University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Toward a Self-aware System for Exascale Architectures

Aaron Landwehr
Stéphane Zuckerman
Guang R. Gao

CAPSL Technical Memo 123
June 2013

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Contents

List of Figures

Abstract

High-performance systems are evolving to a point where performance is not the sole relevant criterion anymore. The current execution and resource management paradigms are no longer sufficient to ensure correctness and performance. Power requirements are presently driving the co-design of HPC systems, which in turn sets the course for a radical change in how to express the need for scarcer and scarcer resources, as well as how to manage them. We believe that systems will need to become more introspective and self-aware with respect to performance, energy, and resiliency. To this end, this document explores the major hardware requirements we believe are central to enabling introspection, the types of interfaces and information that will be needed for introspective system runtimes, and discusses a research path toward a self-aware system for exascale architectures.

1 Introduction

As we move toward an exascale future with ever expanding capacities in terms of both cores and resources, we have reached a point in computing where current execution paradigms will no-longer suffice. High performance computing systems have begun to approach a point where the ever growing multiplicity of transistor counts and components is not sustainable in terms of energy consumption. It has been said that at the current rates, extrapolated into the future, that an exascale computer system would consume over 1.5 GW of power [34]. These ever expanding power requirements necessarily result in the need for a fundamental and radical shift in terms of programmability and adaptation. We believe that systems will need to become hierarchically introspective and self-aware to be able to adapt to these steep performance and energy requirements.

Problem Statement There are number of key facets that need to be addressed to enable a truly introspective and self-aware system capable of performing well and efficiently. The first is that a form of co-design needs to occur in terms of hardware and software. Exascale hardware needs to support a number of integral features to enable controlling system software to monitor and adapt to the current system state and any requirements passed to it in the form of power or performance. In broad terms, there will need to be some form of an observe-decide-act (ODA) loop to monitor, make decisions, and to control both the hardware and software aspects of a system. The second is that the system needs to be capable of adapting for power and performance while at the same time maintaining correct and reliable operation. It is key to recognize that these conflicting goals form a basis for a multi-variable problem which will be further complicated by the need to run multiple programs on a system with thousands of components. There is an open question on how to self-adjust and to meet these goals. The third facet is to propose a hierarchical method to control the system using these ODA loops and the co-designed hardware and software features within the system. This document primarily serves to discuss the first two facets and leaves the third for future discussion. As a second inter-related objective, we discuss the current Traleika Glacier (TG) toolchain as it pertains to our path toward researching and demonstrating self-adaptation.

As mentioned, adaptation will need to occur using ODA loops as shown in Figure 1. These are a type of self-feedback loop consisting of three stages where each stage feeds back to another stage in the process. The observe stage takes in inputs from the system environment. The Decide stage is where decisions are made based upon observed information and possibly some form of external input. In this context, a decision is answering the question of *what* to do in the system not *how* to achieve the objective. The act stage translates a decision into a set of actions which are then committed by adjusting some form of actuators. This stage answers the question of *how* to adapt. For instance, this could be to adjust core frequencies or some combination of actuators or knobs. In section 6.1, we will discuss how ODA loops directly map to the self-adaptive mechanisms of the TG architecture.

The rest of this document is organized into a discussion of the various steps involved in

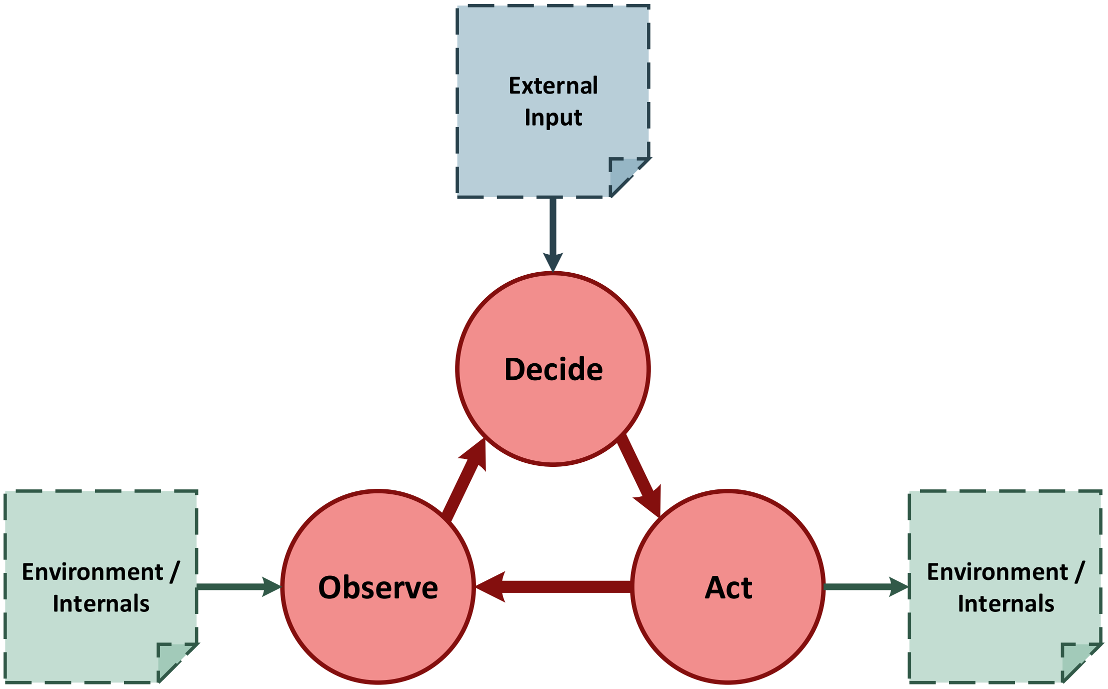

Figure 1: A generalized ODA loop consisting of observe, decide, and act stages.

moving toward a self-aware system for exascale architectures. In section 2, we discuss the background of the TG architecture and program execution models. In section 3, as a form of co-design, we evaluate important hardware requirements for aiding an introspective software system in self-adaptation. In section 4, we evaluate the current TG toolchain to identify any deficiencies that need to be addressed moving forward. In section 5, we expand our problem statement. In section 6, we discuss a research plan in moving toward self-adaptation. This plan is two-fold and involves addressing iteratively both the deficiencies in the tool chain and a primary goal of demonstrating adaptation within TG architecture. Finally, in section 7, we discussion related work in the field.

2 Background

This section is devoted to a discussion of the Traleika Glacier architecture, the codelet execution model, and the challenges and opportunities for self-awareness within exascale architectures.

