A Design/Study of a Self-Aware Codelet Runtime System

Advanced Topics in Computing Systems Spring 2014
Group B

Pouya Fotouhi, Chuanzhen Wu, and Laura Rozo

University of Delaware, 140 Evans Hall, Newark DE 19716, USA
{pfotouhi, kavfwu, lrozo}@udel.edu

Abstract. To tackle the new challenges of high performance systems, we believe that systems should become more self-aware in order to achieve the optimum resource management. An adaptive system that makes decisions based on system's status at the moment and also user defined constrains. Meanwhile, such system should be also proactive and capable of taking the previous systems observations into account. In this paper we aim to define schemes and policies required for a system based on a *fine-grained*, *event-driven* model.

Keywords: Self-aware Computing, Codelet Runtime System

1 Introduction

Within the progress of computer architecture, extreme scale computers have been developed to satisfy academic and industrial needs. While handling diverse software requirements and dynamic hardware modeling, extreme scale computers bring new challenges to computer engineers.
Exascale systems, which may contain billions of transistors and thousands of cores, highly need new efficient resource management methods instead of current existing methods that are no longer sufficient. The expensive costs (e.g. energy consumption) of these extreme scale systems also become unsustainable and therefore force computer engineers to consider the balance between system's performance and costs, especially at runtime system level. Several efforts have been made to address these problems and various system architectures have been proposed.
Self-Aware systems also bring challenges. Based on Salehie and Tahvidaris work [7], new challenges engineers facing will include defining and implement the self-aware properties (some certain characteristics of the system) and adaptation processes, coordinating them, evaluation techniques and how to involve human supervision.
Codelet Program eXecution Model (PXM) is a fine-grain, event-driven model based on the data-flow theory. A codelet is a set of machine instructions in the runtime system and is scheduled atomically as a single, basic unit of computation.

In this report, we combined the idea of self-awareness with the codelet model and tried to use and incorporate fine-grained codelet PXM within a self-aware framework in order to design a self-aware codelet system while taking the advantages from both of the two systems.
In Section 2 we talk about the required background for our work. We have the problem statement of our project in Section 3. Section 4 includes our solution for the project. We also make a introduce some related works in Section 5.

2 Background

To overcome those challenges mentioned before and build a general self-adaptive system, several works try either to build a system within specified domains (e.g. web servers) with a fixed set of actions defined already or simply build the system in software layer disregarding the hardware construction. Both the two approaches affect the generality of these systems.

2.1 Control theory

Control theory dates back to antiquity and has been widely applied in many fields. The objective of using control theory is to deal with the behavior of dynamical system with inputs. Control theory calculates solutions for the proper corrective action from the controller that keeps the system stable. As a result, control system will hold the set point and not oscillate around it. The basic idea of control theory is to form a closed-loop with an observation part, a decision part and an action part, also called ODA loop. Through the ODA loop, the system is able to gain feed backs from itself and adjust itself with these feed backs as input. We will take how computer system applies control theory as an example. First of all, there will be some kinds of reference, which can be an applications goals. The observation part monitors the executing routine, and then passes the information observed to the decision part. The observed result will be compared with reference. If there is difference, the decision part will try to decide what can be adjusted. After that, the last part, the action part will apply the adjustment. As this is a closed-loop, the observation part will keep monitoring the applications execution so on and so forth.

2.2 Scheduler-driven adaptive framework

A Scheduler-driven adaptive framework (ASAFESSS) [8] uses runtime task schedulers to manage runtime system parameters and tasks. In ASAFESSS, programs are divided into different types of tasks, as a reflection of quality of the operation, in order to execute by the compiler. Task schedulers keep monitoring these tasks while scheduling and then make the optimization decisions based on adaptive engine.

