On Self-adaptive Resource Allocation through Reinforcement Learning

Jacopo Panerati,* Filippo Sironi,† Matteo Carminati,† Martina Maggio‡
Giovanni Beltrame,* Piotr J. Gmytrasiewicz,§ Donatella Sciuto,† Marco D. Santambrogio†
*École Polytechnique de Montréal, †Politecnico di Milano, ‡Lund University, §University of Illinois at Chicago
jacopo.panerati@polymtl.ca, sironi@elet.polimi.it, mcarminati@elet.polimi.it, martina.maggio@control.lth.se
giovanni.beltrame@polymtl.ca, piotr@cs.uic.edu, sciuto@elet.polimi.it, santambrogio@elet.polimi.it

***Abstract*—Autonomic computing was proposed as a promising solution to overcome the complexity of modern systems, which is causing management operations to become increasingly difficult for human beings. This work proposes the Adaptation Manager, a comprehensive framework to implement autonomic managers capable of pursuing some of the objectives of autonomic computing (i.e., self-optimization and self-healing). The Adaptation Manager features an active performance monitoring infrastructure and two dynamic knobs to tune the scheduling decisions of an operating system and the working frequency of cores. The Adaptation Manager exploits artificial intelligence and reinforcement learning to close the Monitor-Plan-Analyze-Execute with Knowledge adaptation loop at the very base of every autonomic manager. We evaluate the Adaptation Manager, and especially the adaptation policies it learns by means of reinforcement learning, using a set of representative applications for multicore processors and show the effectiveness of our prototype on commodity computing systems.**

I. Introduction

The vision of autonomic computing [5, 11] is about computing systems capable of managing themselves without (or with limited) intervention by human beings. The term *autonomic* was emblematic of natural (e.g., human bodies), social (e.g., human society), and economic (e.g., world wide economy) self-governing systems consisting of a myriad of interacting autonomous elements. The original belief was that autonomic computing systems should seek inspiration from the real world. Despite the original idea, most of the researchers put a lot of effort in designing and developing autonomic computing systems managed either by centralized entities or by hierarchies of entities employing heuristic solutions (e.g., set of rules, etc.) or control theory (e.g., model predictive control, etc.) to embed domain knowledge and provide self-adaptation capabilities. Although the techniques employed so far proved to be valuable, embedding an effective amount of domain knowledge may be extremely difficult for large-scale autonomic systems.

Recently, researchers are exploiting well-known techniques borrowed from the artificial intelligence community allowing autonomic computing systems to learn effective adaptation policies and to autonomously acquire most of the domain knowledge, leaving users with the task of providing high-level goals [13]. In particular, reinforcement learning has the potential to achieve better performance than traditional adaptation policies while still requiring less "a priori" domain knowledge [21]. Moreover, adaptation policies may exploit utility functions instead of simple goals providing additional expressiveness [12].

Motivated by this trend, we decided to focus on the design and development of *autonomic elements*, the combination of an *autonomic manager* and a *managed element*, exploiting artificial intelligence techniques to achieve self-adaptation. In this paper we make the following contributions:

- We propose the *Adaptation Manager* (AdaM), a framework to realize autonomic managers. AdaM comes with sensors, granting self-awareness, and actuators, providing self-optimization, which can be exploited by adaptation policies to achieve self-adaptation.
- We study and implement two reinforcement learning algorithms harnessing the infrastructure of AdaM, the first is *Adaptive Dynamic Programming* (ADP) while the second is *Q-learning*. Moreover, we employ utility functions to represent user-defined performance goals and drive the reinforcement learning algorithms.
- We evaluate AdaM through a representative subset (i.e., applications showing heterogeneous behaviors) of the PARSEC 2.1 benchmark suite [4] and show how applications can be driven towards meeting user-defined performance goals.

The rest of this paper is organized as follows. Section II introduces related works at the intersection of autonomic computing and artificial intelligence and briefly presents the previous work we embraced. Section III reports a minimal artificial intelligence background and describes the reinforcement learning algorithms we adopted, while Section IV sketches the implementation of AdaM. Section V reports the experimental evaluation of AdaM; finally, Section VI draws the conclusions and outlines possible research directions.

II. Related Works

In the attempt to summarize significant contributions, that we consider strictly related to the topic of this paper, we present related works whose focus is on fundamental attributes of autonomic computing (i.e., self-awareness, self-adjusting, etc.). We also discuss comprehensive frameworks to realize

This work was partially supported by the *Regroupement Stratégique en Microsystèmes du Québec* (ReSMiQ)

autonomic computing systems, showing how artificial intelligence and reinforcement learning have been exploited so far.

Hoffmann et al. [7], proposed *Application Heartbeats* [1], an active performance monitoring infrastructure providing self-awareness. Application Heartbeats is based on the well-known ideas of heartbeats and heart rate [18] and encapsulates them in a compact API. Application Heartbeats gives applications (i.e., managed elements) a method to export user-defined performance goals and signal progresses towards those goals; in addition, it enables autonomic managers to acquire information about the desired and measured performance. Maggio et al. [14] proposed a control-theoretical decision-making mechanism to allocate resources (i.e., number of cores in a multicore processor) to applications; Application Heartbeats was responsible for providing self-awareness. Maggio et al. [15] also compared many different decision-making mechanism ranging from heuristic to control-theoretical and artificial intelligence-backed solutions. A comprehensive framework exploiting the availability of Application Heartbeats as a self-awareness provider and different decision-making techniques is the *SElf-awarE Computing* (SEEC) framework proposed by Hoffmann et al. [8, 9]. SEEC comes with a decision-making mechanism coupling control-theory and machine learning where the second is employed to refine the models of applications and computing systems provided by the designers and developers. Most of these works employ an actuator to change the mapping of threads and an actuator to change the working frequency of cores, a set of dynamic knobs giving autonomic elements self-adjusting capabilities.

