The Autonomic Operating System Research Project – Achievements and Future Directions

Davide B. Bartolini, Riccardo Cattaneo, Gianluca C. Durelli, Martina Maggio*
Marco D. Santambrogio, Filippo Sironi
Politecnico di Milano, *Lund University
{bartolini, rcattaneo, durelli, santambrogio, sironi}@elet.polimi.it, martina.maggio@control.lth.se

ABSTRACT

Traditionally, hypervisors, operating systems, and runtime systems have been providing an abstraction layer over the bare-metal hardware. Traditional abstractions, however, do not consider for non-functional requirements such as system-level constraints or users' objectives. As these requirements are gaining increasing importance, researchers are looking into making user-specified and system-level objectives first-class citizens in the computer systems' realm.

This paper describes the *Autonomic Operating System* (*AcOS*) project; *AcOS* enhances commodity operating systems with an autonomic layer that enables self-* properties through adaptive resource allocation. With *AcOS*, we investigate intelligent resource allocation to achieve user-specified service-level objectives on application performance and to respect system-level thresholds on CPU temperature. We give a broad overview of *AcOS*, elaborate on its achievements, and discuss research perspectives.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: *Hardware/software interfaces; System architectures*; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.4.8 [**Operating Systems**]: Performance—*Measurements; Modeling and prediction; Monitors*

General Terms

Design, Management, Measurement, Performance

Keywords

Autonomic computing, Operating systems, Virtualization, Performance management, Dynamic thermal management

1. INTRODUCTION

In the last decade, the failure of Dennard's scaling law [11] determined the inability to leverage single-threaded performance improvements in order to keep doubling integrated circuits performance every two years, as stated by the established Joy's law. Meanwhile, transistors density has kept its exponential increase, as predicted by Moore's Law [22]. These two phenomena drove chip manufacturers towards embracing parallelism, leading to the prevalence of Chip-MultiProcessors (CMPs) and multi-processor system-on-chips (MPSoCs) throughout most computing systems segments.

On the one hand, mobile and embedded systems feature heterogeneous MPSoCs specialized for efficiency. For instance, the NVIDIA Tegra platform leverages the partner/companion core approach [20], while the ARM big.LITTLE chip implements the single-instruction set architecture (ISA) heterogeneous computing approach, coupling high-throughput and energy-efficient cores [17].

On the other hand, large-scale installations, like warehouse-scale computers, build on nodes equipped with homogeneous CMPs offering an increasing number of on-chip cores. For example, Tilera already offers 64-core solutions for general-purpose processing [5], while Intel and NVIDIA propose 100+thread solutions, like the Intel Xeon Phi and the NVIDIA Tesla, respectively, to accelerate embarrassingly parallel applications.

Thanks to Moore's law, chip manufacturers equip each new generation of CMPs and MPSoCs with an increased amount of on-chip resources (e.g., cores, caches, memory controllers). This unprecedented availability of on-chip resources encourages workload consolidation, realized by co-locating single and / or multi-threaded applications onto the same chip. Co-located applications share on-chip resources and require careful multiplexing in both space (i.e., placement on CPUs) and time (i.e., CPU bandwidth) to maintain performance predictability despite contention over on-chip shared resources [23]. The advent of virtualization and commodity hypervisors for widespread ISAs [3] brings additional complexity, as different users, with different service-level objectives (SLOs), can own the co-located applications. This scenario opens new research issues in the areas of resource allocation, which had settled on well-established techniques for time-shared single-core processors.

With the *Autonomic Operating System* (*AcOS*) project, we target these issues by looking for ways to automatize allocation of on-chip shared resources. We aim at enabling users to easily state SLOs and to automatically tune resource allocations in order to meet user-specified SLOs, while enforcing system-level constraints.

We focus on a specific system-level constraint: maximum processor temperature. This constraint is of primary concern for current and future CMPs and MPSoCs, since recent lithographic technologies cannot keep up the down-scaling of supply voltage with the up-scaling of clock frequency and transistor density, causing power density to increase and therefore reducing the capacity of packages of dissipating the resulting heat. Avoiding high processor temperature means avoiding to impair performance [10], energy efficiency [29], and reliability [30] of integrated circuits.

With *AcOS*, we research cheap and efficient software techniques to keep temperature within a system-specified threshold while minimizing the impact of this cap on performance.

The remainder of this paper first illustrates the high-level ap-

DAC '13, May 29 - June 07 2013, Austin, TX, USA

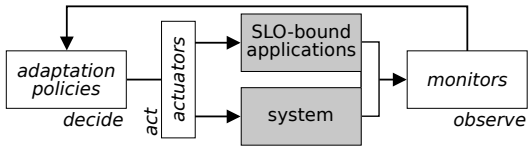


Figure 1: Interaction of the autonomic components with the computing system and the applications; these components realize the *observe*, *decide*, and *act* phases of the ODA control loop.

proach and methodology we adopt in *AcOS* (Section 2) and then focuses on performance (Section 3) and thermal (Section 4) management, going into details and validating *AcOS* through experimental results. We conclude by discussing related research (Section 5) and future directions and perspectives (Section 6). We provide further details on more specific aspects in Appendices A to C.

2. FOUNDATIONS AND METHODOLOGY

AcOS derives its overall methodology from *autonomic computing* [18]. In the last decade, autonomic computing has grown from a futuristic vision of computing systems autonomically taking care of themselves and of their own complexity to a multifaceted and pragmatic research field [9]. The aim of this research is automatically exploiting runtime information to ease user interaction with computing systems. From a theoretical standpoint, autonomic computing has the goal of enhancing computing systems with *self-* properties*, as analyzed by Salehie and Tahvildari [25], who also propose a taxonomy for autonomic computing. According to this taxonomy, *AcOS* is a *closed*, *model-based* solution employing *continued monitoring*, *dynamic decision making*, and *external proactive adaptation*. *AcOS* extends commodity operating systems with self-* properties at the software level (both within the operating system and with a companion runtime system); more specifically, it enables *self-adaptive* and *self-managing* properties. *AcOS* enables these properties by employing feedback control to make active use of runtime information. Different representations exist for such control loops; in *AcOS*, we adopt the most compact of these abstractions: the *observe—decide—act* control loop (ODA).

We specialize the ODA control loop to highlight interactions of the autonomic layer provided by *AcOS* with the computing system and with the applications. Figure 1 shows the three *autonomic components* realizing the steps of the ODA control loop: (1) *monitors* realize the observe phase; (2) *adaptation policies* provide the decide phase; and *actuators* enact the act phase.

Monitors are components in charge of properly exposing runtime information; they can be either passive or active elements, depending on how they gather information [16]. For instance, we use a passive monitor to observe CPU temperature; this monitor simply retrieves data from model-specific registers. Instead, to observe the throughput of an application, we use an active monitor that implements an infrastructure to synthesize this metric, which is not directly available. Each monitor is also in charge of exposing an API to allow setting SLOs on the specific measurement it provides.

Adaptation policies elaborate the measurements and SLOs exposed by monitors to estimate the corrective action needed to drive the measurements towards meeting the objectives. Each adaptation policy uses a specific decision engine; we experimented with heuristics, analytic modeling, and control theory.

Actuators provide mechanisms that adaptation policies can use, through appropriate APIs, to enact corrective actions. For instance, we implemented an idle cycle injection actuator that adaptation policies can use to selectively preempt application threads in favor of the idle task with the goal of capping processor temperature without unnecessarily harming performance.

Figure 1 highlights interactions of monitors, adaptation policies, and actuators with the system as a whole and the applications. Monitors retrieve information on both system-wide parameters and application-specific measurements and SLOs. Adaptation policies elaborate this information and use actuators to affect system and application behavior. This paper illustrates how *AcOS* exploits this structure to enhance commodity operating systems with performance-aware resource allocation and temperature management.

