AcOS: an Autonomic Management Layer Enhancing Commodity Operating Systems

Davide Basilio Bartolini *, Matteo Carminati *, Riccardo Cattaneo *, Jacopo Panerati *,
Filippo Sironi * †, Donatella Sciuto *

* Politecnico di Milano, Dipartimento di Elettronica e Informazione
{bartolini, carminati, sironi, sciuto}@elet.polimi.it,
{riccardo1.cattaneo, jacopo.panerati}@mail.polimi.it

† Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory
sironi@csail.mit.edu

***Abstract*— Traditionally, operating systems have been in charge of serving as a convenient layer between applications and the bare-metal hardware by both providing an abstraction of hardware itself and allocating available resources to the applications. Both of these roles are becoming ever more important due to the increasing complexity of modern computer architectures. The rise of chip multi-processors brought to consolidation of multiple applications and systems (i.e., through hardware-supported virtualization) on a single piece of silicon. Heterogeneous System-on-Chips (SoCs) are becoming ubiquitous, while deep memory hierarchies (i.e., multiple level of caches) are around since the beginning of this millennium. Moreover, modern systems have to face the challenge of meeting a growing set of functional and non-functional requirements (e.g., performance, temperature, power consumption, etc.); a system-wide strategy is needed and operating systems may become the right actors for this role.**

This paper describes an approach to enhance commodity operating systems with an *autonomic* layer, introducing *smart*, automatic resource allocation through self-management capabilities, which in turn leverage the availability of user and system-specified goals and constraints. The methodology for realizing such an *Autonomic Operating System* (AcOS) is illustrated throughout its building blocks: *monitors*, *actuators*, and *adaptation policies*. The applicability and usefulness of this approach are demonstrated by providing experimental evidence gathered from different case studies involving two different operating systems (i.e., GNU/Linux and FreeBSD) and dealing with diverse goals and constraints: performance and temperature requirements.

I. Background and Introduction

The turn of computer architectures from the well-established single-core structure to multiple (and, possibly, heterogeneous) processing elements is a years-long trend. This paradigm shift has been dictated by physical and architectural motivations [1]. Physical causes depend on the so-called power wall (i.e., inability to increase the clock frequency without hurting power consumption), while architectural reasons relate to the well-known ILP-wall (i.e., the diminishing revenue due to further micro-architectural optimizations). These difficulties made it infeasible to keep up with Joy's performance law and, to survive its commitment to performance improvements, the computer industry changed strategy, opening the multicore era.

In the single-core era, software performance improvements were provided by beefier single-core processors and applications experienced the so-called "free lunch", with free-of-charge speedups obtainable by just upgrading to the latest processor. The new parallel course in computer architectures, despite being due to architectural reasons, carries the side effect of ending the free lunch, moving a good part of the burden of improving performance onto the software developers' shoulders. The demanding ability of producing efficient and reliable parallel software adds to the already considerable bulk of expertise needed for successfully coping with requirements in terms of computing performance, functionality, reliability, and constraints satisfaction required by today's IT. This situation leads to an increased need of pushing part of the system management effort into computing systems themselves, which can be achieved leveraging *autonomic computing* techniques [2].

Autonomic computing carries a proposal of embedding *self-* properties* into computing systems with the objective of maintaining "good-enough" working conditions. The term autonomic strongly recall the initial link between autonomic computing and the biological world, in fact, autonomic computing systems were initially supposed to mimic the behavior of the autonomic nervous systems of human beings. In details, computing systems should be able to monitor themselves and their environment, detect significant changes and decide how to react, and act to execute such decisions towards user-specified goals in terms of non-functional requirements (e.g., desired QoS, system health, etc.) [3].

Within this context, realizing support for autonomic management at the operating system level is crucial. The operating system is the central system layer, serving as the bridge between hardware components and applications by both offering abstractions for more convenient access to hardware devices and managing the allocation of system resources (e.g., processor or memory bandwidth). Hence, the operating system has both full access to the bare hardware and overall view of the software being executed. The operating system is where maximum information regarding the status of the computing system as a whole and maximum freedom of action over its behavior are available. For these reasons, we claim that an appropriate infrastructure at the operating system level is fundamental for being able to realize an autonomic apparatus within a computing system, able to integrate the diverse self-management techniques proposed in literature [4, 5, 6, 7, 8, 9, 10, 11]. This paper presents the proposed methodology and an evaluation through case studies of different implementations of this approach to the realization of an *Autonomic Operating System* (AcOS) layer.

In the remainder of this paper, Section II describes the proposed methodology, Section III presents case studies regarding

the implementation of this methodology over two well-known commodity operating systems and reports experimental results based on these case studies. Finally, Section IV gives an overview of related works and Section V concludes the paper.