Tesauro [21] and Kephart and Das [12] designed and developed *Unity*, a framework to build self-adaptive distributed computing systems borrowing from the artificial intelligence community. It relies on multi-agent theory to control the interaction among autonomic elements and exploits utility function to specify goals, and (hybrid) reinforcement learning to acquire an effective amount of domain knowledge. Unity was evaluated in a data center scenario, and results from this research were merged with commercial products such as IBM WebSphere Extended Deployment and IBM Tivoli Intelligent Orchestrator, proving the efficiency and effectiveness of reinforcement learning and utility functions.

Recently, reinforcement learning gained more and more attention even outside the artificial intelligence and the autonomic computing communities: Tan et al. [19] and Wang et al. [22] adopted model-free reinforcement learning to learn robust, optimal or near-optimal dynamic power management policies for various components (e.g., hard disk drives, WLAN cards, etc.). The authors advocate the adoption of reinforcement learning as the single way to deal with uncertainty and variability coming from the hardware, the software, and the environment.

Heo and Abdelzaher [6] designed, developed, and evaluated *AdaptGuard*, a framework for guarding autonomic computing systems from instability caused by software anomalies and faults. The approach of AdaptGuard is somehow complementary to that of reinforcement learning; in fact AdaptGuard may be employed during the exploration phase in which it is likely the reinforcement learning efforts may result in a far from optimal policy.

More recently, Sironi et al. [17] proposed *Metronome*, a framework for self-adaptive computing enhancing the scheduling infrastructure of the Linux kernel. Metronome is made up of two components, the *Heart Rate Monitor* (HRM), which is an evolution of Application Heartbeats towards multi and manycore processors capable of exporting the information it carries to a broader number of elements (e.g., the Linux kernel), and *Performance-Aware Fair Scheduler* (PAFS). AdaM perfectly fits the system architecture at the very base of Metronome since it comprises sensors (i.e., HRM), actuators/effectors (i.e., a core allocator and a frequency scaler), and adaptation policies through reinforcement learning realizing a MAPE-K adaptation loop.

III. Adaptation Policies Learning

This section presents the theoretical artificial intelligence background needed to understand AdaM.

Autonomic computing systems are supposed to deal with uncertain non-episodic environments, heterogeneous resources, and irregular workloads; these conditions usually require to face so called sequential decision problems, where previous decisions affect future ones, and to provide systems with the ability to elaborate a plan/strategy.

A. Markov Decision Processes

In such a context we chose to adopt the Markov Decision process mathematical framework. A *Markov Decision Process* (MDP) [16] is a problem defined in a fully observable, stochastic, stationary environment and it is composed of four elements:

1) A set of states S containing the initial state s_0 and at least one final state.
2) A set of actions $A(s)$ for each state.
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' performing action a.
4) A reward function $R(s)$ returning the reward for each state.

The solution of a MDP is a *policy* (i.e., function) $\pi(s) : s \in S \mapsto a \in A(s)$. The policy is defined over all possible states since all of them are reachable due to the stochastic nature of the environment. A solution that maximizes the expected utility is an *optimal policy*, where the *utility* (i.e., the performance measure) is a value $U([s_0, s_1, \ldots, s_n])$, which depends on the rewards collected over the states covered by the agent along its path. There are several ways to express the expected utility, we adopt infinite horizon with discounted rewards (where γ is the *discount factor*) as reported in Eq. (1).

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

B. Active Reinforcement Learning

Reinforcement learning carries huge expectations due to its novel reward-based programming model that does not require specifying a way to accomplish a task [10]. Russell and Norvig [16] draw a distinction between *passive* and *active* reinforcement learning. The former consists of those algorithms in which an agent learns the utilities of the states of a MDP when a fixed policy is specified. The latter, which is used in this work, consists of those algorithms in which an agent learns how to accomplish a task interacting with the environment through a trial-and-error process. A framework for active reinforcement learning is made up of:

- A set S of states an agent can explore within the environment.
- A set A of actions an agent can perform.
- A scalar reinforcement signal an agent receives.

The agent, which is placed in the environment, performs actions that change the state of the environment; whenever an action is performed and/or the state of the environment changes the agent immediately receives a reinforcement signal (i.e., reward). A reinforcement learning algorithm leverages the trial-and-error process described above to devise a policy that maximizes the long run collection of rewards.

Proving properties of reinforcement learning algorithms is non-trivial. Eventual convergence to optimality and speed of convergence to optimality are interesting and for some algorithms and some specific constraints these properties were effectively proved. However, eventual and speed of convergence to optimality are practically useless because they lack information on the decrease in reward and the behavior of the agent due to the learning process (i.e., regret) that is a far more important and difficult property to prove [10].

At this point, there are two ways to proceed: (1) learn a model and then derive a controller (i.e., model-based), and (2) learn the controller (i.e., model-free).