Autonomic computing can be beneficial throughout the hardware/software stack. However, since the operating system, coupled with runtime systems, is traditionally in charge of managing system resources and can control both the hardware and the applications, we argue that this is the level where these techniques are most needed and can yield most benefits. Moreover, an operating system with autonomic capabilities can both serve as a convenient base to offer interfaces for autonomic applications and exploit additional self-* properties offered at the hardware level. For these reasons, we build *AcOS* as an extension to commodity operating systems.

3. PERFORMANCE MANAGEMENT

Modern computer architecture design follows the principle of optimizing for the common case. For instance, caches, coupled with prefetching, dramatically reduce memory latency for regular access patterns. While this strategy continuously improves performance for many applications, it also makes it unpredictable; this side effect can make advanced architectural features unsuitable for embedded systems, where what matters is often the worst-case execution time (WCET). Moreover, the advent of CMPs and MPSoCs leads to on-chip co-location of applications that must rely on shared hardware resources; this scenario further impairs performance predictability.

We seek answer to the following question: can we leverage autonomic computing to achieve predictable performance for applications co-located onto a multi-core processor? Well-established techniques exist to estimate the WCET of single-threaded applications on single-core processors [31] and recent research focuses on multi-core processors [12, 21]. This research relies on offline profiling to provide strong guarantees for time-critical systems. Instead, we want to dispense from offline profiling and provide users with an intuitive means of stating SLOs on execution time.

We tackle this challenge with *Metronome* [28] and *Metronome++*, respectively a heuristic and model-based feedback control policy; both these policies introduce performance-awareness in the Linux kernel and rely on the *Heart Rate Monitor* (*HRM*) to provide performance measurements and requirements.

3.1 Performance Metrics and Measurement

Our goal is to enforce a SLO defined on the execution time $\bar{t}$ to complete $\bar{n}$ of *units of work*; for instance, a video encoder may be required to process $\bar{n}$ frames in $\bar{t}$ seconds. In order to strike appropriate resource allocation without the need of offline profiling, we need a metric to estimate the execution time of an application at runtime. We leverage the known amount of units of work to define a proxy for the execution time: the *required throughput* to attain the SLO is $\bar{g} = \bar{n}/\bar{t}$. Equation (1) formalizes how we enforce the SLO: given that after k control steps the application completed m units of work, we keep the *global throughput* $g(k)$ up to step k close to $\bar{g}$.

$$\forall k,\, g(k) \equiv \bar{g} \qquad \text{where} \qquad g(k) = \frac{m}{k} \tag{1}$$

Global throughput is an application-specific high-level performance metric that easily allows users to state meaningful SLOs [14, 15,

23, 28]. We leverage this metric for automatic goal-oriented resource allocation, dispensing users and administrators from the laborious process of analyzing application properties (e.g., scalability) to manually determine resource allocations.

Application-specific high-level performance metrics require application support; in our case, applications need to provide progress information. For this reason, we developed *HRM* [28]: an active monitoring infrastructure to synthesize throughput measurements from *heartbeats*. Similarly to previous proposals [13], *HRM* exports a simple API for applications to emit a heartbeat whenever they complete a unit of work. Based on the time stamps of these signals, *HRM* efficiently provides adaptation policies with throughput measurements expressed in heartbeats/s; these measurements directly map to application-specific performance metrics such as frames/s for a video encoder or decoder. *HRM* also exports API calls for users to express SLOs, according to the methodology described in Section 2. The major novelties of *HRM* are support for both multi-threaded and multi-programmed applications, through the definition of monitoring *groups*, and the system-wide (i.e., both in user- and kernel-space) visibility of throughput measurements [28]. Appendix A, provides additional details regarding the usage of *HRM* to instrument various flavors of multi-threaded applications.

We implemented *HRM* both on top of Linux and FreeBSD.[1] Both implementations feature a split design where most of the infrastructure leaves in kernel-space and a small-footprint user-space library, namely *libhrm*, exports an API for applications and user-space adaptation policies. To pass information across address spaces, we exploit shared memory to carefully map shared memory pages and use cache-aligned data structures to avoid poor performance due to caching issues such as false sharing [28].

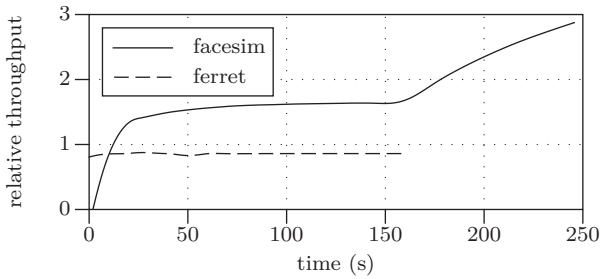
3.2 Metronome

With *Metronome*, we introduce performance-awareness by means of a non-invasive modifications to the *Completely Fair Scheduler* (*CFS*), which is the default scheduling class for the Linux kernel since version 2.6.23. CFS, as most schedulers in commodity operating systems, offers mechanisms (e.g., priorities and resource containers [2]) to allocate resources (e.g., CPUs and CPU bandwidth) to applications; however, the task of determining appropriate settings to respect SLOs is far from easy. To address this issue, *Metronome* automates CPU bandwidth allocation based, at each scheduling step k, on the performance error $e(k)$ between the desired ($\bar{g}$) and measured ($g(k)$) global throughput: $\forall k$, $e(k) = \bar{g} - g(k)$.

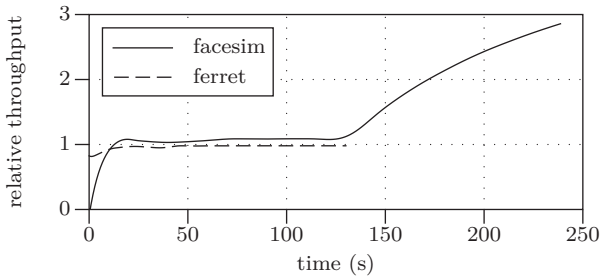
Metronome leverages *HRM* to retrieve throughput measurements and user-specified SLOs and uses this information to modify a single parameter of the CFS scheduler: the *virtual runtime* (*vruntime*). Since CFS picks tasks for execution in ascending order of *vruntime*, modifying this value implicitly defines CPU bandwidth allocation. *Metronome* uses a simple heuristic to tune the *vruntime* of tasks of SLO-bound applications with a scaling factor s that depends, for each SLO-bound application a, on the current performance error $e_a(k)$. If $e_a(k) > 0$, then a needs more CPU bandwidth and *Metronome* will weight the *vruntime* of its tasks with a factor $s = g(k)/\bar{g}$. Otherwise, if $e_a(k) <= 0$, a is delivering sufficient throughput to attain its SLO and *Metronome* will not affect the *vruntime* of its tasks.[2] Notice that, since in Linux and in most commodity operating systems the scheduler resides in kernel-space, *HRM*'s capability of exporting measurements system-wide is crucial for *Metronome* to be effective despite using a very simple heuristic.

[1]We support versions 2.6.35 and 3.2 of the Linux kernel and version 7.2 and 9.0 of the FreeBSD kernel.

[2]For additional details regarding design, implementation, and validation of *HRM* and *Metronome*, refer to [28].



(a) Linux kernel *vanilla*.



(b) Linux kernel enhanced with *Metronome*.

Figure 2: Relative throughput of *facesim* and *ferret*; 1 represents the throughput required to attain per-application SLOs.

We validate *Metronome* on a workstation with an Intel Core i7-870 Processor (we disable Intel Hyper-Threading, Enhanced SpeedStep, and Turbo Boost Technologies), 4 GB of 1066 MHz Single Ranked DIMMs, and the Linux kernel 2.6.35 enhanced with *HRM* and *Metronome*.[2] We used two applications from the PARSEC 2.1 benchmark suite [6]: *facesim* and *ferret*. We measure the performance of *facesim* in frames/s, as it is instrumented with *libhrm* to emit a heartbeat per computed frame; *facesim* yields ≈ 0.67 frames/s when running with 4 threads. We instrumented *ferret* to emit a heartbeat per computed query; therefore, its performance reads in in queries/s. When run with 4 threads, *ferret* yields ≈ 30 queries/s.

