A Performance-Aware Quality of Service-Driven Scheduler for Multicore Processors

Filippo Sironi, Donatella Sciuto, Marco D. Santambrogio
Politecnico di Milano
{filippo.sironi,donatella.sciuto,marco.santambrogio}@polimi.it

ABSTRACT

In the latest decade, the IT industry shifted from single to multicore processors. Multicore processors require better support from operating systems and runtimes to allow applications to achieve predictable performance and guarantee quality of service (QoS). Finding a proper schedule to yield the specified performance for single and multi-threaded applications can be cumbersome; dealing with multi-programmed workloads may be even worse.

We present a performance-aware QoS-driven scheduler for multicore processors, which exploits the availability of runtime application-specific performance measurements to determine a suitable allotment of cores for multi-programmed workloads so as to achieve the desired level of QoS. The proposed scheduler is meant to be implemented in user-mode and harnesses an auto-regressive moving average performance model to put in a relationship performance measurements and resource allocation and is capable of embodying applications' characteristics such as execution phases.

Categories and Subject Descriptors

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

Operating systems, Performance measurement, Performance modeling, Resource allocation

1. INTRODUCTION

Multicore processors are ubiquitous in desktops, servers, and embedded devices [1, 18, 20]. Computer architects designed multicore processors to overcome the limitations of superscalar processors (*e.g.,* poor performance per Watt ratios) whose performance stopped growing at historical rate.

This paradigm shift considerably increased the burden on systems' and applications' programmers. Nowadays, commodity operating systems schedulers fail in taking full advantage of multicore processors when scheduling multi-programmed workloads of multi-threaded applications [21]. They are oblivious of applications' characteristics (*e.g.,* execution phases) and the resulting resource allocation may lead to unpredictable performance [14]. Judicious management of on-chip shared resources is critical to deliver predictable performance and (if possible) guarantee QoS.

Characterizing applications by means of their instructions' throughput [3], miss rate curves [26], speedup, efficiency (*i.e.,* speedup over resource allocation) [10], etc. to overcome the inefficiencies of commodity schedulers is a widely accepted practice [9, 21]. Characterization can be performance either offline, online, or by means of a mix of both. Profiling applications offline using reference (*i.e.,* training) inputs may not uncover execution phases that depends on the input itself. Moreover, anticipating a significant set of environmental conditions (*i.e.,* number of different multi-programmed workloads) is nearly prohibitive and, if possible, is likely to be time-consuming. As an alternative, it is possible collecting information at runtime to infer applications' characteristics and harnessing feedback-based mechanisms.

We understand and appreciate the value of offline analysis that can uncover fine-grained information; however, we advocate online characterization is key to guarantee QoS. Moreover, we cannot expect applications' programmers or even users to employ machine-specific performance measurements like instructions per cycle (IPC) or last-level cache (LLC) miss rate. Instead, we claim application-specific performance measurements (*e.g.,* frames/s for a video encoder or decoder) can be as effective as low-level once when harnessing feedback-based mechanisms. In addition, they are meaningful for applications' programmers and users since they address the impedance-mismatch problem [8] by turning the resource allocation problem into a goal definition problem, which is later bound to resource allocation.

To this end, this paper presents a performance-aware QoS-driven scheduler for multicore processors running multi-programmed workloads and makes the following contributions:

- Introducing a simple yet effective user-mode performance monitoring infrastructure to instrument applications so as to make application-specific performance measurements system-wide accessible and allow users specifying QoS requirements;
- Exploiting a first order discrete-time auto-regressive moving-average (ARMA) performance model to establish the relationship between resource allocation and performance measurements;
- Harnessing online estimation to discover the performance model's parameters through a recursive least square (RLS) filter, thus uncovering applications' characteristics such as coarse-grain execution phases [24];

EWiLi '13, August 26-27, 2013, Toulouse, France.

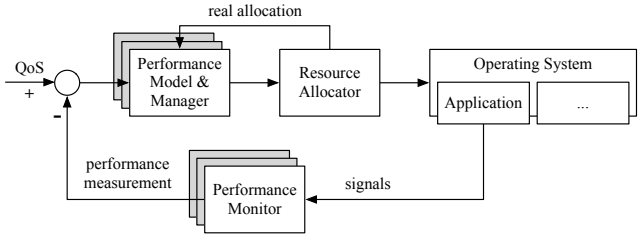


Figure 1: System architecture diagram. Each application is coupled with an instance of the performance monitor and one of the performance model and *performance manager*. A single *resource allocator* normalizes applications' resource demands.

