Coordination of Independent Loops in Self-Adaptive Systems

JACOPO PANERATI, École Polytechnique de Montréal
MARTINA MAGGIO, Lund University
MATTEO CARMINATI, FILIPPO SIRONI, MARCO TRIVERIO, and
MARCO D. SANTAMBROGIO, Politecnico di Milano

Nowadays, the same piece of code should run on different architectures, providing performance guarantees in a variety of environments and situations. To this end, designers often integrate existing systems with ad-hoc adaptive strategies able to tune specific parameters that impact performance or energy—for example, frequency scaling. However, these strategies interfere with one another and unpredictable performance degradation may occur due to the interaction between different entities. In this article, we propose a software approach to reconfiguration when different strategies, called *loops*, are encapsulated in the system and are available to be activated. Our solution to loop coordination is based on machine learning and it selects a policy for the activation of loops inside of a system without prior knowledge. We implemented our solution on top of GNU/Linux and evaluated it with a significant subset of the PARSEC benchmark suite.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *Performance attributes*; G.3 [**Probability and Statistics**]: *Markov Processes*; I.2.6 [**Artificial Intelligence**]: Learning—*Knowledge acquisition*

General Terms: Algorithms, Design, Management, Performance

Additional Key Words and Phrases: Autonomic computing, autonomic manager, performance management

ACM Reference Format:
Jacopo Panerati, Martina Maggio, Matteo Carminati, Filippo Sironi, Marco Triverio, and Marco D. Santambrogio. 2014. Coordination of independent loops in self-adaptive systems.

1. INTRODUCTION

Reconfigurable systems can be seen as interactive collections of elements, capable of managing themselves at runtime. This is very close to the original vision of autonomic computing [Kephart and Chess 2003] that advocates the realization of *autonomic elements* as individual systems consisting of a *managed element* and an *autonomic manager*. These systems are supposed to deal with uncertain environments, heterogeneous resources, and irregular workloads; enforcing autonomicity in such conditions usu-ally requires the ability to plan and to perform multiple, correlated adjustments (i.e., sequential decision making).

At the same time, every system should have some self-adaptive capabilities. In fact, programmers have to deal with the advent of multi/manycore systems, which make the task of writing solid code much more complicated than it was for singlecore systems (e.g., achieving the desired quality of service (QoS) in multiple scenarios that might not

This work was partially supported by the Regroupement Stratégique en Microsystèmes du Québec (ReSMiQ).
Authors' addresses: J. Panerati, Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal; M. Maggio, Department of Automatic Control, Lund University; email: maggio.martina@gmail.com; M. Carminati, F. Sironi, M. Triverio, and M. D. Santambrogio, Dipartimento di Elettronica e Informatica, Politecnico di Milano.

be known a priori). The same code is required to work in multiple architectures, with possible performance guarantees on each of them. The complexity of this task is skyrocketing and it is not advisable to demand that a single programmer has such strong competences both in software design and in architecture-related issues. Therefore, the best way to deal with this problem is to adopt a reconfigurable system. A reconfigurable system is able to mask this complexity to the programmer, by adjusting itself when changes in the environment occur.

As an example, take a piece of code that has to run on multiple types of hardware. If the code has a high degree of parallelism its performance usually benefits from the availability of multiple cores. If a lot of memory is needed, the presence of multiple cores can become a disadvantage, since sharing the data over multiple computation units can actually slow down the computation instead of speeding it up. Since the machine configuration can differ, add-ons such as symmetric multithreading (SMT), sprinting, etc. can alter the behavior of the application. For example, the Intel Turbo Boost Technology can alter the frequency of cores in a completely transparent fashion to the operating system kernel and applications, providing a source of irregularity in the execution environment. All these quantities can be controlled by specific entities, receiving feedback from the system and deciding in which way to act on a specific knob in order to obtain the desired performance, or save power, or whatever policy could be expressed at design time. However, all these entities need coordination and, due to this reason, will be disabled during all the performed experiments.

In this article, we propose a software approach to reconfiguration in which different loops are encapsulated in the system and are available to be activated. A *loop* is an entity that takes decisions on the runtime execution of the system, whichever nature these decisions can have. For example, a decision can be to implement a simple policy such as "if the number of frames to be processed by the hardware and sent over the network for surveillance purposes is greater than 5, then activate some additional hardware accelerator, otherwise keep it off". Loops can be much more complicated, based on techniques that analyze the current execution and react consequently—control theory and reinforcement learning being two examples of these techniques. Loops modify the system behavior, in an *a priori* unpredictable way, since their combination could result in unforeseen configurations. Different loops can act independently on the same or on different quantities, shaping the way the overall system behaves. Since their behavior is not coordinated, different loops can in principle be distruptive for the system and they must be coordinated in some way [Heo and Abdelzaher 2009]. The main idea behind the article is to build a model free solution that is able to select a policy for the activation of internal loops of the system in order to coordinate them, without prior knowledge on the system. This work makes the following contributions.

(1) We propose a framework enabling the coordination of different reconfiguring elements acting in a system. Each of these elements closes one or more loops around the system. Our framework is based on observations of the actual performance and it selects which loops to disconnect and which loops to activate, exploiting a reinforcement learning approach.
(2) We implement policies harnessing the infrastructure of the framework. The framework uses the Heart Rate Monitor (HRM) [Sironi et al. 2012] to retrieve information from the running applications and it uses reinforcement learning to select the new configuration to be applied.
(3) We assess our solution across a significant subset of the PARSEC benchmark suite [Bienia 2011] with core allocation, frequency scaling and dummy loops. The evaluation shows that we are able to enable the loops that make the system tend towards its goals.

The remainder of this article is organized as follows. Section 2 presents the related work, then Section 3 explains our main contribution in designing the decision element in charge of orchestrating all the loops, exposes the theoretical foundations of the algorithms exploited to enforce self-adaptation and describes relevant implementation details. The results obtained on real hardware are illustrated in Section 4, while Section 5 presents further extensions of our approach. Last, Section 6 concludes the article.

2. RELATED WORK

This section summarizes significant research contributions to the coordination of multiple entities acting on a single system in order to make it reconfigurable and achieve user goals or desired properties. First, we present related works whose focus is on fundamental attributes needed for reconfiguration, for example, monitoring capabilities. Then, we discuss comprehensive frameworks to realize reconfigurable systems and, finally, we extend the discussion to self-adaptive systems exploiting artificial intelligence and machine learning.

