Toward a Self-aware System for Exascale Architectures

Aaron Landwehr, Stéphane Zuckerman, and Guang R. Gao

Department of Electrical and Computer Engineering
University of Delaware, United States
aron@udel.edu, {szuckerm,ggao}@capsl.udel.edu

Abstract. High-performance systems are evolving to a point where performance is no longer the sole relevant criterion. 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. It is our opinion that systems will need to become more introspective and self-aware with respect to performance, energy, and resiliency. In this *position paper*, we explore the major hardware requirements we believe are central to enabling introspection and self-awareness, as well as the types of interfaces and information that will be needed for such runtime systems. We also discuss 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 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, an exascale computer system would consume over 1.5 GW of power [18]. 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 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 [7]. 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.

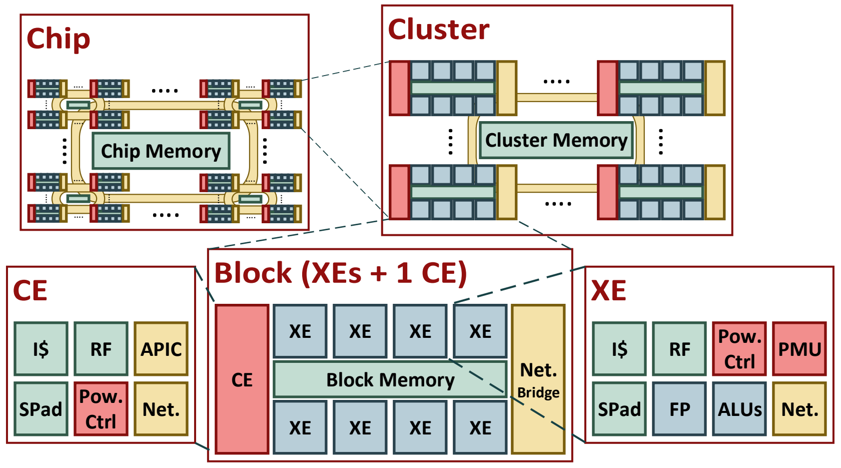
This *position paper* proposes the following contributions: (1) we evaluate and motivate the important hardware requirements for a truly self-aware and introspective system (using a specific target architecture), and (2) we provide a vision and discuss important research venues for self-aware exascale systems.

This paper is organized as follows: Section 2 discusses our target exascale architecture; Section 3 evaluates the hardware requirements essential for an introspective system software; Section 4 provides research venues for self-adaptation at the exascale level; Section 5 discusses the related work; and we conclude in Section 6.

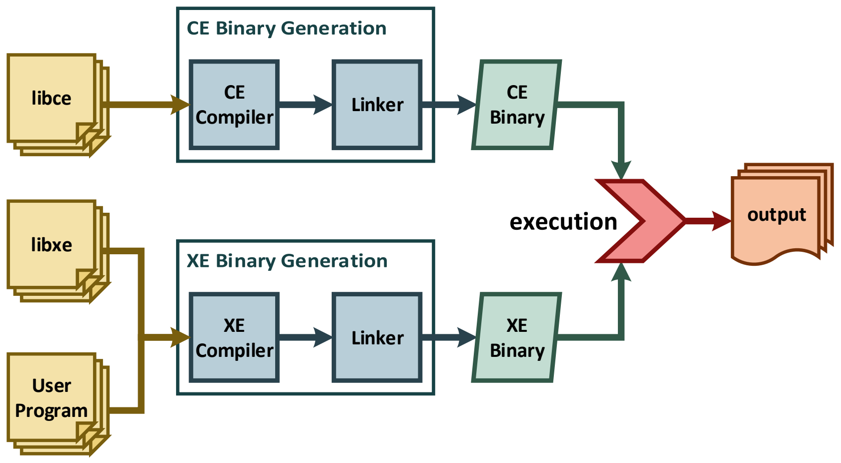
2 Background

Target Exascale Architecture and Associated Toolchain For the purpose of this discussion, we introduce our target exascale architecture (TEA) which is similar to the one described by Knauerhase et al. [11]. At the lowest level, the TEA is organized into blocks consisting of a Control Engine (CE), several eXecution Engines (XEs), and a block shared memory. An XE's fundamental usage is to execute arbitrary code without interruption. The CE on the other hand is designed to control and schedule work to a number of local XEs within a given block. A detailed description of the TEA is provided in Figure 1(a). Both XE and CE cores contain a set of identical components: some local memory (for data and code), ALUs, a clock/power gate control unit, and some network functionality. In addition, XEs also feature a floating-point unit, as well as a performance monitoring unit (PMU). 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. A detailed description of the TEA toolchain is shown in Figure 1(b).

