Self-Aware Adaptation in FPGA-based Systems

F. Sironi*, M. Triverio*, H. Hoffmann†, M. Maggio*†, and M. D. Santambrogio*†
* *Dipartimento di Elettronica e Informazione (DEI) - Politecnico di Milano*
Email: {filippo.sironi, marco.triverio}@dresd.org, maggio@elet.polimi.it, marco.santambrogio@polimi.it
† *Computer Science and Artificial Intelligence Laboratory (CSAIL) - Massachusetts Institute of Technology*
Email:{hank, mmaggio, santambr}@mit.edu

***Abstract*—Self-Aware Adaptive computing systems are capable of adapting their behavior and resources thousands of times based on changing environmental conditions and demands. This allows them to automatically find the best way to accomplish a given goal with the resources at hand. This capability would benefit the full range of computer systems, from embedded devices to servers to supercomputers. Although such a system may seem rather far fetched, we believe that basic semiconductor technology, computer architecture and software systems have advanced to the point that the time is ripe to realize such a system.**

In this paper we present an implementation of an FPGA-based Self-Aware Adaptive computing system which blends techniques developed in different research fields, i.e, monitoring, decision making, and self-adaptation. The result is a system built on top of a set of *enabling technology* that proves the effectiveness of using Self-Aware Adaptive computing systems. We used the *Application Heartbeats* to assess performance goals and to inspect application progress and the *Implementation Switch Service* to switch between different implementations of the same algorithm (both in software and in hardware) at runtime. Preliminary results show the effectiveness and the usability of the proposed approach.

I. Introduction

The amount of available 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 Even though technologies have improved, we continue to apply outdated approaches to our use of these resources, and key computer science abstractions have not changed since the 1960's. On one hand, the advantages of complex, highly-parallel, heterogeneous multicore systems could be applied in various fields, such as: telecommunications (*e.g.,* adaptive intelligent routers and cognitive networks that are looking for high-performance, adaptable mobile devices), Data Center on Chip (*e.g.,* spatially organized modular applications architecture taking lessons from internet-like distributed, fault-tolerant, applications), high accuracy speech processing, intelligent transducers at bio-electronic interfaces, etc. On the other hand, system complexities are skyrocketing, making it unfeasible for the average programmer to weight all the constraints and optimize the program for a wide range of machines and scenarios. Self-Aware Adaptive computing is a research area aimed at leveraging the new balance of resources to improve performance, utilization, reliability and programmability, overcoming the burden imposed by the increasing complexity and the associated workload of modern computing systems. Within this context, imagine an interaction capability of digital systems by which designers and users can specify their desired goals rather than how to perform a task, along with constraints in terms of an energy budget, time, preference for an approximate answer over an exact answer and possibly the confidence level of such answers. 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 Adaptive 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 scenario would benefit the full range of computer infrastructures, from embedded devices to personal computers to servers to supercomputers as demonstrated by the interest of major companies such as IBM [1] (IBM Touchpoint Simulator, the K42 Operating System [2]), Oracle (Oracle Automatic Workload Repository [3]), and Intel (Intel RAS Technologies for Enterprise [4]).

Within this scenario, we,

- developed a first implementation of an FPGA-based Self-Aware Adaptive computing system,
- defined and implemented the Implementation Switch Service, based on the Hot-Swap mechanism, to switch at run-time between different implementations of a functionality which is not performing as expected,
- introduced the Application Heartbeats in the GNU/Linux operating system, executed on the FPGA, to set the performance goals of the applications and to monitor at runtime the performance of the system,
- presented a case study to prove the validity and the effectiveness of the proposed approach.

The remainder of this paper is organized as follows. Section II identifies the key system components of a self-aware systems, while Section III presents the enabling technologies proposed in this paper to realize an FPGA-based Self-Aware

Adaptive system. We present a summary of related work in Section V and in Section IV our experimental results are presented. Finally, our conclusions are summarized in Section VI.

