Enabling Technologies For Self-Aware Adaptive Systems

Marco D. Santambrogio^{1,2}, Henry Hoffmann^1, Jonathan Eastep^1, Anant Agarwal^1

^{1}Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139
{eastep, wingated, santambr, agarwal}@mit.edu

^{2}Politecnico di Milano
Dipartimento di Elettronica e Informazione
20133 Milano, Italy
marco.santambrogio@polimi.it

***Abstract*—Self-aware computer systems will be capable of adapting their behavior and resources thousands of times a second to automatically find the best way to accomplish a given goal despite changing environmental conditions and demands. Such a capability benefits a broad spectrum of computer systems from embedded systems to supercomputers and is particularly useful for meeting power, performance, and resource-metering challenges in mobile computing, cloud computing, multicore computing, adaptive and dynamic compilation environments, and parallel operating systems.**

Some of the challenges in implementing self-aware systems are a) knowing within the system what the goals of applications are and if they are meeting them, b) deciding what actions to take to help applications meet their goals, and c) developing standard techniques that generalize and can be applied to a broad range of self-aware systems.

This work presents our vision for self-aware adaptive systems and proposes enabling technologies to address these three challenges. We describe a framework called Application Heartbeats that provides a general, standardized way for applications to monitor their performance and make that information available to external observers. Then, through a study of a self-optimizing synchronization library called Smartlocks, we demonstrate a powerful technique that systems can use to determine which optimization actions to take. We show that Heartbeats can be applied naturally in the context of reinforcement learning optimization strategies as a reward signal and that, using such a strategy, Smartlocks are able to significantly improve performance of applications on an important emerging class of multicore systems called asymmetric multicores.

I. Introduction

Resources such as quantities of transistors and memory, the level of integration and the speed of components have increased dramatically over the years. Even though the technologies have improved, we continue to apply outdated approaches to our use of these resources. Key computer science abstractions have not changed since the 1960's. The operating systems, languages, etc we use today, were designed for a different era. Therefore this is the time for a fresh approach to the way systems are designed and used. The Self-Aware computing research leverages the new balance of resources to improve performance, utilization, reliability and programmability [1, 2].

Within this context, imagine a revolutionary computing system that can observe its own execution and optimize its behavior around a user's or application's needs. Imagine a programming capability by which users can specify their desired goals rather than how to perform a task, along with constraints in terms of an energy budget, a time constraint, or simply a preference for an approximate answer over an exact answer. Imagine further a computing chip that performs better according to a user's preferred goal the longer it runs an application. Such an architecture will enable, for example, a hand-held radio or a cell phone that can run cooler the longer the connection time. Or, a system that can perform reliably and continuously in a range of environments by tolerating hard and transient failures through self healing. Self-aware computer systems [3] will be able to configure, heal, optimize and protect themselves without the need for human intervention.

Abilities that allow them to automatically find the best way to accomplish a given goal with the resources at hand. Considering the similarity to organic nature we will refer this new class of computer systems as Organic Computing System (OCS). These systems will benefit the full range of computer infrastructures, from embedded devices to servers to supercomputers. Some of the main challenges in realizing such a vision are: to add auto-adaptability capabilities to organic devices, to implement distributed self-training algorithms over such architectures, and to specify and formulate application solutions using such a computing paradigm.

To realize this vision of an Organic Computing System, we must a) enable applications to specify their goals, b) enable system services to determine whether these goals are met, and c) enable adaptive systems to make informed decisions among multiple possible actions. Tackling the first two of these challenges requires a general framework allowing a wide range of applications to express their goals while enabling adaptive systems to read these goals and measure the applications' progress toward them. Addressing the third challenge requires imbuing adaptive systems with the ability to navigate a vast and interconnected decision space. Furthermore, these challenges should be met using techniques that are generalizable, repeatable, and portable.

We address the first two of these challenges by presenting the Application Heartbeats framework (or Heartbeats for short), which provides a general, portable interface for applications to express their performance goals and progress towards those goals [4]. Using Heartbeats applications express their performance using the well-known abstraction of a heartbeat; however, the API also allows applications to express their goals as a desired heart rate (for throughput oriented applications) or a desired latency between heartbeats (for latency sensitive applications). We find the use of Heartbeats within an application allows adaptive systems to make decisions using a direct measure of an application's performance rather than trying to infer it from underlying hardware counters. Additionally, by specifying application goals, the Heartbeats framework allows adaptive systems to perform constraint-based optimizations like minimizing power for a minimum performance target.