2.1. Enforcement of Self-Awareness

Application Heartbeats is an infrastructure for active self-monitoring introduced and developed by Hoffmann et al. [2010c]. Application Heartbeats is built over the well-known ideas of heartbeats and heart rate [Sterritt and Bustard 2003] and implements them through a compact Application Programming Interface (API). Application Heartbeats empowers applications (i.e., managed elements) with the ability to export user-defined performance goals and signal progression towards those goals. In the work by Maggio et al. [2010], Application Heartbeats is exploited by a control-theoretical framework allocating resources to applications; in such a framework, Application Heartbeats APIs provided the self-monitoring capabilities. This work also led to a comparison of many different decision-making mechanism ranging from heuristic to control-theoretical and machine learning [Maggio et al. 2011].

Metronome [Sironi et al. 2012] is a framework enabling self-adaptive computing in the scheduling infrastructure of Linux kernel. The framework contains two components, *Heart Rate Monitor* (HRM), which inherits the ideas of Application Heartbeats [Hoffmann et al. 2010b] improving the support of multi/manycore system and operating system kernels. The framework we propose here perfectly fits the system architecture at the very base of Metronome including sensors, effectors, and adaptation policies. Results demonstrating the effectiveness of reinforcement learning in achieving desired throughput goals have been presented in Panerati et al. [2013].

AdaptGuard is a framework for guarding autonomic systems from instability caused by software anomalies and faults proposed by Heo and Abdelzaher [2009]. The approach of AdaptGuard is not strictly correlated to the one proposed here. Nevertheless, we might envision the two frameworks work hand-in-hand, with AdaptGuard protecting the system during its learning phase. Also, a policy can implement the AdaptGuard rules to check that when the optimal policy changes at runtime during the program execution, this change is detected and acted upon by the framework. The theory behind AdaptGuard is not at present incorporated in the proposed work but it is easy enough to specify new policies with the proposed framework and to define a policy that obeys also to the rules determined by the tool.

2.2. Frameworks for Reconfiguration

SElf-awarE Computing (SEEC) [Hoffmann et al. 2011, 2012] is a comprehensive framework exploiting Application Heartbeats as a self-monitoring provider and different decision-making techniques. The primary decision-making mechanism of

SEEC is based on control-theory, though the framework utilizes machine learning for refining the models of applications and systems provided by developers. SEEC includes an actuator to change the scheduling priorities of threads and an actuator affecting the working frequency of the CPUs. This set of actuators is considered basic to give to reconfigured elements the capability to adjust themselves.

In Sima and Bertels [2009] and Sigdel et al. [2009] the physical implementation, hardware or software, of a requested functionality of the system is taken online during the system execution. Moreover, two similar GNU/Linux-based platforms with support for dynamic reconfiguration, with extensive details regarding the online support, are presented in Williams and Bergmann [2004], Santambrogio et al. [2007], Donato et al. [2007], and Santambrogio and Sciuto [2008]. All these solutions proved to be quite effective: they can be used to monitor the system and to react to modification in the runtime environments but adjustments can be made only if they have been previously envisioned and studied at designed time. Unknown situations cannot be dealt with. The *BORPH Operating System* [So and Brodersen 2007, 2008] is an extended Linux kernel that is able to handle field-programmable gate array (FPGA) resources as native computational resources on the *Berkeley Emulation Engine 2* (BEE2) [Chang et al. 2005]. BORPH introduces the concept of hardware process, which is a hardware component running on an FPGA; this hardware process is a standard user process, so it behaves just like a normal software program running on a processor. However there is no possibility to execute a partitioning on a given application to derive a software and a hardware part, and there is no automatic flow that is able to generate a hardware process from a high-level specification. Thus each change in the high-level specification of the problem has to be directly translated in a manual change of the low-level hardware description. All the works cited so far are effective solutions for promoting adaptive systems and constitute important milestones. However, they still need to be embraced by more comprehensive, autonomic operating systems and runtimes such as the K42 research operating system. *K42* [Krieger et al. 2006] is a research operating system embracing the concepts of autonomic computing and hence making monitoring and adaptation first-class citizens. Taking advantage of solution such as BORPH, and Caronte [Donato et al. 2007] while leveraging autonomic computing like in K42 could enable a whole new class of applications capable of seamlessly employing processors, coprocessors, and fabrics to achieve high performance and full utilization on heterogeneous systems with minimal effort for developers and maximal benefit for users. K42 provides a comprehensive support for both monitoring and adaptation; however, it lacks support and formalism when it comes to decision-making. *AQuoSA* [Palopoli et al. 2009] is an extension of the Linux kernel comprising a runtime to enable reasoning and formal decision-making in operating systems. AQuoSA provides a control theoretical framework to allow soft real-time applications meeting user-specified QoS requirements through adaptive CPU reservation.

2.3. Reinforcement Learning

In Tesauro et al. [2005] and Tesauro [2007], *Unity*, a framework to build self-managing distributed systems, showed how autonomic systems can be dealt with as multiagent systems through utility functions and (hybrid) reinforcement learning. The insight obtained through Unity were used in the development of commercial products such as IBM WebSphere Extended Deployment and IBM Tivoli Intelligent Orchestrator proving the practical utility of reinforcement learning.

Model-free reinforcement learning allowed Tan et al. [2009] to learn robust and near-optimal dynamic power management for devices such as hard drives, WLAN cards, and many more. Wang et al. [2011] claim that reinforcement learning is the only way to deal with uncertainty and variability coming from hardware, software, and environmental context.

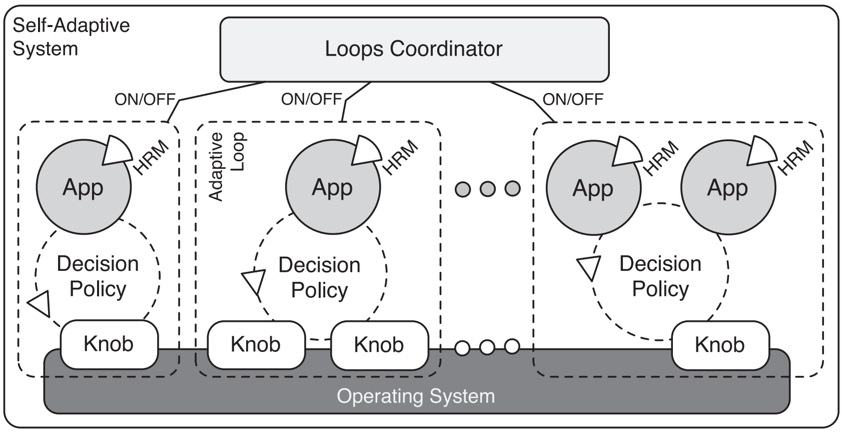

Fig. 1. Multiple adaptive control loops may coexist in a self-adaptive system; the loops coordinator is in charge of properly activating and deactivating these loops to meet the perfomance goals.