C. Model-Based Learning

Some reinforcement learning algorithms are meant to learn the stochastic transition model of a MDP and, then, derive the optimal policy from the MDP. These algorithms are especially important in applications in which computation is considered cheap and real-world experience is considered costly.

Adaptive Dynamic Programming (ADP) is a model-based reinforcement learning algorithm, an agent exploiting ADP may use two tables:

- $N_{sa}[s, a]$, which keeps track of how many times the agent performed action a when in state s.
- $N_{s'|s,a}[s', s, a]$, which maintains information on how many times the agent performed action a when in state s and landed in state s'.

The estimated stochastic transition model is updated at each learning step following the rule reported in Eq. (2) for each pair $< s, a >$ such that $N_{s,a}[s, a]$ is non zero. Once the stochastic transition model is available, the utility of each state is computed using the iterative equation reported in Eq. (3) or, if a fixed policy π is available, solving the linear system of equations reported in Eq. (4).

$$P'(s'|s, a) = \frac{N_{s'|s,a}[s', s, a]}{N_{s,a}[s, a]} \tag{2}$$

$$U_{i+1}(s) \leftarrow R(s) + \gamma max_{a \in A(s)} \sum_{s'} P(s'|s, a)U_i(s') \tag{3}$$

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))U_i(s') \tag{4}$$

D. Model-Free Learning

Model-free reinforcement learning algorithms are meant to learn directly the optimal policy and their core is the update rule used to compute utility values.

Q-learning is one of such algorithms; however, in Q-learning, the agent learns the utility value of performing action a in state s, the so called *Q-Values* $Q(s, a)$. Given the Q-values, the utility of each states can be derived as reported in Eq. (5).

$$U(s) = max_a Q(s, a) \tag{5}$$

The rule to update the utility value when the agent moves from state s to state s' is reported in Eq. (6).

$$Q(s, a) = Q(s, a) + \alpha(R(s) + \gamma max_{a'} Q(s', a') - Q(s, a)) \tag{6}$$

α is a decreasing function of how many times a pair $< s, a >$ was seen; 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.

IV. Adaptation Manager

The original vision of autonomic computing [11] advocates the realization of *autonomic elements* as individual systems consisting of a *managed element* and an *autonomic manager*. AdaM is a framework to realize autonomic managers capable of driving an instrumented application (i.e., a managed element) towards user-defined performance goals thanks to an active monitoring infrastructure and to a set of dynamic knobs. AdaM provides a skeleton to evaluate many different adaptation policies and, as demonstrated in this work, algorithms capable of learning adaptation policies at runtime.

A. Monitor

AdaM relies on HRM to provide self-awareness. As already explained in Section II, HRM [17], much like Application Heartbeats, exploits the well-known ideas of *heartbeat* and *heart rate*, which were used in the past for both indicating availability, measuring performance, and expressing performance goals. HRM is an extension of the Linux kernel supporting 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 algorithms and data structures.

AdaM adopts HRM to deliver the following capabilities:

- Provide a compact interface to instrument applications allowing them to export user-defined performance goals and register progresses.
- Measure the performance delivered by an instrumented application through a generalization of an application-specific metric.
- Support the realization of adaptation policies in user-space through the consumer portion of the interface of HRM.

Instrumented applications translate user-defined performance goals into a general performance measure and register progresses when a significant point in the execution is reached. For example, we instrumented the *x264* video encoder from the PARSEC 2.1 benchmark suite [4] to make a one-to-one translation from *frames/s* to *heartbeats/s* by emitting a heartbeat every time a frame is encoded.

B. Execute

AdaM depends on two different dynamic knobs to provide self-adjusting capabilities, a core allocator and a frequency scaler. Both the core allocator and the frequency scaler are encapsulated within libraries, which abstract their operations as much as possible to better support adaptation policies.

1) Core Allocator: The library implementing the *core allocator* is a wrapper around the `sched_setaffinity(2)` system call [3], which alters the affinity mask of a task (i.e., a thread). The affinity mask of a task determines the set of cores in a multicore processor on which it is eligible to run. By default, each task is eligible to run on every core and the scheduling infrastructure of the Linux kernel tends to balance the load on all the cores; hence, it is likely a parallel application spans all the cores. The core allocator provides a way to decrease or increase the number of cores a parallel application can harness.

The core allocator abstracts the `sched_setaffinity(2)` system call through the concept of *set of actions*. Different sets of actions, with different dimensions, may be adopted by different "Plan and Analyze" solutions. For example, a distinction can be made if the "Plan and Analyze" solution is required to make local (i.e., task-wide) decisions or global (i.e., system-wide) decisions. Another distinction can be made between a set of actions providing the ability to incrementally modify the number of cores assigned (stateful implementation) and a set of actions that fully specifies the number of cores to assign (stateless).

2) Frequency Scaler: The library implementing the *frequency scaler* is a wrapper around the `cpufrequtils` package [2], a set of user-space utilities designed to interface with the Linux kernel and manage dynamic voltage and frequency scaling (DVFS).

Considerations, similar to those made for the core allocator, can be made about the sets of actions for the frequency scaler. However, two reasons make the frequency scaler case simpler: first, local decisions are not supported since they would require to change the working frequency of a core (i.e., the one it is executing or it is going to execute a task of an instrumented application) possibly at each context switch operated by the scheduling infrastructure of the Linux kernel, and second, commodity multicore processors support different frequencies on different cores only when C-states differs.