One promising approach to the third challenge, that of making informed decisions, is to use machine learning in adaptive systems. We explore this approach in an adaptive, self-aware spinlock library called Smartlocks [5]. Smartlocks use an adaptation referred to as *lock acquisition scheduling* to determine the optimal policy for allowing access to a critical section within an application. We find that guiding lock acquisition scheduling using machine learning allows Smartlocks to adopt a near perfect lock scheduling policy on an asymmetric multicore. Furthermore, Smartlocks are able to adapt their behavior in the face of environmental changes, like changes in clock frequency.

The rest of this paper is organized as follows. Section II identifies key system components of a self-aware systems and presents several works proposed in literature. Section III presents our vision on self-aware adaptive systems, proposing enabling technologies for adaptive computing that address these challenges. Section IV presents our experimental results. Finally, Section V concludes.

II. An Overview of Self-Aware Systems

One solution to overcoming the burden imposed by the increasing complexity and the associated workload of modern computing systems is to adopt self-adaptive [3] and autonomic computing techniques [1]. This work includes research on single- and multi-core architectures [6–9], networks [10], self-healing systems [11–13], self-monitoring for anomaly detection in distributed systems [14], automatic techniques to detect and cope with attacks [15] or faults in software systems [16], managing in cloud computing and grid [17–19], complex distributed Internet services [20, 21], self-healing and operating systems [22–28].

Within this context classical reconfigurable and multicore systems are moving to self-aware computing systems [29] where hardware components [7, 30–32], the applications [33, 34] and the operating system [26] can be made to autonomously adapt their behavior to achieve the best performance.

Tackling online monitoring and program optimization together, it is possible to obtain an efficient monitoring system able to improve operating system availability [35] with dynamic updates based on hot-swapable objects [32, 34]. The hot-swap mechanism is used to implement software reconfiguration in the K42 Operating System [26].

A self-optimizing hardware approach can be found in [30], where a self-optimizing memory controller is presented. This controler can optmize its scheduling policy using a reinforcement learning approach which allows it to estimate the performance impact of each action it can take to better respond to the observations made on the system state. Two interesting examples of adaptable chip multiprocessor architecture have been presented in [7] and [31]. In [7], a heterogeneous multicore architecture has been proposed which optimizes power consumption by assigning different parts of an application to the core that will have the best energy characteristics for that computation. This architecture has been designed taking into consideration the fact that typical programs go through phases with different execution characteristics. Therefore the most appropriate core during one phase may not be the right one for a following phase. In [31] the Core Fusion architecture is proposed. This architecture is characterized by the presence of different tiny independent cores that can be used as distinct processing elements or that can be fused into a bigger CPU based on the software demand.

III. Our Work Towards the Definition of Self-Aware Adaptive Systems

To achieve the vision described in the previous sections, a self-aware system must be able to monitor its behavior to update one or more of its components (hardware architecture, operating system and running applications), to achieve its goals. This paper proposes the vision of organic computation that will create such a self-aware computing system. An organic computer is given a goal and a set of resources and their availability, it then finds the best way to accomplish the goal while optimizing constraints of interest. An organic computer has four major properties:

- It is **goal oriented** in that, given application goals, it takes actions automatically to meet them;
- It is **adaptive** in that it observes itself, reflects on its behavior to learn, computes the delta between the goal and observed state, and finally takes actions to optimize its behavior towards the goal;
- It is **self healing** in that it constantly monitors for faults and continues to function through them, taking corrective action as needed;
- It is **approximate** in that it uses the least amount of computation or energy to meet accuracy goals and accomplish a given task.

More importantly, much like biological organisms, an organic computer can go well beyond traditional measures of goodness like performance and can adapt to different environments and even improve itself over time.

To adapt what the organic computer is doing or how it is doing a given task at run time, it is necessary to develop a control system as part of the system that observes execution, measures thresholds and compares them to goals, and then adapts the architecture, the operating system or algorithms as needed. A key challenge is to identify what parts of a computer need to be adapted and to quantify the degree to which adaptation can afford savings in metrics of interest to us. Examples of mechanisms that can be adapted include recent researches on memory controller [30] or on reactive synchronization mechanisms in which the waiting algorithm is tailored at run time to the observed delay in lock acquisition.