Figure 2 shows the dynamics of the global throughput of *facesim* and *ferret* relative to the respective SLOs; we arbitrarily choose a SLO of 0.22 frames/s for *facesim* and 19 queries/s for *ferret*. Figure 2a shows the two applications scheduled by the unmodified CFS. CFS partitions CPU time evenly among the applications till ≈ 150 s, when *ferret* terminates. Since CFS is a work-conserving scheduler, it never idles resources whenever there are runnable tasks; therefore, when *ferret* terminates, *facesim* can use the whole processor and its global throughput grows. Notice that, since CFS is not aware of the SLOs, it does nothing to enforce them: doing so would require manual intervention to increase the relative priority of *ferret*. *Metronome* performs this action automatically, as Figure 2b reports. The simple heuristic at the base of *Metronome* is able to adjust the relative priorities of the two applications to keep both close to a normalized performance of 1, which represents the respective SLOs. *Metronome* maintains all the desirable properties of CFS (e.g., non-starvation) and also the work-conserving behavior; therefore, *facesim* gets the full processor when *facesim* terminates, after approximately 130 s.

3.3 Metronome++

Metronome demonstrates how a simple heuristic can be enough to enable goal-oriented resource allocation by exploiting runtime performance feedback. With *Metronome++*, we devise and evaluate a more advanced adaptation policy to dynamically allocate CPUs to SLO-bound applications in a multi-core processor; again, our goal is achieving performance predictability to meet SLOs. Commodity operating systems provide various mechanisms (e.g.,

task pinning) to define the task to CPU mapping; however, just as for task priorities, these mechanisms only provide knobs that administrators are in charge of manually adjusting.

Similarly to other recent proposals [26], we estimate at runtime the scalability characteristics of applications; however, our goal is respecting SLOs and not maximizing performance. To estimate scalability characteristics, we use the least squares algorithm to fit a second order polynomial that correlates the number of allocated CPUs and the throughput measurements provided by *HRM*; Appendix B justifies this methodology through experimental results.

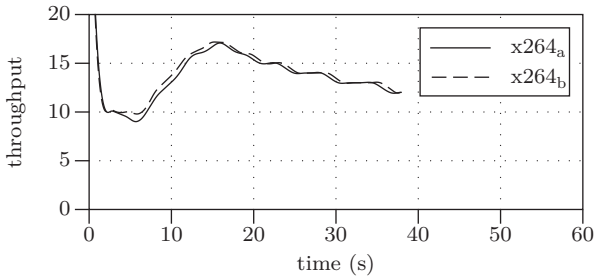
We build *Metronome++* based on a user/kernel-space split design and manage cross-address space communication by careful (i.e., cache-aware) sharing of mapped memory pages. The scalability characteristics estimation runs in user-mode in order to take advantage of linear algebra libraries, while the actuation (i.e., tasks migration among run queues to) runs in kernel-mode, so as to avoid the overhead of synchronous system calls and thus minimize runtime impacts. We devise the kernel-space side of *Metronome++* to care for task migration minimization and load balancing.

Since changing the mapping of tasks to CPUs can be expensive due to task migration, *Metronome++* takes into account the history of throughput measurements to avoid trashing a proper CPU allocation due to noise in the data. The drawback of this choice is reduced reactivity in case applications go through different execution phases. We address this issue by adding a prediction mechanism for execution phase transitions as a second adaptation level. To detect a transition, we use an exponential moving average of the ratio between the throughput and the resource allocation. We describe in more details and evaluate the prediction mechanism in Appendix C.

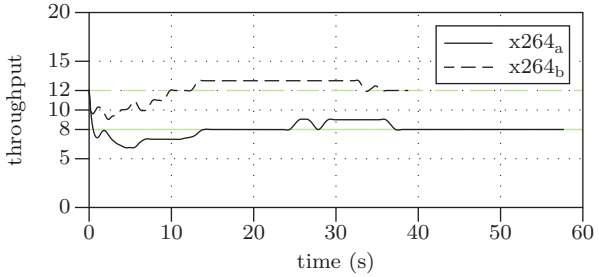
To evaluate *Metronome++*, we use the *x264* application from the PARSEC 2.1 benchmark suite [6] and co-locate two identical instances of the application on a workstation with an Intel Xeon Processor W3670 (we disable Intel Hyper-Threading, Enhanced SpeedStep, and Turbo Boost Technologies), 12 GB of 1333 MHz Single Ranked DIMMs, and the Linux kernel 3.2 enhanced with *HRM* and *Metronome++*. We instrument *x264* to emit a heartbeat per encoded frame: we measure its performance in frames/s.

Figure 3 shows the dynamics of the two instances of *x264* when run on *vanilla* Linux and managed by *Metronome++*. An important characteristic of *x264* is the presence of input-dependent execution phases: the native input of the PARSEC 2.1 benchmark suite presents a lighter-weight (i.e., higher performance) phase between the 70-th and the 300-th frames. The execution phases of *x264* emerge from Figure 3a: the global throughput is far from constant, even though the Linux kernel *vanilla* allocates resources evenly between the two instances (marked $x264_a$ and $x264_b$). Figure 3b shows the performance of the two instances when using *Metronome++* to dynamically allocate CPUs to match SLOs (we arbitrarily choose 8 and 12 frames/s). These results show that *Metronome++* is able to drive both instances to respect their SLO, effectively estimating scalability characteristics to strike proper CPU allocation allocations and responding to execution phase transitions; we validate our transition prediction mechanism in Appendix C.

Notice that *Metronome++* does not expose the same work-conserving behavior of *Metronome*, as it does not over-allocate resources. This feature potentially enables to activate power-saving techniques on idle resources. The use of a more complex analytic model with respect to *Metronome* makes *Metronome++* slower in converging to the desired global throughput due to the need to "warm up" the scalability characteristics and execution phase prediction; however *Metronome++* improves upon *Metronome* in terms of robustness to noise and response to execution phase transitions.



(a) Linux kernel *vanilla*.



(b) Linux kernel enhanced with *Metronome++*.

Figure 3: Throughput of two instances of *x264*; the green constant lines represent the throughput required to attain SLOs.

4. TEMPERATURE MANAGEMENT

We argue that providing autonomic performance-aware resource allocation is just one side of the coin with respect to the benefits embedded systems can draw from autonomic computing; on the other side is smart enforcement of system-level constraints. One of the major emerging system-level constraints in modern computing systems is capping CPU temperature. Maintaining CPUs cool is crucial for energy efficiency [29] and reliability [30], as high temperature increase of leakage power and leads to a reduction of the MTTF. Furthermore, high temperature in embedded systems may lead to usability issues (e.g, in hand-held devices).

Blindly enforcing temperature constraints, as done with classic dynamic thermal management (DTM) techniques, indiscriminately harms applications performance. With *AcOS*, we investigate how to intelligently enforce system-level temperature constraints while avoiding to break performance SLOs. For this purpose, we devise and evaluate *ADAptive Performance and Thermal ManagEment* (*ADAPTME*): a feedback control framework for dynamic performance and thermal management. *ADAPTME* leverages control theory and idle cycle injection [1, 4] to co-manage CPU temperature and performance of multi-programmed workloads.

For *ADAPTME*, we need to observe both performance and temperature. We use *HRM* (see Section 3.1) for throughput measurements and per-CPU machine-specific registers available in modern multi-core processors for temperature measurements: per-CPU high-priority kernel-mode threads periodically sample and make available temperature measurements.