II. Self-Aware Adaptive Systems: Definition and General Overview

To achieve the vision just described, a Self-Aware Adaptive computing system is no longer view as a static bunch of hardware components with a passive set of applications running on top of an operating system, that properly coordinate the underling architecture. It becomes an active system where either the hardware, the applications and the operating system have to be seen as an unique entity that have to be able to autonomously adapt itself to achieve the best performance[1]. Fig. 1 presents the general overview of the hardware and software architecture components of a self-aware computing system. As presented in [5], a Self-Aware

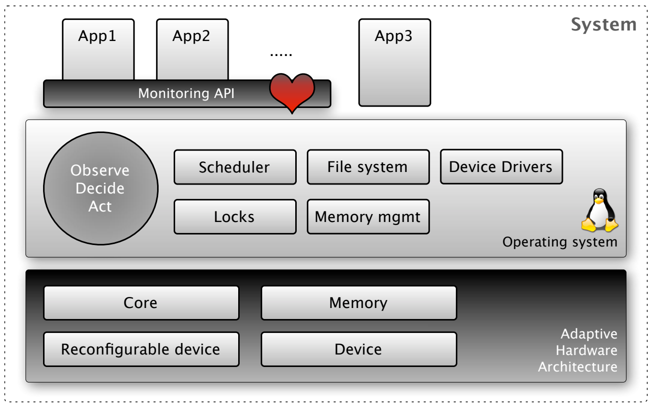

Figure 1. Overview of the proposed self-aware FPGA-based computing system.

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. A self-aware computing system 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.

[1]Performance can have different meaning according to a specific scenario. It is possible to have a system trying to maximize the overall completion time of a set of applications, and, under different constraints, having the same system running trying to minimize its power consumption.

Given a goal, a set of resources and their availability, a Self-Aware Adaptive computing system finds the best way to accomplish the goal while optimizing constraints of interest, *i.e.,* accomplish the goals with the minimal amount of resources and energy. Of course, it is also provided with many possible procedures to accomplish subtasks, each of which might use different types of architectural components. In a mobile phone scenario, a self-aware system is given the goal of maintaining a connection to a receiver with a desired bit rate, using the least amount of energy. The software and architecture collaborate on achieving this goal. The underlying architecture has cognitive hardware mechanisms in its trusted core to both *observe* and to *affect* the execution. Since it is impossible to pre-configure all possible scenarios, the Self-Aware Adaptive computing system also implements learning and decision making engines in a judicious combination of hardware and software to determine the appropriate actions based on given observations. To adapt what the self-aware computing system 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.

III. The Proposed Self-Aware Adaptive Systems

In this paper we present a first implementation of an FPGA-based Self-Aware Adaptive computing system which blends techniques developed in different research fields, *i.e.,* monitoring, decision making, and self adapting. The result is a system built on top of a set of *enabling technology* that proves the effectiveness of using Self-Aware Adaptive computing systems. Our approach merges the potentiality brought by reconfigurable hardware with the state-of-the-art in performance assertions, monitoring, and adaptation: FPGAs offer tremendous computational power while the possibility of running an operating system on top of it makes it possible to observe performance and take actions that can involve both software and hardware adaptation.

The operating system is in charge of choosing at runtime among the set of possible implementations (hardware or software among the available implementations) according to different criteria, such as expected performance, available area (set of resources) on the FPGA, input data type and size, functionalities already implemented and available as hardware components. The runtime decision of the most suitable implementation, based on the *Observe-Decide-Act (ODA)* loop, allows this work to be considered one of the first attempts to the definition of *Self-Aware Adaptive system* in the embedded domain. Within this context the system monitors its performance (*Observe*), the gathered data feed a decision engine (*Decide*), and the decisions can be translated in a variety of actions (*Act*), *i.e.,* , by choosing at runtime the best[2] implementation for functionality which is not performing as expected. These phases have been implemented within the proposed system, by using different software libraries. Such a solution:

- makes use of the *Application Heartbeats* [6, 7] (or simply Heartbeats) to set performance goalsand monitor the progress of the execution;
- has knowledge (*i.e., static* performance, reconfiguration time, etc. . .) over different implementations of the exposed functionality;
- features a decision mechanism to choose the best implementation. The decision on which implementation has to be used can be taken using different techniques, *i.e.,* machine learning [8], control theory, competitive algorithm [9], etc. . . .
- presents an hot-swap mechanism, used to create the *Implementation Switch Service*, to switch between different implementations (*i.e.,* from software to software, from software to hardware, from hardware to software or from hardware to hardware) of a functionality which is not performing as expected.

Fig. 2 presents the overall structure of the proposed system.

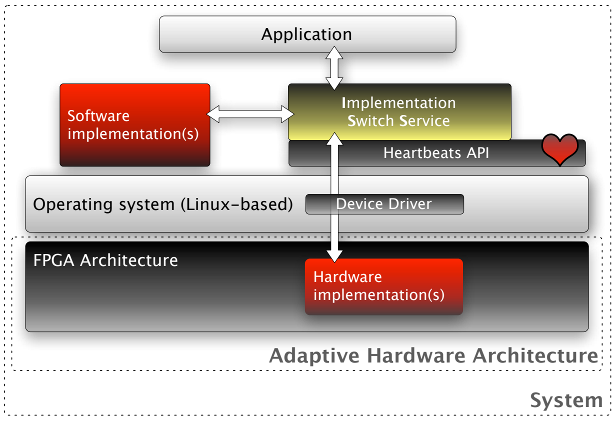
Application
Software implementation(s)
Implementation Switch Service
Heartbeats API
Operating system (Linux-based)
Device Driver
FPGA Architecture
Hardware implementation(s)
Adaptive Hardware Architecture
System

Figure 2. Self-Aware Adaptive computing system

In the following more details on the Application Heartbeats and on the Implementation Switch Service are presented.

A. Application Heartbeats

The *Application Heartbeats* implements a simple yet extremely powerful monitoring infrastructure. Specifically it is an API made of a small set of functions that makes it straightforward to use. Through the Application Heartbeats it is possible to declare performance goals and monitor the progress of the execution. Any software component that wants to make use of the framework first has to register to the Application Heartbeats specifying a certain number of parameters: minimum and maximum heart rate, size of the window of observation, size of the heartbeats history buffer, and others. During execution the same component has to update the progress through the call to a function that signifies an heartbeat. The framework automatically updates all the necessary information about the global heart rate, the windowed heart rate and other internal information. Such data is made available to either external observers or the component itself. In our implementation the *Implementation Switch Service* is in charge of registering to the framework, issuing heartbeats, and retrieve all the information need for the decision process. A typical example of usage is a video encoder: the application might set a certain goal (*e.g.,* 30 frames per second) and issue an heartbeat for each produced frame (*e.g.,* the goal is in this case 30 heartbeats per second).

B. Implementation Switch Service

The adoption of an *Implementation Switch Service (ISS)* is fundamental for the system to take decisions and act. The *Application Heartbeats* makes possible to query the progress of the execution and obtain an overall history of the heart rate on a given time window. Fed with such information, the decision mechanism chooses at runtime the best implementation to use in accordance to given constraints (*e.g.,* avoid oscillations) and goals (*i.e.,* desired heart rate).

The need for a dynamic choice between available implementations is given by the fact that the system is *live* and *lives* in an unpredictable environment: for such reason it is impossible to decide statically which implementation proves to be the most convenient. For instance a static analysis might show that for any given input data size there exists an implementation which outperforms the others; yet many other factors, such as the variable system load, might prevent the static analysis to get real. For this reason the system must monitor the surrounding context and take decisions accordingly hence dynamically balancing all the constraints.