In the following we present our first work in defining enabling technologies for adaptive computing that address these challenges. Specifically, we present the Application Heartbeats framework [4], an open source project [36] which provides a simple, standardized way for applications to monitor their performance and make that information available to external observers. We illustrate how other components of the system can use Heartbeats to adapt and achieve better performance. We discuss an example of Heartbeat usage within Smartlocks [5], a self-aware synchronization library that adapts its internal implementation using reinforcement learning with the Heartbeat as the reward function.

A. Application Heartbeats

The Application Heartbeats framework provides a simple, standardized way for applications to report their performance and goals to external observers [4]. As shown in Figure 1, this progress can then be observed by either the application itself or an external system (such as the OS or another application). Having a simple, standardized interface makes it easy for programmers to add Heartbeats to their applications. A standard interface, or API, is also crucial for portability and inter-operability between different applications, runtime systems, and operating systems. Registering an application's goals with external systems enables adaptive systems to make optimization decisions while monitoring the program's performance directly rather than having to infer that performance from low-level details. If performance is found to be unacceptable, information gleaned from hardware counters can help explain why and what should be changed.

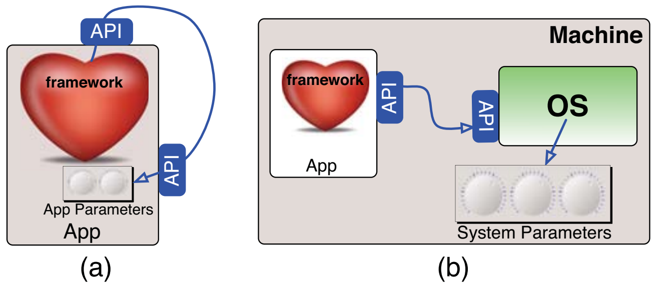

Fig. 1: (a) Self-optimizing application using the Application Heartbeats framework. (b) Optimization of machine parameters by an external observer.

The Application Heartbeats framework measures application progress toward goals using a simple and well-known abstraction: a heartbeat. At significant points, applications make an API call to signify a heartbeat. Over time, the intervals between heartbeats provide key information about progress for the purpose of application auto-tuning and/or externally-driven optimization. The Heartbeats API allows applications to communicate their goals by setting a target heart rate (*i.e.,*number of heartbeats per second) and a target latency between specially tagged heartbeats. Adaptive system services, such as the operating system, runtime systems, hardware, or the application itself, monitor progress through additional API calls and can then use this information to change their behavior and help the application achieve the specified performance goals. As an example, a video codec application could use Application Heartbeats to specify a target throughput of 30 video frames a second. In the encoder example, an adaptive scheduler could ensure that the encoder meets this goal while using the least number of cores, thus saving power or allowing extra cores to be assigned to other purposes.

B. Smartlocks

Smartlocks [5] is a self-aware system designed to help reduce the programming complexity of extracting high performance from today's multicores and asymmetric multicores. Smartlocks is a self-optimizing spin-lock library that can be used as the basis for synchronization, resource sharing, or programming models in multicore software. As it runs, Smartlocks uses Application Heartbeats [4] together with a Machine Learning (ML) engine to monitor and optimize application performance by adapting Smartlock's internal behaviors.

The key adaptable behavior within Smartlocks is the lock acquisition scheduling policy. *Lock acquisition scheduling* is picking which thread or process among those waiting should get the lock next (and thus get to execute the critical section next) for the best long-term effect. [5] demonstrates that intelligent lock acquisition scheduling is an important optimization for multicores with static and especially dynamic performance asymmetries.

As illustrated in Figure 2, Smartlock applications are C/C++ pthread applications that use pthreads for thread spawning, Smartlocks for spin-lock synchronization, and the Heartbeats framework for performance monitoring. Application developers insert heart beats at significant points in the application to indicate progress toward the applicaion's goals. Then, within Smartlock, the Heartbeats *heart rate* signal is used as a reward signal by a Reinforcement Learning (RL) algorithm. The RL algorithm attempts to maximize the heart rate by adjusting Smartlock's lock acquisition scheduling policy. The scheduler is implemented as a priority lock, and the scheduling policy is configured by dynamically manipulating per-thread priority settings.