Even though the frequency scaler is limited with respect to the core allocator, the abstraction of set of actions still exists. It is possible to construct sets with a portion of the available operating frequencies, it is possible to pinpoint the working frequency or moving with increment and decrement operations.

C. Plan and Analyze

This work makes use of AdaM and completing its skeleton with two reinforcement learning algorithms described in Section III: Adaptive Dynamic Programming and Q-learning.

The autonomic element, the union of the reinforcement learning powered AdaM and an instrumented application, is a rational agent and it is immersed in a single-agent and fully observable environment. The environment consists of the underlying hardware architecture, the system software, and the autonomic element itself. Full observability of such an environment is achieved through the self-monitoring capabilities of HRM. A similar minimalist description was proved to be effective by Tesauro et al. [20]. The environment is also uncertain because there is no "a priori" domain knowledge of how the self-adjusting capabilities of the core allocator and the frequency scaler will affect it; hence, the transition model is stochastic. In the most general case, the environment can also be considered sequential if future decisions are affected by the previous ones (e.g., the core allocator acts by incrementing or decrementing by n the number of cores assigned to the instrumented application). No information is known if the environment is stationary or not: one more reason for applying reinforcement learning algorithms since, in presence of a suitable exploration policy, it is possible to cope with slowly evolving non-stationary environments.

Therefore, the reinforcement learning algorithms implement utility-based learning agents moving in an environment treated as Markovian.[1]

V. Experimental Results

This section answers the following research question: can AdaM learn and exploit strategies to allocate resources (i.e., number of cores and/or frequency step) such that instrumented applications deliver user-defined performance goals? In the rest of this section we show how AdaM can learn strategies through different algorithms (i.e., ADP and Q-learning) and exploit diverse set of actions (i.e., core allocation only or core allocation plus frequency scaling).

A. Evaluation Platform

We evaluated AdaM on a workstation with an Intel Core i7-870 Processor and 4 GB of Single Ranked DIMMs. The

[1] In a (fist order) Markov process the current state depends only on the previous state: $P(X_t|X_{0:t-1}) = P(X_t|X_{t-1})$ [16].

processor features 4 cores running at a nominal clock frequency of 2.93 GHz and sharing 8 MB of last-level (L3) cache. The Enhanced Intel SpeedStep Technology allows the 4 cores to operate at 14 different clock frequencies: from 1.20 to 2.93 GHz with steps of 133 MHz. Each memory module runs at a clock frequency of 1066 MHz.

We disabled the Intel Turbo Boost Technology to prevent the 4 cores to operate at clock frequency higher than the nominal without AdaM being notified; in fact, this technology is completely transparent to both the kernel and user-mode applications and thus would make the environment partially observable. We disabled the Intel Hyper-Threading Technology (HTT) to prevent trashing cache lines in private inner-level (L1 and L2) caches that become shared whenever 2 threads run simultaneously on the same physical core.

We modified Debian 6.0 to run the Linux kernel 2.6.35 enhanced with HRM and provide user-mode access to frequency scaling through the *cpufrequtils* and the *userspace* governor.

B. Evaluation Methodology

We instrumented 8 out of 13 applications from the PARSEC 2.1 benchmark suite with *libhrm* [17]. The instrumentation provides users with the ability of setting performance goals and exports performance measurements system-wide so as AdaM can sample them and verify the effectiveness of its strategies. Different applications report performance measurements according to application-meaningful metrics; *blackscholes* employs options/s, *bodytrack*, *fluidanimate*, *raytrace*, and *x264* employ frames/s, *canneal* employs exchanges/s, *dedup* employs chunks/s, *ferret* employs queries/s, and *swaptions* employs simulations/s.

We executed AdaM alongside each instrumented application executing in isolation with 4 threads for 10 consecutive runs; by doing so, most of the experiments lasted 5 to 10 min making the workload more realistic. During the first two runs we let AdaM free to operate randomly varying the number of cores and the frequency step so as to explore the state space: we call this the *exploration* phase. From the third to the tenth run AdaM exploited the knowledge gathered up to that point by varying the number of cores and/or the frequency step so as instrumented applications attained user-defined performance goals: we call this the *optimization* phase.[2]

Whenever AdaM chooses the number of cores and/or the frequency step so as instrumented applications attain user-defined performance goals, we say the exploration phase was successful and that the management is effective.

Figs. 1 to 6 shows experimental results in pair or triple of plots; the first, second, and third (when present) report the performance measurement, the number of cores, and the frequency step (numbered from 1 to 14), respectively. Vertical dotted lines represent the end of one run and the beginning of the subsequent; the thick (second) vertical dotted line represents the end of the exploration phase and the beginning of the optimization phase. The area within horizontal dashed lines represents the user-defined performance goal.

[2]AdaM makes use of application-meaningful metrics computed over a sliding window of 1 s; the choice of the sliding window length has not specific semantic, it simply allows to catch all the relevant execution phases in the instrumented applications we considered.

C. Discussion

Figs. 1 and 2 show two alternative strategies to attain the user-defined performance goal for *blackscholes*. In the first experiment AdaM exploits ADP and core allocation and finds out it can schedule *blackscholes* on two cores while in the second experiment AdaM harnesses Q-learning, core allocation, and frequency scaling and discovers it can schedule *blackscholes* on four cores operating at the lowest clock frequency. Both the strategies are successful and the resulting management is effective even though performance measurements in the second experiment are lower than performance measurements in the first experiment. AdaM cannot disambiguate the two strategies since the user-defined goal is concerned with just performance and the resulting utility is the same. A user-defined power consumption goal[3] could help AdaM choosing the best possible strategies among those learned.