II. Autonomic Operating System Infrastructure

The creation of an infrastructure for autonomic computing in commodity operating systems opens to the possibility of supporting self-management capabilities for any system supported by the OS of choice. For instance, enhancing GNU/Linux, and, more specifically, the Linux kernel with an autonomic layer enables a wide variety of system and devices (ranging from mobiles to supercomputers) to take advantage of the benefits carried by runtime self-adaptation. Autonomic Operating System (AcOS) is aimed at this target, i.e., the definition of a unified infrastructure for autonomic computing and its implementation over commodity operating systems. The methodology at the base of AcOS leverages the *Observe-Decide-Act* (ODA) autonomic loop [3]; Figure 1 represents an overview of a computing system enhanced with the autonomic layer. The AcOS infrastructure defines a model for hosting an

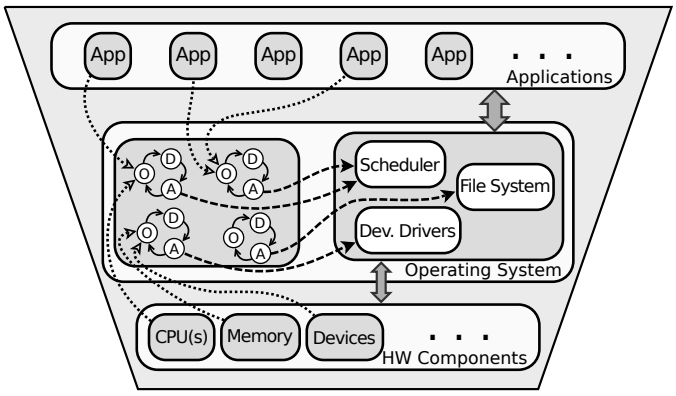


Fig. 1. Overall view of the AcOS infrastructure model. The autonomic layer enhances the operating system implementing different autonomic loops, which evaluate the system status and act within operating system boundary to meet user and system-specified goals and constraints.

array of ODA autonomic loops within the autonomic layer. These self-management loops gather information from running applications, the system, and the hardware components (observation), analyze the data collected comparing it against the desired system status, determining how to intervene (decision), and adjust available knobs (action).

A. AcOS system componenents

AcOS defines common interfaces and templates, standardizing these phases by the definition of three kinds of *autonomic components*:

- *Actuators* are wrappers around a parameter of the system and serve as the actual knobs which can be adjusted by the autonomic layer.
- *Adaptation policies* represent the core autonomic component, as they actually take decisions on how to adapt the system behavior. Adaptation policies implement a decision mechanism to decide how to act on the knobs provided by actuators.
- *Monitors* provide access to information about the current status of the system or of its environment and to user-defined goals. This information is accessed by adaptation policies to apply their decision mechanism.

The *AcOS way* of realizing the three phases of the ODA autonomic loop through autonomic components, along with more details about the overall approach, are further illustrated in the remainder of this section.

B. Acting on the System

Autonomic action on the computing system corresponds to the modification of one or more parameters which affect its runtime behavior. An actuator is a wrapper for one or more parameters, exporting a well-defined action under the form of an Application Programming Interface (API) which can be used by adaptation policies. For instance, an actuator may export the action of assigning a certain number of cores to the tasks of a specific application. The implementation of the calls exported through an actuator's API must take care of performing all the operations required to reliably perform the action. Since the AcOS model places the autonomic layer within the operating system, actuators can affect all the system parameters managed by the operating system through its subsystems (e.g., processor scheduler, device drivers, etc.).

Note that not all the possible actions affect a system parameter: a different class of actuators can be defined at the level of a single application. For instance, an application may be able to vary at runtime the number of its working threads. This kind of actuators, however, still requires support at the operating system level for providing a standardized API to be used by adaptation policies with a broader view on the system than a tight adaptation loop within the application itself. For instance, to support the runtime adaptation of the number of threads in an application, the autonomic layer would define an actuator API for adaptation policies and also another interface for applications capable of this kind of self-adaptation to register and create a communication channel (e.g., based on shared memory) through which be informed of calls to the actuator by adaptation policies. In brief, an application-level actuator is just the same as a system-level one, but its backend is not fully implemented within the operating system but it triggers an action in registered applications. Thanks to this mechanism, AcOS is able to make use of both system and application-level actuators, enabling more powerful adaptation; for instance, when changing the number of cores assigned to an application, an adaptation policy could also change the number of its working threads to match the number of cores, obtaining the best possible scalability.

C. Making Decisions

In the AcOS model, decisions are made by *adaptation policies*, which are defined as a distributed decision making infrastructure. Each adaptation policy chooses which monitors and actuators to use. To do so, adaptation policies embed a decision mechanism able to evaluate information about the system status and user-specified goals, which are made available by monitors (see the next Section II-D). The decision

mechanism within an adaptation policy can vary from a simple heuristic based on empirical observation to more complex control techniques based on control theory or machine learning. Adaptation policies can work at either application-specific (e.g., performance) or system-wide (e.g., system temperature) level or even at both levels. For instance, an adaptation policy seeking the goal of keeping the temperature of the processor below a certain threshold by randomly injecting idle cycles in the CPU(s) would work at system-level, but if the idle cycles injection is selectively performed taking into consideration the performance of the running application, it would operate at application-level, while using system-level information and goals. Clearly, in each case an appropriate actuator to allow the action is needed.

This definition of adaptation policy leads to interaction problems in terms of control stability, for instance, when two adaptation policies want to use the same actuator in opposite directions. Currently, this problem is solved in AcOS by providing the user with the possibility of enabling or disabling each policy. One of the ongoing works is a study of the application of distributed decision theory to this context to automatize the activation and deactivation of conflicting adaptation policies with a system-wide policies coordinator. Even if this is currently an open problem, the AcOS infrastructure has the major advantage of keeping all the components at the operating system level, thusly simplifying the connection of the of the policies coordinator with the autonomic layer.

