The Consensus Object: Coordinating the Behavior of Independent Adaptive Systems

Marco Triverio¹, Martina Maggio¹,², Henry Hoffmann², and Marco D. Santambrogio¹,²

¹Dipartimento di Elettronica e Informazione
Politecnico di Milano, Italy
marco.triverio@mail.polimi.it {maggio,santambr}@elet.polimi.it

²Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, Cambridge
{mmaggio,hank,santambr}@mit.edu

ABSTRACT

Nowadays the complexity of computing systems is skyrocketing. Programmers have to deal with extremely powerful computing systems that take time and considerable skills to be instructed to perform at their best. This work analyzes the stated problem and proposes a simple, yet powerful mechanism for optimizing performance through the coordination of the interaction of multiple, independent adaptive systems called services. In this scenario we developed the Consensus Object, a system-centralized decision engine based on reinforcement learning. The Consensus Object gathers information about the performance goals of the system and it can either turn services on or off. The Consensus Object analyzes the runtime impact of services and of their autonomous decision policies, looking for a combination of services that makes it possible to reach the given goals. The experiments that have been carried out show the ability of the Consensus Object to adapt to changing conditions, confirming the validity and the flexibility of the followed approach.

1. INTRODUCTION

Nowadays computing systems are evolving at an unprecedented rate of change and are becoming increasingly complex. Not only have the performances and the amount of available hardware resources greatly increased in the past decades, but new architectures have appeared, offering multiple processors and multiple cores. On one hand, the advantages of highly-parallel systems could benefit an enormous variety of fields. On the other hand, the growing complexity is making it unfeasible for the average programmer to weight all the constraints and optimize the system for a wide range of machines and scenarios. Even though technologies have improved, making a system perform at its best is a non-trivial task. The burden on programmers is noticeable and many research efforts have worked to address this issue. It is clear that it is not feasible to rely on human intervention to tune a system: conditions change constantly, rapidly, and unpredictably. It would be desirable to have the system automatically adapt to the mutating environment.

There is a wide consensus on the fact that new paradigms have to be explored and new frameworks have to be developed. Among those, autonomic computing systems¹ seem to be the answer to most of the problems previously described [1]. Self-Aware Adaptive computing systems adapt their behavior and resources to automatically find the best way to accomplish a given goal despite changing environmental conditions and demands. Therefore, this kind of system needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions. The life-cycle of self-adaptive softwares does not stop after its development and initial setup but it indeed continues to handle changes on several possible levels.

This work presents a solution able to adjust itself during execution due to a simple, yet powerful mechanism for coordinating the interaction of multiple, existing adaptive systems, each of them with its own decision engine. The architecture used in this paper consider current computing systems augmented through the usage of a modular ecosystem of software components called services, capable of actuating a change on the system. Services can perform changes on the system due to the decision making entities embedded

¹In this paper we will interchangeably use the terms autonomic and self-adaptive.
in the services themselves, see Section 3. In this scenario we developed a system-centralized decision engine, called Consensus Object, based on reinforcement learning. The Consensus Object gathers all the information about system performance and it is able to decide which measures to enact, orchestrating the different elements that commit changes to the environment. The Consensus Object co-exists with other independent and adaptive decision making entities, which reside in services. The Consensus Object turns services either on or off as necessary to meet the performance goals of the system. In this scenario the use of machine learning techniques enables the Consensus Object to handle services in a modular and flexible fashion, to evaluate at runtime their effectiveness, and to quickly adapt its policy when unknown and unpredictable situations are encountered.

Within this scenario, we have:

- developed a decision engine based on machine learning to guide the action of the Self-Aware Adaptive computing framework;
- defined a centralized mechanism for coordinating the interaction of multiple, existing adaptive systems, each of them with its own decision engine;
- built on existing technologies, such as the monitoring infrastructure called Application Heartbeats, with the aim of providing a complete Self-Aware Adaptive computing framework;
- validated the framework with a set of thorough tests, which have proved the validity of the proposed approach.

The remainder of this paper is organized as follows. Section 2.1 identifies the key characteristics of Self-Aware Adaptive computing systems, while Section 3 and Section 4 introduce and describe in detail the proposed methodology and the machine learning-based decision engine that we call Consensus Object. In Section 5 our experimental results are presented. Finally, our conclusions are summarized in Section 6.