2.1 Traleika Glacier Architecture

The overall Traleika Glacier architecture [8, 10] is shown in Figure 2. At the lowest level, TG is organized into blocks consisting of a Control Engine (CE), eight eXecution Engines (XEs), and a block shared memory. This organization is designed to decouple algorithmic workload from system monitoring and control. An XEs fundamental usage is to execute arbitrary code without

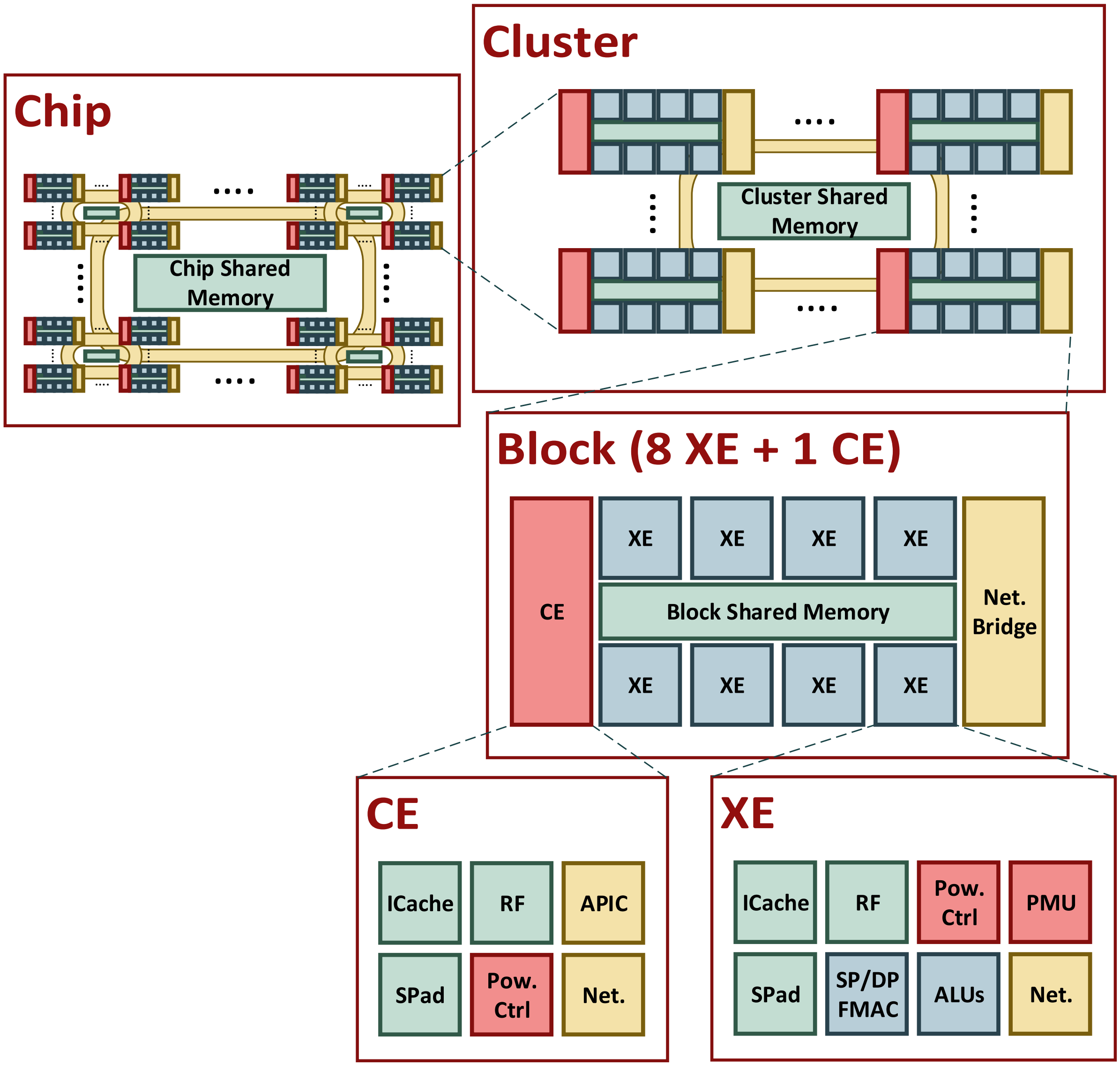

Figure 2: The Traleika Glacier Architecture consists of heterogeneous cores in a hierarchical configuration with network interconnects at each level, organized into blocks, clusters, and chips. The sizes of memories and the amount units per chip and cluster are currently unspecified.

interruption. The CE on the other hand is designed to control and schedule work to a number of local XEs within a given block. XEs and CEs are architecturally unique from one another. Both XE and CE cores contains a register file (RF), instruction cache (ICache), scratchpad memory (SPad), clock/power gate control unit, and network functionality. In addition, XEs also feature a floating-point unit, as well as a performance monitoring unit (PMU). CEs, being designed for control, contain an advanced programmable interrupt controller (APIC). The XEs, being designed solely for execution, do not have logic to directly handle interrupts or traps and instead any interrupts are offloaded to the CE within the block. The architecture itself is designed to operate at close to threshold voltages as well as to control the power and clocks of functional units (FUs) within the system in order to minimize energy usage.

2.2 Traleika Glacier Toolchain

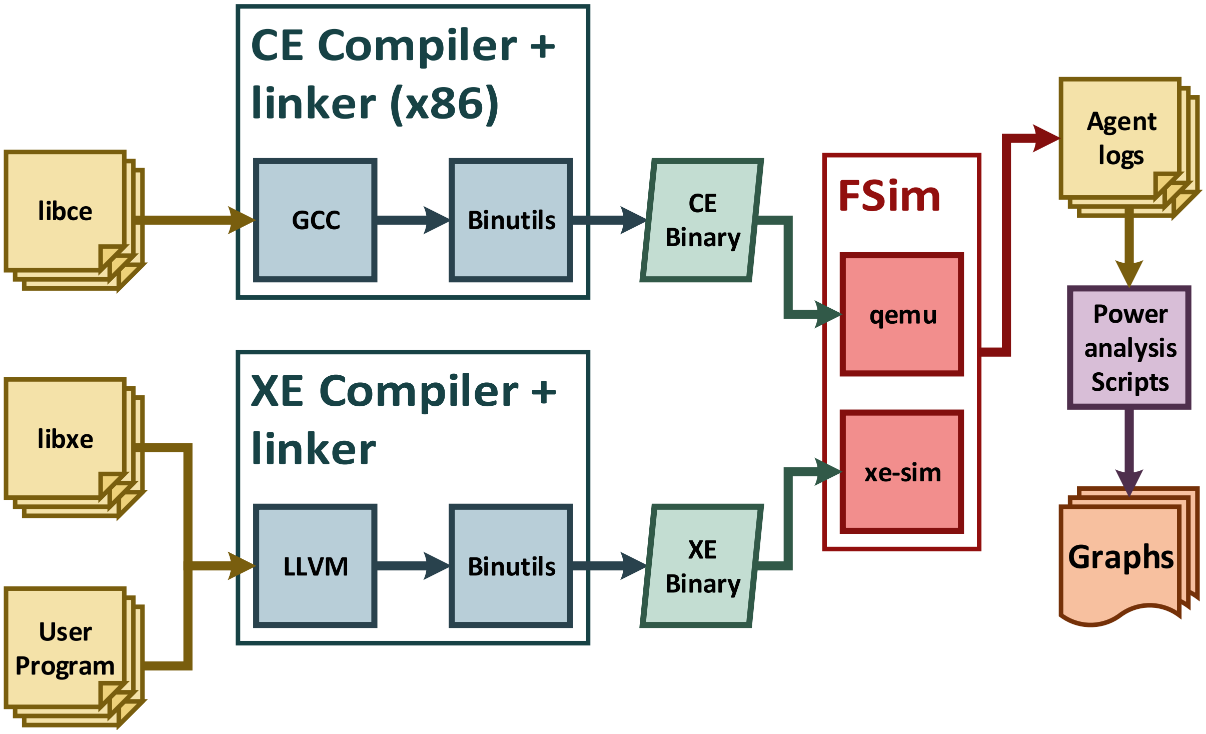

Figure 3: The Traleika Glacier Toolchain from program compilation to simulation output.