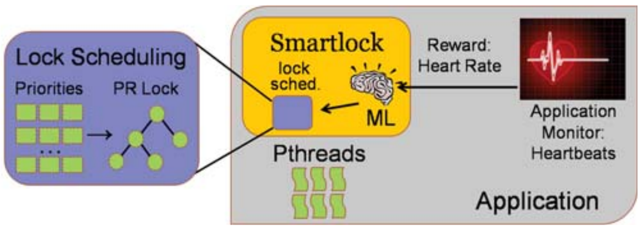

Fig. 2: Smartlocks Architecture. ML engine tunes Smartlock internally to maximize monitor reward signal. Tuning adapts the lock acquisition scheduling policy by configuring a priority lock and per-thread priority settings.

IV. Preliminary Results

This section presents several examples illustrating the use of the Heartbeats framework and Smartlocks. First, a brief study is presented using Heartbeats to instrument the PARSEC benchmark suite [37] and after that the benefits in using Smartlocks have been presented using a synthetic benchmark.

A. Heartbeats in the PARSEC Benchmark Suite

We present several results demonstrating the simplicity and efficacy of the Heartbeat API. These results all make use of our reference implementation of the API which uses file I/O for communication. Results were collected on an Intel x86 server with dual 3.16 GHz Xeon X5460 quad-core processors.

To demonstrate the broad applicability of the Heartbeats framework across a range of applications, we apply it to the PARSEC benchmark suite (v. 1.0). For each benchmark, we find the outer-most loop used to process inputs and insert a call to register a heartbeat in that loop. In some cases, the application is structured so that multiple inputs are consumed during one iteration of the loop. Table I shows both how the heartbeats relate to the input processed by each benchmark and the average heart rate (measured in beats per second) achieved running the "native" input data set[1]. The Heartbeat interface is found to be easy to insert into an application, as it requires adding less than half-a-dozen lines of code per benchmark, and only requires identifying the loop that consumes input data. In addition, the interface is low-overhead, resulting in immeasurable overhead for 9

[1] `freqmine` and `vips` are not included as the unmodified benchmarks did not compile on the target system with our installed version of `gcc`.

of 10 benchmarks and less than 5% for the remaining benchmark. In a deployed system, users may want to adjust the placement of the heartbeat calls to give underlying adaptive services more or less time to respond to changes in heart rate. To demonstrate the use of the API by an external system we develop an adaptive scheduler which assigns cores to a process to keep performance within the target range. The Heartbeat-enabled application communicates performance information and goals to the scheduler which attempts to maintain the required performance using the fewest cores possible. The behavior of `bodytrack` under the external scheduler is illustrated in Figure 3, which shows the average heart rate as a function of time measured in beats. The scheduler quickly increases the assigned cores until the application reaches the target range using seven cores. Performance stays within that range until heartbeat 102, when performance dips below 2.5 beats per second and the eighth and final core is assigned to the application. Then, at beat 141 the computational load decreases and the scheduler is able to reclaim cores while maintaining the desired performance.

TABLE I: Heartbeats in the PARSEC Benchmark Suite

Benchmark	Heartbeat Location	Heart Rate (beat/s)
`blackscholes`	Every 25000 options	561.03
`bodytrack`	Every frame	4.31
`canneal`	Every 1875 moves	1043.76
`dedup`	Every “chunk”	264.30
`facesim`	Every frame	0.72
`ferret`	Every query	40.78
`fluidanimate`	Every frame	41.25
`streamcluster`	Every 200000 points	0.02
`swaptions`	Every “swaption”	2.27
`x264`	Every frame	11.32

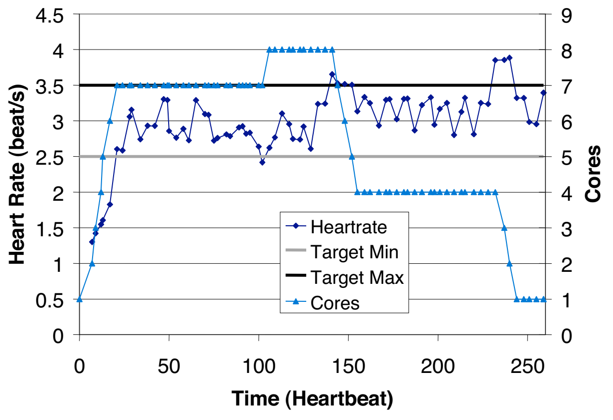

Fig. 3: `bodytrack` coupled with an adaptive scheduler.