3. PROPOSED APPROACH

Common self-adaptive systems may be made up of several adaptive control loops or simply, *loops*; the coordination of these independent, and possibly conflicting, loops is the problem addressed by this work.

The simplest loop is made up by (1) an application that is able to expose relevant information about its behavior, (2) a system knob, i.e., a system feature whose value can be dynamically tuned, and (3) a policy to take decision at runtime on the next value of the knob, given the current value exposed by the application. Applications can provide relevant information to the system if instrumented with *monitors*; a performance monitor, HRM, is exploited in our approach; a core allocator and a frequency scaler are available as system knobs (for further implementation related details, please refer to Section 3.2). Decision making strategies range from simple rule-based engines to control theoretical machine learning techniques. More complex loops, containing multiple knobs and/or multiple applications, are possible, as shown in Figure 1.

At each time instant, each loop in the system can be active or not. The purpose of the coordinator is to select which loops should be enabled in order to achieve the desired user-specified performance goals. Therefore, based on the results achieved in previous iterations, the coordinator exploits the knowledge obtained by activating and deactivating loops or explores the space of possible solutions to build such a knowledge. Regarding exploration, it is worth noting that, the amount of states exponentially grows with the available loops. Adaptive and exploratory capabilities are given by a machine learning engine that exploits different active reinforcement learning techniques in order to meet the user goals. These techniques are further analyzed in Section 3.1.

3.1. Background

In this section, we introduce the theoretical concepts about decision making and learning used to develop the optimal policy based on quantities that we can measure on the system.

Markov Decision Process Framework. A *Markov Decision Process* (MDP) [Russell and Norvig 2009] is a problem defined in a fully observable, stochastic, stationary environment and it is composed by four elements:

(1) a finite set of states S,
(2) a finite set of actions A,

(3) a stochastic transition model (i.e., a probability distribution) $P(s'|s, a)$ defining the probability of going from state s to state s' by performing action a,
(4) a reward function $R(s)$ returning the reward for each state.

The solution of a MDP is a *policy* $\pi(s) : s \in S \mapsto a \in A$ (i.e., function returning an action in each state). The domain of such function is the entire set S since all states are reachable in reason of the stochastic transition model. The *optimal policy* is the solution that maximizes the expected utility $E[U]$, where the *utility* U (i.e., the performance measure) is itself a function $U([s_0, s_1, \ldots, s_n])$ of the states covered by the agent along its path. There are several ways to express utility, we adopt infinite horizon with discounted rewards, whose equation reads:

$$U([s_0, s_1, \ldots, s_i, \ldots]) = R(s_0) + \sum_{i=1}^{\infty} \gamma^i R(s_i), \tag{1}$$

where γ is the *discount factor*. A possible interpretation of this factor is the "probability of being alive in the next time step" of the agent. Using infinite horizon we do not need to specify a lifespan that might be unknown a priori. The discount factor, however, forces the summation of infinite elements to have a finite value thanks to the property of infinite geometric series. Wo chose discounted-rewards over average-rewards to discourage solutions that might be very penalizing during the initial stages.

Reinforcement Learning. Reinforcement learning is arguably the most flexible methodology in machine learning due to its reward-based programming model that does not require specifying a way to accomplish a task [Kaelbling et al. 1996]. Reinforcement learning algorithms in which an agent learns how to accomplish a task through the interaction with the environment, in a trial-and-error process, go under the label of *active reinforcement learning* algorithms, in contrast to *passive reinforcement learning* algorithms for which a policy is given a priori [Russell and Norvig 2009]. A framework for active reinforcement learning is made up of elements (1), (2), and (4) described in the previous Markov Decision Process Framework section definition.

Hence, such a framework contains the same elements of the MDP framework with the exception of the transition model, which is exactly what the learning agent should find out. The learning entity is placed in the environment and it performs actions that affects the state of the environment; whenever an action is performed and a new state of the environment is reached, the agent receives an immediate reinforcement signal (i.e., reward). A reinforcement learning algorithm takes advantage of the trial-and-error process in order to derive a policy that maximizes the long run collection of rewards. In order to solve the problem, there are two ways to proceed: (1) develop a model to derive a controller (i.e., model-based), and (2) directly develop a controller through learning (i.e., model-free).

A simple model-based learning algorithm exploits *adaptive dynamic programming* (ADP) in order to infer the transition model of the underlying MDP. This is achieved by keeping count of how many times an action a has been performed in a given state s and how many times action a has brought the agent from state s to state s'. Given this approximate transition model, the optimal policy can by computed in a straightforward fashion by using the *value iteration* or the *policy iteration* algorithm [Russell and Norvig 2009].[1]

In the following a model-free solution is proposed, since we assume no prior knowledge on the system. The solution is based on *Q-Learning* [Watkins and Dayan 1992]. In Q-Learning, the agent learns the utility value of performing action a in state s, the

[1]These are well-established methodologies therefore we omit their details.

so called *Q-Values* $Q(s,a)$. Given the Q-Values, the utility of each state can be derived as the largest Q-value among those associated with a state.

$$U(s) = max_a Q(s,a). \tag{2}$$

The rule to update the utility value when the agent moves from state s to state s' is:

$$Q(s,a) = Q(s,a) + \alpha(R(s) + \gamma \ max_{a'} Q(s',a') - Q(s,a)), \tag{3}$$

where γ is the *discount factor* and α is a parameter that decreases to zero as a function of how many times a pair $< s, a >$ has been observed; as long as this property stands, Q-Learning will eventually converge. In Q-Learning, the agent chooses its next action a after deciding if it is going to explore the environment or exploit the knowledge.

In our experiments, the state corresponds to the loops that are kept active and an action from one state to another implies breaking an existing or creating a new loop. Loops that are active allow their code to act on manipulated variables – for example the number of core assigned to a specific application, its nice number, the frequency of the cores on the machine – to obtain a specific objective, like the speed of a certain application or a reduction in power consumption.