The Traleika Glacer toolchain is shown in Figure 3. In Traleika Glacier, XEs have a distinct instruction set architecture (ISA) and as a result a special compiler and linker is used to produce XE programs. As input, a special library providing basic functionality called libxe is compiled and linked to a user program. The CE on the other hand uses an x86 binary produced from a generic x86 compiler and linker. As input, the CE compiler takes a special library called libce which implements the CE runtime. From there, the binaries are loaded into the the functional simulator FSim. In order to simulate the TG architecture, FSim uses two distinct simulators with a communication layer between them. QEmu is used to to simulate a CE and to run the CE binary whereas xe-sim is used to simulate XEs and the different layers of memory. In the simulation framework, each component is called an *agent* (each execution engine, block shared memory, etc.) and log files are produced during a run for each agent with various levels of logged information. This could be such information as standard output or detailed execution information such as different types of operations, etc. From the logs, a power analysis can be done to produce detailed energy usage information in the form of graphs for a given run of a program.

2.3 Codelet Program Execution Model and System Software Model

As we discussed in the previous section, the TG architecture will be hierarchical in nature with numerous components. Coarse grain or monolithic approaches to adaptation do not befit this type of architecture. We strongly believe that these types of program execution models (PXMs) will be incapable of effective self-adaptation. Instead we believe that the focus should be on using and incorporating fine-grained PXMs within a self-adaptation framework. There have been number of discussions and effort specifically tailored to the TG architecture [20]. One such runtime is the Open Community Runtime (OCR) [33] effort designed to explore various methods of high-core-count programming. UD has proposed and implemented their own fine-grained codelet execution model [46, 47] for which several implementations exist [23, 39]. Moreover, the codelet model can be implemented using OCR as vehicle for exploring the model within the TG architecture.

2.4 Challenges and Opportunities for Self-awareness

There are number of challenges in the move toward exascale architectures. Energy becomes a primary and ever increasing issue because of the aforementioned multiplicity of components and transistors. Moreover, current execution paradigms will no longer suffice in terms of either performance or energy as communication and data access becomes prohibitively expensive between far memories and cores. This leads to the need for intelligent system runtimes designed to minimize energy while also maximizing performance. Furthermore, this leads to the need for additional hardware mechanisms to enable system runtimes to monitor system state efficiently and accurately. The details of which will be discussed in the following section.

3 Requirements for Self-adaptation

In this section, we will discuss the underlying hardware requirements in order to implement a self-adaptive system. Many of these will be integral toward enabling a self-adaptive system software and others will simply improve or make its job easier in adapting. Primarily this section serves as a "wish" list of features that we believe are important for adaptation in any exascale architecture. However, before we discuss these requirements in detail, we will first discuss the types of adaptation that we target followed by a discussion of the benefits of a tailored performance monitoring unit

3.1 Types of Adaptation

As we mentioned briefly in section 1, when it comes to a self-aware system, there are a number of types of adaptation that need to occur. These can be categorized broadly into three distinct types of adaptation: adapting for energy or power consumption, adapting for performance, and adapting for resiliency.

Energy adaptation covers any decisions that take into account energy and power in some manner. This can include such things as monitoring the energy usage of tasks directly or indirectly as well as maintaining a given energy or power envelope. This can also include any decisions that increase performance with a primary focus on reducing energy. An example of energy adaptation would be to identify whether a task is utilizing specific FUs and to clock or power gate them if they are not in use. Performance adaption covers any decision that has a primary goal of increasing performance. Such decisions might take into account system and network utilization as well as characterize tasks by the types of operations they are doing. For example, this could be to identify whether a task is CPU bound, I/O bound, etc. Resiliency adaptation covers any sort of detection and/or prevention of faults as well as the handling of task recovery in the event of failure. Task recovery can entail a multitude of different features such as task migration and memory redundancy.

While we will discuss all three types of adaption, the primary purpose of this document is to focus on a path forward for demonstrating energy adaptation in the TG architecture. It is important to note that there is necessary coupling between all three categories of adaptation. An example of overlap between performance and energy adaptation is intelligently scheduling tasks and movement of data to increase performance while at the same time reducing energy consumption. An example of overlap between all three types of adaptation is building in hardware support for the detection of data corruption. Without hardware support, a resilient runtime would be forced to duplicate work and provide checksumming which increases energy and decrease performance at the same time.

3.2 Role of the PMU in Exascale Systems

Before moving onto the detailed requirements in the subsequent subsections, we will discuss the PMU as an important mechanism toward enabling adaptation. From an energy perspective, the counters can be combined with instruction energy cost metrics in order to indirectly monitor energy. From the perspective of performance monitoring, the PMU can directly give many different instruction count metrics that are useful in characterizing performance. From a resiliency perspective, counts of correctable errors can be used to aid in proactively monitoring for potential issues and as a mechanism to determine whether the system software should be cautious with the work it is scheduling. Given a plethora of counters and the ability to run them concurrently will greatly aid in information gathering for a self-adaptive system. As such, it is our belief that the PMU will serve as the primary means of information gathering for both performance and energy adaptation and will be one of the most important mechanisms of a self-adaptive system.

3.3 Energy Requirements

For any large-scale computer system (including current petascale ones), the primary goal in self-adaptation is to minimize energy consumption. As discussed previously, the PMU will be integral in this goal.

At the most basic level, the PMU provides various performance related metrics. This includes counts of various different instruction types such as local/remotereads, writes, ALU operations, FPU operations, DMA operations, etc. These counts are useful directly for determining the workload characteristics and optimality of running tasks (and of higher level components in the system). For example, if the runtime system is able to determine that a task is spending the majority of its time idling while waiting for remote memory through the usage of some combination of remote read and DMA operations, it could clock gate the processor running the task, while the data of the task is moved to a more localized memory. For another example, through the count of FPU operations, the runtime system could determine that only integer calculations are performed on a given XE, and thus decide to power-gate its FPU.

The PMU can also be used indirectly to estimate energy usage. This is possible if the energy cost of various instructions and components in the system are known or estimable, and an energy model is developed. Essentially, the costs of instructions could be combined with the counts read from the PMU to form a picture about the overall energy usage. This information would then be used in conjunction with specified power budgets to determine if actions need to be taken in order to meet goals.

The TG architecture is expected to be capable of adjusting the state of FUs. The Extended Document Specification (EAS) for the TG architecture [10] states that this will be at a functional unit block (FUB) granularity. To clarify, this is at a finer granularity than FUs, meaning that individual sub-unit pipelines can be clock gated or power gated. We will discuss this aspect in more detail in section 4.1.2. Here we focus on the granularity we believe is useful for adaptation. At an individual core level, it is useful to be able to adjust both the clock rates, and to power down unused cores in order to save energy. An example of the former is simply adjusting the clock rates to meet a specified power budget. Another example outside of the realm of energy only adaptation is that some cores may not be able to run at a full clock-rate due to physical defects or the current thermal characteristics of the system. It may be advantageous in this case to use the cores at a lower clock-rate than to simply put them into a power-gated state. As a final motivating example, a self-adaptive runtime could identify a case where the rate of data being streamed into a block is lower than the XE using the data for computation. In this case, the computation XEs would need to idle waiting for the data. The CE could identify this and then individually slow the clock rate of the XEs to match that of the rate at which the data is being streamed in.

We believe it is advantageous to adjust at an even finer-granularity than simply the core. One interesting motivation would be for task kernel hinting. Given a compiler with the capability to identify the types of instructions used by a task kernel and given mechanism for the runtime to hook into this information, it could identify explicitly which units would not be used by a given task and simply power gate them. Furthermore, the same strategy could be applied at an even finer-granularity to turn off individual pipelines within FUs.