Additional case studies are described in [38] and [39]. [38] has more information on our adaptive scheduler and studies using heartbeats to develop adaptive video encoders. [39] describes the SpeedGuard run-time system which can be automatically inserted into applications by the SpeedPress compiler. SpeedGuard uses heartbeats to monitor application performance and trade quality-of-service for performance in the presence of faults such as core failures or clock-frequency changes.

B. *Smartlocks Versus Frequency Variation*

This section demonstrates how Smartlocks helps address asymmetry in multicores by applying Smartlocks to the problem of thermal throttling and dynamic clock speed variations. It describes our experimental setup then presents results.

Our setup emulates an asymmetric multicore with six cores where core frequencies are drawn from the set {3.16 GHz, 2.11 GHz}. The benchmark is synthetic, and represents a simple work-pile programming model (without work-stealing). The app uses pthreads for thread spawning and Smartlocks within the work-pile data structure. The app is compiled using `gcc` v.4.3.2. The benchmark uses 6 threads: one for the main thread, four for workers, and one reserved for Smartlock. The main thread generates work while the workers pull work items from the queue and perform the work; each work item requires a constant number of cycles to complete. On the asymmetric multicore, workers will, in general, execute on cores running at different speeds; thus, x cycles on one core may take more wall-clock time to complete than on another core.

Since asymmetric multicores are not widely available yet, the experiment models an asymmetric multicore but runs on a homogeneous 8-core (dual quad core) Intel Xeon(r) X5460 CPU with 8 GB of DRAM running Debian Linux kernel version 2.6.26. In hardware, each core runs at its native 3.16 GHz frequency. Linux system tools like *cpufrequtils* could be used to dynamically manipulate hardware core frequencies, but our experiment instead models clock frequency asymmetry using a simpler yet powerful software method: adjusting the virtual performance of threads by manipulating the reward signal supplied by the application monitor. The experiment uses Application Heartbeats [38] as the monitor and manipulates the number of heartbeats such that at each point where threads would ordinarily issue 1 beat, they instead issue 2 or 3, depending on whether they are emulating a 2.11 GHz or 3.16 GHz core.

The experiment simulates a throttling runtime environment and two thermal-throttling events that change

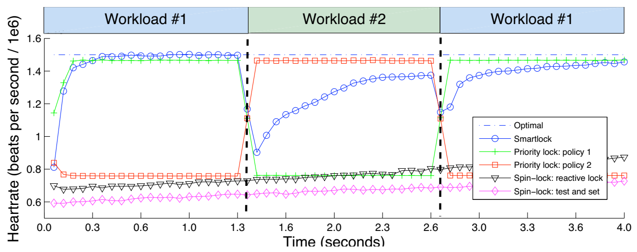

Fig. 4: Performance results on thermal throttling experiment. Smartlocks adapt to different workloads; no single static policy is optimal for all of the different conditions.

core speeds.[2] No thread migration is assumed. Instead, the virtual performance of each thread is adjusted by adjusting heartbeats. The main thread always runs at 3.16 GHz. At any given time, 1 worker runs at 3.16 GHz and the others run at 2.11 GHz. The thermal throttling events change which worker is running at 3.16 GHz. The first event occurs at time 1.4s. The second occurs at time 2.7s and reverses the first event.

Figure 4 shows several things. First, it shows the performance of the Smartlock against existing reactive lock techniques. Smartlock performance is the curve labeled "Smartlock" and reactive lock performance is labeled "Spin-lock: reactive lock." The performance of any reactive lock implementation is upper-bounded by its best-performing internal algorithm at any given time. The best algorithm for this experiment is the write-biased readers-writer lock so the reactive lock is implemented as that.[3] The graph also compares Smartlock against a baseline Test and Set spin-lock labeled "Spin-lock: test and set" for reference. The number of cycles required to perform each unit of work has been chosen so that the difference in acquire and release overheads between lock algorithms is not distracting but so that lock contention is high; what *is* important is the policy intrinsic to the lock algorithm (and the adaptivity of the policy in the case of the Smartlock). As the figure shows, Smartlock outperforms the reactive lock and the baseline, implying that reactive locks are sub-optimal for this and similar benchmark scenarios.

