware resource, in turn, is associated with a set of parameters that describe the resource usage and characteristics. For example, each core in the system is associated with three classes of parameters: `performance`, `power` and `resilience`. Although current processor architectures do not usually expose power or resiliency information to the user or the OS, we use the per-core proxy power sensor described in Section III-A. For architectures that provide accurate per-core power sensors, we can simply replace the proxy sensor with the real one, without any modification to SMI, uRTM or the application. The `sysfs` files that represent a parameter (e.g., `flops`, `power`) can be used in two ways: the observed values o_i can be obtained by reading each `sysfs` file; the reference values r_i can be set by writing to the proper `sysfs` file ($r_i = 0$ resets the reference value). For performance reason, SMI provides a comprehensive `sysfs` file (`stats`) to collect all information with one single read operation. User or user runtimes can register the controlling process by writing its `PID` to `/smi/cpu/cpuX/pid` (`-PID` de-registers the process). Note that the entity that will receive the signal can be application's process/thread, a specific application process, or an external process.

C. Controller

HPC systems are used to solve complex scientific problems in a "reasonable time". Slowing down the system to unconditionally save power and failing to provide the solution within the expected time frame defeats the purpose of using large machines to solve scientific problems. In an extreme case, running an application in single thread mode would probably provide the lower power consumption but would not provide the desired level of performance. On the other hand, using all available hardware resources for applications that do not scale would not provide higher performance (possibly performance degradation) but would increase the amount of wasted power. Since performance and power affect each other, it seems clear that they need to be addressed at the same time.

Adapting to changes in the execution environment, whether because of a fault or of power considerations, may require to reschedule tasks, repartition data and/or re-allocate resources. With our decoupled approach we can implement adaptive algorithms in different ways, such as an external thread, a stand-alone dynamic library or a programming model runtime module. Since several important information on data structure allocation, level of parallelism and synchronization are available in the programming model runtime, in this work, we present an implementation as part of the OpenMP and MPI runtimes. However, similar adaptive systems can be implemented as part of other programming models without the need of modifying the underlying monitor (kRTM) or interface (SMI). Our Controller follows an economic model: increasing the number of active computing elements (cores or hardware threads) requires an investment (power) that may provide a profit (performance). uRTM seeks the configuration with the highest return on investment: uRTM increases the number of OpenMP threads if the benefits (performance) are higher than the investments (power), i.e., highest performance per watt invested. uRTM is implemented as a dynamic library that can be pre-loaded before executing the application. Once loaded in memory, uRTM intercepts calls to the MPI functions to initialize/finalize the Controller environment. Initial reference values can be specified through a set of environment variables. We assume that the number of MPI processes is statically determined at the beginning of the application and that it does not change during the execution. The number of OpenMP threads per MPI process (*actuator*), instead, can be modified during the execution of the application to increase performance/reduce power. Performance is assessed in terms of floating point operations/second and active power consumption is obtained from the per-core proxy power sensors discussed in Section III-A. Note that there are ways to reconfigure malleable MPI applications to use a different set of hardware resources. Ribeiro et al. [39] propose a low-cost algorithm to transform traditional MPI applications into malleable versions that are capable of adjusting the number of running processes. This mechanism could be used to enhance our Controller.

The uRTM task scheduler assigns each MPI process a group of CPUs. The size of the group depends on the number of available CPUs (32 in our system) and the number of MPI processes. Computing elements are equally distributed among the MPI tasks. During the execution of the application, uRTM gradually increases the number of OpenMP threads/MPI process and checks the application's performance and active