- Implementing a *performance manager* leveraging a proportional-integral (PI) controller feeding resource demands to a fair *resource allocator*, which exploits well-established resource allocation mechanisms.

In the rest of this paper: Section 2 illustrates the design principles and gives development insights. Section 3 presents an experimental campaign showing the validity of the proposed approach. Section 4 goes through the related works and, finally, Section 5 concludes the paper.

2. DESIGN AND DEVELOPMENT

The scheduler proposed in this paper leverages a classic feedback-based structure consisting of three distinct phases respectively devoted to:

1. Monitor applications to gather performance measurements;
2. Evaluate the scheduler's policy devising a thread to core mapping so as to guarantee (if possible) QoS;
3. Apply the mapping migrating threads as needed.

These three phases construct a closed loop as depicted in Figure 1.

2.1 Performance Monitoring

The Heart Rate Monitor (HRM) [25] is an open source monitoring infrastructure that compute application-specific performance measurements on which users can specify QoS requirements through applications' instrumentation. Adaptation policies, which can be implemented in kernel [5, 25] and user-space [19], can exploit the availability of performance measurements and QoS requirements to affect applications' execution. HRM is tightly integrated with the Linux kernel since its primary goal was to export application-specific performance measurements to the kernel-space.

This paper proposes a more general approach leveraging a user-space scheduler that can run on top of most POSIX-compliant operating systems (*e.g.,* GNU/Linux, BSD, ...) with minimal changes. Due to this reason, we developed a portable *user-mode* performance monitoring infrastructure: *libthroughput* with performance comparable to that of HRM. *libthroughput* delivers competitive performance with respect to HRM and provides similar functionality: first, performance measurements for both single and multi-threaded applications and multi-programmed applications, second, QoS requirements specification.

libthroughput collects application-specific performance measurements representing applications' throughput. As an example, consider the *x264* video encoder, which implements the H.264/MPEG-4 Part 10 or Advanced Video Coding (AVC) standard, included in the PARSEC 2.1 benchmark suite [6]. The parallel algorithm of *x264* harnesses a virtual pipeline with one stage per frame. *x264* processes in parallel a number of pipeline stages equal to the number of encoder threads realizing a sliding window moving from the beginning to the end of the pipeline. Once the execution of a stage finishes the encoder thread handling the stage issues a *signal*, which in the context of *x264* signifies the completion of a frame. Users can specify QoS requirements through a meaningful performance measurement like frames/s.

2.2 Performance Modeling and Management

The proposed user-space scheduler leverages a first order discrete-time ARMA *performance model* that established the relationship between the resource allocation and performance measurements. Equation (1) reports the mathematical formulation where $r(k)$ and $r(k+1)$ are the performance measurements at the k-th and $(k+1)$-th control steps, respectively. $c(k)$ is the subset of cores allocated to the application. a and b are the performance model's parameters whose values depend on the application, the workload, and the system.

$$r(k+1) = a \cdot r(k) + b \cdot c(k) \quad (1)$$

The proposed user-space scheduler employs a recursive least squares (RLS) filter, which is a common choices among least squares and Kalman filters [22], to perform online estimation of the performance model's parameters. Online estimation allows the user-space scheduler to capture applications' characteristics such as execution phases, which changes the relationship between the resource allocation and performance measurements. Section 3.1 reports a thorough validation of the performance model.

Starting from the performance model reported in Equation (1) we devised a *performance manager* leveraging a PI controller that responds to errors (*i.e.,* difference between the QoS requirements and performance measurements) by means of two terms: a proportional and an integral term. The proportional term changes its effect according to the current value of the error and in a way that reduces the future values of the error. The integral term changes its effect incrementally accounting for the past values of the error. The choice of neglecting the derivative term, thus the use of a PID controller, translates into a little loss of control but, at the same time, leads to notable less noise.