The second thing that Figure 4 shows is the gap between reactive lock performance and optimal performance. One lock algorithm / policy that can outperform standard techniques is the priority lock and prioritized access. The graph compares reactive locks against two priority locks / hand-coded priority settings (the curves labeled "Priority lock: policy 1" and "Priority lock: policy 2"). Policy 1 is optimal for two regions of the graph: from the beginning to the first throttling event and from the second throttling event to the end. Its policy sets the main thread and worker 0 to a high priority value and all other threads to a low priority value (e.g. high = 2.0, low = 1.0). Policy 2 is optimal for the region of the graph between the two throttling events; its policy sets the main thread and worker 3 to a high priority value and all other threads to a low priority value. In each region, a priority lock outperforms the reactive lock, clearly demonstrating the gap between reactive lock performance and optimal performance.

The final thing that Figure 4 illustrates is that Smartlock approaches optimal performance and readily adapts to the two thermal throttling events. Within each region of the graph, Smartlock approaches the performance of the two hand-coded priority lock policies. Performance dips after the throttling events (time=1.4s and time=2.7s) but improves quickly.

V. Conclusion

Adaptive techniques promise to reduce the burden modern computing systems place on application developers; however there are several obstacles to overcome before we see widespread usage of adaptive systems. Among these challenges, adaptive techniques must be developed which are generally applicable to a wide range of applications and systems. Furthermore, adaptive systems should incorporate the goals of the applications they are designed to support using standard and broadly applicable methods.

This paper presents our work addressing these challenges. First, we have defined the key characteristics of adaptive systems. We have also presented the Heartbeat API which provides a standard interface allowing applications to express their goals and progress to adaptive systems using a standard interface. With the Smartlocks library we have provided an example of using machine learning in combination with application performance measurements to adapt to a challenging computing environment. Both Heartbeats and Smartlocks are built upon techniques which can be generalized to a wide range of adaptive systems.

Our results show that Heartbeats are low-overhead and easy to add to a variety of different applications. In addition, we have demonstrated the use of Heartbeats in an adaptive resource allocator to perform constraint based optimization of an application incorporating that application's performance and goals. Furthermore, results using the Smartlocks library indicate that using Heartbeats as a reward function for a reinforcement

[2]We inject throttling events as opposed to recording natural events so we can determine a priori some illustrative scheduling policies to compare Smartlock against.

[3]This is the highest performing algorithm for this problem known to the authors to be included in a reactive lock implementation.

learning engine is an effective technique for adapting an application's behavior. Specifically, we show that the Smartlock machine learning engine can dynamically learn optimal lock acquisition policies even faced with disruptive events like core frequency changes.

ACKNOWLEDGMENT

We'd like to thank all the many people we worked with over the last two years for all the useful discussions and for their thoughtful ideas. Special thanks to Jason E. Miller and all the members of the CARBON research groups and to the Rocca Foundation for its support.

REFERENCES

[1] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

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

[3] P. Dini. Internet, GRID, self-adaptability and beyond: Are we ready? Aug 2004.

[4] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *PPOPP*, pages 347–348, 2010.

[5] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: Self-aware synchronization through lock acquisition scheduling. SMART 2010: Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion, 2010. Online document, http://ctuning.org/dissemination/smart10-05.pdf.

[6] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002.

[7] R. Kumar, K. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Processor power reduction via single-isa heterogeneous multi-core architectures. *Computer Architecture Letters*, 2(1):2–2, January-December 2003.

[8] Intel Inc. Intel itanium architecture software developer's manual, 2006.

[9] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th Inter. Conf. on Supercomputing*, pages 101–110, 2005.

[10] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[11] HP Labs. HP open view self-healing services: Overview and technical introduction.

[12] David Breitgand, Maayan Goldstein, Ealan Henis, Onn Shehory, and Yaron Weinsberg. Panacea towards a self-healing development framework. In *Integrated Network Management*, pages 169–178. IEEE, 2007.

[13] C. M. Garcia-Arellano, S. Lightstone, G. Lohman, V. Markl, and A.Storm. A self-managing relational database server: Examples from IBMs DB2 universal database for linux unix and windows. *IEEE Transactions on Systems, Man and Cybernetics*, 36(3):365– 376, 2006.