3.2. Implementation

An infrastructure is needed in order to realize the proposed system. A monitoring system is necessary in order to obtain the execution speed of a specific program, to understand which actions are to be taken on it. Every loop can use the same infrastructure to compute a control signal with whichever technique is considered to be more useful. The coordinator uses the feedback to understand if the single loops are acting correctly or failing in driving the feedback signal to its reference value.

Monitoring Infrastructure. The coordinator relies on HRM to provide self-monitoring capabilities. HRM exploits the well-established ideas of *heartbeat* and *heart rate*, which have been used in several other works both as a measure of availability or performance, and as a metric for the declaration of performance objectives. HRM is implemented as an extension of the Linux kernel and it supports diverse parallelization models (i.e., multiprocessing, multithreading, and feasible mixes) through the concept of *group* and exploiting multicore processors by avoiding synchronization and adopting cache-friendly data structures.

HRM provides the coordinator with:

(1) a compact API to instrument applications, allowing them to specify user-defined performance goals and to register progression;
(2) a generic performance measure delivered by each instrumented application as a generalization of an application-specific metric;

Applications instrumented with HRM translate user-defined performance goals into a generic performance measure. For instance, the *x264* application from the PARSEC 2.1 benchmark suite [Bienia 2011] is instrumented to make a one-to-one translation from *frames/s* to *heartbeats/s* and it emits a heartbeat for each frame is computed. If the performance goal specified by the application (or the user) says that it should produce 30 heartbeats per second, it means that the applications has to encode 30 frames in a second.

Adaptive Loops. The system exploits different knobs in order to provide self-adjusting capabilities, examples being a *core allocator* and a *frequency scaler*. Knobs are encapsulated within libraries, which abstract their operations in the form of finite sets of actions to better support adaptation policies.

The *core allocator* is implemented as a wrapper library around the `sched_setaffinity(2)` system call[2], which alters the affinity mask of tasks. The affinity mask of a task specifies on which cores (of a multicore processor) the task is allowed to be scheduled on. In Linux systems, by default, tasks are eligible to be scheduled on every core and the Linux kernel tends to balance the load on all the cores. The *core allocator* is a knob implementing the ability to decrease or increase the number of cores that can be exploited by a parallel application. Local decision can be made for a specific application, as well as global decision on the set of all active and monitored applications.

The library implementing the *frequency scaler* is a wrapper around the `cpufrequtils` package[3], a set of user-space utilities designed to interface with the Linux kernel assisting with frequency scaling. In this case, local decisions are not supported since they would require to change the working frequency of a core possibly at each context switch operated by the scheduling infrastructure of the Linux kernel, and second, the frequency scaler supports a single system-wide working frequency.

Another loop, implemented for the only purpose of testing, simply does nothing to help the application in reaching its goals. The coordinator should learn that it is useless and deactivate it. Much more loops can be envisioned. For example, one loop can insert idle cycles in cores to maintain the core temperature in a certain range [Bailis et al. 2011].

4. EXPERIMENTAL RESULTS

Experimental results were collected on a workstation equipped with a single Intel Core i7-870 quad-core processor running natively at 2.93 GHz with 14 different operating points[4] featuring 8 MB of shared LLC (L3), 4 GB of DDR3-1066 non-ECC RAM, and a 500 GB 7200 RPM SATA2 hard disk.[5] Advanced features such as Intel Hyper-Threading Technology and Intel Turbo Boost Technology were disabled while the governor of Enhanced Intel SpeedStep Technology was set to *userspace* through the `cpufrequtils` package granting ad-hoc tuning capabilities. The AMD64 version of Debian 6.0 alias "squeeze" was configured to run Linux 2.6.35.14 extended with HRM.

Six applications from the PARSEC 2.1 benchmark suite [Bienia 2011], were instrumented with HRM in order to gather experimental evidence that our framework is capable of learning at runtime how to drive applications towards their performance goals. Applications composing the PARSEC benchmark suite are gathered from many different areas: *blackscholes* is a financial application computing option pricing; *canneal* performs a heuristic optimization of the routing costs in a chip design; *dedup* is a compression algorithm; *swaptions*, another financial application, computes the pricing of a portfolio; *x264* is a video encoding algorithm; and *raytrace* is a real-time rendering algorithm. Their instrumentation aims at measuring *options/s* in *blackscholes*; *exchanges/s* in *canneal*; *chucks/s* in *dedup*; *simulations/s* in *swaptions*; and *frames/s* in *x264* and *raytrace*. Table I reports reference metrics for the heart rate of each application. We let our machine learning framework run alongside these applications, one at a time. Each application was run ten consecutive times with its native input bundled with the benchmark suite. During iterations 1 and 2, the machine learning engine was let free to perform random adjustments in the number of cores and the operating frequency, this phase is called *exploration*. From iteration 3 to 10, the machine learning

[2]http://kernel.org/doc/man-pages/online/pages/man2/sched_setaffinity.2.html.

[3]http://kernel.org/pub/linux/utils/kernel/cpufreq/.

[4]The working frequencies of the available operating points are: 1200, 1333, 1466, 1599, 1732, 1865, 1998, 2131, 2264, 2397, 2530, 2663, 2796, and 2929 MHz.

[5]Hard drive specs, however, do not have a significant impact on the experimental results because the benchmark applications under test are not disk-bound.

Table I.
Heart rate semantic, desired heart rate range, average, minimum and maximum heart rate registered in the experiments for the six applications from the PARSEC benchmark suite.

application	HR Semantic	Goal HR Range	Avg. HR	Std. Dev.	Min. HR	Max. HR
blackscholes	options/s	$8.23 \cdot 10^6 - 12.35 \cdot 10^6$	$8.57 \cdot 10^6$	$3.38 \cdot 10^6$	$0.11 \cdot 10^6$	$20.58 \cdot 10^6$
canneal	exchanges/s	$0.8 \cdot 10^6 - 1.25 \cdot 10^6$	$1.04 \cdot 10^6$	$0.26 \cdot 10^6$	$0.04 \cdot 10^6$	$2.17 \cdot 10^6$
dedup	chucks/s	$3.16 \cdot 10^3 - 4.74 \cdot 10^3$	$5.11 \cdot 10^3$	$3.62 \cdot 10^3$	$0.07 \cdot 10^3$	$26.40 \cdot 10^3$
raytrace	frames/s	$5.60 - 8.40$	6.67	2.40	3.00	17.00
swaptions	simulations/s	$37.96 \cdot 10^3 - 56.94 \cdot 10^3$	$41.92 \cdot 10^3$	$13.66 \cdot 10^3$	$2.71 \cdot 10^3$	$95.26 \cdot 10^3$
x264	frames/s	$6.40 - 9.60$	7.90	4.51	2.00	38.00