2. SELF-AWARE ADAPTIVE SYSTEMS

Self-Aware and Self-Adaptive systems augment software engineering with a novel notion of change. Differently from usual software, they are in fact proactive and suggestive towards changes, introducing the concept of runtime adaptiveness. Software is a key element of modern computing systems: it drives the hardware and enables exploitation of its full power. The complexity of software is growing exponentially [2]: programs are often developed by several programmers, built of several inter-communicating components, and run in unknown and mostly unpredictable contexts. Even with modern testing techniques, this scenario often leads to software being deployed with relevant issues. Bugs and security issues have to (or should be) patched in a fast cost-efficient manner through update mechanisms built in the shipped software. But this is only part of the whole problem: in fact, not only software has to be polished and perfected after installation but it has to be designed in order to accommodate needs and fulfill requirements that become evident only time after deployment.

2.1 Context Description

In order to understand this aspect it is useful to review the evolution of software design. Software engineering was born in the late 1960’s when the study of a series of techniques to approach software design, development, and maintenance arose [3]. The goal was desirable: being able to improve the quality of products building them in a more systematic and disciplined way. For the first time a solid alternative to the code-and-fix” approach was proposed: creating software was not a synonym for programming anymore. In particular software started to be developed through the following phases:

- definition of stable requirements through the analysis of the context in which the product will be used, the users that will make use of it, and the needs to be fulfilled; software design, usually driven by tools such as the Unified Modeling Language (UML) and it is used to architect the software before it is implemented: the purpose is to make the software maintainable, modular, expandable, and reusable; software implementation, which is the final step in engineering a software and it consists of programming and testing. It was indeed argued that a conscientious definition of requirements could avoid radical changes in response to the demand for possibly simple new features: this clearly was a big step towards better quality.

Yet, after just a few years, this approach became not satisfactory because it was based on the arguable assumption that such thing as stable requirements existed [4]. When software products became something more than static, monolithic, and centralized tools internal to organizations the so-called closed world assumption [4] did not hold anymore. Evolution is indeed intrinsic to software [5] and for this reason software engineers must design programs that can easily accommodate future changes. As outlined by [6] programs must be designed for change.

If in the previous years one single change could have implied recompilation and redeployment of the whole system, changes gradually became easier to introduce, also thanks to object-oriented programming languages. Software became modular, dynamic and distributed (also across networks).

Yet the growth in the dynamism of change has not shown signs of stopping any time soon. Even though most softwares developed in the past few years are easily updatable and have little downtime, new application and new domains demand more flexibility and performance. Software still requires a lot of (costly) manual intervention, it sometimes still fails, and it has almost no knowledge about itself: the problem lays in the open-loop structure [1] followed in the engineering of the software. There is a wide consensus on the fact that new paradigms have to be explored and new frameworks have to be developed. Among those, autonomic computing systems seem to be the answer to most of the problems previously described [1]. They introduce a new framework based on the closed-loop approach, which, through a feedback loop, adjusts the software while it is running. This kind of system needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions. The life-cycle of self-adaptive softwares does not stop after its development and initial setup but it indeed continues to handle changes on several possible levels.

2.2 Basic Definitions

Several definition of Self-Aware Adaptive computing software have been provided through the years. Two of them are:

- “Self-adaptive software evaluates its own behavior and
changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible”[7]

- “Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation”[8]

But autonomic computing was thoroughly formalized only in 2001 by IBM with its “Autonomic Computing Manifesto”[2], first presented by Dr. Paul Horn’s during the National Academy of Engineering meeting of the same year. Horn outlined eight key properties of autonomic computing systems:

1. **Self-Awareness or Introspection** — the system must know itself in detail: components, layers, statuses, connections, extents,... are necessary parts to be monitored and kept under inspection;

2. **Self-Reconfigurability** — the system must be able to dynamically change its behavior at runtime;

3. **Continuous Self-Optimization** — the system must never settle but it must continuously monitor itself to discover possible optimizations;

4. **Self-Healing Capabilities** — in case of failures (caused by routine or extraordinary events) the system must be able to recover;

5. **Self-Protection Techniques** — the system must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity;

6. **Context-Awareness** — the system must know the environment (defined as: everything in the operating environment that affects the system’s properties [1]) and the context that surround its activity and take it into consideration when acting;

7. **Openness** — the system must be able to interact with heterogeneous components that might not be under its control;

8. **Anticipation and Support** — the system must be able to provide and anticipate the correct status to handle future changes.