[14] Andres Quiroz, Nathan Gnanasambandam, Manish Parashar, and Naveen Sharma. Robust clustering analysis for the management of self-monitoring distributed systems. *Cluster Computing*, 12(1):73–85, 2009.

[15] Salim Hariri, Guangzhi Qu, R. Modukuri, Huoping Chen, and Mazin S. Yousif. Quality-of-protection (qop)-an online monitoring and self-protection mechanism. *IEEE Journal on Selected Areas in Communications*, 23(10):1983–1993, 2005.

[16] Onn Shehory. Shadows: Self-healing complex software systems. In *ASE Workshops*, pages 71–76, 2008.

[17] S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in grid computing. *Concurr. Comput. : Pract. Exper.*, 17(2-4):235–257, 2005.

[18] J. Buisson, F. André, and J. L. Pazat. Dynamic adaptation for grid computing. *Lecture Notes in Computer Science. Advances in Grid Computing - EGC*, pages 538–547, 2005.

[19] P. Reinecke and K. Wolter. Adaptivity metric and performance for restart strategies in web services reliable messaging. In *WOSP '08: Proceedings of the 7th International Workshop on Software and Performance*, pages 201–212. ACM, 2008.

[20] John Strassner, Sung-Su Kim, and James Won-Ki Hong. The design of an autonomic communication element to manage future internet services. In Choong Seon Hong, Toshio Tonouchi, Yan Ma, and Chi-Shih Chao, editors, *APNOMS*, volume 5787 of *Lecture Notes in Computer Science*, pages 122–132. Springer, 2009.

[21] Armando Fox, Emre Kiciman, and David Patterson. Combining statistical monitoring and predictable recovery for self-management. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop*

on Self-managed systems, pages 49–53, New York, NY, USA, 2004. ACM.

[22] J.S. Vetter and P.H. Worley. Asserting performance expectations. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 33–33, Nov. 2002.

[23] M. Caporuscio, A. Di Marco, and P. Inverardi. Run-time performance management of the siena publish/subscribe middleware. In *WOSP '05: Proc. of the 5th Inter. Work. on Software and performance*, pages 65–74, 2005.

[24] L. A. De Rose and D. A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *Inter. Conf. on Parallel Processing*, 1999.

[25] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM J. Res. Dev.*, 50(2/3):239–248, 2006.

[26] O. Krieger, M. Auslander, B. Rosenburg, R. Wisniewski J. W., Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: building a complete operating system. pages 133–145, 2006.

[27] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC '03: Proc. of the ACM/IEEE conf. on Supercomputing*, Nov 2003.

[28] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI '99: Proc. of the third symp. on Operating systems design and implementation*, 1999.

[29] Marco D. Santambrogio. From reconfigurable architectures to self-adaptive autonomic systems. *IEEE International Conference on Computational Science and Engineering*, pages 926 – 931, 2009.

[30] E. Ipek, O. Mutlu, J. F. Martnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08: Proc. of the 35th Inter. Symp. on Comp. Arch.*, 2008.

[31] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, 2007.

[32] J. Appavoo, K. Hui, M Stumm, R. W. Wisniewski, D. Da Silva, O. Krieger, and C. A. N. Soules. An infrastructure for multiprocessor run-time adaptation. In *WOSS '02: Proceedings of the first Workshop on Self-healing Systems*, pages 3–8, New York, NY, USA, 2002. ACM.

[33] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *PPoPP '05: Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 277–288, New York, NY, USA, 2005. ACM.

[34] C. A. N. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, Ostrowski M., B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.

[35] A. Baumann, D. Da Silva, O. Krieger, and R. W. Wisniewski. Improving operating system availability with dynamic update. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[36] Henry Hoffmann, Jonathan Eastep, Marco Santambrogio, Jason Miller, and Anant Agarwal. Application Heartbeats Website. MIT, 2009. Online document, http://groups.csail.mit.edu/carbon/heartbeats.

[37] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT-2008: Proceedings of the 17th Inter. Conf. on Parallel Architectures and Compilation Techniques*, Oct 2008.

[38] Henry Hoffmann, Jonathan Eastep, Marco Santambrogio, Jason Miller, and Anant Agarwal. Application heartbeats for software performance and health. Technical Report MIT-CSAIL-TR-2009-035, MIT, Aug 2009.

[39] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT, September 2009.