2.3 Toward a Self-aware System for Exascale Architectures

As computing move toward exascale architectures, current existing resource management techniques are no longer sufficient for intersecting goals such as performance, energy consumption and reliability. Landwehr et al. has proposed a Self-aware System for Exascale Architectures [5] to tackle this challenge. In their work, self-awareness has three main objectives: Performance, Energy Consumption and Resiliency. While instruction count metrics, observed directly from Performance Monitoring Unit PMU, are used for evaluating systems performance, they are also combined with instruction energy cost metrics in order to estimate energy consumption, and from a reliability point of view, counts of correctable errors are used for enhancing resiliency. In addition to information from PMU, observations from sensors and etc. are combined with goals determined by user in order to make the optimization decisions. Eventually, based on the decision that system has made modifications like frequency changes, moving data, changes in states, scheduling policies and etc. would be applied to system through power control, runtime and so on.

2.4 Codelet Program eXecution Model

Codelet Program eXecution Model (PXM) [9] is a fine-grain event-driven execution model designed for exascale applications based on the dataflow theory [2]. Codelet PXM has two levels of granularity, Codelets and Threaded Procedures (TP). A Codelet is a set of machine instructions that scheduled atomically as a basic unit of computation. Compared to traditional Thread, Codelets are more fine grain and also non-preemptive. Therefore, when a Codelet is allocated and scheduled to a Computation Unit (CU), that CU will be busy until the Codelets completion. A Codelet becomes enabled once all required tokens present on each of its input arcs and it could be fired (executed) when it has all required resources and scheduled for execution. A TP is the second level of granularity in the Codelet PXM and its a container for codelets. It can serve not only as a label to invoke a set of Codelets, but also as a frame for efficient operation of Codelets while increasing the locality of shared data.

3 Problem Statement

While technology is leading computer engineers to design exascale systems with more resources and cores, according to complexity of functions, shrinking devices size, capacity expansion and other existing constrains there would be a variety of measures to be taken into account [Cite borkar] and performance is not the main concern any more [1].
It is our strong believe that the solution to these problems would be through a fine-grained, event-driven system which is also expected to be adaptive and proactive.
Our aim is to combine the Observe-Decide-Act (ODA) loop and the Codelet

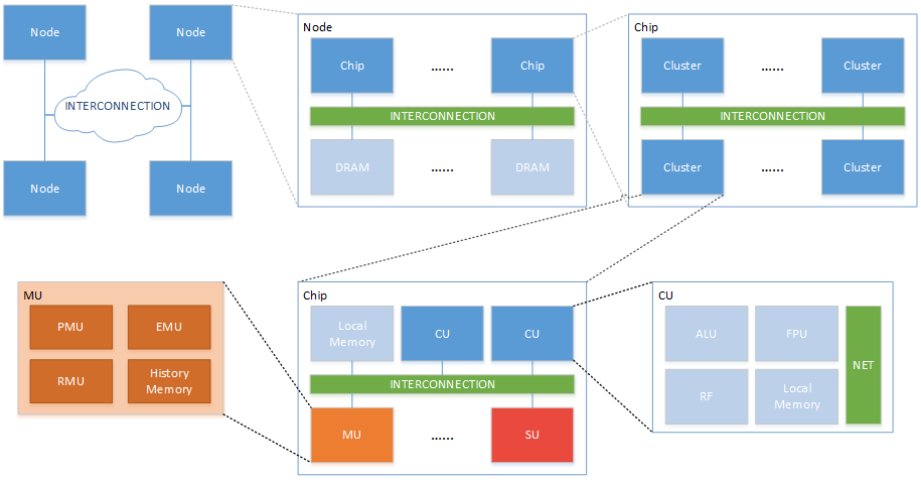

Fig. 1. New architectural elements in the abstract machine

PXM in order to design a Self-aware Codelet system for exascale architectures. In our approach, system would be evaluated in three domains: Performance, Energy Efficiency and Resiliency. Our goal is to optimize systems resource management and scheduling policies, with respect to the trade-off between maximizing performance, minimizing energy consumption and enhancing reliability.