There is an enormous amount of possibilities created by Self-Aware Adaptive computing systems. 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. For instance, in a mobile phone scenario, a Self-Aware Adaptive computing system is given the goal of maintaining a connection to a receiver with a desired bit rate, using the least amount of energy. The underlying architecture has cognitive hardware mechanisms to both observe and to affect the execution. A decision policy (either static or dynamic) makes use of the monitored data to take an informed decision and determine the appropriate actions to be taken in any given moment.

Self-Aware Adaptive computing systems are usually designed and implemented through the Observe-Decide-Act (ODA) loop, which means that autonomic systems execute, monitor themselves and the surrounding environment, and adapt to the sensed conditions. Observation is carried out by components called sensors, decisions are taken by a decision engine, and actions are performed by components called actuators. As described in Section 3, the ODA loop is the approach that we have followed.

### 3. THE PROPOSED METHODOLOGY

The scenario that we are addressing is made of:

- **Applications** – Pieces of software written to accomplish a specific task.
- **Processes** – Instances of an application.
- **Monitored Processes** – Process that are making one or more entities of the system aware of their performance goals and actual progresses. This is possible, for instance, adopting the Application Heartbeats framework. Monitored Processes are what has so far been called targets in that their performance is altered by actuators.
- **Services** – It is an extension of what has so far been called actuator. It represent a component capable of performing changes on one or more applications or on the whole system. Services are enabled or disabled by the decision engine; when enabled, they are autonomous components that can decide and act on the system, for example reading the performance signals of the applications and computing some reaction to changes in the external environment.

- **Consensus object** – It knows about monitored processes and services, gathers data about applications performance levels and acts enabling or disabling services. Its action policy is based on machine-learning technique.

---

**Figure 1:** Consensus Object role: (1) dynamically update the list of available actuators and possible targets (via the information retrieval channels); (2) decide which actuator to enable on which target, sending commands to the services. For the sake of clarity it is not shown that services might also accesses performance data.

In this context applications set their performance goals and the Consensus Object orchestrates the available service boosting (or reducing) performance in order to make it possible to achieve the given goals. This scenario features the Consensus Object as a central element that has a global...
vision of the system. The general realization of this system is shown in Figure 1. We believe this approach has the power to achieve, if possible, a global optimum. While the Consensus Object represents a single point of failure, this condition has been mitigated by our implementation, which allows each component of the framework to run even in case of failure of one or more other components. We designed an architecture made of a decision engine, a set of services, the interface to the services, and a mechanism to discover new actuators, which can be mapped to an ODA loop, as described in the following Sections.

3.1 Observe

Observation is what makes a system self-aware. Through it, a system understands its state, its current progress, and its possible future actions. This introduces at least three levels of awareness:

1. awareness through the data gathered by sensors
2. awareness of the possible targets of actions, in this case monitored applications
3. awareness of the availability of services

The focus is set on applications performance, therefore Level 1 can be achieved through a monitoring infrastructure that gathers the relevant data. The Application Heartbeats [9, 10], are a portable and general method of monitoring an application progresses towards its performance goals and seem particularly appropriate for this work. In fact, the Application Heartbeats framework implements a simple yet extremely powerful monitoring infrastructure. Any application making use of this Application Programming Interface (API) has then a standardized method to assert its performance goals registering to the Heartbeats and specifying a certain number of parameters, e.g., minimum and maximum heart rate, size of the window of observation, size of the heartbeats history buffer. Moreover, the application updates at runtime its progresses 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. The framework is capable of monitoring the progress of the execution. The calculated information is made available to either external observers or the application itself, creating a powerful mechanism for designing control loops. A typical example of use is a video encoder, which might want to be able to deliver, for instance, 30 frames per second. It then communicates this information to the Heartbeat API and it issues an heartbeat for every produced frame. An external observer can consequently improve (or reduce) performance through the modification of some system-level or application-level parameters, such as the number of cores assigned to the video encoder application or the encoding algorithm. If needed, the application itself could impersonate the external observer, but this is not the case here presented. A monitor that resides outside of the application has access to global information unavailable inside of the application and can then make informed decisions.