power consumption. If the application's performance has increased more than the power consumption, uRTM moves the application to the new configuration. If, instead, power consumption increases more than the performance improvement, uRTM rolls back to the previous configuration. Note that, the new configuration is active from the next OpenMP parallel phase, i.e., the increased/decreased number of OpenMP threads will only be used once an MPI process start a new OpenMP parallel region. This process repeats until either the optimal configuration has been found or all available computing elements have been used. Once an optimal configuration has been reached, uRTM sets the reference values (FLOPS, r_{flops}, and power consumption, r_{watts}) of each MPI task and lets kRTM monitor the evolution of the application. At this point, uRTM does not perform any other action unless a change in the execution environment occurs, in which case kRTM notifies uRTM. This occurs if the observed performance is below the reference value ($o_{flops} < r_{flops}$) or if the power consumption exceeds the expectations ($o_{watt} > r_{watt}$). Upon receiving a message from kRTM, the user-level controller has to determine what has caused the change in the execution environment and take the proper actions. Currently, we implemented several scenarios: If uRTM is notified because of a loss of performance or an excess of power consumption and it can determine that the issue is temporal, our runtime takes no further action (*transient event*). For example, if an external process or a kernel thread preempts an application's threads ($o_{preemption} > 0$), uRTM only records the event and the state of the system for offline analysis and debugging. If a series of transient events occur within a given time frame (`transient_period`), uRTM assumes that the application has moved to a new phase and restarts the process of seeking the optimal configuration (*transition event*). Finally, if too many transition events occur in a given period of time (`transition_period`), uRTM assumes that there is a permanent problem with the system or the application. This may, in fact, be a hardware problem or may be related to the fact that the application performance or power model dynamically built by uRTM does not represent the application behavior. In both cases, uRTM records the events and the observed values obtained from SMI and stops searching for the optimal configuration to avoid unnecessary oscillations.

IV. Experimental Results

In this section we present the evaluation of our self-aware/self-adaptive system. We performed our experiment on a dual-socket AMD Opteron 6272 (Interlagos) [35] compute node: each socket is a Clustered Multi-Threaded (CMT) chip divided into two Multi-Chip Modules (MCMs), and each MCM consists of four Clustered Integrated Cores, a shared 8MB L3 cache, and a memory controller. A Clustered Integrated Core couples two "conventional" x86 out-of-order Integer Cores together; the two cores share fetch and decode units, the 64KB L1I and the 2MB L2 cache (the 64KB L1D is private to a single Integer Core), and the floating point unit. The system is equipped with 64GB main memory organized in 4 NUMA domains (one for each MCM). All applications are compiled with `gcc 4.8.2` for 64-bit architectures with all common optimization and linked to `OpenMPI 1.7.3`. NPB applications use the class C input set, while *AMG*, *Sphot*, *GTC*, and *NEKBone* use their respective reference input sets. The system runs Linux 3.12.0.

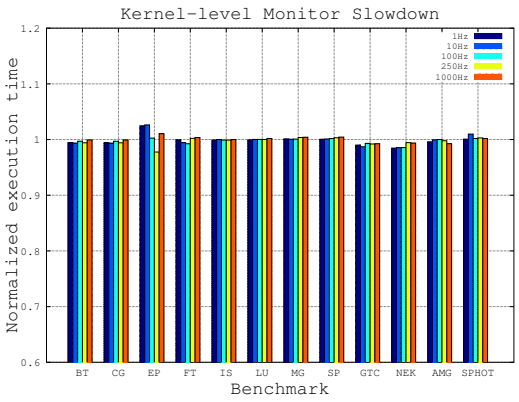
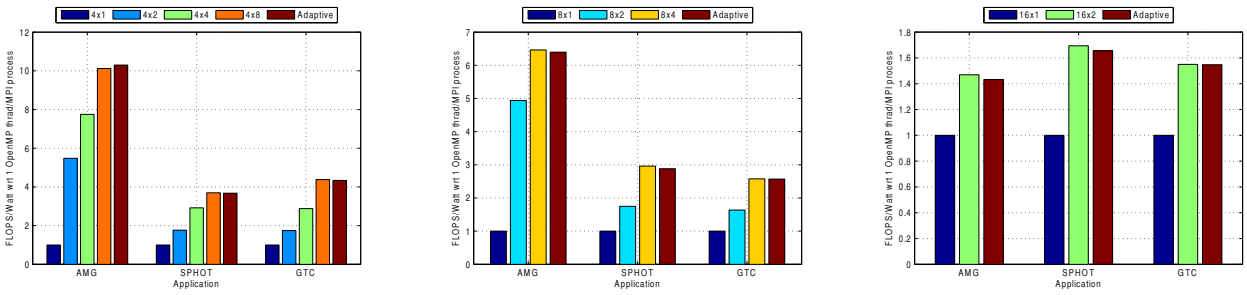


Fig. 3: kRTM runtime slowdown with different frequencies.