D. Gathering Information

The autonomic layer needs constantly updated informations regarding the system status in order to be able to take informed decisions through the adaptation policies. In the AcOS model, this information is made available by monitors. Just as actuators, monitors can simply be wrappers around an information source already available to the operating system (e.g., the temperature of each core in the processor). In this case, a monitor is said to be passive [12], since it simply wraps already available information. On the other hand, active [12] monitors process in some way the information to synthesize a metric for characterizing a certain runtime property (e.g., throughput). Other than making runtime information available through APIs, monitors also manage the specification of goals by the user and expose this additional data to adaptation policies. Hence, a monitor is characterize by the metric of the measurements it takes and it must expose at least two APIs: one towards adaptation policies, allowing to get the data on both the current status and the associated goal, and one towards users, allowing to define goals on the monitor's measurement.

E. Autonomic Components Interoperation

In order to have a working autonomic layer, the three classes of autonomic components must be able to reliably and efficiently communicate. The AcOS model has been devised as an enhancement layer for contemporary commodity operating systems, which, in most of the cases, feature a monolithic kernel. In particular, our current design is strongly biased towards UNIX-like operating systems, with a strong separation between system code, which usually runs in kernel mode and application code, which runs in user mode [13][1]. One of the challenges in the design of the autonomic infrastructure was to allow efficient communications between the user and kernel-space, since diverse, interacting components can be placed in different spaces or a single component may need to be partitioned. To avoid the use of system calls, which are time-consuming and possibility harmful [14, 15], on hot code-paths, the AcOS infrastructure leverages shared memory, mapping portions of physical memory in both the user and kernel-space allowing for fast, low-latency communication without unnecessary overheads [10]. This is done considering both security (i.e., managing permissions on the shared memory) and efficiency (i.e., laying out data in a cache-friendly way to avoid false-sharing [16] and other subtle issues). Thanks to this approach, the most common operations are managed in the fastest possible way, while system calls (or alternative interfaces, such as Linux's `sysfs` and `procfs`) are employed to handle uncommon operations [10]. According to this model, an autonomic component can be implemented across user and kernel-space, making use of specially mapped shared memory to inexpensively pass information across address spaces. Moreover, we foresee similar optimization between the different agents involved in the global system optimization.

III. Cases for Autonomic Components

We employed the methodology described in Section II to extend two well-known commodity operating systems (i.e., Linux and FreeBSD) with an autonomic layer. This section reports case studies involving the creation of autonomic components integrated in either Linux or FreeBSD and aimed at tackling interesting runtime management problems in the context of server systems. More into details, one of the interesting problems in server systems is the management of the quality of service (QoS) yielded by the running applications. A typical class of workloads in this scenario is made of throughput-based application processing a stream of data (e.g., a video encoder for online streaming). To enable autonomic QoS management within this context, we designed and implemented an active monitor for throughput called Heart Rate Monitor (HRM) and different adaptation policies and actuators exporting actions over the task scheduler. Another compelling problem is that of thermal management, which can be a significant issue in server farms and data centers. To tackle this problem, we implemented a passive temperature monitor and an adaptation policy exploiting the available actuators. The remaining of this section gives a brief overview of these autonomic components and presents an experimental characterization of what it's possible to realize by applying the proposed methodology. For the experimental evaluation, we employed workstations equipped with current quad-core Intel processors (Xeon and Core i7) and applications from the PARSEC 2.1 [17] parallel benchmark suite.

[1] A different design that exploits a micro-kernel, message-passing based, distributed operating system is foreseeable and maybe even more suitable for the distributed, agent-like environment we describe.

A. Throughput Monitoring: the Heart Rate Monitor

The Heart Rate Monitor (HRM) is the AcOS solution for measuring throughput and it has been employed to characterize the performance of parallel applications. As a motivating example, think of a set of parallel applications executing on a server system (e.g., video encoders for web streaming). When different applications run concurrently, they contend for the available resources and this may lead to performance degradation below a desired QoS threshold (i.e., a minimum frame rate). In this case, a throughput measure can be used to indicate the frame rate at which each encoder is producing the video to be streamed and the QoS goal could be easily set referring to this meaningful metric. Having this information available at runtime enables dynamic adaptation (e.g., by acting on scheduling) to driver applications matching their QoS goal.

Within this context, HRM lets software developers instrument the resource-demanding section (called the *kernel*, or *hotspot*) of the application to emit a *heartbeat* for each encoded frame and provides throughput measures in terms of a *heart rate* [10]. Moreover, according to the AcOS model, HRM permits to express application-level goals in term of desired heart rate (which maps, in the example, to a desired frame rate).

A black-box view of HRM can be given according to a producer/consumer model similar to that used in PEM [18]: producers emit heartbeats to signal events and consumers access the heart rates computed by the monitor. Going back to the example proposed above, the threads doing the frame compression work within the video encoder are the producers, while the component acting on the scheduler to modify the system behavior is a consumer. HRM acts as an interface between producers and consumers, elaborating the heartbeats emitted by the former into a metric used by the latter to evaluate the system status with respect to the goals and take corrective actions as needed. In this way, HRM permits a goal oriented approach towards the control of any phenomena within a computing system which can be characterized by a throughput measure.