Level 2 can also be achieved with the Application Heartbeats, which keeps track of all the processes currently registering their performance goals.

Level 3 allows the system to find the available services and understand their potential in terms of system-wide or application-specific effect. This duty is carried out by the Services API designed for the Consensus Object, which keeps track of registered services on a constantly updated list.

3.2 Decide

Artificial Intelligence might be applied to make Self-Aware Adaptive computing softwares flexible and able to learn from experience, to recognize patterns, and to boost the decision efficacy. The Consensus Objectis our decision engine. It stands at the center of the ODA loop and exploits the awareness given by the observation mechanisms to elaborate a plan for future behavior. The aim is to tune performance in order to make each monitored process achieve its performance goals. In particular, the Consensus Object:

- exploits the observation phase in order to determine the available services and the monitored applications;
- constantly verifies the availability of new or old services and the presence of new or old monitored processes;
- gathers the information coming from the monitored processes;
- analyzes the performance-related data in order to understand whether to enact a correction policy;
- decides which services to enable or disable;
- performs the decided actions.

The decision policy that drives the Consensus Objectis based on machine learning techniques, which were proved powerful tools for managing the increasing complexity of computing systems. For example, machine learning has been used to build a self-optimizing memory controller [11], to coordinate management of interacting chip resources [12], and to handle synchronization in a spin-lock library [13].

We implemented R-learning, a reinforcement learning algorithm [14]. In general, Reinforcement Learning augment a system with the possibility to learn from experience through the use of a reward signal that drives the learning process. In particular the algorithm calculates a signal that is a synthesis of the current state (and of its performance characteristics, such as whether the applications are in the desired performance range); it then selects actions attempting to maximize the given reward. More details about the R-learning algorithm and our implementation are provided in Section 4.

3.3 Act

Once a strategy has been decided by the Consensus Object, it must be enacted. In our frameworks the actuators are called services. There are many kinds of services that might be available and that might affect the system in different ways: some might affect performance, accuracy, or both; some might impact on the whole system while others might target one or more applications; some might access performance data of the applications. For instance: Unix niceness enhances performance leaving accuracy intact while techniques such as loop perforation enhance performance altering accuracy; frequency scaling affects performance of the whole system while loop perforation can target a single application, and so on.

In an autonomic system we can define two main categories of actuators:
• **Blind Actuators** – This kind of actuators enact their actions without knowing the actual effect on the system.

• **Conscious Actuators** – This kind of actuators have a feedback mechanism to be aware of the results of their action.

In our framework **blind** services are those that do not access the information about the monitored processes’ performances, while **conscious** services make their decisions taking into account about the heartbeats generated by the monitored processes. Conscious services usually implement a feedback loop that allows them to create an independent ODA-based system.

A list of possible services and a first classification is here presented:

- **Core allocation** – Under modern operating systems it is possible to force a given process to be run only on certain processors or cores, for example by calling the `taskset` Unix command that sets the affinity mask for a process. Ideally, when each process is coupled to some determined cores and the number of cores is sufficiently high to serve all the application requests, there would be no need for time multiplexing on one or some of the cores. This optimization is enabled on a per-application basis [15, 16]. For this service and in the following, it is worth stressing that this means that a service per application could be activated. In this case each of the services would implement the feedback loop presented in the cited references, taking as a feedback measure the heart rate of the single application the service is active for and producing as an output (and applying) the affinity mask the controller chooses to be better to match the performance goals of the application.

- **Memory allocation** – The core allocation service is particularly useful when used on CPU-bound or CPU-intensive processes. Different processes may require other resources, e.g., memory. Such programs are said memory-bound or memory-intensive: in order to be optimized they require memory to be allocated easily and efficiently. This optimization can be enabled on a per-application basis, as shown in [15].