Runtime Overhead Analysis: Minimal runtime overhead is a critical aspect of a monitoring system for self-aware/self-adaptive system software for exascale supercomputers. Figure 3 reports the execution time of each application monitored by kRTM with a given frequency f over the standard execution of the application without kRTM (i.e., lower is better). We use 16 MPI tasks (we do not use the secondary core in each CIC to minimize execution time variation) and bind each MPI task to a specific core through the `--bind-to-core` option of `mpiexec` to avoid process migration. In this experiment kRTM does not notify any user-level runtime system but continuously monitors the evolution of the application and the system by collecting information from hardware and software performance counters, the per-core proxy power sensors and kernel state variables. More specifically, kRTM collects the following information: number of instructions, number of cycles, number of FP operations, number of last-level cache misses, number of load instructions (hardware performance counters), per-core power consumption (proxy power sensor), number of process preemptions, number of process migrations, number of page faults, number of page migrations (kernel information). The graph shows that kRTM runtime overhead is negligible for all tested frequencies across all applications and indistinguishable from the normal execution time variation (1-2%) that applications running on real systems experience from run to run. In some cases (e.g., *GTC* and *NEKBone*) executing the application with kRTM provides "better performance". We believe that this is caused by the normal execution time variation, hence we conclude that kRTM does not introduce any measurable performance degradation for these applications as well. *EP* shows execution time variation in both directions: the runtime overhead with $f = 1Hz$ and $f = 10Hz$ appears to be larger than the runtime overhead with $f > 10Hz$, while the application appear to be faster than the standard case when running on kRTM with $f = 250Hz$. As for the case of *GTC* and *NEKBone*, we believe these variation are well within the expected execution time variation on a real system.

Adaptive OpenMP Runtime: As described above, the AMD Opteron 6272 is a complex and hierarchical system where hardware resources are shared at different levels. For example, the L2 cache is shared by the Integrated Cores in the same cluster, the L3 cache and the memory controller are shared by the four Clustered Integrated Cores (eight Integrated Cores) in the same MCM. Mapping a parallel application to such a



(a) 4 MPI processes (b) 8 MPI processes (c) 16 MPI processes

Fig. 4: Performance efficiency of MPI+OpenMP applications. Values are normalized to configurations with 1 OpenMP thread.

system in a way to minimize resource contention, maximize performance and reduce power consumption is a complicated task that is usually performed manually by the programmer and needs to be performed again when the input set changes. In these experiments uRTM dynamically adapts the application to the system's characteristics and explores opportunities to increase performance while maintaining an acceptable level of power consumption, by varying the number of OpenMP threads per MPI task.[2] Once the optimal configuration has been identified, uRTM sets performance and power reference parameters and registers to be notified by kRTM if the execution does not follow the requested performance or power levels. In these experiments we use 1kHz frequency.