Figs. 3 and 4 show two alternative strategies to achieve the user-defined performance goal for *canneal*. Once again, we exhibit the impact of the set of actions; in the first experiment AdaM exploits core allocation and ends up scheduling *canneal* on three cores while in the second experiment AdaM exploits both core allocation and frequency scaling and finds out it can either schedule *canneal* on two or three cores and select between these options depending on the very first performance measurement of the run, which is biased by how fast the load balancer inside the Linux kernel scheduling infrastructure spreads threads among cores. The second experiment also shows how AdaM strengthens its knowledge even after the exploration phase is finished; the third run displays high oscillations since AdaM is switching between two strategies (i.e., four cores at the lowest clock frequency and three cores at a slightly higher clock frequency). However, both these strategies do not allow to achieve the user-defined performance goal and their utility lowers in favor of more stable solutions (i.e., either two or three cores at the highest clock frequency).

Conversely to the previous experiments discussed in this section, it is worth mentioning that AdaM learns the two strategies by harnessing ADP only; this means both the active reinforcement learning algorithms (i.e., ADP and Q-learning) work and strategies are mostly influenced by the set of actions.

Fig. 5 shows the strategy AdaM chooses to achieve the user-defined performance goal for *dedup* when exploiting Q-learning, core allocation, and frequency scaling. The learning and thus the management are not completely satisfactory since performance measurements oscillate and sometimes do not match the user-defined performance goal. However, it is still worth highlighting the self-healing capability of AdaM that recovers from its own errors (i.e., see runs 7 to 9) and comes up with a better strategy for the last run. Errors can always arise, especially with applications exposing diverse execution phases, because MDPs are intrinsically stochastic.

[3]Frequency (and voltage) scaling has direct impact on power consumption

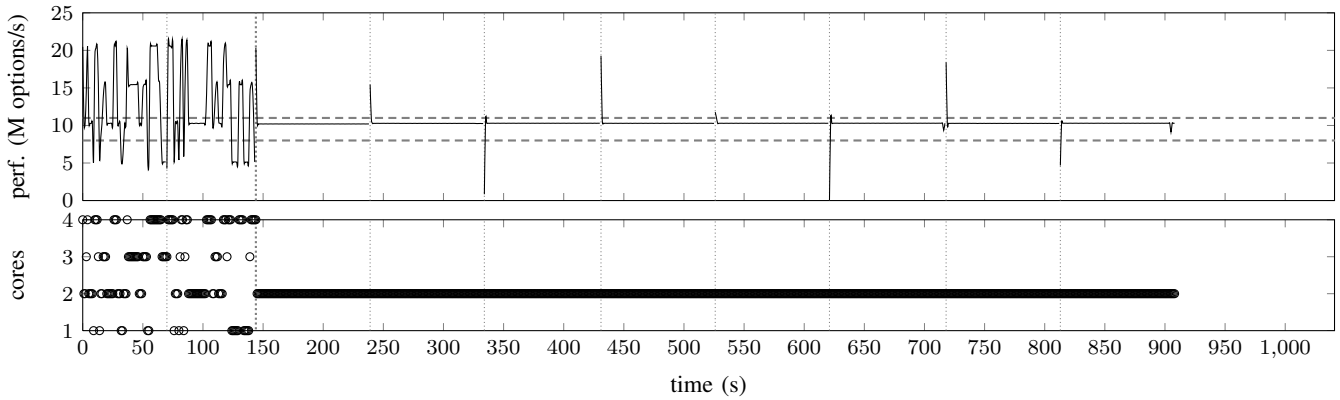

Fig. 1. *blackscholes* managed by AdaM exploiting ADP and core allocation.