- **Nice ness (priority) adjusting** – Modifying the nice number of a process, being it related to the process priority, can enhance or reduce performance of a given application. This optimization is enabled on a per-application basis.

- **Locks** – Changing the policies on the management of locks might also affect performance, as shown in [13].

- **Frequency scaling** – It is a widely popular service that reduces processor clock frequency slowing down the computation on the entire system or increase the clock speed speeding up the computation. When using such optimization it is of course necessary to take into account its global effect. Details are given in [15], however prior work has considered just actions on a per application basis, while future works require to compute this action on a system basis.

- **Dynamic perforation** – Some optimizations increase performance to the detriment of accuracy of calculation. For instance, this is the case with dynamic loop perforation [17, 18]. This optimization is enabled on a per-application basis.

Within this work we will focus on two of the above mentioned services, namely the core allocator and the priority adjuster. As shown in Figure 2, the communication between services and the Consensus Object occurs through a standard interface called Services API. All services have peculiar characteristics that make them unique. Because new services might be implemented at any time we would like to stress the fact that we designed the whole framework to be as modular and flexible as possible. Services do not advertise through the Services API any of their characteristics, apart from their potential effect on the system (local or global). This way the Consensus Object does not know since the beginning of execution the impact of each service, but it has to learn it at runtime.

This has some important advantages. First, the interface between the Consensus Object and the services is extremely simple. Second, there is no need to model a service – something that might prove truly difficult given the heterogeneity of the computing systems on which the framework could run. Moreover, the Consensus Object do not need to be updated or restarted when new services are plugged in, to the extent of self-configuration. At any time during execution, all the services are treated equally by the Consensus Object. The only aim of the Consensus Object is to reach the performance goals, regardless of which services are employed. This allows, as shown in the experimental section, the Consensus Object to disable a service when it is not useful anymore. All services embed a decision making mechanism that is independent from the Consensus Object: the Consensus Object, given the knowledge acquired through the machine learning algorithm, enables or disables services. When a conscious service is enabled its own loop-based decision mechanisms is activated, in order to make the monitored process reach their performance goals. The Consensus Object learns to turn on and off these decision mechanisms either on a per-application basis or globally, as the principal conductor of an orchestra directs a musical performance.

4. **IMPLEMENTATION DETAILS**

The Consensus Object has the responsibility of deciding which services to activate or disable on a given target or on a system basis. The Consensus Object does not know which is the potential of each service (i.e., how much a given service can improve or decrease the application performance)
and, more in general, it does not have a thorough description of the service characteristics. This means that, at least at the beginning of the execution, all services are considered equal. The algorithm has then to evince from past behavior which services are most appropriate in a certain situation. The Consensus Object is therefore able to handle unknown and unpredictable situations, giving its great flexibility. The Consensus Object can either enable a service on a given target (or globally) or disable it. From the runtime performance of the service it then infers when enabling or disabling the service leads to the desired performance levels. This is accomplished through the R-learning algorithm.

A Markov Decision Process (MDP), is the fundamental formalism to understand reinforcement learning and is a tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where:

- $S$ is the set of states. It is important that the state includes both information about current sensors measurements and past “memories”. While it is of course not required that the state represents everything about the environment, information about past observations might be retained to improve the quality of decisions. Quoting [14], “we don’t fault an agent for not knowing something that matters, but only for having known something and then forgotten it”;
- $A$ is the set of actions;
- $\{P_{sa}\}$ define the probability of shifting from a state $s$ to a state $s'$ when a the action $a$ is taken. Basically $\{P_{sa}\}$ gives the transition probability distribution if we take action $a$ in state $s$;
- $\gamma \in [0,1)$ is the discount factor;
- $R$ is the reinforcement.

The dynamics of a MDP proceed from an initial state $s_0 \in S$, wherever some action $a_0 \in A$ is chosen therefore transitioning to a new state $s_1$, according to the distribution probability defined by $\{P_{sa}\}$. From $s_1$, a new action $a_1$ is chosen, and taken, iterating the process. This can be described by the sequence:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$$ (1)

Whenever a new state is reached, a reward might be given to the learning agent. The reward is accumulated over all the states transition and can be expressed as:

$$R = R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \ldots$$ (2)

Since we aim at optimizing the obtained reward we might say that we want to maximize the average of $R$ over time, $E[R]$. Some important definitions are:

- A policy $\pi : S \rightarrow A$ that maps states into actions.