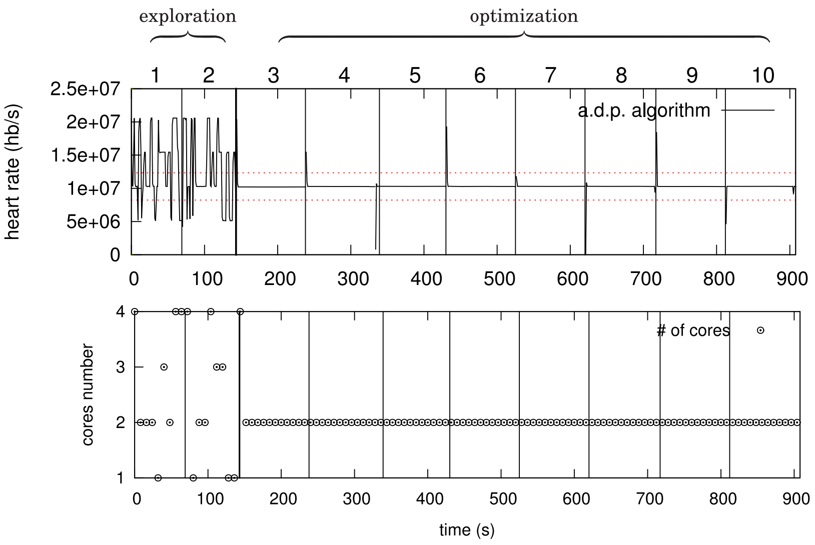

Fig. 2. Benchmark application *blackscholes* controlled *via* core allocation.

engine exploits the knowledge gathered up to this point so far in order to drive the application heart rate in a predefined window/range (please notice that this window *does not* have any specific semantic), we call this phase *optimization*. We chose to split the two phases in 2 learning iterations and 8 optimization iterations because we wanted to show that our framework is capable of learning with a limited amount of data. However, we believe it is still important to have two different samples, so that behaviors that might be only episodic would not be learned unless they repeat in consecutive iterations.

When our framework is able to choose the number of cores allocated to the applications from the PARSEC benchmark suite and the operating frequency so that, since iteration 3, the application heart rate falls inside the pre-defined window, we say the learning was successful and the control is effective. Figures from 2 to 7 show our results as pairs of plots. In each figure, the top plot shows the heart rate of the application, the bottom one shows the number of cores the application can be scheduled on and the operating frequency. Vertical bars are used to separate the ten consecutive runs.

Figure 2 shows how the machine learning engine can easily learn which is the right number of cores[6] on which the the *blackscholes* application has to be mapped in order to keep its heart rate in the desired window. However, Figure 3 bring another interesting

[6]Experiments where only *core allocation* is enabled were carried out with the system frequency fixed to its maximum.

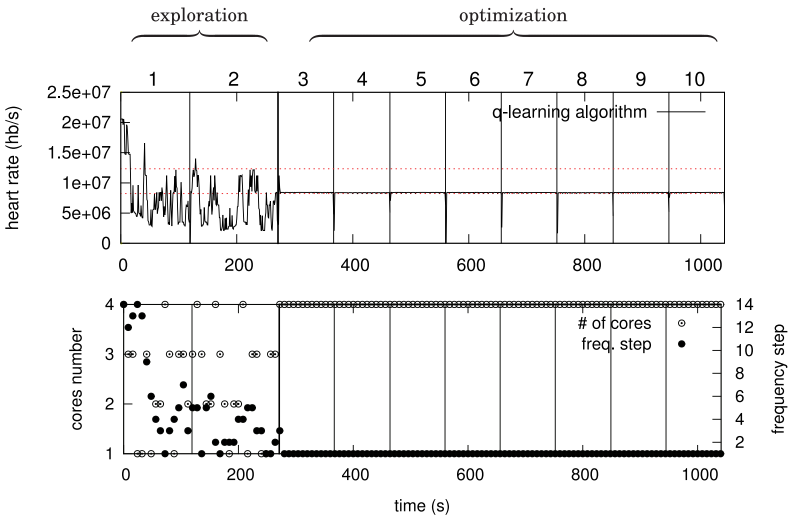

Fig. 3. Benchmark application *blackscholes* controlled *via* core allocation and frequency scaling.

result. Using the Q-learning algorithm, a machine learning engine capable of exploiting both the *core allocation* and the *frequency scaling* finds a different solution to keep the same application in the same heart rate window. Here, the application is scheduled on four cores but the operating frequency is kept at the lowest step. The resulting heart rate is lower than in Figure 2, but still inside the desired window. One of the advantages of using a machine learning solution over a control theory one lies in this multiple optimal choice, that a control theoretical approach would not find. One of the main disadvantages, however, is the absence of formal guarantees that this solution will be found. In practical cases, we experienced that our machine learning engine has always been able to find a viable solution. More details about qualitative comparisons among different decision making techniques can be found in Maggio et al. [2012].

Figures 4 and 5 show how, similarly, the heart rate of the application *canneal* can be kept inside the desired window. Again, it is worth noting that the experiment in which the machine learning engine uses both the *core allocator* and the *frequency scaler* results in a slightly different solution. By using only the *core allocator*, the application is scheduled on three cores; when using also the *frequency scaler*, the learning algorithm discovers that the application can be scheduled on only two cores if the frequency is kept at the highest step. Moreover, notice how, on iteration 3 of the experiment using both the *core allocator* and the *frequency scaler*, the learning framework is trying to exploit another, different, configuration: four cores but the lowest frequency. This results in a strongly oscillating heart rate. Because of the fact that the learning process continues even after the end of the pure exploratory phase, the algorithm converges to a better solution before the end of the third iteration.