ADAPTME uses an adaptation policy leveraging discrete-time linear models for performance and temperature reported in Equations (2) and (3), respectively.

$$r_i(k+1) = r_i(k) + \eta_i \cdot p_i(k) \tag{2}$$

$$T_j(k+1) = T_j(k) + \mu_j \cdot I_j(k) \tag{3}$$

The model in Equation (2) assumes that performance $r_i(k+1)$ of application i at control step $k+1$ can be derived through a linear combination of its performance $r_i(k)$ and the priority $p_i(k)$ (e.g., *nice* value) of i's tasks, weighted with a parameter η_i, at the previous step. Equation (3) states the same relation for temperature

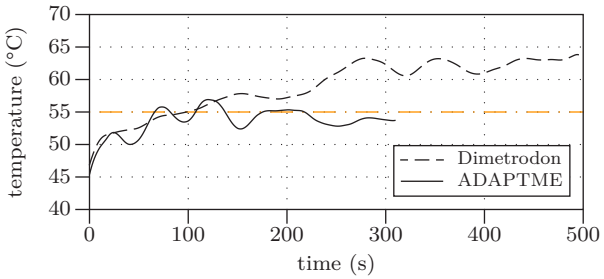


Figure 4: Average temperature for 5 consecutive runs of *swaptions* when capping with either *Dimetrodon* (50 % chance of injecting idle time) or *ADAPTME* (55 °C temperature constraint).

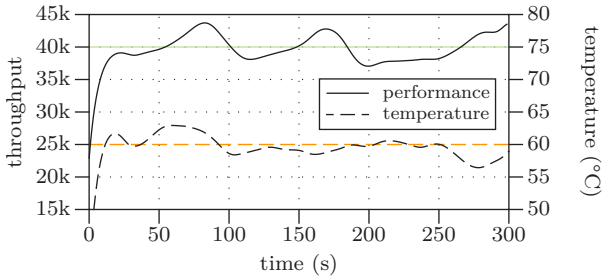


Figure 5: Throughput of one SLO-bound instance of *swaptions* among the 4 co-located and average temperature when using *ADAPTME* for performance (SLO: 40000 swaptions/s) and temperature (constraint: 60 °C).

$T_j(k+1)$ of CPU j, where $I_j(k)$ is the fraction of idle time injected during the previous control period.

Based on these two models, we synthesize two deadbeat adaptive controllers to respectively estimate priority and idle time required by each SLO-bound application and CPU. We couple each SLO-bound application with a priority controller and each CPU with an idle time controller. Since the control context is variable, finding static values for the parameters η_i and μ_j for each application and CPU is both impractical and ineffective. Therefore, we employ an adaptive filter, in this case an exponential moving average, to estimate parameters online.

Since idle cycle injection has the side effect of impairing applications performance, this action can conflict with the process of adjusting priorities to respect SLOs. For this reason, to avoid instability, we need to define a policy to coordinate controllers. We use a probabilistic solution: whenever a temperature controller requires the injection of idle cycles and a performance controller requires an application not to be preempted (which happens when the application is not respecting its SLO), *ADAPTME* actually injects idle time over that application with a tunable probability.[3]

We implemented *ADAPTME* in FreeBSD 7.2, which we also extend with a port of *HRM*. We use the *swaptions* applications from the PARSEC 2.1 benchmark suite [6] as our reference application to evaluate *ADAPTME*; we instrumented *swaptions* with *HRM* to measure performance in swaptions/s. We realize two different experiments to evaluate *ADAPTME*. First, we consider the temperature controller alone to test the ability of enforcing constraints; this experiment allows us to compare against *Dimetrodon* [1]: a state-of-the-art extension of FreeBSD 7.2 for preventive DTM. Second, we evaluate coupled temperature and performance controllers. Experimental results were collected on the workstations described in Sections 3.2 and 3.3, respectively.

Figure 4 shows the dynamics of average CPU temperature of *ADAPTME* and *Dimetrodon* in the first experiment. *Dimetrodon* employs probabilistic feedforward control and allows to specify an idle cycle injection probability; however, it does not provide any guarantee with respect to actual temperature capping. Instead, *ADAPTME* exploits feedback control and allows to specify a temperature constraint; moreover, *ADAPTME* is devised so as to minimize the impact of idle cycle injection on performance. These characteristics allow *ADAPTME* to attain better performance (i.e., faster execution time) and keep lower temperature than *Dimetrodon*.

Figure 5 shows the results of the second experiment: we co-locate four instances of *swaptions* running with 4 threads on an infinite input dataset and set a SLO (i.e., 40000 swaptions/s, which is twice as much as each instance achieves by default) for one them and a temperature constraint (i.e., 60 °C). Experimental results show that *ADAPTME* is able to both cap temperature and prioritize the SLO-bound application to attain its SLO.

[3]For additional details regarding the design, implementation, and validation of *ADAPTME*, refer to [4].

5. RELATED WORK

With *AcOS*, we investigate how to leverage autonomic computing at the operating system and runtime level to ease the management of computing systems. In this paper, we focus on performance-aware resource allocation for multi-core processors (with *Metronome* and *Metronome++*) and intelligent thermal capping (with *ADAPTME*). The rest of this Section discusses related work, first focusing on each of these two topics and then on wider projects.

Application Heartbeats [13] was an early proposal for a framework allowing applications to easily export measurements and specify SLOs on their performance in high-level metrics. These ideas gave momentum to initial work on *HRM* and *Metronome* [28]. *HRM* improves *Application Heartbeats* in the observation phase, introducing the concept of *groups* and leveraging a design split across user and kernel-space to improve efficiency and enable system-wide visibility of throughput measurements. This last feature enables *Metronome* to achieve performance-aware CPU bandwidth allocation by means of a heuristic acting on applications *vruntime*. *Metronome++* shares the idea of performance-aware CPU allocation with *PDPA* [8] and the use of application scalability characteristics estimates like *SBMP* [26]. However, both these works pursue performance maximization, while *Metronome++* aims for SLOs satisfaction and performance predictability. Moreover, we leverage a high-level application-specific performance metric, which leads to the advantages discussed in Appendix B.

A relevant but orthogonal technique with respect to *ADAPTME* is thermal-aware scheduling; relevant works are *Heat-and-Run* [24] and *ThreshHot* [32]. *Heat-and-Run* [24] exploits simultaneous multi-threading to place on the same core tasks requiring different functional units and it manages tasks migration to balance temperature across a CMP. *ThreshHot* [32] schedules tasks ordered from the "hottest" (most CPU-intensive) to the "coldest" (most I/O-intensive); this schedule guarantees to minimize temperature at the end of an epoch. The goal of thermal-aware schedulers is minimizing temperature without degrading performance. *ADAPTME*, which enforces temperature requirements through DTM, is orthogonal to thermal-aware scheduling: DTM tackles situations where minimizing temperature is not enough and capping is necessary.

HybDTM [19] exploits the *hot* and *cold* tasks classification for DTM: whenever temperature exceeds the threshold, it throttles "hot" tasks first by lowering their priority. *HybDTM* is meant for single-core processors and many of its considerations do not apply to multi-core processors. *Dimetrodon* Bailis et al. [1] is a framework on top of FreeBSD that leverages idle cycle injection to decrease temperature with a probabilistic feedforward approach. We com-

pare *ADAPTME* against *Dimetrodon* in Section 4.