One alternative mechanism for hardware supported energy monitoring is the Running Average Power Limit (RAPL) interfaces found in the Sandybridge, Ivybridge, and Haswell architectures [16]. These directly provide energy/power information, power limiting, and policy controls in the form of machine specific registers (MSRs). Using these interfaces, it is possible to retrieve both power (watts) and energy (joules) directly as well as specify a goal in terms of watts over a given time interval. Finally, these interfaces are able to give information about the amount of time that a component has been throttled in order to meet a specified goal.

3.4 Performance Requirements

In X-stack, the system software will be responsible for task scheduling and resource allocation. Thus it needs to be able to monitor performance in order to achieve adaptation. There are various types of performance metrics that will be important. These can range from different types of resource utilization (network, CPU, memory, etc.) to workload distribution, etc. Characterizing sections of the system will require monitoring to be relatively fine-grained. We believe that the ideal granularity is at the task level. We discussed previously some examples of the dual energy/performance adaptations that can be yielded from PMU metrics. As we mentioned, the key importance here is that the PMU is useful in determining the workload characteristics and optimality of running tasks within the system.

Using the PMU events described, performance within the runtime can be evaluated. To further motivate the usefulness of knowing this information, let us consider another example. By knowing the frequency and types of memory counts, the system software can determine network utilization and whether the communication is relatively localized. This information is useful for determining how optimal the current task scheduling is in terms of performance and energy efficiency. If for instance, the system software can determine that groups of tasks are communicating frequently but are not localized to the same block, it could migrate the tasks to one block in order to localize the network traffic.

3.5 Resiliency Requirements

Fault tolerance is one of the most important aspects of a self-adaptive system. Without proper hardware support, the software will be unable to cope with failures or to meet goals in exascale systems. Furthermore, the system software would necessarily be burdened with the detection and prevention of faults through costly primitive means. This could potentially entail such things as the duplication of tasks and verifying the results of all task computations within the system. In short, lack of proper hardware support for resiliency will significantly affect other aspects of self-adaptation.

It is our belief that the hardware must be able to detect faults within the components of the system. The primary motivation for this is to minimize runtime scope and energy costs. A system software burdened with the aforementioned details will be extensive and inefficient. This leads not only to a high cost in software support, but also a reduction in energy efficiency and performance. For example, task duplication could force the same task to be re-run three times simply to determine which set of components is faulty.

However, the hardware needs not only to detect and/or correct faults but also a means to deliver information about the failure to the system software. It is absolutely essential for an introspective system software to know which FUs have failed in order to reschedule any tasks that require or depend on the failed hardware resources.

Even with proper hardware detection for faults, the software system is burdened by the need to schedule tasks efficiently in a non-ideal environment. From a self-adaptive standpoint, one of the primary tasks will be to achieve a known working state. On current generation systems, the primary means by which this is achieved is through extensive quality control tests. In the software stack of an extreme scale environment, a map of known permanently bad components from quality of control testing will be essential for the system-software to learn and adapt as quickly and efficiently as possible. This will yield a known good state on the start up of the system without spending large amounts of energy determining which parts of the system are bad and/or scheduling tasks to faulty components. This map would need to contain two types of information: all components that fail under any circumstances, and components that fail under certain known operating conditions. For example, the latter could contain Vdd requirements, thermal requirements, limits to the number of XEs that can operate together, etc.

Another aspect of fault tolerance is proactive preventive measures that can be taken in both hardware and software. For instance, if the system-software can identify components with a high probability of failure, it can avoid scheduling critical tasks to those components. The envisioned form of support is through the dual usage of error correction and the tabulation of errors exposed through performance monitoring units. Given such details, the system-software could keep track of the errors and use some form of built in risk assessment when scheduling and allocating resources.

4 Simulator and System Runtime Status

This section will discuss the current status of the simulator (FSim) and system runtime with respect to the hardware and software requirements we have identified as integral dependencies for the implementation of an introspective and self-aware system runtime. We will first discuss the simulator followed by the system runtime.

4.1 Simulator

The simulator carries a heavy burden in terms of what it must support in order to enable the study of a self-adaptive system. It must be capable of providing both performance monitoring and energy monitoring interfaces to the runtime software as well as power management interfaces. Moreover, it must be able to provide a mechanism to quantify the energy usage of a program. This section will discuss in detail the different requirements for the simulator as well as the current state of the implementation with respect to those requirements.

4.1.1 Performance Monitoring Unit

The simulator needs to support a number of introspective capabilities and interfaces. The first and most important of these is the PMU as we discussed in section 3. For completeness sake, we repeat here that the PMU must be capable of counting many different types of instructions (Floating point, DMA, remote memory, local memory, etc). These will be the primary method for an introspective system to identify the workload characteristics of running code at a fine-grained level. Interface wise, PMU registers will need to support both polling and interrupt triggering as well as the ability to specify and enable threshold triggers. For example, one might track remote load operations by enabling the remote load counter and specifying a CE interrupt to occur every N operations. The level of granularity of PMUs (per XE, per Block, . . .) is an important consideration because it limits the level at which an introspective runtime can characterize the workload of the system. It is our belief that both the hardware knobs (clock gating, power gating, etc.) and software knobs (scheduling, data movement, etc.) should be taken into account when considering the level of PMU granularity. For example, even without the ability to clock-gate individual XEs, decisions about where to schedule tasks and whether to localize data can still be made using information gathered from XE level PMUs. Moreover, at this granularity, event driven tasks (EDTs) can be characterized by whether they are computationally intensive, I/O intensive, or some combination in-between. This information could feasibly be used to identify which blocks to schedule future EDTs on or even future instantiations of the same EDT type.

The simulator currently contains a rudimentary PMU with support for register polling using memory mapped I/O. Feature wise it lacks support for threshold triggering and thus cannot wake a CE when a specific event count is reached. The EAS [10] specifies that an XE can either enter a clock gated state or continue executing instructions depending on the PMU trigger control register configuration. In the latter case, it is important to note that the PMU alarm condition can be overwritten by further alarms. This will be discussed in more detail in section 6.2.1.

There are also a number of timing limitations when accessing PMU registers within the simulator. The EAS states the following timing constraints for PMU counter access:

1. The end of the current clock period will reflect the PMU counters that correspond to the state of the machine at the end of the prior clock period,

2. a read of a PMU register will occur in the top-half of the cycle in which the read occurs and this read will return the state of the register from the end of the prior cycle,

3. a counter update or a software write to a PMU register occurs in the bottom-half of the cycle in which a write occurs,

4. in the case of a conflict between a core unit reporting a PMU event conflicting with a software-based write instruction, the software-based instruction always over-writes any other input to determine the final state, and

5. there is no promise of single-cycle access to the PMU registers; however, the timing to the PMU registers will be uniform such that whatever latency is incurred is the same regardless of which PMU register is involved.

Although the aforementioned constraints will hold true in the hardware implementation, the simulator lacks cycle accurate timing and thus these conditions will not hold true. Simply put there are no guarantees as to the cycle-accuracy or uniform latencies of reads and writes to PMU registers within the simulator. Although we do not anticipate these specific limitations to be of a large concern for our demonstration of self-adaptation, they will no doubt be important for future consideration.

4.1.2 Power Management

The TG architecture will be capable of adjusting the state of FUs. In section 3, we previously motivated our beliefs as to the granularity that this should be adjustable at. In this section, we will discuss adjusting the state of FUBs as a function of the simulator.