For Self-Aware Adaptive computing system the ability to switch between different implementations of the same functionality while the system is running proves to be fundamental. To *hot-swap* an implementation with another one is a non-trivial process; threads within a single process can access data structures concurrently and different implementations can use completely different data structures; state quiescence and state translation are the most visible problems the hot-swap process generates. In Self-Adaptive computing literature this is a well-known problem and a general framework to solve it has already been proposed in [2]. This general framework inspired our hot-swap mechanism which is divided in 3 sub-phases as illustrated in Fig. 3: (a) a prior phase representing a common working condition; (b) a transfer phase in which new requests are blocked in order

[2]Remembering that a Self-Aware Adaptive system is *approximate*, *best* does not mean the optimal solution but the one that can guarantee the best performance considering the runtime conditions in which it has to be executed.

to reach a quiescent state which is translated to fit another data structure; (c) a post phase in which both blocked and new requests are allowed to proceed.

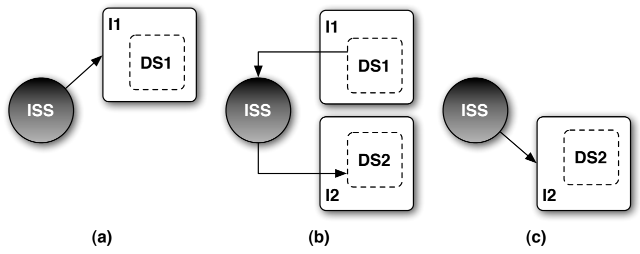

Figure 3. Hot-Swap mechanism

Even though our approach is inspired by the *state-of-the-art* framework, it differs since it works solely on data structures instead of objects because the different implementations of a certain functionality are provided by means of Dynamic-Link Libraries (DLLs) that decouple the data structure from the actual implementation. This means only the data structures are actively involved in the hot-swap process. This translates in the advantage of not requiring every implementation to conform to a single interface. Furthermore, our approach defines a technology able to manage also the adaptation of the underlying physical architecture using, where possible, hardware implementations.

IV. Preliminary Results

This Section presents preliminary results to support the validity of the proposed approach. In Section IV-A we give more details about the final implementation while the case study used to present the effectiveness of our techniques and to validate them has been described in Section IV-B.

A. Implementation

Our environment is based on a *Xilinx University Program Virtex-II Pro* board featuring a *PowerPC 405* processor and 256 MB of system memory. On top of this architecture a streamlined *Linux*-based operating system is loaded to execute applications accessing both the software implementation and the hardware implementation (eventually dynamically reconfigured). Given the system presented in Fig. 2 we implemented the *Data Encryption Standard (DES)* [10], to prove the validity of the proposed technology, not as a breakthrough in cryptography.

It is indeed true that many problems are more efficiently solved using hardware implementations instead of software implementations. Yet this claim actually depends on the expected *Quality-of-Service (QoS)* which may be such that a software implementation might perform sufficiently well with respect to given constraints. Therefore we designed two DES implementations, one in software and one in hardware. The applications specify[3] an expected performance goal over time *i.e., heart rate* while the current heart rate is updated every time a block is computed. The Application Heartbeats makes it possible to check if the current heart rate fits the expected goal enabling the library to take decisions in accordance. When the throughput (*i.e.,* heart rate) over a certain window of time drops under or excessively overcomes the expectations a heuristic is activated. The heuristic knows the available implementations and acts to improve performance hot-swapping them when needed. It takes into account several parameters such as the last instant when the implementation has been swapped (to avoid short-term oscillations), the setup time needed to activate an implementation, and so on. Fig. 4 shows a possible execution progress. Even though in $(t_0,\, t_1)$ the application heart rate is dropping due to the context switches we are not observing any change in the implementation because the current heart rate is still inside the desired heart rate window. In (t_1, t_2) the computed heart rate exits this window for more than Δ_1 time instants hence the ISS decides to spend $(t_2,\, t_3)$ reconfiguring the FPGA[4] and to switch to the hardware implementation. In $(t_3,\, t_4)$ the heart rate increases entering the desired heart rate window. The sudden decrease of the heart rate in $(t_4,\, t_5)$ proceeding in $(t_5,\, t_6)$ is caused by resources contention (*e.g.,* buses, memory, etc. . .). Therefore, after waiting for Δ_2 time instants the software implementation is swapped in to try increasing the heart rate as it happens in $(t_6,\, t_7)$. With Δ_i we denote the time spent on the decision mechanism used within the system. So far, we used an heuristic based decision mechanism to identify at run-time the best implementation, nevertheless other decision mechanism *i.e.,* machine learning, control theory, competitive algorithm, etc. . . , can be used to improve the overall quality of the application.