4 Design of a Self-Aware Codelet Framework

4.1 New architectural elements

This section will introduce and discuss about the infrastructure of the self-aware codelet based runtime system. As a design but not an implementation, we focus on the abstraction of the system, how its structure should look like, what kinds components model it should have and so on. All these are to achieve the desired requirements and features as we plan. We build our system based on the Codelet Programming eXecution Environment (PXE).

Within an extrascale architecture containing thousands of cores, it is necessary to have a hierarchy abstraction model. The highest level of an extrascale system consists of many nodes and interconnection among them. Each node can break into several chips connected to each other through interconnection and data storage. The next level is clusters that compose a chip plus interconnection and memory. The inside of a cluster is the lowest level of the abstract architecture. A cluster consists of multiple basic components including computing units (CUs), synchronization units (SUs), memory, interconnection and an additional but important component, a monitoring unit (MU). A computing unit is used to proceed different sets of instructions within a codelet. Because it mainly deals with logical processes, there are usually arithmetic logic units (ALUs),

local memory, floating-point unit (FPUs) as well as some network functionality inside a computing unit. Computing units are designed solely to executing, therefore all computations about scheduling, handling interrupts and management of resource are offloaded from the CUs to SUs. Compared with CUs, SUs have additional knowledge of managing resource, making scheduling decisions and communicating with other SUs in different cluster. However, SUs can also work as a computing unit if needed. Beside these basic ideas, we require some specific components added to the system to help achieving our goals.
Differed from many other architecture designs, we focus on performance, energy consumption and resiliency as three quality attributes we concern about our system. To observe accurate information when executing an application or to estimate and foresee the latency features of an application before it start running, we employ several components to collect data, which include performance monitoring unit (PMU), energy monitoring unit (EMU) and resiliency monitoring unit (RMU).
PMU - Performance Monitoring Unit serves as an important role in achieving performance goals. In real hardware implementation, the PMU will connect to hardware sensors to collect information about performance, e.g. instructions counters and cycle counters, and collect bandwidth usage information from cores.
EMU - Energy Monitoring Unit works as an observer of energy consumption and as a controller that controls components power. EMU collects data from temperature observers and contains a matrix to estimate energy consumption through approximation according to the number of instructions, the type of an instruction and the average energy consumption of different types of instructions. The temperature gained can implicitly indicate the energy consumption.
RMU - Resilience Monitoring Unit is used to collect and store the fault rate of a codelet and reliable level.

4.2 Codelets Meta-data

In order to minimize real-time computations, some useful information can be computed offline for each of the codelets. These information or properties of each codelet can be used further to assign codelets to the most suitable core among all available cores of the system. Accordingly, for each codelet, three properties are to be computed:
Criticality: Based on the number of dependents of a codelet, the criticality would be determined. During the program execution, a codelet with many dependents (i.e. a Load/Store codelet) is critical in a sense that it has a considerable effect on systems performance. Therefore, this codelet should be allocated and scheduled to a core which can execute it quickly compared to the other cores. Another measure to be taken into account is the number of instructions in a codelet. For instance, the more instructions mean the higher price of re-execution. Therefore, such codelets should be executed on the healthiest cores of the system to avoid re-execution.

Deadline: In some applications, specifically real-time applications, system should meet deadlines. These time constrains can be defined by nature of the

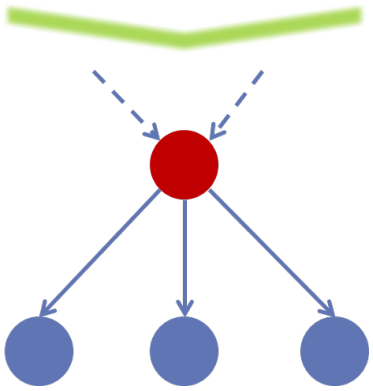

Fig. 2. a critical codelet in the program execution flow

application, and in some cases directly by user.
Intensity: During the offline computations, a new tag can be assigned to each of the codelets. According to types of instructions in a codelet, the codelet can be tagged as Floating Point Intensive, Load/Store Intensive or Arithmetic/Logic Intensive. These labels can be used for making optimization decisions for Performance and Energy Consumption.