Figure 2 represents this black box view, highlighting the flow of information from producers to consumers through HRM, which enables the realization of the ODA control loop. Flexibility and expressiveness are achieved in HRM through the definition of *monitoring groups* (marked as $\mathbf{G}_i$ in Figure 2), by providing measures as multiple *moving averages*, and by letting tie the goal to one of the measures. These concepts are further illustrated in the next subsubsections.

1) Groups: To provide flexibility and be useful in current and future parallel systems, HRM must support monitoring any kind of parallel workload (i.e., multithreaded, multiprocessed, or any feasible mix of the two); this is attained by defining *monitoring groups*. A *group* is a set of tasks (i.e., either processes or threads) constituting the atomic monitoring entity. Referring to the video encoders example, the working threads of each encoder would be grouped together, contributing to the frame rate of that encoder. In general, each group represents an activity which throughput is to be measured.

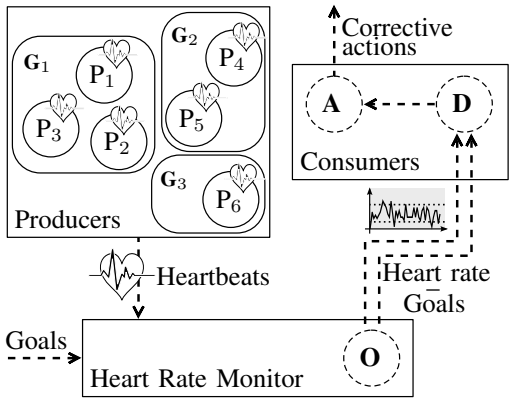


Fig. 2. Black box view of the Heart Rate Monitor. The inputs are heartbeats emitted by instrumented *producers*, organized in *groups*, and goals set by the users. HRM makes heart rate measures available to *consumers*, enabling the creation of an ODA loop.

2) Moving averages: Throughput measures are computed by HRM as heart rates, i.e., for each group, emitted heartbeats count of all its producers over the elapsed time. Clearly, for such a measure, the considered time horizon matters: considering the whole execution time provides a smoothed average, while considering only the heartbeats emitted within a shorter time window discards the old history and allows to better highlight short-term trends. For this reason, HRM provides both a *global heart rate* and *window heart rates*, allowing to tune the focus on longer- or shorter-term trends as required by the specific monitoring context. Initially [10], HRM was able to provide, apart from the global measure, only one window heart rate at a time. A recent advancement allows to output throughput measures on multiple moving averages at the same time, enabling to highlight different trends and providing richer information to consumers.

3) Goal Specification: HRM aims at being a general-purpose throughput monitor, serving as an infrastructure to be used by a variety of consumers to get information regarding any phenomenon, within a computing system, measurable as a throughput. For this reason, it must allow for a simple yet generic way of setting a desired value for the heart rate of a group. This is achieved by allowing to define a desired heart rate range between a *minimum* and a *maximum* heart rate; moreover, it is possible to tie the goal to a specific window heart rate. For instance, in the familiar example of the video encoder, the minimum heart rate could be set to the minimum frame rate to guarantee the desired QoS (e.g., 30 frames/s), the maximum could be set to a value over which no sensible benefit would be achieved and the goal could be tied to a certain time horizon according to how much buffering space is available for the encoded video.

The availability of different window heart rates enables catching different trends in the execution of an application. To provide an example, we implemented and deployed on a workstation equipped with an AcOS-enhanced Linux 3.3 an ad-hoc microbenchmark which attaches four producers to a monitoring group and starting to emit heartbeats as fast as possible in an infinite loop. Clearly, this simple application is very regular and it reaches a peak throughput of about $40 \times 10^6 \frac{heartbeats}{s}$ on the target workstation. Different performance

trends have been artificially created by running a variable number of instances of another CPU-bound application (i.e., the cpuburn stress test), to simulate collateral system load. Figure 3 shows a plot of the microbenchmark's throughput, representing the global heart rate and six additional window heart rates over different moving averages of sizes in the set $\{1, 5, 10, 15, 30, 60\}$[seconds][2]. The execution presents six

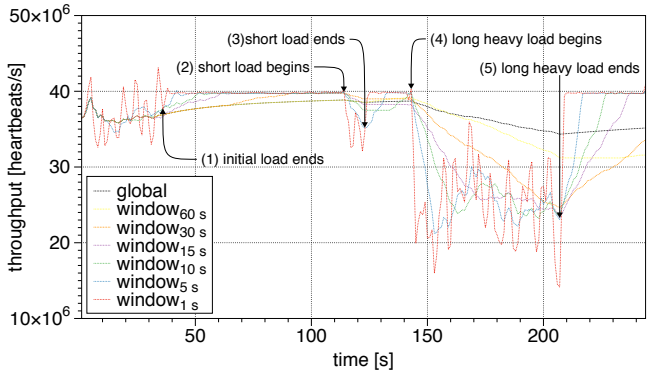


Fig. 3. Global and six different window heart rates of an ad-hoc application showing different performance trends.

different phases: initially, up to the point marked (1), there is a light additional load which makes the heart rates over shorter time windows quite noisy. Then, this load terminates and the application reaches its peak performance up to point (2), when another external load is started. It is apparent from the figure how this disturb is clearly visible looking at the heart rate measured on the short term, while it could go almost unnoticed looking at heart rates take over longer periods. At point (3) the second external load terminates and the microbenchmark goes back to its peak throughput but, at point (4), a heavier and longer-lasting load is applied. Again, it can be noticed how heart rates measured on shorter time windows give a prompt feedback when a change in the performance happens, but tend to become noisy when the execution becomes more regular. Finally, at point (5), the final load terminates and, after some more time, the experiment is concluded.