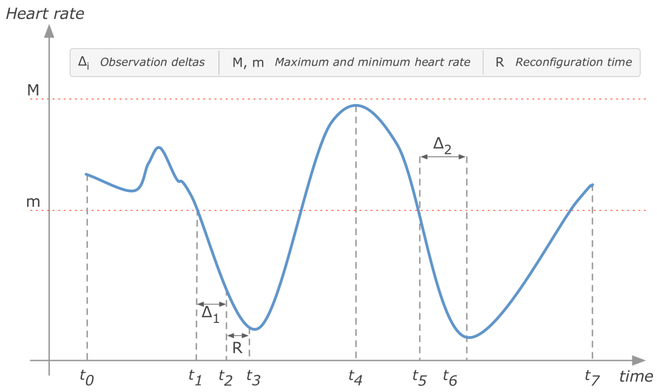

Figure 4. Application execution behavior. The interval between m and M defines the desired heart rate window

The advantage of this library is the fact that it hides complexity from the applications which are unaware of the

[3]In particular programmers might have to manually tune the minimum and maximum heart rate parameters depending on the targeted systems.

[4]The reconfiguration time R is spent only when the desired hardware implementation is not already configured on the FPGA.

presence of multiple implementations and are only required to set an expected performance goal.

B. Results validation

The first result of our work is to statically analyze the ideal behavior of the two DES implementations. We run both the implementations varying the input data size and averaging the results on tens of different trials. The results are shown in Fig. 5: the hardware implementation (*square*) is faster than the software implementation (*circle*) when the input size is bigger than 50 blocks. The hardware implementation might need to be configured hence the hardware implementation considering the reconfiguration time (*triangles*) gets faster than the software implementation when the input size is bigger than 400 blocks. This analysis might seem to prove

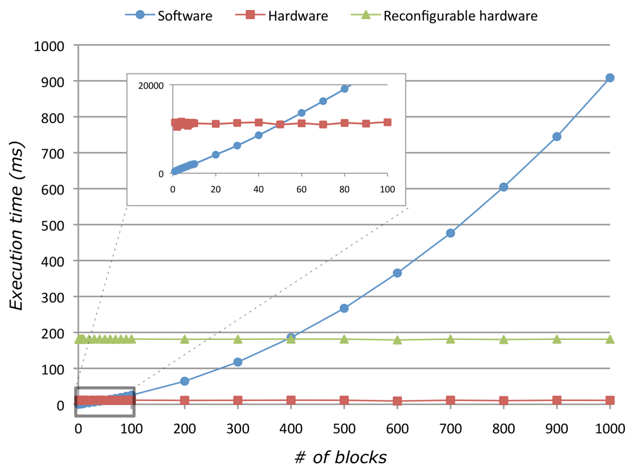

Figure 5. Execution times

that, depending on the input size (number of blocks), it is possible to choose the best implementation statically. Yet in a dynamic scenario such execution times might change completely due to the system load or constraints (*e.g.,* power consumption) and a statical approach might fail. Since plenty of factors cannot be predicted it is necessary to monitor the throughput of the active implementation and decide when to switch one in favor of the other at run-time.