4.3 Historical data for codelets

To design a Self-aware framework, system is expected to be not only Self-adaptive but also introspective. In order to make our system proactive, it should be able to use the results observed from execution of codelets in past. Once a codelet has been allocated, scheduled and executed, system will record a set of observations and store them in the historical database of codelets. System has one database of codelets per level of hierarchy and they include:
Codelets Meta-data Database should include executed Codelets meta-data and they can be used as index for codelets.
Instruction Per Cycle By observing the number of instruction counts and cycle counts provided by the Monitoring Unit, Instruction Per Cycle (IPC) can be computed and then stored in the database.
Temperature The temperature of chip during the execution of codelet should be stored in the database.
This information stored in tables (one per levels of hierarchy) and each time a new codelet is going to be allocated and scheduled to a cluster or core, system will choose the suitable cluster or core based on the existing data in tables and codelets meta-data.

4.4 Historical data for cores

In addition to the properties of codelets, historical data for each core is considered in the offline computations. Data collected by Monitoring Unit (MU) during executions of codelets in the past on each of the cores would be stored in the historical tables. Having this information and also the meta-data for each codelet, they can be used for choosing the right core in order to schedule a codelet on. For each core we would have a history of:
Fault rate Having the number of executed instructions and also the number of detected fault, the fault rate of a core can be computed and stored.
Core diagnosis Based on the sequences of fault detections, reliability level of each core would be determined. In addition, a core can be labeled as an unhealthy or healthy core with respect to fault rate and fault detection sequences.
Temperature The temperature of core during the execution of codelets should be stored in the historical table.

4.5 Achieving Self-awareness

We divide the goal of achieving self-awareness of our system into two parts, achieving proactivity and achieving self-adaptivity. Proactivity means the system is able to make proactive decisions once it learns a new application. Specifically, after an application is broke into several threaded procedures and assigned to clusters, the clusters can use the codelet metadata provided to predict some properties of the task, e.g. the number of resources needed. Self-adaptivity refers to the ability that the system can respond and adapt to changeable events occurred, e.g. the change of a performance goal of the application. We will further discuss being proactive and being self-adaptive below. The reason we self-awareness into these two parts is that proactivity and self-adaptivity are two important phases of a runtime system. After a program is compiled and broke into small executable pieces, which are called threaded procedures in a codelet PXE, by the compiler, its execution mission is passed to the runtime system. RS need to use the information provided to allocate enough resources to these TPs and assign them to fitted cluster to execute. Proactivity is achieved in the phase in which the RS assigns TPs to clusters. Once they are executing, self-adaptivity needs to be achieved.

Proactivity In the prior sections, we have discussed a lot about codelet metadata and historical data. These contain general information about codelets executing properties and the system, especially every clusters status. Knowing these enables the system to predict an upcoming applications properties as well as to evaluate its internal changes so that it can try to avoid system failing from happening.

Predicting a threaded procedure's runtime properties The codelet metadata exposes codelets possible feature. According to these, a cluster can adjust itself to befit the ready to execute codelet. For example, when a cluster knows that the threaded procedure just assigned to it has a maximum value of width, 5,

the cluster may predict that the number of resource this subroutine needed will approximately be no more than five at a time. Hence, it can power-gate other cores and only leave 5 cores to stay enabled so that it can reduce the power consumption of unused components.

Evaluate systems status and adjust itself accordingly Evaluating itself is an important feature of the system. Using the trends of historical data, the system estimates its current status and then adjust itself to avoid negative effects from the current status. If a cluster learns from its historical fault rate that its fault rate is increasing, it will assume that it has high fault rate currently and is likely to exceed the predefine threshold. The adjustments made accordingly can be the change of resource allocation policy, or the change of energy budgets. E.g., the cluster will avoid assigning any codelet to those CUs that seem have great probability to fail, or even power-gate these CUs and distribute the additional energy budgets it has to nearer clusters.