We report the results of our experiments in terms of FLOPS/Watt ratio (performance efficiency), where the power consumption is the active power consumption of the active cores estimated through the proxy per-core power sensors described in Section III-A. Figure 4 shows the power efficiency *AMG*, *SPHOT* and *GTC* with various OpenMP+MPI configurations. The values in the graphs are normalized to the performance efficiency of the application running with 1 OpenMP thread/MPI process (initial configuration). The results are shown for different number of MPI processes, 4, 8 and 16, and each MPI process is assigned with an equal number of CPUs that it can use to run additional OpenMP threads. We present the results for static configurations of X MPI processes with Y OpenMP threads ("XxY") and our adaptive system ("Adaptive"). A configuration A with a OpenMP threads/MPI task is more efficient than a configuration B with $b < a$ OpenMP threads/MPI task if the performance improvement (benefits) of A over B is higher the extra power consumption (investment) required to run additional OpenMP threads in A. We notice that, in general, increasing the number of OpenMP threads/MPI process provides a positive return of investment (performance improvements are higher than extra power consumption). For example, Figure 4a shows that *AMG* achieves 10x higher performance efficiency with 8 OpenMP threads/MPI process (4x8) with respect to 1 OpenMP thread/MPI process (4x1). The graphs also show that our adaptive solution is able to reach the best configuration found by manually setting the number of OpenMP threads/MPI process (upper bound). Interestingly, for *GTC* uRTM is able to match the performance efficiency of the best static configuration for all test (except a small variation with 4 MPI processes). For *AMG*, instead, uRTM needs some time to converge to the optimal configuration, especially with 8 and 16 MPI processes. We inspected the code and noticed that the first iterations of the application are longer than the following ones. This means that, although uRTM attempts to increase the number of OpenMP threads, the effects on the application's performance will only be seen in the next iteration (i.e., the next OpenMP parallel loop). At this point a part of the application has been already executed with a sub-optimal number of OpenMP threads.

In conclusion, our results highlight that a cross-layer self-aware/self-adaptive solution has the can achieve high performance efficiency with minimal runtime overhead.

[2]With MPI applications data structures are statically partitioned, thus changing the number of MPI processes at runtime to adapt the application to the underlying architecture usually requires data re-partitioning or even to recompile the application, as it is the case for the NPB benchmarks.

V. Conclusions

To achieve the desired level of performance, power efficiency and correctness, exascale system will have to manage multiple, conflicting goals and optimize performance, power and resiliency in concert. Exascale systems will require an autonomic and holistic solution where all levels of the system stack cooperate towards a common goal. Such an holistic solution will require fundamental new research at all level of the system stack and, in particular, for the system software.

In this paper we propose a new self-aware/self-adaptive system software design that accurately monitors the execution of the application, autonomously explores opportunities to contemporarily increase performance and power efficiency and react to changes in the execution environment. Our approach combines introspective performance, power and resilience observations and in-situ analysis and adaptive algorithms based on feedback. We achieve the desired level of accuracy with minimal runtime overhead (1-2%) by de-coupling our solution into a kernel-level, low-latency system monitor (kRTM) and an adaptive runtime system tightly integrated with the programming model runtime (uRTM). We show that our approach is capable of achieving 98% of the performance and power efficiency of manually-tuned applications' configurations.

ACKNOWLEDGMENTS

This work was supported by the DOE Office of Science, Advanced Scientific Computing Research, under award number 62855 "Beyond the Standard Model – Towards an Integrated Modeling Methodology for Performance and Power"; Program Manager Karen Pao.

REFERENCES

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. A. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA IPTO, Tech. Rep. DARPA-2008-13, September 2008.

[2] C. Boneti, R. Gioiosa, F. Cazorla, and M. Valero, "A dynamic scheduler for balancing HPC applications," in *SC '08: Proc. of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[3] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: towards automated performance tuning," in *SC*, 2002, pp. 1–11.

[4] B. Rountree, D. K. Lowenthal., B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: Making DVS practical for complex HPC applications," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. New York, NY, USA: ACM, 2009.

[5] P. Beckman, R. Brightwell, B. de Supinski, M. Gokhale, S. Hofmey, S. Krishnamoorthy, M. Land, B. Maccabe, J. Shalf, and M. Snir, "Exascale operating system and runtime software report," U.S. Departement of Energy, Tech. Rep., December 2012.

[6] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt, "Hobbes: composition and virtualization as the foundations of an extreme-scale os/r," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, 2013.

[7] "Argo: An exascale operating system." [Online]. Available: http://www.mcs.anl.gov/project/argo-exascale-operating-system

[8] "Hobbes: composition and virtualization as the foundations of an extreme-scale OS/R." [Online]. Available: http://xstack.sandia.gov/hobbes/

[9] G. Long, D. Fan, and J. Zhang, "Architectural support for Cilk computations on many-core architectures," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09, 2009.