We adopted the Application Heartbeats to support the decision process, however, to truly improve performance it is necessary for the monitoring infrastructure to be lightweight. To prove this we have considered the encryption of 1 to 1000 blocks with and without the Application Heartbeats: the average overhead on the execution time is a moderate 3.52%. This overhead is due to a series of system calls the Application Heartbeats requires in order to initialize its data structures and updates the global heart rate. Moreover, the overhead seems to decrease while the number of blocks increases due to the fact that the weight of the initialization tends to decrease.

V. Related Work

Different FPGA-based adaptive systems have been presented in the literature. In the following, we are summarizing, at the best of our knowledge, the key contributions considering the two main aspects described in our work: system monitoring and the runtime reconfiguration management.

FPGA-based reconfigurable architecture has been used in cooperation with ad-hoc, or extended version of standard compiler to create runtime adaptable system, as presented in [11, 12]. These approaches take advantages by the use of the compiler optimization to create the supposed to be, best configuration for a given application. Unfortunately these solutions lack in online adaptability, limit that has been faced in [13] and in [14]. In [14], the FPGA has been used as a sort of filter to monitor, using the dependability analysis, the data flowing through a certain part of the system. This approach, even without introducing overhead into the computation, cannot be considered as non intrusive with respect to the overall system. A partial reconfiguration approach, due to the reconfiguration capabilities of modern FPGAs, has been proposed in [13] to implement an online adaptive system, able to update its underlying architectural implementation to optimize the power consumption. This is an interesting approach but it is only proving that partial reconfiguration can be used to implement an online solution, but the runtime environment has not been realized nor information on how to monitor online its behavior has been outlined.

In the context of reconfigurable systems, many approaches focused on effective utilization of the dynamically reconfigurable hardware resources. In [15] custom monitoring components, implemented using a text based description language called SiLLis, have been presented in a reconfigurable FPGA scenario. These components, called *listeners*, are able to filter the data and use it for monitoring, debugging, and control purposes. This approach is very promising, but compared to the one proposed in this paper, it requires extra hardware to be included in the final architecture. Noguera and Badira [16] proposed a design framework for dynamically reconfigurable systems, introducing a dynamic context scheduler and HW/SW partitioner. Banerjee et al. [17] introduced a partitioning scheme that is aware of the placement constraints during the context scheduling of the partially reconfigurable datapath of the SoC. In [18] the authors propose a new methodology to allow the platforms to hot-swap application specific modules without disturbing the operation of the rest of the system. This goal is achieved through the use of partial dynamic reconfiguration. The authors presented an effective and flexible reconfigurable architecture, but unfortunately no information on how to take the decision on when it would be necessary to reconfigure the system have been presented. Several works [19–21] have focused their attention of the effective combined utilization of dynamically reconfigurable hardware and software resources at runtime. Within this context, the runtime implementation, hardware or software, of specific elements of the systems is taken online during the system execution. These works lack (i) in the usage of an online monitoring system able to observe the performance of the system itself and (ii) in the definition of a self-aware mechanism able to autonomously take decision, based on the online observations, regarding the hardware or software implementation for a certain functionality. Decision that can be in contrast to standard behaviors, as described in Section IV-A.

VI. Conclusion and Future Work

This paper presents an implementation of an FPGA-based Self-Aware Adaptive system which blends techniques developed in different research fields. Our approach merges the potentiality brought by reconfigurable hardware with the state-of-the-art in performance assertions, monitoring, and adaptation. With the Implementation Switch Service we have shown an example of action that can take advantage by the application performance measurements to adapt to a challenging computing environment. Our results show that the Application Heartbeats is low-overhead and, in addition, we have demonstrated the use of such a technology in a GNU/Linux operating system to perform constraint based optimization of an application incorporating that application's performance and goals. Extensions of the present work will use the same framework for a multi-core system together with an FPGA, used as external accelerator, where the operating system is executed on top of the multi-core machine and the hot-swap mechanism allows to use the FPGA when needed without modifying the original code. Within this context we will also able to combine multiple effects deriving by the simultaneous usage of different services, e.g., the Implementation Switch Service, the Core Allocator [6], Smartlocks [8].