Using the right moving average can help identify execution phases of a workload; for instance, Figure 4 shows a plot of an instance of the *x264* video encoder application from the PARSEC 2.1 suite working on the reference input and instrumented to emit one heartbeat per encoded frame. The figure shows both the global and window heart rate, which helps highlighting a much lighter phase in the execution, due to the characteristics of the input. These two examples give evidence of how proper instrumentation with HRM can yield accurate runtime information, helping characterize applications' performance. Since these data are available at runtime, in the AcOS layer, in both kernel and user-space, adaptation policies can use them to pursue user-specified goals. This is what is presented in the remainder of this section.

[2]Note that, when there are no enough data to compute a window heart rate over its full size, the measure is still provided using the available data

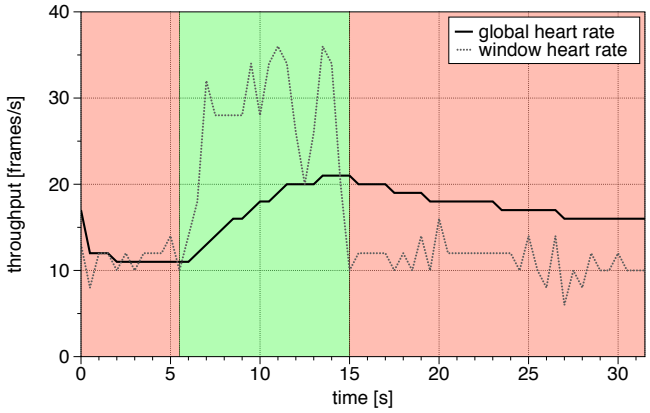
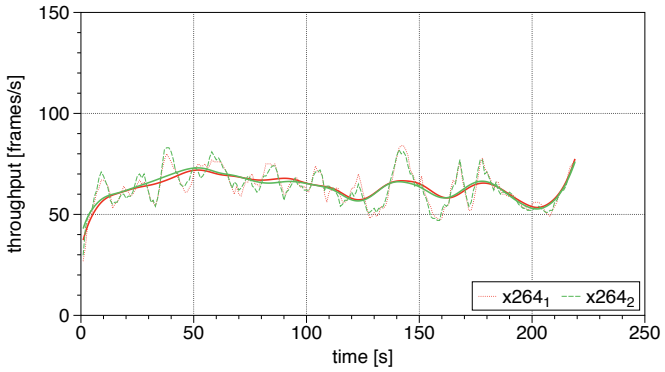


Fig. 4. Execution phases of the x264 video encoder working on the native input of the PARSEC 2.1 suite.

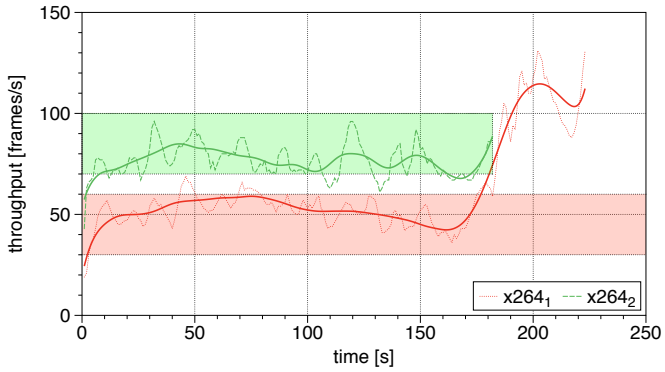
B. Performance-Aware Scheduling

The information provided by HRM has been employed by different adaptation policies affecting the task scheduler in different ways. These case studies are based on Linux 3.3 and appropriate actuators were implemented to allow the autonomic layer access two different scheduling parameters: tasks priority (obtained by scaling their *virtual runtime* [10]) and CPU affinity. The first actuator is used by an adaptation policy built for the Metronome framework [10] and called *Performance-Aware Fair Scheduler* (PAFS), while the second actuator is employed by another adaptation policy named *Performance-Aware Processor Allocator* ((PA)[2]).

1) Performance-Aware Fair Scheduling: The rationale behind PAFS is adapting the priority of the tasks belonging to HRM-instrumented applications according to whether their user-specified throughput goal is being attained or not. If an application is running *too slowly*, i.e., under its minimum desired heart rate, its priority is increased, and conversely if it is running over its maximum desired heart rate. This simple scheme has been implemented in an adaptation policy based on an heuristic decision mechanism, which affects the priority of the tasks belonging to instrumented applications (i.e., comprised in an HRM group) by scaling their virtual runtime, which is the metric used by Linux's Completely Fair Scheduler (CFS) to choose, at each context switch, which task to execute next. To evaluate this adaptation policy, two 4-threaded instances of the x264 video encoder have been run, both instrumented with HRM and attached to different groups, on a quad-core workstation equipped with an AcOS-enhanced Linux 3.3. This time, the workload consists in encoding a copy of the full-lenght *Big Buck Bunny* full HD video [19]. Figure 5a shows the two encoders managed by the Linux Completely Fair Scheduler (CFS), which is perfectly fair in assigning the bandwidth of the cores to the two instances of the same, application resulting in equal performance. The CFS features a sophisticated mechanism to assign different CPU bandwidths to different objects (e.g., applications); however, this mechanism is difficult to use when systems and administrators are faced with high-level performance goals (e.g., frames/s). High-level performance goals make HRM shine since it provides general performance measures understandable



(a) Unmanaged instances of x264.