An XE has both FUB clock and power controls. These allow fine-grained control over clocking and powering specific FUB pipelines (e.g. 16-bit Integer *Mul* pipeline) as well as coarser grain controls over Mega-Block pipelines (e.g. 32-bit integer pipeline). There are also flags to control the power to local caches and scratchpad memory. Currently the simulator does not support controlling clock/power states at this granularity. Moreover, there have been some concerns as to how to accurately model the energy usage at this level within the simulator.

Each major component in the system (XE, CE, Block SRAM, Intra-block network ports, and Inter-block network port) has a set of MSRs to control the clock and power states of a given component. Depending on the component, the state can be adjusted at a finer-granularity. For instance, the power to specific memory banks of block shared memory can be controlled. Currently in the simulator, it possible to clock gate XEs and CEs; however, we do not currently know the status of the support for clock gating and power gating other components. It is worth noting that for self-adaptive purposes accurate power modeling is needed to demonstrate the importance of adjusting component state. This will be discussed in the following section.

4.1.3 Power Modeling

Power modeling is another important aspect to quantifying the benefits of self-adaptation. In order to effectively demonstrate the benefits of self-adaption for energy, the simulator will need to measure both dynamic and static energy. At a high level, quantifying dynamic energy allows us to see the impact of instruction costs, different component costs, and network communication costs. This information is useful for determining the optimality of task scheduling and data distribution. However, only measuring dynamic energy is unrealistic as it does not take into account the leakage costs of keeping components powered or the costs of the logic to keep components clocked and transitioning. Given that the TG architecture will operate at close to the threshold voltage, this becomes center stage because up to fifty percent of the energy costs can come from the leakage in idle components. As such, it is imperative that there be a way to quantify static energy costs. From an introspective perspective, measurements of static energy allow us to infer the benefits or costs of dynamically adjusting an FUs states as a form of self-adaptation.

Currently, the simulator is capable of measuring the dynamic energy usage of instructions, DMA, cache, scratchpad, network communication, memory, memory controller, and off chip communication in terms of joules or joules per operation [28, 41]. However, currently, static leakage energy is only modeled as an estimated constant multiple of the total dynamic energy. As mentioned above this is an unrealistic assumption given the aforementioned assumptions regarding the TG architecture and the fact that FUs can be put into clock gate or power gate states dynamically.

4.2 Runtime

This section will discuss the current status of the runtime as method to implement and demonstrate self-adaptation. Fundamentally, at the block level, the CE runtime will be responsible for observing and acting upon any gathered information. In order to do this, a CE will take on the role of a decision engine that operates within a localized observe-decide-act loop. There needs to be interfaces in place for information gathering, and for communication between any FUs that a decision engine is making decisions for. At a higher level, there needs to be mechanisms in place for communication between different decision engines in order to communicate system state. This will be relied heavily upon in the future for non-localized self-adaptive decisions.

4.2.1 Communications

XE-to-CE Within a block, there exists a synchronous message subsystem for communication between a CE and the XE. When an XE needs to communicate with the CE, it will write the message to a designated slot in memory reserved for messages, signal the CE to wake up, and then enter a clock gated state to wait for the CE to process the message. The CE will then process the message and wake the XE once it is done. Any self-adaptive mechanisms that originate from the XE outside of the CE's observation will need to occur through this communication channel.

CE-to-CE At the inter-block level, communication between CEs occurs using an asynchronous doorbell system. CEs can send different types of messages to one another using this system. When a CE wants to send a message, it will write it to an available doorbell slot located within a remote CEs local memory, “ring” the doorbell, and then continue to do other operations. Because the communication is asynchronous, the sender can specify that an acknowledgement be sent back once a given message is processed by the receiving CE. Using this mechanism, callback subroutines can be run on the sender once an ACK is received and processed by the sender. From a self-adapting perspective, aggregated and generalized information about local state will be sent through this communication channel. For example, the relative health and utilization of a block can be communicated to other blocks this way. For instance, if the self-adaptive system is organized into a hierarchy of decision engines, information can be communicated through the hierarchy for non-localized scheduling decisions to be made.

4.2.2 Interfaces

The self-adaptive aspects of the system runtime will at a low-level depend on the underlying interfaces provided by the simulator. If the implementation depends on polling registers, the runtime will have to take a more active role in observation and information gathering; however, if the runtime relies upon alarm triggers to be raised when certain conditions are true, then it will take on a more passive role in observation. Fundamentally, regardless of the information gathering process, the information should be recorded in such a way as to not impact the decision making process. That is to say, the information gathering process should be separated from the decision making process, and that an interface should be designed for a decision engine to act upon gathered information uniformly outside of its own collection of information.

5 Problem Formulation

This section will expand upon the problem statement stated in section 1 by asking three central questions. From there we will put forth a foundation toward solving these questions.

5.1 Questions

- How to monitor, make decisions, and control the hardware and software aspects of a exascale system efficiently?
- How to resolve a multi-variable objective function using a self-aware strategy?
- How to use the TG toolchain and simulation framework as it pertains to our research and our goal of demonstrating self-adaption?

5.2 Solution Methodology

How to monitor, make decisions, and control the hardware and software aspects of a exascale system efficiently? The TG architecture will need to support a number of hardware features to enable monitoring within a block as we mentioned in detail in section 3. From there a system runtime on TG will need to make intelligent decisions to control various hardware features within the architecture to reduce energy consumption. This will take the form of clock rate adjustments and clock gating/power gating of components. From a software decision perspective, a runtime will also need to schedule tasks and move data based upon information from monitoring the system. We mentioned briefly that this forms the basis for an ODA loop. In TG, we foresee each block within the system having some independent ODA loop for localized decisions. Figure 4 shows at a high level how we foresee TG mapping to an ODA mechanism. A CE will implement an ODA loop for self-adaptation. Information will come from various hardware and software mechanisms within the system and goals will come from the user or program. And finally, various actions will occur in the form of adjustments to hardware state or some type software change.

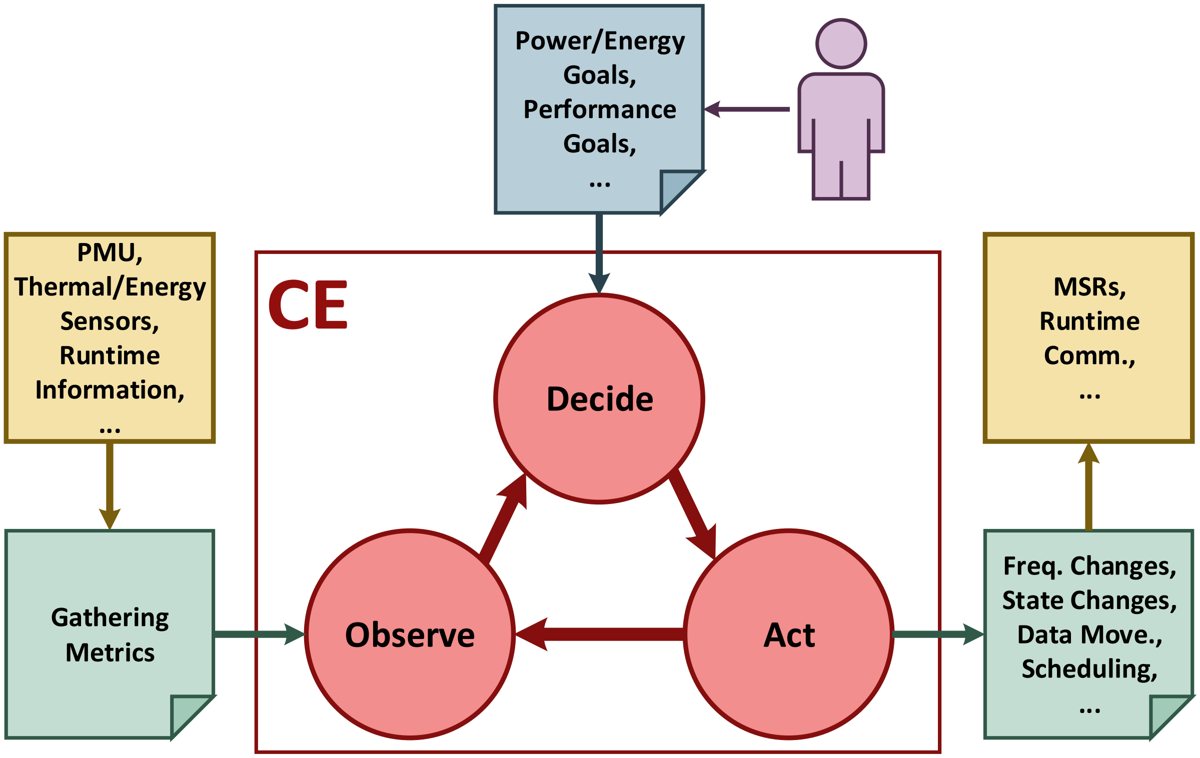
Power/Energy Goals, Performance Goals, ...
PMU, Thermal/Energy Sensors, Runtime Information, ...
CE
Decide
MSRs, Runtime Comm., ...
Gathering Metrics
Observe
Act
Freq. Changes, State Changes, Data Move., Scheduling, ...