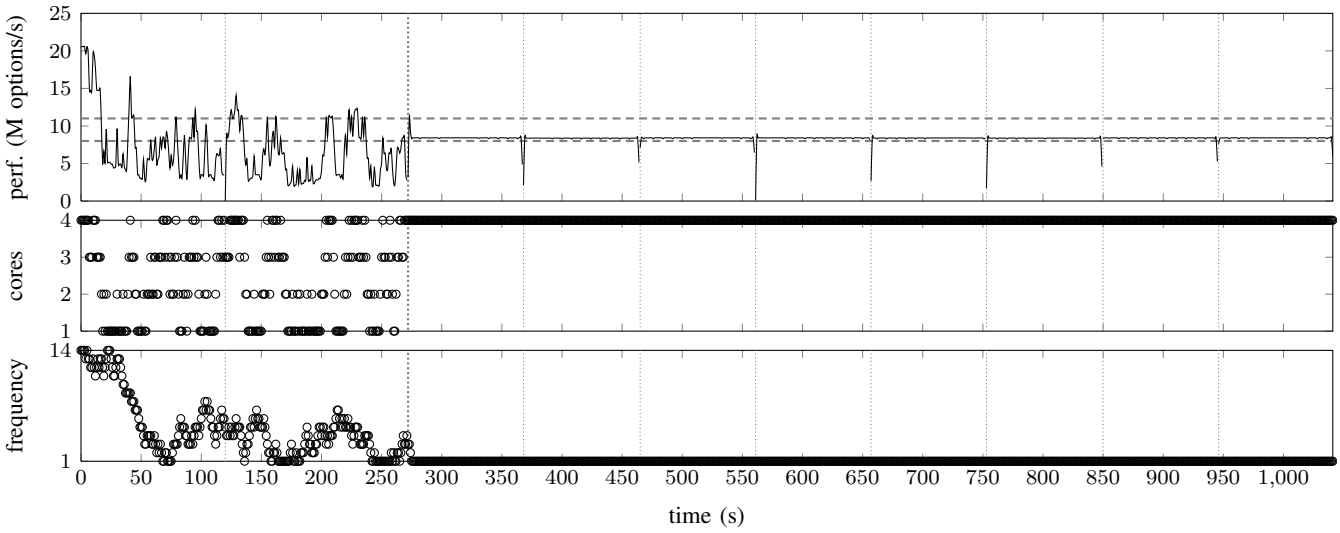

Fig. 2. *blackscholes* managed by AdaM exploiting Q-learning, core allocation, and frequency scaling.

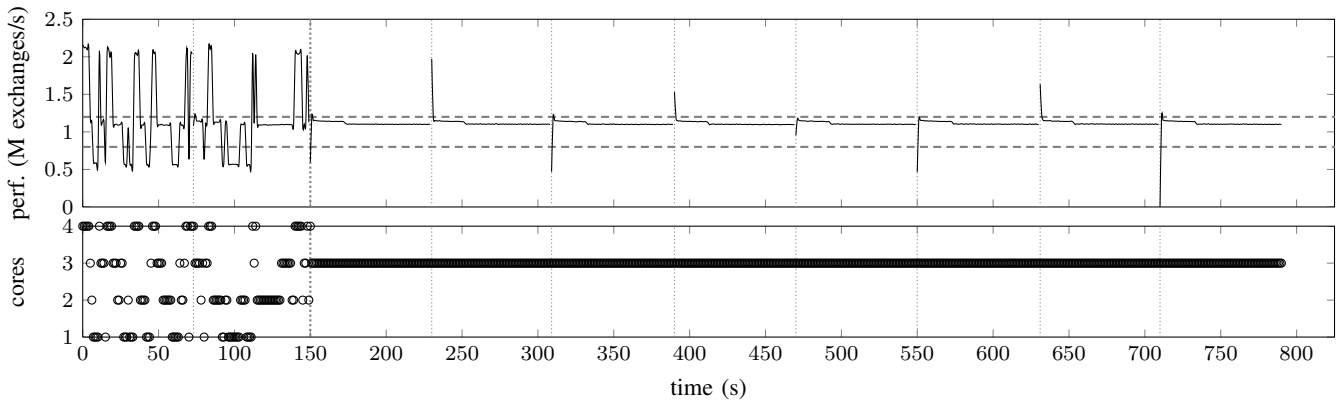

Fig. 3. *canneal* managed by AdaM exploiting ADP and core allocation.

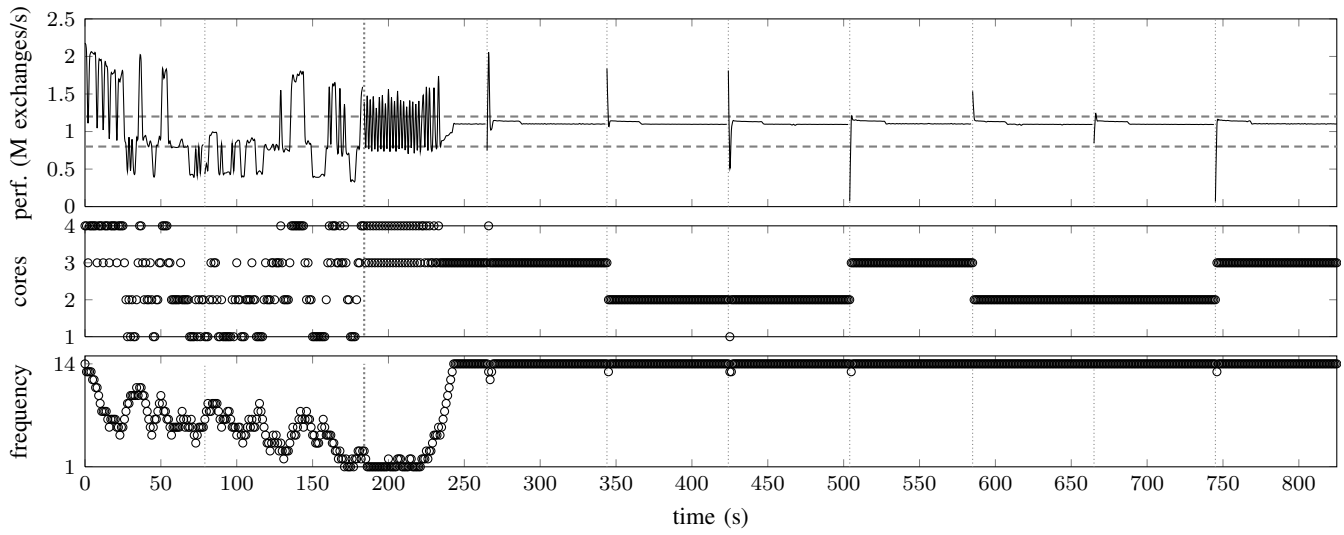

Fig. 4. *canneal* managed by AdaM exploiting ADP, core allocation, and frequency scaling.