METE [27] is a control-theoretical framework for CMPs to meet QoS by managing CPU, cache ways, and memory bandwidth allocation for multi-threaded applications. *AcOS* focuses on only CPU and CPU bandwidth allocation, but also considers temperature capping; taking into consideration automatic allocation of additional resources is one of the future directions for *AcOS*. *SEEC* [14, 15] is a runtime system that performs resource allocation to respect SLOs by exploiting control theory and machine learning; *SEEC* focuses on balancing performance and power consumption requirements, while *AcOS* can balance performance and thermal requirements. *Tessellation* [7] is an operating system for multi and many-core processors and client computing systems based on the concept of adaptive resource-centric computing (ARCC). *Tessellation* restructures the operating system around QoS-guaranteed resource containers called *cells*. *AcOS* is orthogonal to *Tessellation*: we focus on feedback control, while *Tessellation* focuses on providing adaptation mechanism within the operating itself. *AcOS* could exploit an operating system like *Tessellation* as a base for further research.

6. PERSPECTIVES

We started the *AcOS* project to seek an answer to a question: *can we enhance commodity operating systems with an autonomic layer so as to respect user-specified SLOs and enforce system-level constraints?*

With the control loops (i.e., ODA loops) we propose and validate in this paper, we contribute to moving a step towards an affirmative answer. *Metronome* and *Metronome++* demonstrate the ability to respect user-specified SLOs on performance measurements by means of CPU bandwidth and CPU allocation, while *ADAPTME* is able to enforce a system-level temperature constraints while still accounting for performance. However, several open problems require further research before we can definitively answer this question.

An interesting direction is evaluating SLOs defined on different performance metrics. For instance, instead of requesting a bound on WCET, users may define the desired QoS on latency or real-time constraints. Feedback control techniques to automatically attain such requirements with on-chip shared resources may need adaptation policies based on different mechanisms and algorithms.

Possibly, different QoS definitions may require the management of a wider set of resources (e.g., cache ways, memory bandwidth, file system cache, disk bandwidth, network bandwidth, etc.). One of the challenges towards this direction is enabling mechanisms to effectively manage such resources at runtime. If coordinate management of multiple resources was demonstrated in a simulation environment [27], actual hardware and software mechanisms are needed to experiment with similar adaptation policies on commodity computing systems [7].

Having an increasing pool of resources to manage and an increasing number of control loops leads to a third compelling challenge: properly orchestrating a large number of possibly conflicting adaptation policies. With *ADAPTME*, we propose a probabilistic heuristics to define the interaction of two conflicting adaptation policies aimed at respecting performance SLOs and enforcing a temperature constraint. A solution of this kind, however, does not scale well with an increasing number of control loops: we need to research a more systematic methodology.

7. ACKNOWLEDGMENTS

The authors would like to thank Simone Campanoni, Fabio Cancarè, Henry Hoffmann, Giovanni F. Del Nero, and Donatella Sciuto for their contributions to the *Autonomic Operating System* project.

8. REFERENCES

[1] P. Bailis *et al.*, "Dimetrodon: Processor-Level Preventive Thermal Management via Idle Cycle Injection," in *Proc. of DAC*, 2011.
[2] G. Banga *et al.*, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proc. of OSDI*, 1999.
[3] P. Barham *et al.*, "Xen and the Art of Virtualization," in *Proc. of SOSP*, 2003.
[4] D. B. Bartolini *et al.*, "A Framework for Thermal and Performance Management," in *Proc. of MAD*, 2012.
[5] S. Bell *et al.*, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," in *Proc. of ISSCC*, 2008.
[6] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.
[7] J. A. Colmenares *et al.*, "Tessellation: Refactoring the OS around Explicit Resource Containers with Continuous Adaptation," in *Proc. of DAC*, 2013.
[8] J. Corbalan *et al.*, "Performance-Driven Processor Allocation," in *Proc. of OSDI*, 2000.
[9] S. Dobson *et al.*, "Fulfilling the Vision of Autonomic Computing," *IEEE Computer*, vol. 43, no. 1, 2010.
[10] N. El-Sayed *et al.*, "Temperature Management in Data Centers: Why Some (Might) Like It Hot," in *Proc. of SIGMETRICS*, 2012.
[11] H. Esmaeilzadeh *et al.*, "Power Challenges May End the Multicore Era," *Commun. ACM*, vol. 56, no. 2, 2013.
[12] G. Giannopoulou *et al.*, "Timed Model Checking with Abstractions: Towards Worst-Case Response Time Analysis in Resource-Sharing Manycore Systems," in *Proc. of EMSOFT*, 2012.
[13] H. Hoffmann *et al.*, "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *Proc. of ICAC*, 2010.
[14] ——, "SEEC: A General and Extensible Framework for Self-Aware Computing," Massachusetts Institute of Technology, Tech. Rep., 2011.
[15] ——, "Self-aware Computing in the Angstrom Processor," in *Proc. of DAC*, 2012.
[16] M. C. Huebscher and J. a. McCann, "A survey of Autonomic Computing – degrees, models and applications," *ACM Comp. Surv.*, vol. 40, no. 3, 2008.
[17] B. Jeff, "Big.little system architecture from arm: Saving power through heterogeneous multiprocessing and task context migration," in *Proc. of DAC*, 2012.
[18] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, 2003.
[19] A. Kumar *et al.*, "HybDTM: A Coordinated Hardware-Software Approach for Dynamic Thermal Management," in *Proc. of DAC*, 2006.
[20] E. Lau *et al.*, "Multicore Performance Optimization Using Partner Cores," in *Proc. of HotPar*, 2011.
[21] M. Lv *et al.*, "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software," in *Proc. of RTSS*, 2010.
[22] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, 1965.
[23] R. Nathuji *et al.*, "Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds," in *Proc. of EuroSys*, 2010.
[24] M. D. Powell *et al.*, "Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System," in *Proc. of ASPLOS*, 2004.
[25] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, 2009.
[26] H. Sasaki *et al.*, "Scalability-Based Manycore Partitioning," in *Proc. of PACT*, 2012.
[27] A. Sharifi *et al.*, "METE: Meeting End-to-End QoS in Multicores through System-Wide Resource Management," in *Proc. of SIGMETRICS*, 2011.
[28] F. Sironi *et al.*, "Metronome: Operating System Level Performance Management via Self-Adaptive Computing," in *Proc. of DAC*, 2012.
[29] K. Skadron *et al.*, "Temperature-Aware Microarchitecture: Modeling and Implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, 2004.
[30] J. Srinivasan *et al.*, "The Case for Lifetime Reliability-Aware Microprocessors," in *Proc. of ISCA*, 2004.
[31] R. Wilhelm *et al.*, "The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 2008.
[32] X. Zhou *et al.*, "Performance-Aware Thermal Management via Task Scheduling," *ACM Trans. Archit. Code Optim.*, vol. 7, no. 1, 2010.

APPENDIX

A. HRM USAGE AND VALIDATION

HRM [28] is an active monitor providing: *libhrm*, which is a simple API to instrument parallel applications, and system-wide available application-specific throughput measurements and requirements (i.e., SLOs). This section provides several use cases for *libhrm*, which is employed to instrument parallel applications with diverse multi-threading models, and demonstrates the efficiency of *HRM* by evaluating its runtime impact.

A.1 Instrumenting Parallel Applications

HRM organizes instrumented applications in *groups*, where each group is a set of tasks cooperating on a certain activity (e.g., encoding a video) that is bound to a SLO. Therefore, to properly use *libhrm*, a multi-threaded application must: (1) *attach* its tasks (i.e., threads) to a group; (2) *set* the SLO;[4] (3) emit one *heartbeat* upon completion of a unit of work; and (4) *detach* its tasks from the group upon termination.

We analyze the instrumentation of 11 out of 13 multi-threaded applications from the PARSEC 2.1 benchmark suite [6] employing diverse multi-threading models, which allows to show the flexibility and ease of use of *libhrm*.

We focus on the following applications: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *facesim*, *ferret*, *fluidanimate*, *raytrace*, *streamcluster*, *swaptions*, and *x264*. These 11 applications can be grouped in four categories according to their multi-threading model:

- *category 1—blackscholes*, *canneal*, *fluidanimate*, *streamcluster*, and *swaptions* use "fork & join" of workers;
- *category 2—bodytrack*, *facesim*, and *raytrace* leverage pools of workers running different jobs in parallel;
- *category 3—dedup* and *ferret* use a pipeline with pools of workers serving the stages;
- *category 4—x264* employs "spawn & kill" of workers to realize a virtual pipeline.