Self-adaptivity The basic idea is to form an ODA loop using those components we have discussed at the beginning of this chapter. An ODA loop contains an observing part, a deciding part and an acting part. In our case, SU, MU along with PMU, EMU and RMU form a control engine that has the whole three parts of an ODA loop. PMU, EMU and RMU from a core gather the information observed by them in the core and send to the SU and MU in the cluster. Based on what have been observed, SU and MU together decide the adjustment needed for optimization. Then they will apply those changes. The three self-adaptive properties of the system, performance, energy consumption and resilience will be discussed separately about how it is achieved.

Achieving Performance Performance is the basic measurement of a system. It mainly takes the time efficiency as an important indicator to present an executing applications performance, which means the more quickly the system completes an execution, the higher performance it has. Although performance is no more the only measurement of a system and there is no doubt that modern architecture can easily achieve high performance, it is still very important. There are simple ways to improve performance, e.g. incrementing the frequency of a core. Besides, the codelet-based system we design is able to achieve high performance through providing a fine-grained, event-driven parallelism. However, the mechanisms for codelet scheduling and resource allocation within our system are important and should be well defined. For the aspect of self-adaptivity, the system should have the ability to optimize the scheduling mechanism.
The PMU keeps observing instruction counts, cycle counts and bandwidth usage. After information arrives at the MU, the MU first compares the observed result with predefined thresholds and calculates the difference between them. Then, if the result shows the performance goal is no met, MU and SU will work together to decide how to improve the performance or even adjust the scheduling mechanism.

Achieving Energy consumption As mentioned in the beginning of this report, energy efficiency is becoming more and more important in exascale machines because the energy consumption of thousands of cores is enormous. Traditional

technologies of energy consumption quickly become unsupportable. The basic idea of reducing energy consumption is to shut down cores quickly when they become not useful or to execute a routine on a core that costs less. The system also needs to take care of the cores that tend to exceed the power threshold. If necessary, the system will clock-gate or power-gate some clusters to reduce the whole systems energy consumption, even though such approach will harm the performance. Hence, determining which cores can be shut down or allocated to codelets execution and when they need to be clock-gated or power-gated is the major task of our system to achieve energy consumption goal. Since determining which cores can be shut down has been completed in the proactivity phase, the self-adaptivity phase will focus on determining the low cost cores and dealing with excess energy consumption.
The first thing for reducing energy consumption is to monitor the energy consumption. We employ several hardware sensors to observe the hardware temperature and count number and types of instructions executed. We originally planned to have hardware sensors to monitor energy consumption directly. However, it is hard for a sensor to monitor the energy consumption for each core directly. Therefore, the system estimates the energy consumption for each core and cluster using temperature and instructions count indirectly instead.
According to the result, MU will compare every optimization method it has and finally come out the decision that which method is the best in current situation. Then it will apply the adjustments from that method, e.g. power-gating a core.

Achieving Resiliency Resiliency is the last property of our system. Being resilient means the system can cope with failures. To achieve resilience, the first thing we need is error detection. We try to detect error that occurs in an applications execution through comparison among the results from duplication copies. Specifically, there are level of error detection in our system, threaded procedure level error detection and codelet level error detection. In threaded procedure level error detection, the system will duplicate the target threaded procedure at least once and compare the result after two threaded procedures finish executing. In codelet level error detection, comparisons happen each time a codelet and its duplicate copies finish execution. Using software to detect errors will definitely result in the problem of energy consumption. Hence, we are trying to relieve the side effect of energy consumption by dynamically controlling the number of duplicate copies generated so that the cost of energy will reduce.
If resiliency is needed by an application, the system will generate one copy of the application by default. As mentioned in prior section, once a fault is detected, it will be stored. MU will use the fault rate collected by the RMU to decide whether more copies are needed.

4.6 Interaction and precedency

As the system provides three adaptivity properties for applications, two serious problem are how these properties affect each other and how to prioritize these properties if they cannot be achieved at the same time.