Figure 4: Mapping Traleika Glacier to an observe-decide-act mechanism.

How to resolve a multi-variable objective function using a self-aware strategy? It is important to recognize that exascale architectures are burdened with a more complicated decision making process than what is typically seen within current generation systems. Exascale systems must adapt to minimize energy, maximize performance, and to be reliable all at the very same time. There are two key challenges: (1) conflict resolution, and (2) achieving multi-dimensional efficiency [14]. As we stated previously, this is an open research problem that has yet to be solved in the current state of the art [32]. Multi-variable problems are inherently difficult because the search space is too large to do an exhaustive search on and an optimal solution is not known ahead of time (or possibly even the set of actions toward an optimal solution). In TG the problem is even further compounded by the hierarchical and complex nature of the system as well as because a local control engine will not have complete system information and thus local decisions may not meet overall system goals. We foresee the need for some form of distributed control in TG as well as a need for a system model. Ultimately whether the solution lies in machine learning or some other type of decision making process is currently unknown. However, it is worth noting that previous research has suggested that a possible solution lies in using economic models as these are capable of solving for multi-variable problems [14].

How to use the TG toolchain and simulation framework as it pertains to our research and our goal of demonstrating self-adaption? At a high level, we will implement and demonstrate adaptation within the system runtime using monitoring and decision making processes. As we have discussed in section 3, much of the monitoring will be enabled via the architectural features found in the TG architecture. The TG simulation framework provides a basis for monitoring the hardware aspects of the TG architecture and for modifying the state of components. As we noted previously, many of the required features are currently unimplemented and we will need to implement them as we move forward. Moreover, We will need to implement an observation loop within the runtime system to gather information and a mechanism to make and apply decisions. Ultimately, energy usage information will need to be gathered via power modeling in order to quantify the benefits as we discussed in section 4.1.3. The following section discusses in detail our research plan on using the TG toolchain and simulation framework.

6 Research Plan

This section will discuss our overall research plan toward a demonstration of introspection and self-adaption within the TG runtime and simulation framework. Firstly, we will discuss our self-adaptive research plan. Then we will discuss the current tool-chain deficiencies.

6.1 Research Venues

In this section, we discuss our vision of exascale adaptation. We start with an discussion of our research focus of adaptation at fine-grained level and from there move toward a discussion of adaptation at a more coarse-grained level. The former constitutes localized decisions engines that have direct control over local resources. The latter consists of higher level decisions engines which will largely pass high level decisions to lower level decision engines. Our plan to take a bottom-up approach. we will focus in adapting at the block level and then eventually expand to demonstrating adaptation using a hierarchy of decision engines.

6.1.1 Fine-grained Adaptation

Our initial focus for self-awareness will be on power management at the block level. We plan to use a combination of PMU counters in order to achieve energy reductions. The kinds of decisions that we want to initially focus on are clock gating and power gating of components as well as adjusting the frequency at the block level. For example, this could be to lower the clock rate of a block if certain conditions are true, or to clock/power gate components that are currently not in use. Following this, we plan to expand our focus to demonstrating I/O management using the PMU counters.

For a small example of adaptation within TG consider Figure 5. The right side shows a block's state at various stages during execution. The left side shows three distinct decisions occurring between those stages. In the example, all blocks are initially enabled. However, the CE observes that only two scheduled tasks are currently running and from there decides to disable the unused XEs. This decision is then translated into an action of writing to each XE's Clock Control to put them into a clock gated state. Next, the CE observes that there are many remote DMA operations occurring and decides to lower the frequency of the block to conserve energy. This decision is then translated into an action of writing to the Block Frequency Control. Finally, the CE observes that there are no floating point operations occurring within the second XE and decides to power gate the FP functional unit. This decision is then translated into an action of writing to the XE Power Control.

6.1.2 Coarse-Grain Adaptation

One of our far-term goals it to implement a form of hierarchical management or non-localized adaptation. For this, we would use the communication subsystems discussed in section 4.2.1. In terms of energy and performance, it will be important to be able to generalize localized information about specific sections of the system and to communicate that information without transferring large amounts of data. In short, CEs will need to aggregate their local information and to package it into condensed form for usage by nodes higher up in the hierarchy. This aggregated information would include the health and resource allocations in subsections of the system. The higher-level nodes would be tasked with making decisions based upon this information. For example, whether to allocate data or schedule a task to a particular section of the system.

6.2 Addressing Tool-chain Deficiencies

Our current and short-term focus is on addressing some of the deficiencies within the tool-chain. Moving forward, this will be an iterative process. Firstly, we will address some of the more pressing issues and then begin to focus on our research avenues. This subsection is organized into a discussion of simulator and runtime extensions.