  - The state-value function for policy $\pi$ is known as $V_\pi(s)$ and is the expected sum of rewards (discounted by the factor $\gamma$) received from starting from state $s$ and then following the policy $\pi$.

  - The action-value function for policy $\pi$ is indicated with $Q_\pi(s,a)$ and is the expected value of the sum of rewards (discounted by $\gamma$) starting from a state $s$, taking action $a$, and then following policy $\pi$.

  - Given a policy $\pi$, its state-value function satisfies the Bellman equation, which states that $V_\pi$ is made of two components: the immediate reward and the expected sum of future discounted rewards

    $$V_\pi(s) = R(s) + \gamma \sum_{s' \in S} P_{sa}(s') V_\pi(s')$$ (3)

  - The optimal state-value function is defined as:

    $$V^*(s) = \max_{\pi} V_\pi(s)$$ (4)

    which is the best possible expected sum of discounted rewards.

  - The optimal policy is defined as:

    $$\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s').$$ (5)

Given the problem described in the previous sections some considerations are in order. First, the states $S$ are given by the current performance states of applications (i.e., above performance, below performance, in range, and so on) and by the current states of services (i.e., service $S_1$ is globally enabled, service $S_2$ is enabled only on a certain application, and so on). Second, the actions $A$ are all the possible activation/deactivation of services either globally or on a specific application. The set of available actions depends on the current state since, for instance, if a service is already enabled on a certain target, it might only be disabled. Moreover it might be necessary to consider a null action which represents the decision of not taking any action. Third, the transition probabilities $\{P_{sa}\}$ are not known and they have to be estimated at runtime. Moreover, balance between exploration (choosing actions almost randomly in order to discover potential benefits) and exploitation (choosing actions that in the past have brought high rewards) has to be reached. Two methods are typically used: $\epsilon$-greedy and Softmax. We opted for the latter, which induces action probabilities proportional to the value of the action-value. Notice also that the reward might be given by a function of the performance of applications.

In this context the R-learning [19] algorithm fits particularly well since it works without needing a model of the environment and it does not divide experience into episodes, allowing for continuous learning (even when terminal states are reached). This also means that there is no need to discount past rewards; in our implementation $\gamma$ is therefore set to 1. $\alpha$ and $\beta$ are two step-size parameters ($0 < \alpha, \beta < 1$) that represent the learning rate on $Q$ and on $\rho$, where $\rho$ is the average reward.

The pseudo code of the R-learning algorithm follows:

- Initialize $\rho$ to zero
- Initialize $\alpha$ and $\beta$
- for all $s, a$ do
- Initialize $Q(s, a)$ to zero
- end for

- loop
- s ← current state
- choose action $a$ in $s$ using behavior policy (Softmax) take action $a$, observe $r$ and $s'$
- $Q(s, a) ← Q(s, a) + \alpha[\gamma r + \max_{a'} Q(s', a') - Q(s, a)]$
- if $Q(s, a) = \max_{a} Q(s, a)$ then
- $\rho ← \rho + \beta[r - \rho + \max_{a'} Q(s', a') - \max_{a} Q(s, a)]$
- end if
end if
end loop

On the implementation side, exchange of information among the different components of the framework is achieved through several inter-process communication mechanisms. The APIs for the Application Heartbeats, the services, and the Consensus Object mask the use of these mechanisms. Figure 3 summarizes the communication channels, described below.

Figure 3: Communication channels of the framework

For each monitored process the Application Heartbeats opens a shared memory segment where to save performance statistics and writes to a file some information necessary to access them. The Consensus Object constantly scans the folder where such files are stored and, when a new application is started, it begins to monitor its performance. Moreover, it collects (through the update_list_of_services() function) services registrations and cancellations, which are written in a predetermined named pipe through the register_service() function.

Communication with services is achieved with the use of a shared memory segment, details are shared during registration. The Consensus Object may call the enable_service() and disable_service() functions, which take the service and the targeted process as arguments. Services retrieve commands through the read_command() function.