Goal\ Impact	Performance	Energy	Resiliency
Performance		The faster tasks are executed, higher energy levels are reached.	Additional tasks can be executed faster.
Energy	The less energy the slower tasks are executed.		Additional computations are delayed.
Resiliency	Additional tasks that will cause the application to run slower.	More tasks to be executed, higher energy consumption.	

Fig. 3. Trade-off between three objectives of the system

4.7 Interaction among the three adaptivity properties

Each time we try to improve one of the three properties, it will have side effect on achieving other two properties. For the aspect of performance, if the application is required to execute with a high performance, it will be hard to lower the energy consumption. Though we can still power-gate those unused cores, we cannot DVFS the executing cores. However, requiring high performance doesnt affect achieving resilience much. For the aspect of energy, once the application is required to consume as little energy as it can, the system will simply DVFS those executing cores and let them execute at the slowest way. This is an extreme example to present the side effect on achieving performance goal. For the similar reason, additional computations for resilience are delayed or disabled. The side effects from achieving resilience have been discussed when discussing performance and energy consumption.

4.8 Multiple prioritizations

As discuss above, achieving each property will have side effect on other properties. We address this problem through defining a prioritization list. The goal for the property with the highest prioritization will be achieved firstly. Then the second prioritized propertys goal will be achieved. Then is the third one. If there is collision happened, the system will strictly follow the order of the prioritization list. Application are encouraged to make their own prioritization list of the three properties. The default prioritization list of the system will be: energy consumption, performance, resilience.

4.9 Self-adaptive procedures and examples

To visualize how the system achieves self-awareness, some examples has been provided.

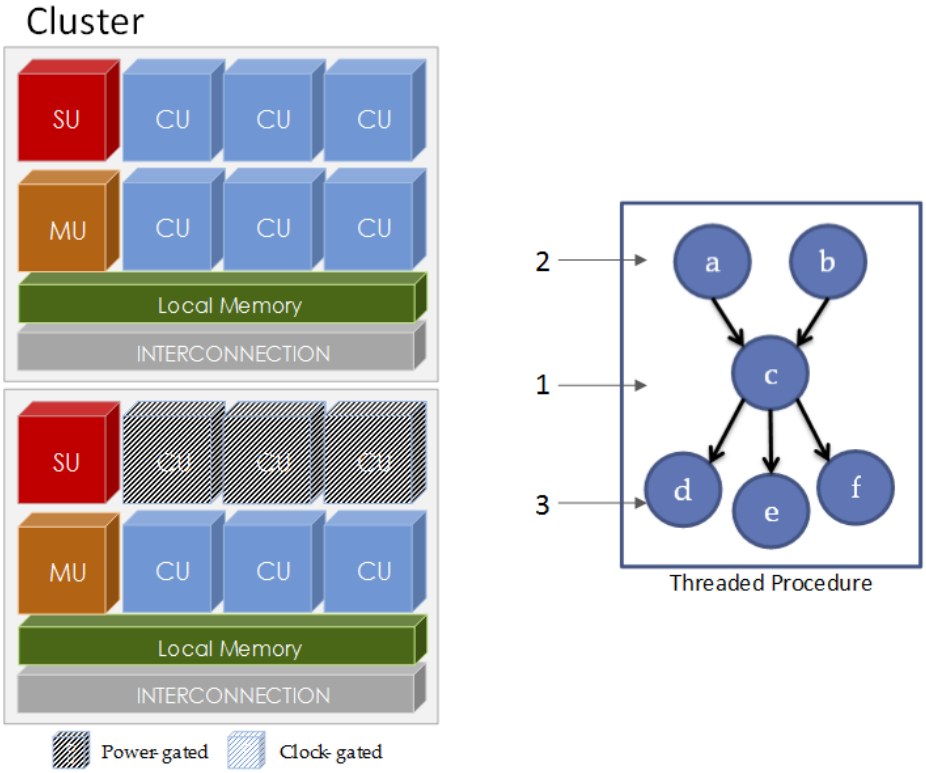

Fig. 4. An example of Proactivity

Assume that a cluster is idle with 6 CUs, a SU, a MU, local memory and interconnection. A threaded procedure with codelet graph showed in the figure is assigned to it. Once the SU gets the data about the codelet graph, it will scan through all the nodes and dependencies inside it. At the same time, it calculates that the maximum value of the width is 3. According to the result, MU decides to power-gate 3 of its 6 CUs and only leaves 3 CUs enabled to working. Therefore the cluster will not consume extra energy.