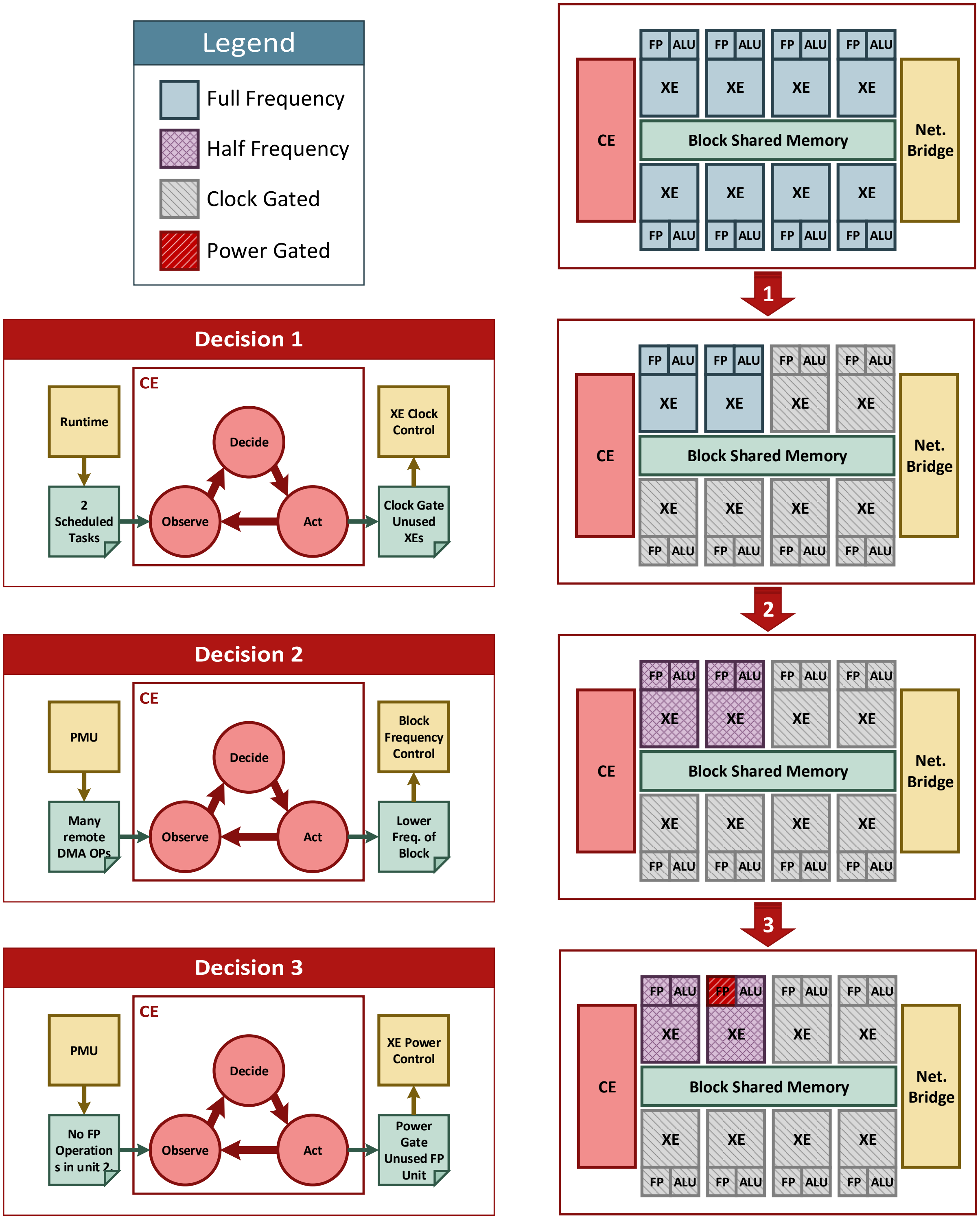

Figure 5: An example of adaptation within Traleika Glacier. Shown is the TG runtime making decisions at the block level and the corresponding block state after each decision.

6.2.1 Simulator Extensions

As discussed in section 4.1.1, there are two ways to access PMU information. The first is polling and the second is interrupt triggering when a specified threshold is reached. In the short term, we will focus on evaluating and correcting any issues with the current register polling implementation. This will largely entail adding any missing counters we need. In the future, once we have small demonstration of the CE performing some type of adaptation within a block, we will focus on the on the ultimate goal to have a working threshold triggering/interrupt implementation.

As we noted in section 4.1.1, the alarm condition register can be overwritten by further alarms before the CE is able to process the original alarm. Moreover, for the PMU in particular, the PMU Trigger Flag registers can also be overwritten in the same manner. These are maskable registers with a bit field for each type of PMU trigger event. This means that multiple triggers for the same event could occur before the CE processes the interrupt which would result information loss. Furthermore, it is also possible that events that are written by a XE while the CE is concurrently processing a PMU interrupt could be lost because the CE is responsible for clearing the trigger register in software. For the simulator, these issues are primarily only a concern if an XE is not put into a clock gated state when an alarm occurs.

6.2.2 Runtime Extensions

Runtime extensions will be in three areas. The first is to develop and implement interfaces that allow retrieval and aggregation of PMU and other monitored information. The second is to develop interfaces to allow adjustment of actuators (clock rates, core state, etc.). And the third is to develop a mechanism to make decisions based upon the aggregated information. Together, this will form the basis of an observe-decide-act loop.

In the far-term, we will need to design and implement mechanisms to generalize and share information between control engines. This will require an appropriate communication layer and a formulation of how to generalize information in a useful and concise manner for remote decision engines to act upon.

7 Related Work

This section discusses previous and related works in self-adaptation. First, we discuss a number of different approaches to adaptation. These fall into three categories: “Application Centric Adaptation,” “Component Centric Adaptation,” and “System Centric Adaptation.” Following that, we discuss a state of the art self-adaptive framework. Finally, we discuss the aspects of exascale adaptation that make it fundamentally different from prior research.

7.1 Application Centric Adaptation

Application centric approaches focus on adaption for specific applications or a subclass of applications within a given domain. Quality of service (QoS) is one large area of active research [1, 7, 18, 26, 45] due to the real time requirements and the ever changing field of computing resources. Many new works focus on cloud computing in particular [22, 42]. Other works focus on changing application specific algorithm policy [40]. Some approaches are more akin to toolkits designed for application programmers to use to enable adaptation within their software [12, 13]. Many of these approaches listed incorporate some form of classical control theory because of various provable guarantees such as stability and linearity, etc.; however, in more recent years, there has also been a shift toward heuristic based tuning and machine learning techniques [27].

7.2 Component Centric Adaptation

Component centric approaches are a form of adaptation that focuses on monitoring and adapting a particular component or resource of a system. For example, there has been research in adapting cache policies [19], dynamic reconfiguration of memory hierarchies [5], self-optimizing memory controllers [17], and online cache modeling [43].

7.3 System Centric Adaptation

System centric approaches focus on adaptation at the system level. Historically the role of resource management has been given to the operating system (OS). In the case of highly parallel systems, we can divide these approaches into several categories: full OSes, lightweight kernels, micro kernels, and high-level runtime systems.

Full OSes High-performance systems which use so-called Full OSes take advantage of off-the-shelf systems, and tune them to reduce system noise. Such approaches are often embodied in cluster-like environments [38]. Even on dedicated supercomputers, these approaches have been followed, as they provide a programming environment which allows for maximal flexibility. However, such systems are in general ill-prepared for the requirements of future extreme-scale/exascale high-performance environments: their control over the power envelope is only at a very coarse-granularity; resilience is left to third party systems, and is not considered as part of the whole; they are oblivious of the needs of the application they host; etc. Finally, full OSes leave very little room for specialization.

Light-Weight Kernels Another approach is to rely on so-called light-weight kernels or LWKs [11, 6]. LWKs offer many advantages over full OSes: They are usually written from scratch, and only reimplement features needed for an HPC environment. The source code being much smaller, means bugs are easier to track and fix. Finally, being specialized kernels, they usually emit very little noise when running an application on top of LWKs. They also do expose significant limitations: they very often require the user to learn a new API to communicate with the system; if a feature usually provided by full OSes is missing from the LWK, the user's application may not be portable to the system. In general, the application programmer does require features found in traditional OSes. Some LWKs do forward calls to "missing" features to a "heavier" kernel however.

Micro-Kernels Micro-kernels strip down the OS to its bare minimum (i.e. address space management, process/thread management, inter-process communications). This quintessential kernel runs in privileged mode (e.g. ring0 on x86 architectures), while providing "servers" or "satellites" which enrich the overall system with additional features, but in an unprivileged mode (for example, a file system driver). Micro-kernel OSes have shown they could be robust and thus fulfill the resiliency and maybe even the power and energy requirements (as only the required services are running).