5. PRELIMINARY RESULTS

This Section presents preliminary results to support the validity of the proposed approach, both in terms of overhead and correctness.

5.1 Application Heartbeats and Services API

Self-Aware Adaptive computing systems have a tremendous potential. It would be desirable not to vanish it because of overhead: the infrastructure needs to be lightweight. Previous research has shown that the Application Heartbeats have a very limited overhead; in particular [20] has confirmed an average overhead of circa 4% on an application encrypting and decrypting through the Data Encryption Standard (DES) algorithm and producing an heartbeat every fraction of work.

We have then analyzed the Consensus Object and the Services API in order to quantify the overhead of the framework. In particular, we run x264 without the Consensus Object and with the Consensus Object and the core allocator on an x86 Intel Core 2 Duo\(^2\) and compared the results. The same experiment has been repeated ten times and then averaged. The Consensus Object-enabled version has an overhead of circa 5%, an encouraging result.

5.2 Machine learning-based Consensus Object

As shown in Section 4, the decision engine of the Consensus Object is based on machine learning and, in particular, on the reinforcement learning algorithm called R-learning. The implemented algorithm makes use of Softmax for the choice of actions; such method is not deterministic, making the whole system behavior not a priori definable. We have therefore repeated our tests several times, to validate the reliability of the decision engine. Notice that the R-learning algorithm explores the state-action space by evaluating which actions bring to a higher reward from the actual state. The Consensus Object is able to gradually gather experience and learn from past exploration. It is also able to react to changing conditions, finding the best actions for new contexts. The following experiments aim at showing the validity and the flexibility of the proposed approach.

We present seven different tests, carried out on an x86-64 machine featuring an Intel Core i7-870 processor\(^3\). The experimental evaluation was able to enable/disable a multi-application version of the core-allocator presented in [16] and a newly developed service named priority adjuster. Note that the proposed methodology is general and may be applied with any other service that implements the Services API. The monitored processes were instances of x264, possibly with different parameters, a different number of threads, and different requirements.

5.2.1 Free execution

This first test has been carried out to explore the heartbeats’ generation by x264 and see whether it has major variations during an uncontrolled execution (neither the Consensus Object nor any of the services are running). As shown in Figure 4, the heart rate is bound between 80 and 90 heartbeats per second. On average, with the given parameters and the given number of cores, it signals 85.49 heartbeats per second.

5.2.2 A single service management

\(^2\)Clock speed: 2.4GHz, 2MB L2 cache, and 4 GB RAM, OS: Ubuntu 9.10.

\(^3\)Clock speed: 2.93 GHz, 4 GB of SDRAM DDR3-1333, NVIDIA GeForce GT 240 graphic card, OS: Ubuntu 9.10.
This second case study shows one instance of x264 running with the core allocator controlled by the Consensus Object. The desired heart rate is between a minimum of 20 and a maximum of 30 heartbeats per second. As shown in figure 5, initially the heart rate increases, while the Consensus Object is still exploring the state-action space and discovering the effects of its possible actions (enabling or disabling the core allocator for the given application). Around 6 seconds from the beginning, the target performance level is met, meaning that the Consensus Object was able, before that time, to find the actions that maximize the reinforcement, bringing the heart rate in the desired range, i.e., enabling the core allocator and not disabling it even if the goal is reached. This test shows the ability of the machine learning-based decision engine to learn to choose the correct action in a static context.

Notice that in this experiment, as well as in the following ones, the decision engine is learning the best action to take (enabling and disabling services based on the performance signals of the applications), while these services are driving the performance signals. In the case just described, the Consensus Object learns to enable the core allocator which is able to drive the heart rate signal.