[10] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[11] L. Kale, "Charm++," in *Encyclopedia of Parallel Computing (to appear)*, D. Padua, Ed. Springer Verlag, 2011.

[12] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, Aug. 2007.

[13] "NAMD scalable molecular dynamics," http://www.ks.uiuc.edu/Research/namd/.

[14] Advanced Scientific Computing Research (ASCR), "Scientific discovery through advanced computing (SciDAC) Co-Design," http://science.energy.gov/ascr/research/scidac/co-design/.

[15] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, May 2009.

[16] K. Huck, S. Shende, A. Malony, H. Kaiser, A. Porterfield, R. Fowler, and R. Brightwell, "An early prototype of an autonomic performance environment for exascale," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '13, 2013.

[17] K. B. Ferreira., P. G. Bridges, and R. Brightwell, "Characterizing application sensitivity to OS interference using kernel-level noise injection," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[18] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *The 4th IEEE Int. Symp. on Signal Processing and Information Technology (ISSPIT 2004)*, Rome, Italy, December 2004.

[19] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of os noise," in *25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[20] B. P. Miller and A. R. Bernat, "Anywhere, any time binary instrumentation," in *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Hungary, September 2011.

[21] S. Pakin and P. McCormick, "Hardware-Independent Application Characterization," in *2013 IEEE International Symposium on Workload Characterization*, Portland, Oregon, September 2013.

[22] R. Ge, X. Feng, W. Feng, and K. W. Cameron, "CPU MISER: A performance-directed, run-time system for power-aware clusters," in *International Conference on Parallel Processing*, XiAn, China, 2007.

[23] O. Sarood and L. V. Kale, "A'cool'load balancer for parallel applications," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[24] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé, "Thermal aware automated load balancing for hpc applications," in *the 2013 IEEE International Conference on Cluster Computing*, 2013.

[25] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva, "Controlling software applications via resource allocation within the heartbeats framework," in *CDC*, 2010.

[26] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments," in *Proceedings of the 7th international conference on Autonomic computing*, ser. ICAC '10, New York, NY, USA, 2010.

[27] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *SIGPLAN Not.*, vol. 46, no. 3, Mar. 2011.

[28] P. Mackerras, T. S. Mathews, and R. C. Swanberg, "Operating system exploitation of the POWER5 system," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 533–539, 2005.

[29] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Using performance modeling to design large-scale systems," *Computer*, vol. 42, no. 11, Nov. 2009.

[30] C. Boneti, R. Gioiosa, F. Cazorla, J. Corbalan, J. Labarta, and M. Valero, "Balancing HPC applications through smart allocation of resources in MT processors," in *IPDPS '08: 22nd IEEE Int. Parallel and Distributed Processing Symp.*, Miami, FL, 2008.

[31] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ser. ISLPED '10, New York, NY, USA, 2010.

[32] U. of Tennessee Knoxville (UTK), "PAPI: Performance Application Programming Interface," http://icl.cs.utk.edu/papi/.

[33] V. Weaver, "Linux perf_event features and overhead," in *Proceedings of the 2nd FastPath Worskhop*, 2013.

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

[35] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, Mar. 2011.

[36] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, "Portable, scalable, per-core power estimation for intelligent resource management," in *Proceedings of the Int. Conference on Green Computing*, 2010, pp. 135–146.

[37] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Enabling accurate power profiling of hpc applications on exascale systems," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, 2013.

[38] C. Bolchini, A. Miele, F. Salice, and D. Sciuto, "A model of soft error e.ects in generic ip processors," in *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, ser. DFT '05, 2005.

[39] A. Sena, F. Ribeiro, V. Rebello, A. Nascimento, and C. Boeres, "Autonomic malleability in iterative mpi applications," in *Proceedings of the 2013 25th International Symposium on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '13, Washington, DC, USA, 2013.