For each category, we give additional details regarding the structure and instrumentation of one application.

Applications in *category 1* are straightforward to instrument, as they use a simple multi-threading model. The sequence diagram in Figure 6 shows the structure and instrumentation of these applications. The main thread of the applications is responsible for forking (i.e., `pthread_create(3)`) the worker threads and joining them (i.e., `pthread_join(3)`) when they terminate. The first worker thread attaches to (and implicitly creates) the group, and it sets the SLO; the other worker threads attach to the group and, just like the first worker thread, start their computation. Figure 7 visualizes the typical structure of computation of a worker thread, which runs in a loop terminating a unit of work, and subsequently emitting a *heartbeat* for each iteration. When the application is terminating, before re-joining, the worker threads detach from the group.

Applications in *category 2* use a pool of worker threads running parallel kernels; therefore, none of the threads completes a unit of work alone. Figure 8 represents the common structure of these applications. The main thread, which acts as a dispatcher, is the first to attach to the group. Due to the structure of these applications, which is represented in Figure 9, the main thread emits heartbeats. However, we still attach all the worker threads to the group to make adaptation policies aware that they are actually relevant.

Applications of *category 3* employ many pools of worker threads organized in a pipeline. In these applications, the main thread is responsible for forking the pools of worker threads and waiting for

[4]We also allow users and administrators to change the SLO.

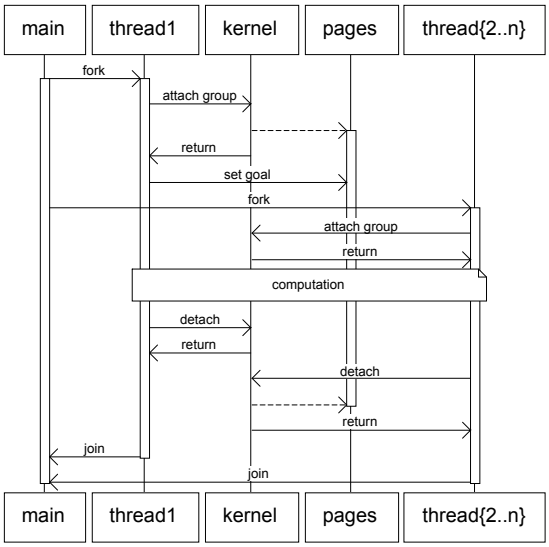


Figure 6: Structure of the applications in category 1.

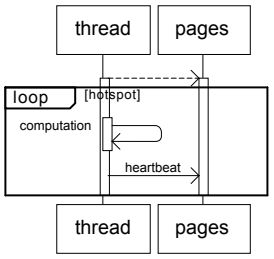


Figure 7: Computation the applications in category 1.

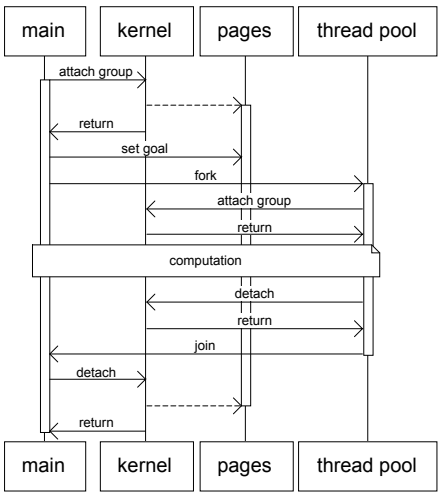


Figure 8: Structure of the applications in category 2.

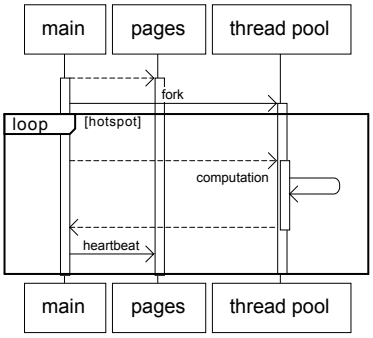


Figure 9: Computation of the applications in category 2.

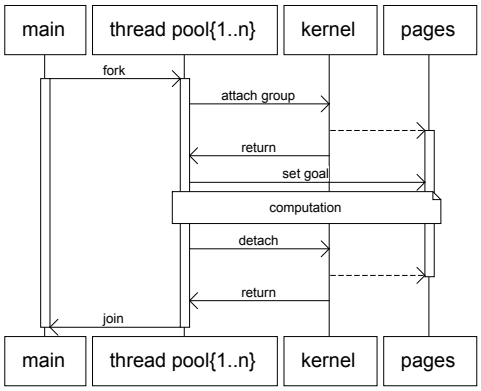


Figure 10: Structure of the applications in category 3.

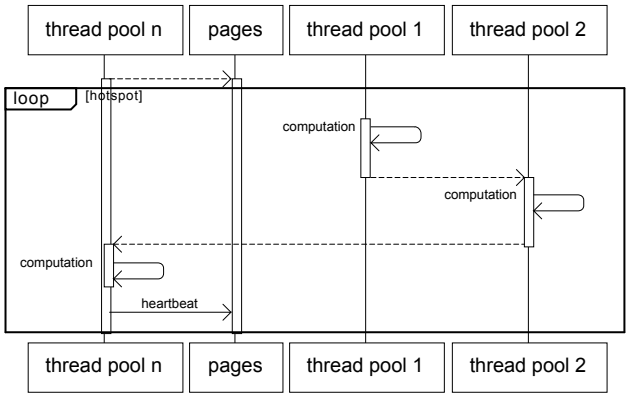


Figure 11: Computation of the applications in category 3.

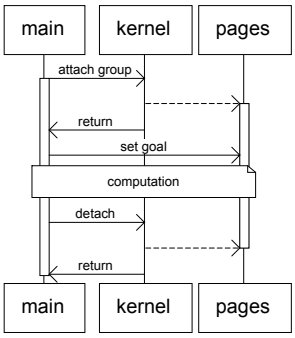


Figure 12: Structure of the applications in category 4.

them to join upon application completion. All the spawned threads attach to the group (the first thread automatically creates the group and sets the SLO); Figure 10 shows the general structure of these applications. The worker threads contained in the *n*-th pool (i.e., the last stage of the pipeline) are the ones committing each unit of work and are responsible for emitting heartbeats, as Figure 11 illustrates.

The last category, i.e., *category 4*, contains only one application, namely *x264*. *x264* creates a virtual pipeline based on a "spawn & kill" multi-threading model, which makes the instrumentation straightforward. Figure 12 illustrates the structure of the instrumentation of *x264*. The main thread is responsible for creating and attaching to the group. Figure 13 focuses on the computation phase: the main thread spawns many different worker threads that re-join when their computation ends. The main thread maintains the notion of advancement (i.e., encoding of frames in *x264*); hence, it is responsible for emitting heartbeats. Just as for applications in *category 2*, we still attach all the worker threads to the group to inform

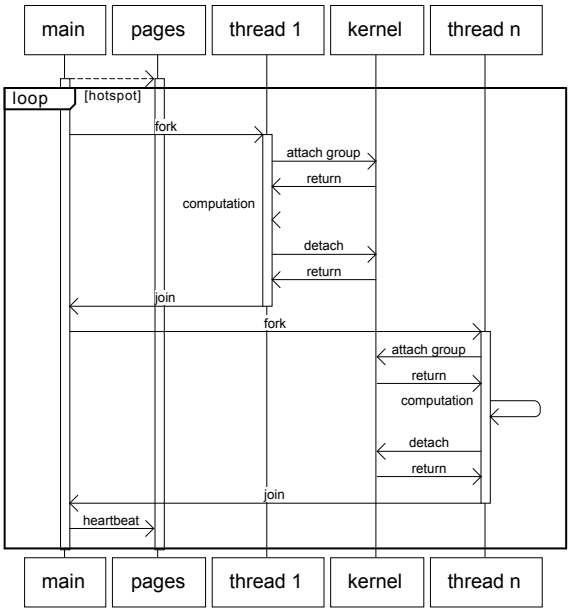


Figure 13: Computation of the applications in category 4.

adaptation policies about their relevance.