References

[1] IBM Inc., "IBM autonomic computing website," 2009. [Online]. Available: http://www.research.ibm.com/autonomic/

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

[3] Oracle Corp., "Automatic Workload Repository (AWR) in Oracle Database 10g." [Online]. Available: http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php

[4] Intel Inc., "Reliability, availability, and serviceability for the always-on enterprise," 2005. [Online]. Available: www.intel.com/assets/pdf/whitepaper/ras.pdf

[5] M. D. Santambrogio, H. Hoffmann, J. Eastep, and A. Agarwal, "Enabling technologies for self-aware adaptive systems," in *Adaptive Hardware and System*, 2010.

[6] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats for software performance and health," in *PPOPP*, 2010, pp. 347–348.

[7] H. D. Team, "Application Heartbeats Website," MIT, 2009. Online document, http://groups.csail.mit.edu/carbon/heartbeats.

[8] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal, "Smartlocks: Self-aware synchronization through lock acquisition scheduling," Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion, 2010. Online document, http://ctuning.org/dissemination/smart10-05.pdf.

[9] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," in *SFCS '86: Proceedings of the 27th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1986, pp. 244–254.

[10] U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, *DATA ENCRYPTION STANDARD (DES)*. The Bureau ; for sale by the National Technical Information Service, Washington : Springfield, Va. :, 1999.

[11] E. M. Panainte, K. Bertels, and S. Vassiliadis, "The molen compiler for reconfigurable processors," *Transactions on Embedded Computing Systems*, vol. 6, no. 1, p. 6, 2007.

[12] D. Bhatia, P. Kannan, K. Simha, and K. M. G. Purna, "React: Reactive environment for runtime reconfiguration," in *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*. London, UK: Springer-Verlag, 1998, pp. 209–217.

[13] K. Paulsson, M. Hübner, and J. Becker, "On-line optimization of fpga power-dissipation by exploiting run-time adaption of communication primitives," in *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*. New York, NY, USA: ACM, 2006, pp. 173–178.

[14] N. Bartzoudis and K. McDonald-Maier, "Online monitoring of fpga-based co-processing engines embedded in dependable workstations," in *Proceedings of the 13th IEEE International On-Line Testing Symposium*, July 2007, pp. 79–84.

[15] P. R. Grassi, M. D. Santambrogio, J. Hagemeyer, C. Pohl, and M. Porrmann, "Sillis: A simplified language for monitoring and debugging of reconfigurable systems." in *International Conference on Engineering of Reconfigurable Systems and Algorithm*, 2009, pp. 174–180.

[16] J. Noguera and R. M. Badia, "Hw/sw codesign techniques for dynamically reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 10, no. 4, pp. 399–415, 2002.

[17] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *DAC '05*. ACM Press, 2005, pp. 335–340.

[18] E. L. Horta, J. W. Lockwood, and D. Parlour, "Dynamic hardware plugins in an fpga with partial run-time reconfigurtion," pp. 844–848, 1993.

[19] V. Sima and K. Bertels, "Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform," in *IEEE International Symposium on Parallel and Distributed Processing*, May 2009.

[20] K. Sigdel, M. Thompson, A. Pimente, K. Bertels, and C. Galuzzi, "System level runtime mapping exploration of reconfigurable architectures," in *IEEE International Symposium on Parallel and Distributed Processing*, May 2009.

[21] M. D. Santambrogio, S. O. Memik, V. Rana, U. A. Acar, and D. Sciuto, "A novel soc design methodology combining adaptive software and reconfigurable hardware," in *ICCAD 2007*. Piscataway, NJ, USA: IEEE Press, 2007, pp. 303–308.