Figure 6 shows the experiment with application *dedup*. We used the Q-learning algorithm and the two control loops *core allocator* and *frequency scaler*. Even though, these results are only fairly good (the heart rate is inside the desired window most of the time in iterations from three to six and in the last one, but it shows an oscillating behavior), it is worth noting that the learning algorithm is able to recover from its own errors (errors that may always arise because of the intrinsic stochasticity in an MDP). To clarify why the heart rate can be outside the desired window, we want to stress that during the iteration the heart rate depends on the application execution. An application that has phases might have a phase when the application is faster and a

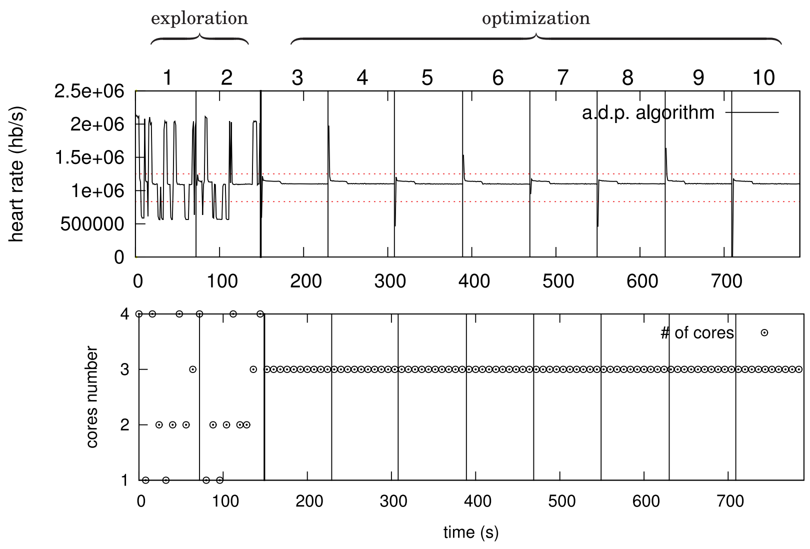

Fig. 4. Benchmark application *canneal* controlled *via* core allocation.

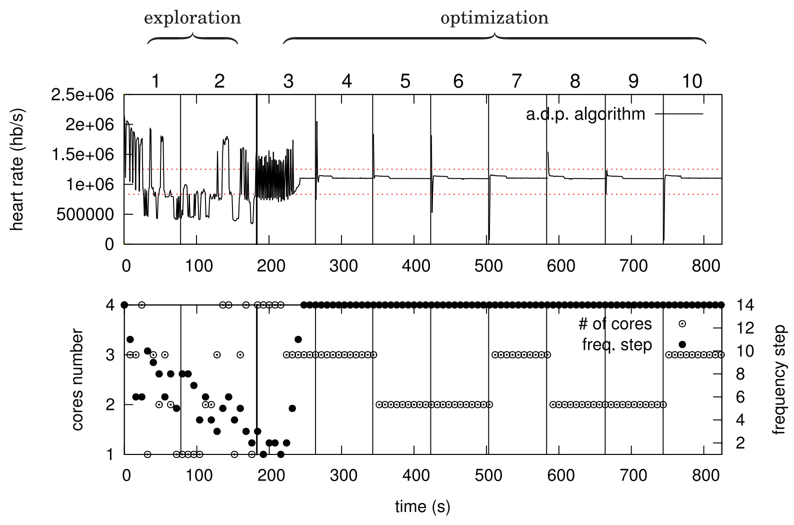

Fig. 5. Benchmark application *canneal* controlled *via* core allocation and frequency scaling.

phase when the application is slower. However, in iterations 7, 8 and 9, our framework starts to incorrectly assume that the application should be scheduled on all four cores. However, by the time of the tenth iteration, the heart rate is led back inside the desired window.

Figure 7 shows another example of an application (*swaptions*, in this case) controlled *via core allocator* and *frequency scaler*. As a remark, in this experiment, when only the core allocator is active, the machine learning algorithm ends up providing the same solution in both cases: always scheduling the application on two cores.

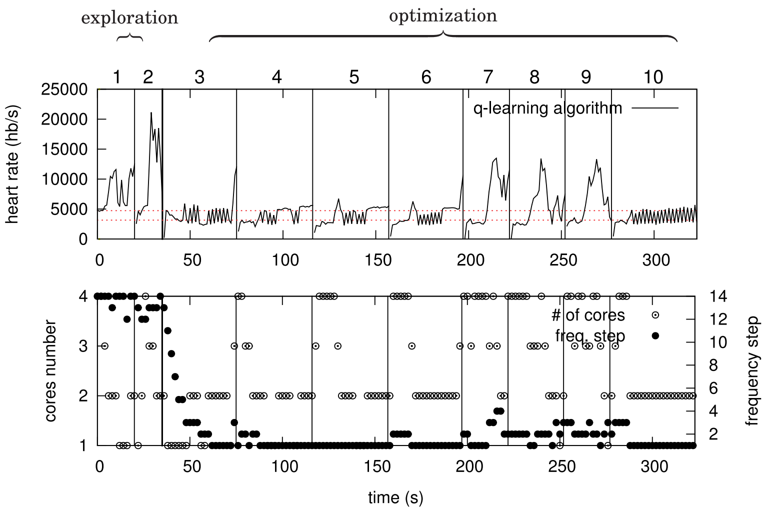

Fig. 6. Benchmark application *dedup* controlled *via* core allocation and frequency scaling.

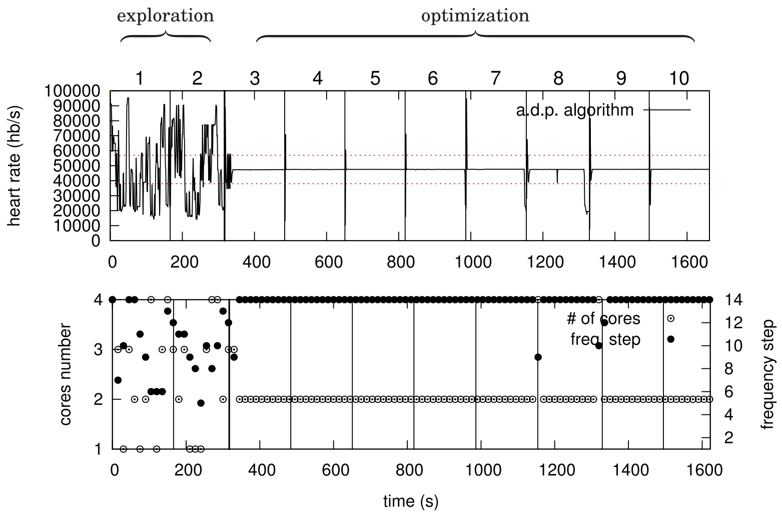

Fig. 7. Benchmark application *swaptions* controlled *via* core allocation and frequency scaling.