A.2 Evaluating the Runtime Impact

We evaluate the overhead of *HRM* on all the applications we instrumented from the PARSEC 2.1 benchmark suite.

Table 1 reports average execution time and its standard deviation of 100 consecutive runs of unmodified (i.e., *vanilla*) and instrumented applications with the native input and the computed runtime impact (i.e., overhead).

Experimental results where collected on a workstation with an Intel Xeon Processor W3570 (we disable Intel Hyper-Threading, Enhanced SpeedStep, and Turbo Boost Technologies), 12 GB of 1333 MHz Single Ranked DIMMs, and the Linux kernel 3.2 enhanced with *HRM*. We configured *HRM* to compute throughput measurements every 100 ms.

The highest runtime impact we measured is 2.80 % for *dedup*; with the exception of *x264*, higher runtime impacts (e.g., *bodytrack* and *dedup*) coincide with short execution times and we argue this is due to "non-amortized" costs of creating the group and attaching worker threads, which are the most expensive operations. According to experimental results we can state that *HRM* is efficient and imposes negligible runtime impact.

Table 1: Comparison between *vanilla* and instrumented applications from the PARSEC 2.1 benchmark suite

category	application	*vanilla*		instrumented		overhead
		avg. (ms)	std. (ms)	avg. (ms)	std. (ms)	
1	*blackscholes*	68731.67	1998.33	68902.53	221.21	0.25%
	canneal	96405.94	1846.36	96913.76	488.74	0.53%
	fluidanimate	95785.44	627.38	96077.83	103.19	0.31%
	streamcluster	147536.15	2393.04	147460.57	333.19	-0.05%
	swaptions	75308.29	308.39	75508.16	249.35	0.27%
2	bodytrack	52849.39	412.03	53732.33	878.61	1.67%
	facesim	145175.15	2256.19	145408.80	787.26	0.16%
	raytrace	124036.75	901.47	124441.34	750.43	0.33%
3	dedup	33509.96	955.21	34448.43	1187.31	2.80%
	ferret	113626.21	527.36	114106.41	218.52	0.42%
4	*x264*	32657.13	252.06	32713.23	255.53	0.17%

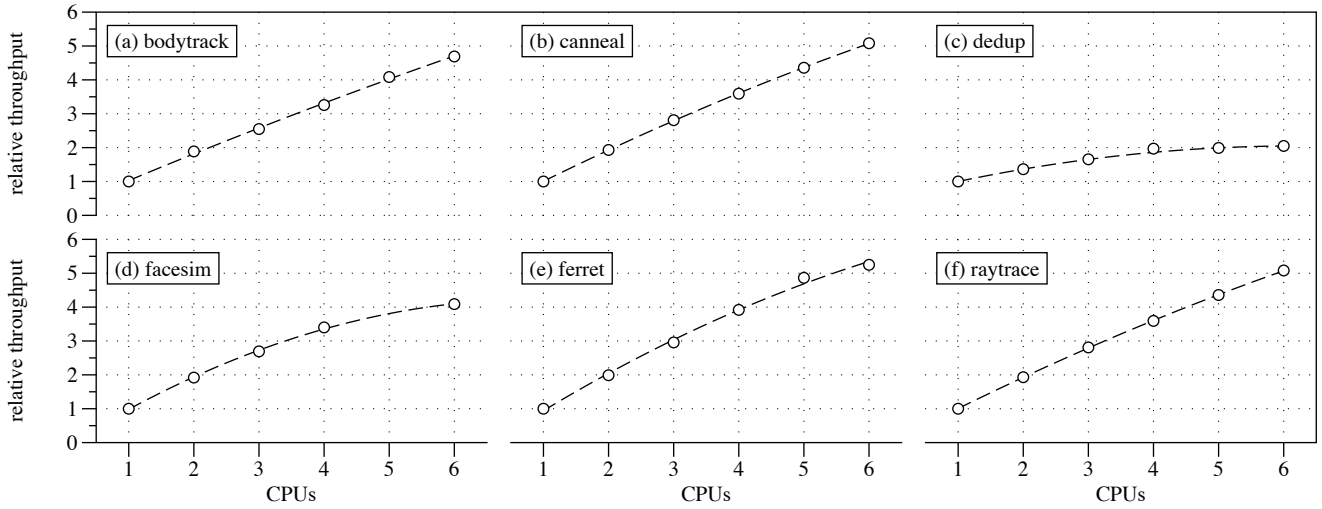


Figure 14: Scalability characteristics of 6 out of 13 applications from the PARSEC 2.1 benchmark suite showing sub-linear behavior.

B. SCALABILITY CHARACTERISTICS

This section elaborates on scalability characteristics and justifies our model exploiting a second order polynomial.

Figure 14 displays the scalability characteristics of 6 out of 13 applications from the PARSEC 2.1 benchmark suite [6]. The *x*-axes indicate the number n of allocated CPUs (in each experiment, we execute applications with n threads); *y*-axes indicate the relative (to the minimum) throughput measurements. Experimental results were collected on the workstation described in Section 3.3.

Most applications (i.e., *canneal*—Figure 14b, *ferret*—Figure 14e, and *raytrace*—Figure 14f) show quasi-linear scalability characteristics, with more than $5\times$ speedup with 6 CPUs. However, other applications (i.e., *dedup* and *facesim*—Figure 14c and d) present sub-linear scalability characteristics even with a relatively small number of CPUs. Previous research [26] analyzed an intersecting subset of applications reported that scalability characteristics bend drastically with 12 or more CPUs.

Previous work [8, 26] describes scalability characteristics through variations of Amdahl's law trying to account for the overheads introduced by synchronization primitives. Instead, we model scalability characteristics through a second order polynomial that puts in direct relationship the number of allocated CPUs with the throughput measurement. We justify our choice by fitting the data in Figure 14 with first and second order polynomials. Each data point is the average over experiments repeated until the width of the 95% confidence interval was below 1%. Fitting with a second order polynomial (i.e., our model) yields, for all applications, a coefficient of determination $R^2 \geq 0.99$; instead, the same metric with a first order polynomial, which is comparable to using Amdahl's law, varies more (e.g., down to $R^2 \approx 0.94$ for *dedup*). Therefore, our choice is justified for the applications we consider.

AcOS exploits this model with *Metronome++* (see Section 3.3), which estimates scalability characteristics at runtime by periodically collecting high-level throughput measurements through *HRM* and fitting them with the least squares algorithm. Conversely to previous research [26], we use high-level application-specific metrics instead of machine-specific metrics such as the number of retired instructions; this choice derives from two main reasons. First, high-level application-specific metrics are meaningful to users, who can easily state SLOs (see Section 3.1). Second, the number of retired instructions is not constant across different CPU allocations [26] and is also sensitive to applications employing non-sleeping synchronization primitives (e.g., spinlocks); instead, the number of *heartbeats* an application emits is constant across different CPU allocations for a given dataset size.