New approaches try to revive micro-kernels, and remove the "layers" that used to make them slow [3, 21, 29, 31]. Indeed, message-driven communication in micro-kernels tend to suit multi and many core systems very well. However, there is no approach (to the best of our knowledge) that tries to provide a holistic view of our three target goals (performance, power/energy, resiliency), for different granularities. However the latest efforts around micro-kernels have evolved toward a "library-oriented" type of operating systems [2]. Such approaches tend to have goals that are closer to our own.

High-level Runtime Systems High level runtime systems typically implement some form of resource management on top of an existing OS. Typically they intend to provide some type of policy management that doesn't exist in the underlying OS. Some focus on providing QoS of system resource [24, 25, 35] or provide resource management with dynamic policies [44]. Others are in the form of languages or frameworks that provide mechanism for an application to adapt [9, 15, 30, 36].

7.4 State of the Art

The current trend in adaptive computing has been to focus on energy adaptation in some form [4, 37] as this is integral for low power domains and now increasingly for exascale. Recent research in the field [14] has focused on both energy and performance adaptation using a notion of "application heartbeats." These allow for an application to communicate goals to a system as well as progress toward those goals. The results have shown that various applications can be instrumented with heart beats and are capable of adapting to meet both energy and performance goals.

7.5 Limitations of Current Work

There are a number of shortcomings that need to be addressed for exascale architectures. Application and component centric approaches lack a wholistic view of adaptation. A coordination between system resources, components, and applications will be integral at the exascale level. Moreover, current system centric approaches lack fine-grained control over components due to limitations in hardware. Exascale architectures will need to adjust the state of components at a very fine granularity in order to conserve energy and to meet power envelopes. This also means that applications will need to become first class citizens in the sense that their goals will need to be accounted for by a self-adapting system. The state of the art research has shown interest in this aspect, but the current research does not resolve the open problem of multiple conflicting goals or of hierarchical non-localized self-adaptation.

Acknowledgment

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717.

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessaril state or reflect those of the United States Government or any agency thereof.

References

[1] T. Abdelzaher and K. Shin. End-host architecture for qos-adaptive communication. In *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*, pages 121–130, 1998.

[2] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: a library operating

system for a jvm in a virtualized execution environment. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 44–54, New York, NY, USA, 2007. ACM.

[3] J. Appavoo, M. Auslander, M. Butrico, D. Da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. Wisniewski, and J. Xenidis. Experience with k42, an open-source, linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[4] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.*, 45(6):198–209, June 2010.

[5] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 245–257, New York, NY, USA, 2000. ACM.

[6] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador. Nix: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[7] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola. Moses: A framework for qos driven runtime adaptation of service-oriented systems. *Software Engineering, IEEE Transactions on*, 38(5):1138–1159, 2012.

[8] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, et al. Runnemede: An architecture for ubiquitous high-performance computing. HPCA, 2013.

[9] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 11–20, 2000.

[10] J. Fryman. *RMD Strawman Platform XE, CE, Block, and Networks Extended Architecture Specification Version 9.1B.* Intel, December 2011.

[11] M. Giampapa, T. Gooding, T. Inglett, and R. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from blue gene's cnk. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–10, 2010.

[12] A. Goel, D. Steere, C. Pu, and J. Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. Technical report, Oregon Graduate Institute, 1998.

[13] A. Goel, D. Steere, C. Pu, and J. Walpole. Adaptive resource management via modular feedback control. Technical report, Oregon Graduate Institute, 1999.

[14] H. Hoffmann. *SEEC: A Framework for Self-Aware Management of Goals and Constraints in Computing Systems.* PhD thesis, Massachusetts Institute of Technology, February 2013.

[15] J. Hollingsworth and P. Keleher. Prediction and adaptation in active harmony. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 180–188, 1998.

[16] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C.*

[17] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. *SIGARCH Comput. Archit. News*, 36(3):39–50, June 2008.

[18] D. Ivanovic, M. Carro, and M. Hermenegildo. Towards data-aware qos-driven adaptation for service orchestrations. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 107–114, 2010.

[19] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGOPS Oper. Syst. Rev.*, 36(5):211–222, Oct. 2002.

[20] R. Knauerhase, R. Cledat, and J. Teller. For extreme parallelism, your os is sooooo last-millennium. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism, ser. HotPar*, volume 12, pages 3–3, 2012.

[21] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.

[22] N. Kumar, N. Chilamkurti, S. Zeadally, and Y.-S. Jeong. Achieving quality of service (qos) using resource allocation and adaptive scheduling in computing with grid support. *The Computer Journal*, 2013.

[23] C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 21–26. ACM, 2012.

[24] B. Li and K. Nahrstedt. Impact of control theory on qos adaptation in distributed middleware systems. In *American Control Conference, 2001. Proceedings of the 2001*, volume 4, pages 2987–2991 vol.4, 2001.

[25] B. Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE J.Sel. A. Commun.*, 17(9):1632–1650, Sept. 2006.

[26] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. H. Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *Parallel and Distributed Systems, IEEE Transactions on*, 17(9):1014–1027, 2006.

[27] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. A comparison of autonomic decision making techniques. 2011.

[28] A. Mishra. *Energy Model for Sunshine Network Architecture.* Intel, May 2012.

[29] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[30] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: adaptive control of distributed applications. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 172–179, 1998.

[31] J. Sacha, J. Napper, S. Mullender, and J. McKie. Osprey: Operating system for predictable clouds. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, 2012.

[32] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

[33] V. Sarkar, B. Chapman, W. Grop, and R. Knauerhase. Birds-of-a-feather session: Building an open community runtime (ocr) framework for exascale systems. `http://sc12.supercomputing.org/schedule/event_detail.php?evid=bof219`, November 2012.

[34] T. Scogland, B. Subramaniam, and W. chun Feng. Emerging trends on the evolving green500: Year three. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 822–828, 2011.

[35] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. Mete: meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, June 2011.

[36] F. Sironi, D. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. Santambrogio. Metronome: Operating system level performance management via self-adaptive computing. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 856–865, 2012.

[37] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 161–174, New York, NY, USA, 2007. ACM.

[38] T. L. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *ICPP (1)*, pages 11–14, 1995.

[39] J. Suetterlein, S. Zuckerman, and G. R. Gao. An Implementation of the Codelet Model. In *Euro-Par 2013*, 2013.

[40] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 277–288, New York, NY, USA, 2005. ACM.

[41] G. Venkatesh. *Power Model for Sunshine Architecture.* Intel, February 2012.

[42] W.-J. Wang, Y.-S. Chang, W.-T. Lo, and Y.-K. Lee. Adaptive scheduling for parallel tasks with qos satisfaction for hybrid cloud environments. *The Journal of Supercomputing*, pages 1–29, 2013.

[43] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang. Online cache modeling for commodity multicore processors. *SIGOPS Oper. Syst. Rev.*, 44(4):19–29, Dec. 2010.

[44] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: a middleware architecture for feedback control of software performance. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 301–310, 2002.

[45] Z. Zheng and M. Lyu. An adaptive qos-aware fault tolerance strategy for web services. *Empirical Software Engineering*, 15(4):323–345, 2010.

[46] S. Zuckerman, J. Suetterlein, and G. R. Gao. Toward an execution model for extreme-scale systems-runnemede and beyond. Technical report, April 2011.

[47] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a codelet program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69. ACM, 2011.