Assume the cluster has detected that energy consumption of a CU is exceeding the threshold. Based on the information, the MU evaluates every possible approach to reduce the energy consumption. E.g. MU can simply power-gate that core, or MU can complexly enable a power-gated core to immigrate and continue the execution and clock-gate the core that exceeds power threshold.

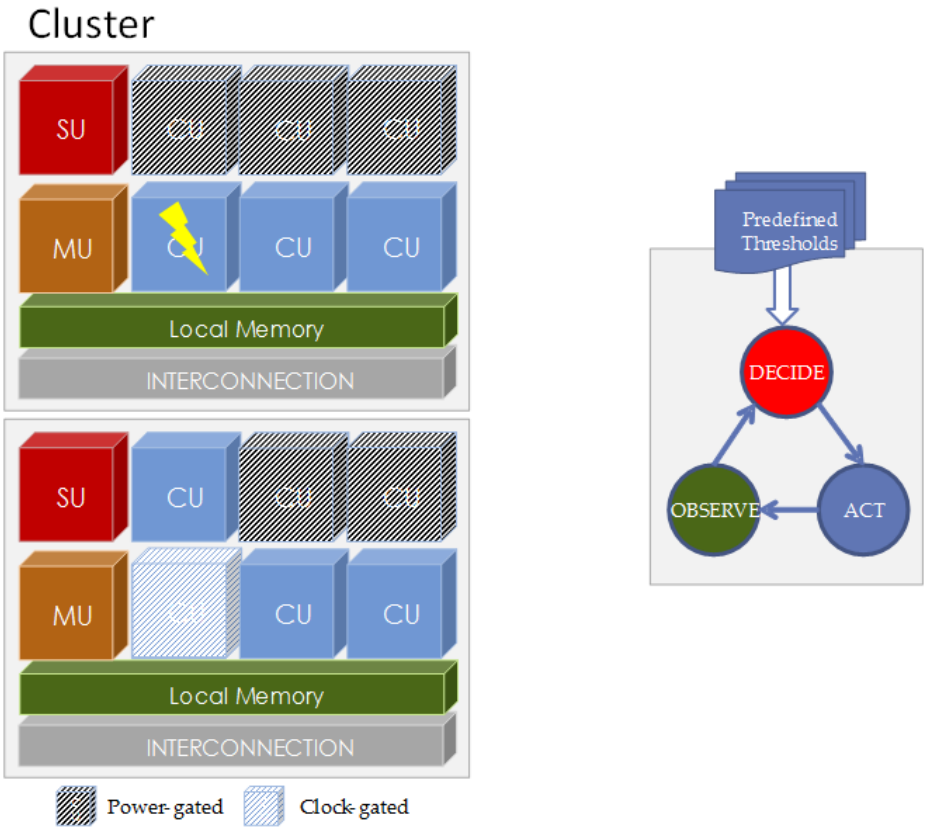

Fig. 5. An Example of Self-adaptivity

5 Related work

In recent years, many attentions have devoted into using control theory to achieve self-adaptive or self-aware system. As mentioned before, control theory is consisting of three parts. Observation part keeps monitoring the execution performance of applications using feedback from system and defines the deviation between current performance and desired performance. Decision part changes and optimizes the allocation of system resources according to the observed result to achieve dynamic (e.g. time varying) requirements and energy consumption goals. Action part applies these optimized changes. Self-adaptive systems use monitor (for observation), control engine (for decision) and system interface (for action) to keep optimizing the system resources allocation and somehow result in the balance between performance and energy consumption. Although control theory has been widely employed in many systems to build adaptive loop, many of the contributions are limited in some certain scopes. E.g., some works focus on WEB

server domain, optimizing request transactions or dealing with connection delay, because of the special characteristics in such domain that generally other systems do not expose. Some remarkable proposed works try to fully exploit control theory for general purpose.