To estimate scalability characteristics, *Metronome++* initially allocates 1 CPU to each instrumented application and collects the first data point (i.e., the number of allocated CPUs, throughput measurement pair). Then, it varies the allocations to collect two additional data points in order to have the three initial data points required to run the least squares algorithm and fit the second order polynomial reported in Equation (4).

$$\mathrm{r} = c + b \cdot \mathrm{p} + a \cdot \mathrm{p}^2 \qquad \text{where} \qquad a < 0 \tag{4}$$

We model the throughput measurement r of an application as a quadratic function of the number p of allocated CPUs; a, b, and c are parameters describing the scalability characteristic. The initial quasi-linearity of scalability characteristics is captured by b, while the final flattening is captured by a, whose influence becomes stronger as number of allocated CPUs grows.

Metronome++ adjusts the fitting with additional data points whenever the "environment" changes (i.e., the number of instrumented applications grows or shrinks); in this way, it can catch the effects of contention over on-chip shared resources. It is worth noticing that, even though *Metronome++* employs a second order polynomial for modeling purpose, there exists a single feasible solution $\bar{p}$ given a throughput requirement $\bar{r}$, which substitutes r when Equation (4) is used to predict the right CPU allocation $\bar{p}$.

C. EXECUTION PHASES

Two different events can trigger *Metronome++* to adjust CPU allocation. The first event (as already mentioned in Appendix B) is a change in the multi-programmed workload due to an instrumented application starting or finishing. This behavior is natural since changing the "environment" may alter the way co-located applications interact with on-chip shared resources. The second event triggering CPU allocation adjustment is a change in the execution phase of an application. Applications can go through execution phases with different scalability characteristics (e.g., CPU and I/O-bound execution phases have dramatically different scalability characteristics) or different performance for a given resource allocation. *Metronome++* attempts to address both of these issues.

To address the first issue, *Metronome++* re-evaluates the scalability characteristics of instrumented applications when their through-

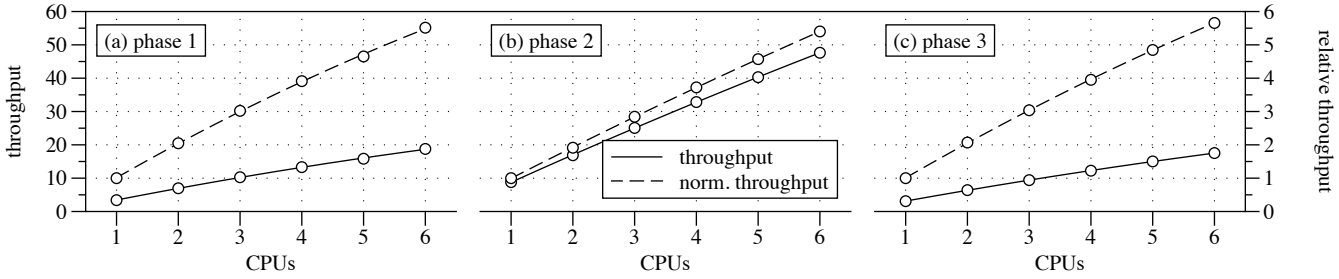


Figure 15: Scalability characteristics of the different execution phases of x264 from the PARSEC 2.1 benchmark suite.

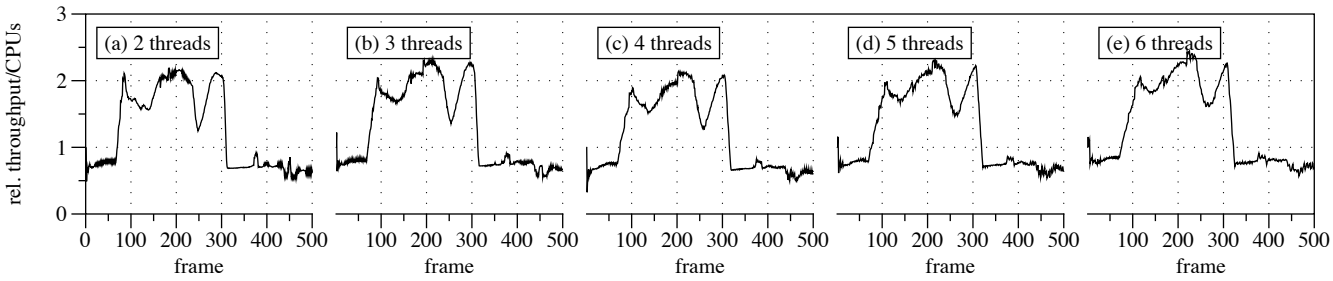


Figure 16: Relative ratio between the window throughput (100 ms) and the CPU allocation for different runs of x264 with 2 to 6 threads; the ratio remains roughly constant across CPU allocations, making it a robust metric to detect execution phases.

put measurements and requirements diverge, even though the "environment" did not change. It archives the previous the scalability characteristic and starts over as if the diverging application has just started. *Metronome++* addresses the second issue by weighing CPU allocations by means of a *workload predictor* to consider for changing execution phases.

The *x264* application in the PARSEC 2.1 benchmark suite [6] is a representative application presenting input-dependent execution phases. Figure 15a, b, and c show three execution phases, which *x264* goes through with the native input. The *x*-axes indicate the number *n* of allocated CPUs (in each experiment, we execute applications with *n* threads); *y*-axes indicate both the absolute (left) and relative (to the minimum, right) throughput measurements. Each data point is the average over experiments repeated until the width of the 95% confidence interval was below 1%. Experimental results were collected on the workstation described in Section 3.3.

The relative throughput measurements show that the scalability characteristics of *x264* do not change significantly across execution phases, thus leaving Equation (4) fitting stable. However, the absolute throughput measurements show a sharp performance increase in the second execution phase suggesting the presence of input-dependent execution phases.

To further analyze the execution phases of *x264*, we collect its frame-by-frame window (100 ms) throughput measurements; *HRM* provides such metric [28]. The window throughput metric is more sensitive (depending on the window length) to short-term trends than the global throughput metric and can highlight execution phase transitions. Figure 16 shows the frame-by-frame relative (to the initial) ratio between the window throughput measurements and the number of allocated CPUs for five different runs of *x264* from 2 to 6 threads. Experimental results were collected on the workstation described in Section 3.3.

The relative ratio increases sharply at the 70-th frame and decreases with similar intensity at the 300-th frame. The rising edge of the ratio indicates a transition from a high complexity subset to a low complexity subset of the dataset, while the falling edge represents the opposite. Since the characteristics of the ratio (i.e., "shape" and "intensity") are roughly the same across the different CPU allocations, this metric proves to be a good proxy for execution phase detection, regardless of the current allocation.

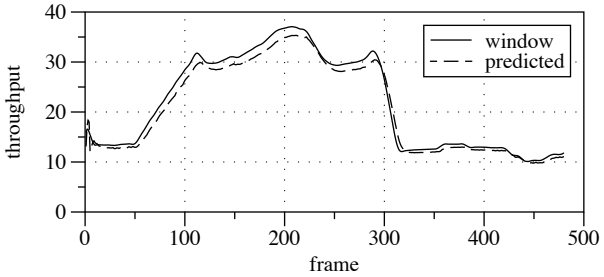


Figure 17: Window (1 s) predicted throughput for x264 running with 4 threads.

For this reason, *Metronome++* uses an exponential moving average[5] of the values of this relative ratio as a workload predictor to detect execution phase transitions. *Metronome++* weighs throughput requirements with the workload predictor to realize the second adaption level (i.e., execution phase adaptation); the result goes through the first adaptation level that instead leverages the scalability characteristic as described in Appendix B.

We conclude with the evaluation of the accuracy of the workload predictor for a run of *x264* with 4 threads. Figure 17 shows the frames on the *x*-axis and both the window (1 s) and the predicted throughput on the *y*-axis. Experimental results were collected on the workstation described in Section 3.3. The predicted throughput, which is computed multiplying the workload predictor with the output of the scalability characteristic of the first execution phase, tracks almost perfectly the window throughput proving the accuracy of our approach.

[5]The use of an exponential moving average helps smoothing the occasional noise in the window throughput.