(b) Managed instances of x264; the performance goals, in frames/s, are [30,60] and [70,100] for the two instances.

Fig. 5. Window heart rate and its LOESS interpolation for each instance of x264.

by users, administrators, and systems and effective adaptation policies can be designed and developed to exploit these data. In Figure 5b, two instances of x264 are executed with different performance goals (i.e., the red and green areas, which are respectively 30 to 60 frames/s and 70 to 100 frames/s) and are driven by the adaptive scheduler towards their performance goal, successfully exploiting the information HRM provides to adjust their *virtual runtimes* (which boils down to assigning them different processor bandwidth). In Figure 5b, the throughput of the slower instance of x264 receives a sudden speedup when the faster instance of x264 teminates. This is due to the fact that the adaptive scheduler from Metronome is designed to account for performance goals whenever there is resource contention within the system, and the maximum heart rate is considered as a *soft bound* on the QoS, and not as a performance cap; when there is no contention applications run as they were unmanaged and thus all the processor time is given to the only running application.

A different interpretation of the maximum heart rate can be given by an adaptation policy employing the CPU affinity actuator, which allows to decide which cores a thread can be run on. In fact, with this kind of action it is possible to both boost and limit the performance of a single application being executed. $(PA)^2$ is an adaptation policy implementing a core-allocation mechanism based on throughput goals as provided by HRM. This adaptation policy features an autoregressive (AR) model to decide the number of cores to assign to each thread of a managed application and a workload estimation adaptive filter. Also $(PA)^2$ was implemented on Linux 3.3 and evaluated on a quad-core workstation to manage a 4-threaded instance of the x264 application. Figure 6 represents the results of such evaluation. The black solid and dotted lines represent respectively the global and window heart rate of a managed run of the encoder, while the red dashed lines show the throughput of the application when run with a fixed number of cores assigned to its four threads. The bottom part of the plot shows the action of the adaptation policy in the CPU affinity actuator (i.e., number of cores to be assigned) and the estimation of the current workload, used to weigh the decision of the number of cores to be assigned. The experiment shows that, despite the changing workload, the autonomic layer is able to keep the performance of the application close to its goal.

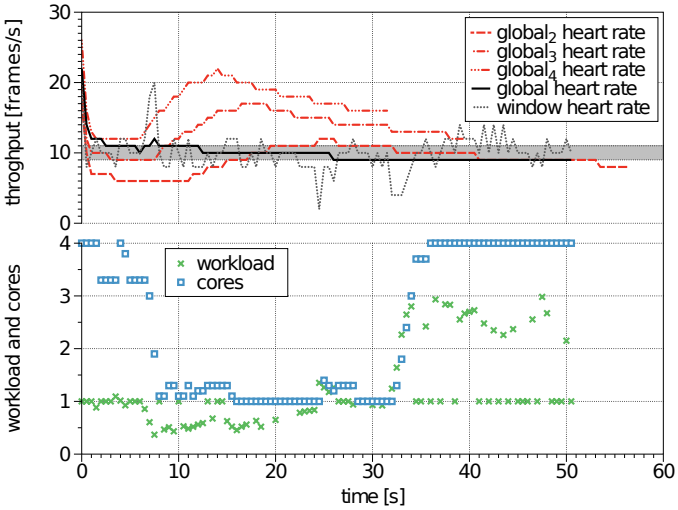


Fig. 6. 4-threaded instance of x264 managed by the $(PA)^2$ adaptation policy towards a throughput goal of $[9, 11]$ frames/second.

C. Adding Thermal-Awareness

An interesting case study extending those regarding QoS management with performance goals comes from adding also system temperature constraints. This example has been implemented over FreeBSD 7.2, which got a porting of the HRM monitor and of the task priority actuator, an implementation of a CPU temperature monitor, which is simply a wrapper around the information available in the model-specific register (MSR), and an idle cycles injection actuator. In this case, an adaptation policy was implemented with the goal of managing performance under the constraints of a maximum working temperature. The decision mechanism is based on control theory and the adaptation policy is split into two parts: one prioritizes tasks according to performance goals and the other selectively injects idle cycles to cool down the processor. This adaptation policy has been evaluated on a quad-core workstation running a patched version of FreeBSD 7.2 by executing four different 4-threaded instances of the *swaptions* application from the PARSEC 2.1 suite. Figure 7 represents the results of this experiment. The black lines represent the average temperature and the heart rate of the four applications (which are very similar, since the scheduler tries to be fair) in