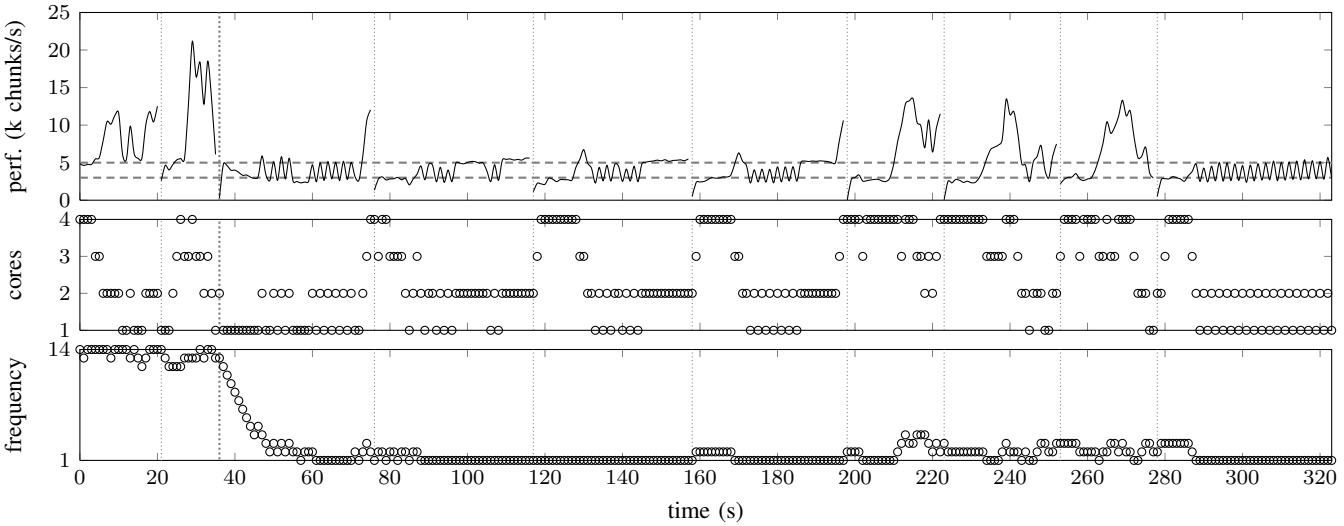

Fig. 5. *dedup* managed by AdaM exploiting Q-learning, core allocation, and frequency scaling.

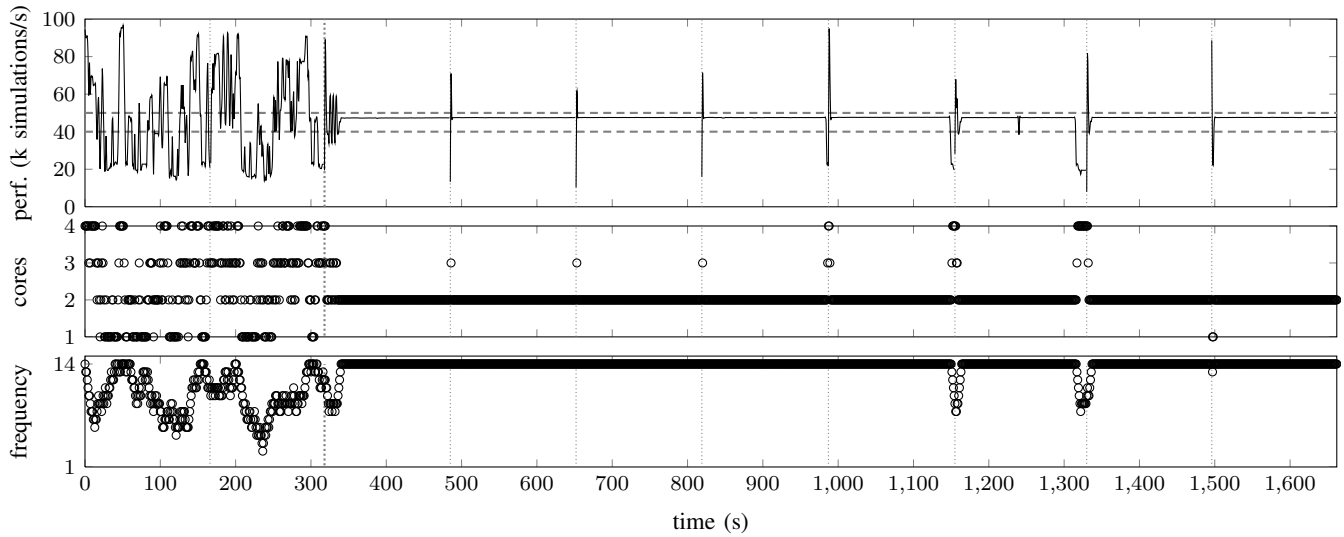

Fig. 6. *swaptions* managed by AdaM exploiting ADP, core allocation, and frequency scaling.