Codelet Program Execution Model and System Software Model The TEA 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 to decide which program execution model should be used for exascale systems. One such PXM is the codelet model [24], which has already been implemented in various ways [13,21]. It features an abstract machine model that fits very well with our TEA.



(a) Our Target Exascale Architecture consists of heterogeneous cores in a hierarchical configuration with network interconnects at each level, organized into blocks, clusters, and chips. The exact number of cores and levels is not important for our discussion.



(b) Target Exascale Architecture Toolchain: XEs and CEs are specialized and may have a distinct instruction set architecture. As a result, a special compiler and linker may need to be used to produce the binaries for each. Each has a library providing a basic runtime system. In the case of an XE, the library is linked with the user program.

Fig. 1. Our Target Exascale Architecture and associated toolchain.

3 Requirements for Self-adaptation

In this section, we discuss the underlying hardware requirements in order to implement a self-adaptive, self-aware system. Many of these will be integral for 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. We target three main objectives for self-aware systems: energy/power efficiency, performance, and resiliency. However, the benefits of a tailored performance monitoring unit are discussed first.

Role of the PMU in Exascale Systems Before moving on to the detailed requirements in the subsequent subsections, we 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 proactive monitoring for potential issues and as a mechanism to determine whether the system software should be cautious with the work it is scheduling. Being 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.

Energy For any large-scale computer system (including current petascale systems), 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/remote reads, 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.

Our TEA is expected to be capable of adjusting the state of FUs, at least at a functional unit block (FUB) granularity – which is at a finer granularity than FUs. To motivate this, consider 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.

Performance We expect the TEA's system software to 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 codelet level.

Using the PMU events described, performance within the runtime can be evaluated. For 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.

Resiliency 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 duplicating 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.

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.

4 Research Venues

In this section, we discuss our vision of exascale self-adaptation. At a high level, the runtime system of exascale architectures will need to make intelligent decisions to control various hardware features to reduce power consumption, as well as to schedule tasks and move data. First, we start with a discussion of adaptation at a fine-grained level and from there move toward a discussion of adaptation at a more coarse-grained level.

Fine-grained Adaptation An exascale architecture must include power management at a fine-grained level. We foresee adaptation occurring through an ODA mechanism as shown in Figure 2. 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. Finally, various actions will occur in the form of adjustments to hardware state or some type of software change.

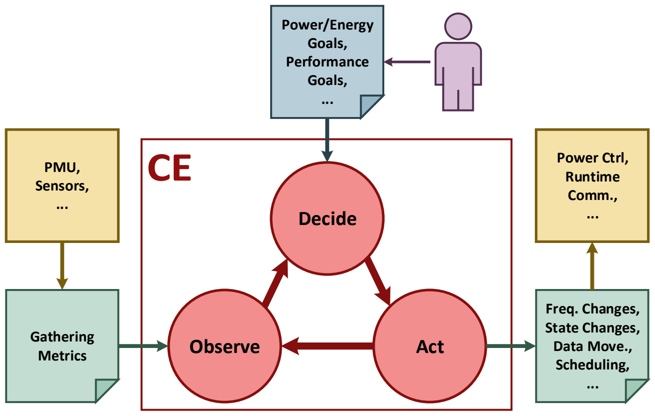

Power/Energy Goals, Performance Goals, ...
PMU, Sensors, ...
CE
Decide
Power Ctrl, Runtime Comm., ...
Gathering Metrics
Observe
Act
Freq. Changes, State Changes, Data Move., Scheduling, ...

Fig. 2. Mapping our target exascale architecture to an observe-decide-act mechanism.

Some of the decisions that we foresee are whether to clock gate or power gate components, as well as whether to adjust the frequency of components. For a small example of adaptation within our TEA, consider Figure 3. 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.

Coarse-Grain Adaptation At a coarse-grain level, a form of hierarchical management will need to occur. For this, there will need to be a communication subsystem between CEs in order to communicate system state. 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.

5 Related Work

Below we discuss other previous work in self-adaptation. The approaches fall into three categories: "Application-Centric Adaptation," "Component-Centric Adaptation," and "System-Centric Adaptation."

Application-Centric Adaptation Application-centric approaches focus on adaptation for specific applications or a subclass of applications within a given domain. Quality of service (QoS) is one large area of active research [1,9] due to the real time requirements and the ever-changing field of computing resources. Other works focus on changing application specific algorithm policy [22]. Some approaches are more akin to toolkits designed for application programmers to use to enable adaptation within their software [6].

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 [10], dynamic reconfiguration of memory hierarchies [3], and self-optimizing memory controllers [8].