4.1. Comparison with Previous Work

Maggio et al. [2012] dealt with the same problem of controlling the behavior of an application instrumented with heartbeat calls through operating system level parameters, using the PARSEC benchmark suite as well. The objective of Maggio et al. [2012] is to compare decision making strategies—control theory, machine learning and heuristic methods. Therefore, the solutions adopted in Maggio et al. [2012] for each of these domains are well assessed. On the other hand, in this work, we evaluate a novel framework and the focus is on the performance and flexibility achieved by it.

On the implementation side, the PARSEC benchmark suite is here instrumented with HRM [Sironi et al. 2012], while Application Heartbeat [Hoffmann et al. 2010a] was used in Maggio et al. [2012]. It is worth noticing that the usage of different

Table II.
Mean squared error normalized to the user-defined performance goal for every combination of reinforcement learning algorithms and dynamic knobs.

Adaptive Approach	PARSEC Application					
	blackscholes	*canneal*	*dedup*	*raytrace*	*swaptions*	*x264*
ADP (cores)	0.16	0.11	0.57	0.17	**0.10**	**0.27**
ADP (cores & freq.)	**0.11**	0.11	**0.34**	0.17	0.10	0.28
QL (cores)	0.12	0.12	1.02	**0.14**	0.11	0.47
QL (cores & freq.)	0.12	**0.10**	0.47	0.19	0.11	0.39

Table III. The ISE_w Error Metric of the Control Approaches Proposed by Maggio et al. [2012] Compared to Our Best Results

application	Heuristic Control		Control Theory		Neural Networks		best MSE
	cores	cores & freq.	cores	cores & freq.	cores	cores & freq.	in this work
blackscholes	**0.70**	**0.20**	**0.37**	**0.64**	**0.27**	**0.15**	0.11
canneal	**0.16**	**0.37**	**0.21**	**0.19**	**0.43**	**0.30**	0.10
dedup	0.02	0.03	0.03	0.03	0.04	0.04	0.34
raytrace	0.00	**0.29**	0.00	0.01	**0.18**	**0.19**	0.14
swaptions	0.02	**0.14**	0.03	0.03	**0.16**	**0.22**	0.10
x264	**0.54**	**0.53**	0.14	0.20	**0.51**	0.25	0.27

Note: The best control theory approach for each application among the three implemented in Maggio et al. [2012].

instrumentation API could make any confrontation of heart rates absolute values meaningless. We cope with this issue by only looking at normalized error measures.

We also have to consider the fact that, while rational learning agents implemented in this work have reward functions assigning different values to different heart rate ranges, the controllers implemented in Maggio et al. [2012] have a “target value” computed, in this case, as the average value between the upper bound and the lower bound of the highest rewarded heart rate range. Because of this, when we report the error, this value is computed in two different way for our reinforcement learning approach and the controllers from Maggio et al. [2012]. In our approach, the computed error is the average distance from the highest rewarded heart rate range, for the other controllers in Maggio et al. [2012] the error is the distance from their target values.

Table II reports the mean squared error normalized to the performance level requested by the user for the reinforcement learning algorithms and the actuators they use. These errors are quite low and most benchmarks can be controlled with each of these algorithms with small performance variations. The case of *dedup* is quite interesting since the benchmark seems to be harder than the others to be controlled. Table III reports the ISE_w error metric, as defined in [Maggio et al. 2012] to perform a comparison between the reinforcement learning techniques introduced in this article and previous work. As we can see, for *blackscholes* and *canneal*, the reinforcement learning techniques introduced here perform better than any previous solution, while for *raytrace* and *swaptions* the techniques are inline with previous results. Again, *dedup* seems to be the only case where reinforcement learning performs poorly with respect to existing solutions. In synthesis, we believe that the reinforcement learning techniques introduced in this article add a contribution to the scenario of existing solutions.

5. MULTIPLE APPLICATIONS SCENARIO

Experimental results reported so far deal with a single-application scenario. Extending this work in the context of multiple applications brings new challenges. Two different approaches are available:

Table IV.
Standard mean, standard deviation, and speed-up over the execution time of each workload run either unmanaged or managed. Experiments were repeated 100 times

	Unmanaged Execution Time		Managed Execution Time		
Workload	Std. Mean [s]	Std. Deviation [s]	Std. Mean [s]	Std. Deviation [s]	Speed-Up
mix 1	151.25	5.10	**118.00**	0.70	1.28×
mix 2	176.25	2.90	**142.50**	1.10	1.24×
mix 3	**216.00**	0.20	217.00	0.20	0.99×

—A centralized adaptive system (a single MDP). *Pros:* the solution is the optimal global strategy. *Cons:* exponential state-space explosion with the number of applications.
—A distributed adaptive system (a collection of MDPs). *Pros:* scalable with the number of application. *Cons:* sub-optimal, need for a *resource arbiter*.

In order to avoid the scalability issues, the same platform described in Sections 3 was used to achieve individual (distributed) adaptiveness in four instrumented applications at the same time. Some of these applications have multiple threads contending (i.e., lock/unlock of a critical section) a per-application shared resource, some others execute without the need for synchronization. An ad-hoc synchronization library was developed by exploiting HRM to measure the contention over a shared resource, the higher the heart rate, the higher the contention (i.e., number of failed locking operating per second). The synchronization library is responsible for providing self-monitoring capabilities. Self-adjusting capabilities are granted by the core allocator.

The loop coordinator is expected to learn either "to force the interleaved execution of contending threads placing them on the same core" or "to enforce the parallel execution of noncontending threads" depending on the application and assuming that serialized execution is advantageous in presence of fine-grain synchronization while parallel execution of threads is advantageous in absence of synchronization.

Experimental results in Table IV were collected using two different micro-benchmarking applications executed concurrently. The first application, *sync*, shows an high degree of synchronization among its threads while the second application, *nosync*, has no synchronization among its threads. Three different workloads were devised.

—*mix 1*. four 4-threaded instances of *sync*
—*mix 2*. two 4-threaded instances of *sync* and two 4-threaded instances of *nosync*
—*mix 3*. four 4-threaded instances of *nosync*

It is worth noting the speed-up values of the three cases. The first and the second workloads experience a speed-up between 1.24–1.28× since moving threads can actually improve performance due to contention reduction. On the other hand, the third workload experiences a negligible slow-down due to the presence of the self-monitoring and self-adjusting capabilities that take time without being actually exercised.