TABLE I
MEAN SQUARED ERROR NORMALIZED TO THE USER-DEFINED PERFORMANCE GOAL FOR EVERY COMBINATION OF REINFORCEMENT LEARNING ALGORITHMS AND DYNAMIC KNOBS

application	ADP		Q-learning	
	cores	cores & freq.	cores	cores & freq.
blackscholes	0.16	0.11	0.12	0.12
bodytrack	0.18	0.22	0.19	0.20
canneal	0.11	0.11	0.12	0.10
dedup	0.57	0.34	1.02	0.47
fluidanimate	0.16	0.15	0.15	0.16
raytrace	0.17	0.17	0.14	0.19
swaptions	0.10	0.10	0.11	0.11
x264	0.27	0.28	0.47	0.39

Fig. 6 shows the strategy AdaM chooses to drive *swaptions* towards the user-defined performance goal when using ADP, core allocation, and frequency scaling. The learning is successful and the management is effective since performance measurements match the user-defined performance goal; as a remark, the use of a reduced set of actions (i.e., core allocation only) alongside with ADP leads to the exact same strategy. Variations in the number of cores and frequency step during the optimization phase (i.e., see runs 7 to 9) are due non observable variables. *swaptions* does not require threads to synchronize, thus executing four of them on less then four cores may lead to asymmetries in their runtime. Since less then four threads may be active, performance measurements drop and AdaM responds varying the number of cores and the frequency step.

We performed an extensive evaluation with all the applications we instrumented exploiting either ADP or Q-learning, core allocation and/or frequency scaling. Table I reports the mean squared errors of the performance measurements normalized to the user-defined performance goals of the performance; in general, more freedom (i.e., using both the core allocator and the frequency scaler at the same time) coincides with lower values of the error. As a final remark, we can say that most of the applications except for *dedup* and *x264*, which expose diverse execution phases and require better exploration policies, behave properly with errors comparable or lower than those obtained by previous works [17].

VI. CONCLUSIONS

This paper describes the design of AdaM, a skeleton to create a self-aware, self-optimizing, and self-healing autonomic element embodying a managed element (i.e., an instrumented application) and an autonomic manager.

The reference implementation of AdaM exploits HRM [17] to provide self-awareness, thus collecting performance measurements, and requires application to be instrumented with *libhrm* so as to signal progress and specify user-defined performance goals. Our implementation of AdaM makes use of two dynamic knobs to affect the execution of applications; the first is core allocation, which changes the number of cores assigned to an applications, while the second is frequency scaling, which varies the clock frequency at which cores operate. We demonstrate the use of two active reinforcement learning algorithms, namely ADP and Q-learning, to discover self-optimizing strategies to drive applications towards user-defined performance goals given different set of actions and eventually recover from errors through self-healing.

An extensive evaluation of our working prototype of AdaM show very interesting experimental results in exploiting artificial intelligence in the field of autonomic computing.

Further developments of this work will involve the adoption of more advanced exploration strategies and will go deeper into a distributed approach, borrowing from the multi-agent branch of artificial intelligence.

REFERENCES

[1] "Application Heartbeats," http://code.google.com/p/heartbeats/.

[2] "`cpufrequils`," http://kernel.org/pub/linux/utils/kernel/cpufreq/.

[3] "`sched_affinity(2)` Linux Programmer's Manual," http://kernel.org/doc/man-pages/online/pages/man2/sched_setaffinity.2.html.

[4] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[5] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Syst. J.*, vol. 42, no. 1, 2003.

[6] J. Heo and T. Abdelzaher, "AdaptGuard: Guarding Adaptive Systems from Instability," in *ICAC*, 2009, pp. 77–86.

[7] H. Hoffmann *et al.*, "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *ICAC*, 2010.

[8] ——, "SEEC: A General and Extensible Framework for Self-Aware Computing," Massachusetts Institute of Technology, Tech. Rep., 2011.

[9] ——, "Self-aware Computing in the Angstrom Processor," in *Proc. of DAC*, 2012.

[10] L. P. Kaelbling *et al.*, "Reinforcement Learning: A Survey," *J. Artif. Int. Res.*, vol. 4, no. 1, 1996.

[11] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, 2003.

[12] J. O. Kephart and R. Das, "Achieving Self-Management via Utility Functions," *IEEE Internet Computing*, vol. 11, no. 1, 2007.

[13] J. Kephart and W. Walsh, "An Artificial Intelligence Perspective on Autonomic Computing Policies," in *POLICY*, 2004.

[14] M. Maggio *et al.*, "Controlling software applications via resource allocation within the Heartbeats framework," in *CDC*, 2010.

[15] ——, "Comparison of Decision-Making Strategies for Self-Optimization in Autonomic Computing Systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 7, no. 4, 2012.

[16] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2009.

[17] F. Sironi *et al.*, "Metronome: Operating System Level Performance Management via Self-Adaptive Computing," in *DAC*, 2012.

[18] R. Sterritt and D. Bustard, "Towards an Autonomic Computing Environment," in *DEXA*, 2003.

[19] Y. Tan *et al.*, "Adaptive Power Management Using Reinforcement Learning," in *ICCAD*, 2009.

[20] G. Tesauro *et al.*, "Utility-Function-Driven Resource Allocation in Autonomic Systems," in *ICAC*, 2005.

[21] G. Tesauro, "Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies," *IEEE Internet Computing*, vol. 11, no. 1, 2007.

[22] Y. Wang *et al.*, "Deriving a Near-optimal Power Management Policy Using Model-Free Reinforcement Learning and Bayesian Classification," in *DAC*, 2011.