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 adapted to cluster-like environments [20] are mainstream OSes customized for high-performance needs. Lightweight kernels [4,5] implement a reduced set of features directly accessible to the application programmer and forward calls for missing features to a "fuller" OS. Micro kernels [12,15,17] implement basic OS services (memory addressing, process management, inter-process communications) as privileged code and implement every other service as some form of unprivileged library [2]. Finally, high-level runtime systems implement some form of resource management on top of the host OS [14,19,23] or expose some form of languages (or frameworks) that provide mechanisms for an application to adapt [7,16].

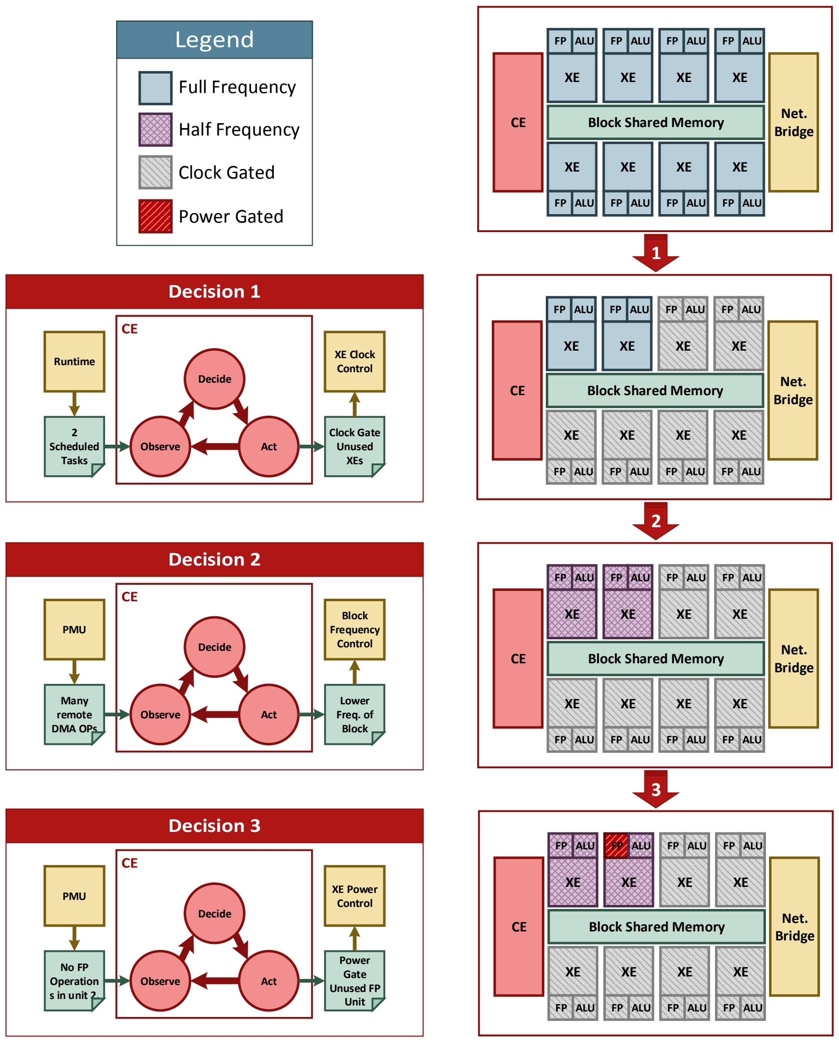


Fig. 3. Target exascale architecture adaptation example.

Discussion There are a number of shortcomings that need to be addressed for exascale architectures. Application and component-centric approaches lack a holistic 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.

6 Conclusion

This paper takes the position that, due to the draconian requirements of future exascale systems with respect to performance, energy, and resiliency, only a proactive, self-aware, adaptive system will be able to correctly manage resources, both at the global, coarse-grain level, as well as at the finest-grain level. To this end, a target exascale architecture was presented, along with the challenges to overcome, and the research venues to solve those problems.

Acknowledgment

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717. This work was partly supported by European FP7 project TERAFLUX, id. 249013.

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. 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.
4. 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.
5. 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.
6. A. Goel, D. Steere, C. Pu, and J. Walpole. Swift: A feedback control and dynamic reconfiguration toolkit. Technical report, Oregon Graduate Institute, 1998.
7. H. Hoffmann. *SEEC: A Framework for Self-Aware Management of Goals and Constraints in Computing Systems.* PhD thesis, Massachusetts Institute of Technology, February 2013.
8. 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.
9. 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.
10. 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.

11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. J. Suetterlein, S. Zuckerman, and G. R. Gao. An Implementation of the Codelet Model. In *Euro-Par 2013*, 2013.
22. 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.
23. 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.
24. 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.