Control Theoretical CPU Allocation In a proposed solution [6], which employs control theory with limitation that the design only deals with one single resource allocation, the CPU allocation. The aim is to meet a set point (e.g. a goal of application performance) through rescaling the CPU allocation. Application Heartbeat framework [3] is used as a measurement of performance. It will omit a heartbeat when the application's execution reaches an important point. The interval of two heartbeats indicates a performance level of the application. Then control system is developed to model the performance level of an application and improve it through allocating more CPU resource to it.

SEEC Self-aware computing framework (SEEC) [4] focuses on its generality and extensibility so that it can handle unseen applications and system components changes. SEEC decouples the ODA loop into separate parts, still the observation, the decision and the action. By decoupling the ODA loop, SEEC enables application programmers and system developers to specify observations and actions separately. Application programmers will specify the desired goals of their application and system developers will actions and models available in the system. Based on the observation part and action part well defined, SEEC adds adaptive feedback control methods and reinforcement learning to enhance its control engine. Adaptive control methods enable SEEC to deal with the dynamic changes of the application goals and reinforcement learning provides SEEC with the ability to handle possible system's resource changes (e.g. processor's frequency change) as well as hardware faults.

Comparison SEEC has only one prioritization for optimization, which means that SEEC prioritizes energy consumption than other features. In addition, SEEC is not designed for extreme scale machines, which may limit it on befitting extreme scale systems. Differed from SEEC, our self-aware system targets on exascale architecture. It has three adaptivity properties, performance, resilience and energy consumption, and it has several optimization prioritizations that can be determined by the application for handling the interaction among three features.

6 Conclusion

This report presents a Self-aware framework based on the codelet model. We believe the solution for challenges in exascale systems would be through a combination of compile time computation and real-time observations. Based on the observations of system, and also threshold and prioritization defined bu user, the interaction between three objectives of system could be handled.

References

1. Borkar, S.: Thousand core chips: A technology perspective. In: Proceedings of the 44th Annual Design Automation Conference. pp. 746–749. DAC '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1278480.1278667
2. Dennis, J.B.: First version of a data flow procedure language. In: Programming Symposium, Proceedings Colloque Sur La Programmation. pp. 362–376. Springer-Verlag, London, UK, UK (1974), http://dl.acm.org/citation.cfm?id=647323.721501
3. Hoffmann, H., Eastep, J., Santambrogio, M.D., Miller, J.E., Agarwal, A.: 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. pp. 79–88. ICAC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1809049.1809065
4. Hoffmann, H., Maggio, M., Santambrogio, M.D., Leva, A., Agarwal, A.: Seec: A general and extensible framework for self-aware computing (2011)
5. Landwehr, A., Zuckerman, S., Gao, G.R.: Toward a self-aware system for exascale architectures (2013)
6. Maggio, M., Hoffmann, H., Agarwal, A., Leva, A.: Control-theoretical cpu allocation: Design and implementation with feedback control (2011)
7. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 14:1–14:42 (May 2009), http://doi.acm.org/10.1145/1516533.1516538
8. St. John, T., Meister, B., Marquez, A., Manzano, J.B., Gao, G.R., Li, X.: Asafesss: A scheduler-driven adaptive framework for extreme scale software stacks. In: Proceedings of International Workshop on Adaptive Self-tuning Computing Systems. pp. 21:21–21:23. ADAPT '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2553062.2553063
9. Zuckerman, S., Suetterlein, J., Knauerhase, R., Gao, G.R.: 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. pp. 64–69. EXADAPT '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2000417.2000424