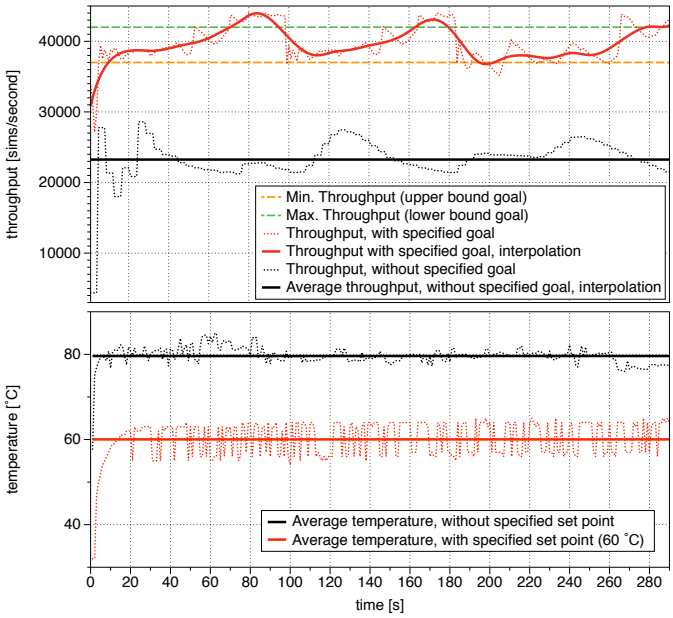


Fig. 7. Four instances of swaptions, each one running with four threads managed with performance and temperature goals. One of the instances is run with a performance goal set between 37000 and 42000 Monte Carlo simulations. The maximum temperature constraint is set at 60 °C.

the base case. The red lines represent the temperature and the heart rate of one of the four applications, which was privileged with a higher performance goal (between the orange and green lines, with the maximum desired temperature set to 60 °C. This experiment shows how it is possible, through coordinating the use of different monitors and actuators, to selectively boost the performance of one application while maintaining the desired border conditions in term of maximum temperature.

D. Learning Adaptation Policies

An alternative approach to the creation of an adaptation policy is leveraging machine learning techniques to let the autonomic layer learn how to drive the actuators in order to achieve the desired goals. The AcOS methodology supports this kind of approach, as actuators can be controlled by reinforcement learning policies harnessing the autonomic infrastructure. As an example, consider the test case represented in Figure 8, where an adaptation policy is learned through Adaptive Dynamic Programming (ADP) to drive two actuators working on frequency scaling and core allocation for letting a 4-threaded instance of the swaptions benchmark achieve a desired throughput setpoint. The application is run 10 consecutive times and the plot shows both the application's heart rate (in red) and the actions taken by the policy. The first two runs are used for exploration (i.e., learning), while in the remaining eight the learnt policy is applied, showing how the machine learning engine is mostly capable of satisfying the performance goal set to 50000 Monte Carlo simulations per second.

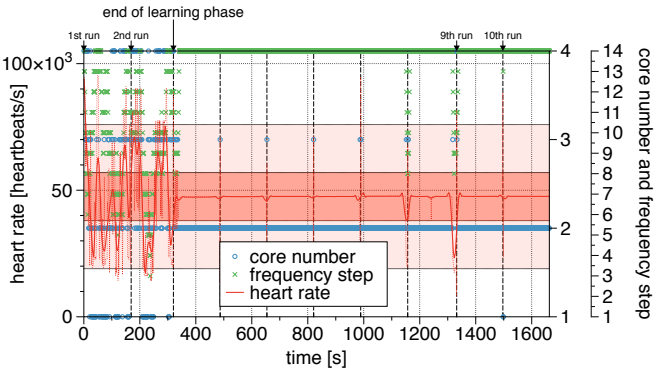


Fig. 8. Learning of an adaptation policy working on frequency scaling and core allocation through Adaptive Dynamic Programming (ADP).

IV. Related Works

This section provides an overview of works related to the topic of this paper. Some of these works deal with an overall approach to adaptivity at the OS level, while others focus on a specific topic such as monitoring or decision making.

The need of a framework for measuring performance and specifying applications' goals, which is the problem tackled in AcOS with the Heart Rate Monitor, is acknowledged within the Tessellation OS [20, 11]. The authors notice that using low-level information for determining applications performance is labor-intensive and error-prone, and that higher-level measures (e.g., *frame rate*), are more suitable for letting users and software developers state performance goals. A work going into this direction is *Application Heartbeats* [4, 21], a performance monitoring infrastructure for self-adaptive systems which shares the basic ideas with HRM: developers are provided with a method for instrumenting applications to signal progress (by issuing signals called *heartbeats*) and expressing performance goals.

Tesauro, Kephart, et al [22, 23, 24, 6, 7] designed and developed *Unity*, a framework to build self-managing distributed systems borrowing from the artificial intelligence community. Unity relies on a multi-agent approach to control the interaction among autonomic elements and exploits utility-function to specify objectives and (hybrid) reinforcement learning to acquire an effective amount of knowledge domain. Heo et al [25] designed, implemented, and evaluated *AdaptGuard*, a framework for guarding autonomic systems from instability caused by software anomalies and faults. This is somehow complementary to that of reinforcement learning, which is leveraged in AcOS to learn adaptation policies at runtime; 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.