5.2.3 Multiple services

A single instance of x264 is used in this scenario as well. However, the Consensus Object could choose between two different services: the priority adjuster and the core allocator, which have been both started and could be both activated or disabled. The desired performance level is different from the previous case and spans from a minimum of 30 and a maximum of 50 heartbeats per second, to prove the framework flexibility. In the first seconds of execution the heart rate decreases, the Consensus Object is still learning how the two services influences the heart rate. After a few seconds the Consensus Object finds the action that maximizes the reinforcement. In particular, the priority adjuster service is disabled and the core allocator service is enabled. This test shows the ability of the machine learning-based decision engine to choose the correct action among multiple choices. Notice that the number of possible choices grows exponentially with the number of available services and in this case the Consensus Object chose among four different configurations the best one. It is also worth stating that if a malicious service was introduced in the system, or a non-working one, the Consensus Object is proven able to learn and correct its activation on the fly.

5.2.4 Changing scenario

This experiment tests the reaction of the system in a multi-application and multi-service domain, more in detail a second application is started around 30 seconds from the beginning of the experiment. Both the applications are instances of x264, with the same performance goals, the heart rate being in the range 30 to 50 heartbeats per second. The two services available for the Consensus Object are the core allocator for the first instance of x264 and the same service for the second instance of the application. As shown in Figure 7, the decision engine is able to handle one application and when the second instance is executed it learns how to keep both applications in their performance range. The heart rate oscillates a little bit, probably due to en-
coding variations and/or to the core allocator actions. This experiment shows that the Consensus Object can react to a changing scenario, exploring the state-action space and correctly balancing exploitation and exploration.

5.2.5 Different performance goals

This test verifies that the behavior highlighted in the previous test is still valid when the performance goals for the two instances of x264 are different. The system behavior is shown in figure 8, when the second x264 instance is introduced the Consensus Object explores the new state-action space entering a suboptimal condition for the first application. It is however able to recognize the best action to be taken almost immediately, allowing both instances to meet their goals.

5.2.6 Increasing the state-action space

In this case study, four instances of x264 are started five seconds one after the other. Four services are active in the system, these being the four core allocator, one for each instance. With respect to the previous tests, the solution space is much bigger, comprehending 16 possible solutions. The purpose of this test is to verify the capability of the algorithm to quickly react to changing conditions. The heart rate of the four applications converges in a matter of seconds, showing a proper reaction time. Figure 9 shows the results of the test. All the x264 instances are executed with 8 threads and converges to the desired performance levels.

5.2.7 Stress test

This test has been run with eight x264 monitored instances, each of them trying to achieve the same performance goals, their heart rate being between 5 and 10 heartbeats per second. In this experiment eight services that can be activated and deactivated, each one being the core allocator for a specific x264 process. The solution space comprehends 256 possible configurations and this experiment shows that the decision engine is still able to choose actions in order to make the heart rate of the applications converge. Figure 10 shows the results for this test. The time needed to learn the system reactions is intuitively higher than in the previous experiments; nevertheless this test confirms the flexibility of the algorithm. Moreover, the Consensus Object reaction is fast enough to drive all the eight applications to their desired performance level.

6. CONCLUSIONS AND FUTURE WORK

In this paper we propose an autonomic architecture capable of performing optimizations on itself, and capable of adapting to unpredictable, unknown, and unfavorable conditions through a decision engine that can learn from experience and optimize the performance of the system. We start from the Application Heartbeats as a simple, flexible, and lightweight monitoring facility and we have literally built around it. We have developed two novel components: the Consensus Object and the Services API. The Consensus Object is the central entity that gathers all the information coming from Heartbeats-enabled applications and decides which actions to undertake in order to make the processes reach the desired goals; its decision engine is based on R-learning, a machine learning algorithm that makes it possible for the Consensus Object to learn from experience. The Services API is a programming interface suggested for those...
processes that can improve or decrease performance (either at system level and at application level): services adopting such interface are controlled by the Consensus Object which can either enable or disable them. The Services API is equal among all possible services and the Consensus Object uses the same commands to control each service. The decision engine does not have any static information on the services and dynamically learns which services to enable and when. In this way new services can be added extremely easily and the whole architecture proves to be modular, flexible, and extensible. Results have been effective in demonstrating the applicability of the proposed approach. It has been proved that the whole infrastructure is lightweight, the overhead of the Consensus Object and of the Services API is quite small, and ready to be expanded in the future to support more services.

Acknowledgment

We’d like to thank the Rocca Foundation and the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) for their support through the Progetto Rocca and the HiPEAC collaboration grant.

References