6. CONCLUSIONS

In this article we presented a coordination strategy, that acts as a decision engine in a self-aware architecture capable of self-optimization and self-adaptation to unpredictable, unknown, and unfavorable conditions. The architecture is composed by multiple loops and a central coordinator, which learns from experience how to optimize the performance of the whole system. The introduction of such central entity, based on machine learning, is the main contribution of this article over previous work described in literature.

REFERENCES

Peter Bailis, Vijay Janapa Reddi, Sanjay Gandhi, David Brooks, and Margo Seltzer. 2011. Dimetrodon: Processor-level preventive thermal management via idle cycle injection. In *Proceedings of the 48th Design Automation Conference (DAC'11)*. ACM, New York, 89–94. DOI:http://dx.doi.org/10.1145/2024724.2024745

Christian Bienia. 2011. Benchmarking modern multiprocessors. Ph.D. Dissertation, Princeton University.

C. Chang, J. Wawrzynek, and R. W. Brodersen. 2005. BEE2: A high-end reconfigurable computing system. *IEEE Des. Test Comput.* 22, 2, 114–125. DOI:http://dx.doi.org/10.1109/MDT.2005.30

Alberto Donato, Fabrizio Ferrandi, Massimo Redaelli, Marco D. Santambrogio, and Donatella Sciuto. 2007. Exploiting partial dynamic reconfiguration for SoC design of complex application on FPGA platforms. http://www.dresd.org/files/IFIP_VLSI_02.pdf.

Jin Heo and Tarek Abdelzaher. 2009. AdaptGuard: Guarding adaptive systems from instability. In *Proceedings of the 6th International Conference on Autonomic Computing*. 77–86.

Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. 2010a. Application heartbeats for software performance and health. In *Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 347–348.

Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. 2010b. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th International Conference on Autonomic Computing*. 79–88.

Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. 2010c. Application heartbeats for software performance and health. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*. 347–348.

Henry Hoffmann, Jim Holt, George Kurian, et al. 2012. Self-aware computing in the Angstrom processor. In *Proceedings of the IEEE/ACM Design Automation Conference*, Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, Eds. ACM, 259–264.

Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. 2011. SEEC: A framework for self-aware management of multicore resources. Tech. Rep., MIT–CSAIL–TR–2011–016, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement learning: A survey. *J. Artif. Int. Res.* 4, 1, 237–285.

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

O. Krieger, M. Auslander, B. Rosenburg, R. J. W. Wisniewski, Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. 2006. K42: Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*. 133–145. DOI:http://dx.doi.org/10.1145/1217935.1217949

Martina Maggio, Henry Hoffmann, Alessandro V. Papadopoulos, Jacopo Panerati, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. 2012. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.* 7, 4, Article 36. DOI:http://dx.doi.org/10.1145/2382570.2382572

M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. 2010. Controlling software applications via resource allocation within the Heartbeats framework. In *Proceedings of the 49th Conference on Decision and Control*. 3736–3741.

Martina Maggio, Henry Hoffmann, Marco D. Santambrogio, Anant Agarwal, and Alberto Leva. 2011. Decision making in autonomic computing systems: Comparison of approaches and techniques. In *Proceedings of the 8th International Conference on Autonomic Computing*. 201–204.

L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. 2009. AQuoSA - adaptive quality of service architecture. *Softw. Pract. Exper.* 39, 1, 1–31. DOI:http://dx.doi.org/10.1002/spe.v39:1

Jacopo Panerati, Filippo Sironi, Matteo Carminati, Martina Maggio, Giovanni Beltrame, Piotr J. Gmytrasiewicz, Donatella Sciuto, and Marco Domenico Santambrogio. 2013. On Self-adaptive resource allocation through reinforcement learning. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*.

Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* 3rd ED. Prentice Hall.

M. D. Santambrogio and D. Sciuto. 2008. Design methodology for partial dynamic reconfiguration: A new degree of freedom in the HW/SW codesign. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*. 1–8. DOI:http://dx.doi.org/10.1109/IPDPS.2008.4536542

Marco D. Santambrogio, Seda Ogrenci Memik, Vincenzo Rana, Umut A. Acar, and Donatella Sciuto. 2007. A novel SoC design methodology combining adaptive software and reconfigurable hardware. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'07)*. IEEE, 303–308.

Kamana Sigdel, Mark Thompson, Andy Pimente, Koen Bertels, and Carlo Galuzzi. 2009. System level runtime mapping exploration of reconfigurable architectures. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*.

Vlad Sima and Koen Bertels. 2009. Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*.

Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffman, Donatella Sciuto, and Marco D. Santambrogio. 2012. Metronome: Operating system level performance management via self-adaptive computing. In *Proceedings of the 49th Design Automation Conference*.

H. K.-H. So and R. Brodersen. 2008. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Trans. Embed. Comput. Syst.* 7, 2, 14:1–14:28. DOI:http://dx.doi.org/10.1145/1331331.1331338

H. K.-H. So and R. W. Brodersen. 2007. BORPH: An operating system for FPGA-based reconfigurable computers. Tech. Rep., UCB/EECS-2007-92, University of California, Berkeley.

R. Sterritt and D. Bustard. 2003. Towards an autonomic computing environment. In *Proceedings of the 14*th *International Workshop on Database and Expert Systems Applications*. 694–698.

Ying Tan, Wei Liu, and Qinru Qiu. 2009. Adaptive power management using reinforcement learning. In *Proceedings of the International Conference on Computer-Aided Design*. 461–467.

Gerald Tesauro. 2007. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Comput.* 11, 1, 22–30.

G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart. 2005. Utility-function-driven resource allocation in autonomic systems. In *Proceedings of the 2nd International Conference on Autonomic Computing*. 342–343.

Yanzhi Wang, Qing Xie, Ahmed Ammari, and Massoud Pedram. 2011. Deriving a near-optimal power management policy using model-free reinforcement learning and Bayesian classification. In *Proceedings of the 48th Design Automation Conference*. 41–46.

Christopher J. C. H. Watkins and Peter Dayan. 1992. Technical note Q-Learning. *Machine Learn.* 8, 279–292.

J. Williams and N. Bergmann. 2004. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*. Toomas P. Plaks (Ed.), CSREA Press, 163–169.