The problem of maximizing performance and/or providing (global or per-application) quality-of-service (QoS) in chip multi-processors (CMPs) for both multi-threaded applications and multi-programmed mixes, which is tackled by some of the proposed autonomic components, has received quite a lot of attention in the latest years. Many approaches focused on the management and partitioning of the cache hierarchy [26, 27, 28, 29, 30]. Other researchers addressed the

problem at the very end of the memory hierarchy altering the behavior of memory controllers [31, 32, 33, 34]. Other works addressed the problem through dynamic assignment of processors [35, 4, 36] and CPU bandwidth [10]. Researchers have also tackled the problem with more comprehensive frameworks capable of managing more than one resource. Bitirgen et al. [37] exploited machine learning to distribute shared resources on a CMP. Srikantaiah et al [27] devised a strategy to partition both processors and caches. Hoffmann et al. [5] proposed SElf-awarE Computing (SEEC), harnessing both control theory and machine learning to meet user-define performance goals allocating cores and scaling frequencies. Sharifi et al. [9] presented METE, a framework for meeting QoS through control theory.

V. Conclusions and Future Works

This paper presents AcOS: a methodology for enhancing commodity operating systems with an autonomic management layer. This approach is based on three basic blocks, collectively called autonomic components, which ensure the availability of information and goals, manage adaptation decisions, and allow to modify system parameters through appropriate knobs. This methodology has been applied towards the enhancement of two widespread opensource kernels (i.e., Linux and FreeBSD) with autonomic management of performance and temperature requirements. The case studies show that the applications of the proposed methodology able achieve the different goals.

This work on AcOS opens the way to many developments; in particular, we are working towards the implementation of parts of the model which have not been extensively experimentally evaluated yet (e.g., the use of application-level actuators in concert with system-level ones). Another open issue is investigating the possibility of applying distributed decision theory to automatize the activation and deactivation of possibly conflicting adaptation policies with a system-wide coordinator.

References

[1] Samuel H. Fuller and Lynette I. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1), 2011.

[2] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[3] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2), 2009.

[4] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *Proceedings of the 7^{th} International Conference on Autonomic computing*, pages 79–88, 2010.

[5] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A Framework for Self–aware Management of Multicore Resources. Technical Report MIT–CSAIL–TR–2011–016, Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, 2011.

[6] Jeffrey O. Kephart and Rajarshi Das. Achieving Self–Management via Utility Functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[7] Gerald Tesauro. Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies. *IEEE Internet Computing*, 11(1):22–30, 2007.

[8] Robert W. Wisniewski, Dilma Da Silva, Marc A. Auslander, Orran Krieger, Michal Ostrowski, and Bryan S. Rosenburg. K42: lessons for the OS community. *SIGOPS Oper. Syst. Rev.*, 42(1), 2008.

[9] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. METE: Meeting End-to-End QoS in Multicores through System-Wide Resource Management. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 2011.

[10] Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D. Santambrogio. Metronome: Operating System Level Performance Management via Self-Adaptive Computing. In *Proc. of the 49^{th} Design Automation Conference*, 2012.

[11] Juan A Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalf, Steven Hofmeyr, Krste Asanovic, and John Kubiatowicz. Resource Management in the Tessellation Manycore OS. In *Proc. of the 2^{nd} Workshop on Hot Topics in Parallelism*, 2010.

[12] Markus C. Huebscher and Julie a. McCann. A survey of Autonomic Computing – degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[13] Andrew S Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[14] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[15] Livio Soares and Michael Stumm. Exception-Less System Calls for Event-Driven Servers. In *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[16] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[17] Princeton University. Parsec benchmark suite website, November 2011.

[18] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development*, 50(2.3), 2006.

[19] Big Buck Bunny. `http://www.bigbuckbunny.org/`.

[20] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *Proc. of the 1^{st} Workshop on Hot Topics in Parallelism*, 2009.

[21] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats for Software Performance and Health. In *Proceedings of the 15^{th} Symposium on Principles and Practice of Parallel Programming*, pages 347–348, 2010.

[22] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A Multi–Agent Systems Approach to Autonomic Computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 464–471, 2004.

[23] G. Tesauro, R. Das, W.E. Walsh, and J.O. Kephart. Utility–Function–Driven Resource Allocation in Autonomic Systems. In *Proceedings of the Second International Conference on Autonomic Computing*, pages 342–343, 2005.

[24] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the Third Internation Conference on Autonomic Computing*, pages 65–73, 2006.

[25] Jin Heo and Tarek Abdelzaher. AdaptGuard: Guarding Adaptive Systems from Instability. In *Proceedings of the 6^{th} International Conference on Autonomic Computing*, pages 77–86, 2009.

[26] Jichuan Chang and Gurindar S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*, 2007.

[27] Shekhar Srikantaiah, Reetuparna Das, Asit K. Mishra, Chita R. Das, and Mahmut Kandemir. A Case for Integrated Processor-Cache Partitioning in Chip Multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[28] S. Srikantaiah, E. Kultursay, Tao Zhang, M. Kandemir, M.J. Irwin, and Yuan Xie. MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[29] Mahmut Kandemir, Taylan Yemliha, and Emre Kultursay. A Helper Thread Based Dynamic Cache Partitioning Scheme for Multithreaded Applications. In *Proceedings of the 48th Design Automation Conference*, 2011.

[30] Akbar Sharifi, Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Courteous Cache Sharing: Being Nice to Others in Capacity Management. In *Proceedings of the 49th Annual Design Automation Conference*, 2012 (to appear).

[31] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[32] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[33] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.

[34] Fang Liu and Yan Solihin. Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 2011.

[35] Julita Corbalán, Xavier Martorell, and Jesús Labarta. Performance-Driven Processor Allocation. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, 2000.

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

[37] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008.