Equation (2) reports the mathematical formulation of the PI controller for application i.

$$c_i(k) = c_i(k-1) + \frac{1-p}{b} \cdot e_i(k) - a \cdot \frac{1-p}{b} \cdot e_i(k-1) \quad (2)$$

$c_i(k)$ and $c_i(k-1)$ are the subset of cores required by the application between the k-th and the $(k+1)$-th control steps to respect the QoS requirement and the subset of cores allocated to the application between the $(k-1)$-th and the k-th control steps, respectively. $e_i(k)$ and $e_i(k-1)$ are the errors at the k-th and $(k-1)$-th control steps, respectively; the errors are computed as $\bar{r_i} - r_i(k)$ and $\bar{r_i} - r_i(k-1)$, where $\bar{r_i}$ is the QoS requirement.

We synthesized the PI controller by applying classical control theory techniques as explained by Levin [16] and constrained the transfer function to have a single pole in p. The controller's parameter p affects the responsiveness;

if the value is chosen in the interval $(-1, 1)$ the system is guaranteed to be stable as long as the performance model holds.[1] Furthermore, if the value is chosen in the interval $(0, 1)$ the system is guaranteed to avoid oscillations. In general, large values in the interval $(0, 1)$ translate into a slower but smoother response, while small values translate into a faster but rougher response.

2.3 Resource Allocation

Each *performance manager* i computes the subset of cores the application i requires to satisfy the QoS requirement. These computations are carried on independently, thus resulting in either a demand that can or cannot be satisfied by the number of cores available in the system.

Whenever the demand can be satisfied the user-space scheduler can exploit an energy-aware policy shutting down unused cores through clock and power gating.

On the other hand, if the demand cannot be satisfied, the *resource allocator* can harness many different policies. We propose re-distributing cores according to *performance managers*' demands following a performance-aware fair policy similar to one we proposed with Metronome [25]. Equation (3) reports the mathematical formulation of the re-distributing filter where $\bar{c}$ is number of cores available in the system and $\tilde{c}_i(k)$ is the proportional demand for application i at the k-th control step.

$$\tilde{c}_i(k) = c_i(k) \cdot \frac{\bar{c}}{\sum_j c_j(k)} \tag{3}$$

The *resource allocator* is also in charge of rounding the floating-point number of cores as needed and inform *performance managers* to avoid compromising the auto-tuning process. Moreover, the *resource allocator* ensures that applications receive at least one core at every control step maintaining the highly desirable non-starvation property of most commodity schedulers.

Alternative policies can employ weights to provide additional knobs and different service levels for different users.

2.4 Prototype for GNU/Linux

The overall design of the user-space scheduler is general enough to be implemented on top of most POSIX-compliant operating systems. Given our design, we developed a prototype on top of the GNU/Linux operating system and its commodity scheduler.

Both performance monitoring and resource allocation heavily relies on the infrastructure provided by GNU/Linux. *libthroughput* exploits *cgroups* [4] to inform the *resource allocator* about which threads belong to which application by creating a new cgroup for each application. The *resource allocator* maps threads belonging to an application to a subsets of cores by harnessing the *cpuset* subsystem of cgroups.

3. EVALUATION

We run all the experiments to evaluate the effectiveness of the performance-aware QoS-driven scheduler on a Dell Precision T3500 server with an Intel Xeon Processor W3670 and 12 GB of Single Ranked DIMMs. The processor features 6 cores clocked at 3.20 GHz and sharing 12 MB of last-level cache. Each memory module runs at 1066 MHz. We disabled the Enhanced Intel SpeedStep Technology, the Intel TurboBoost Technology, and the Intel Hyper-Threading Technology to simplify our analysis. The operating system is Debian GNU/Linux 7.0 x86-64 with the Linux kernel 3.2.

[1]The use of adaptive control through the RLS filter enforces the performance model.

Table 1: Performance model assessment through the average and the standard deviation of the coefficient of determination and the mean absolute percentage error over six static resource allocations

application	R^2 $[0, 1]$		MAPE [%]	
	avg.	std dev.	avg.	std dev.
blackscholes	0.97	0.01	0.63	0.04
bodytrack	0.86	0.02	1.10	0.05
canneal	0.96	0.01	0.95	0.04
dedup	0.73	0.02	3.38	0.17
facesim	0.89	0.02	1.03	0.04
ferret	0.92	0.02	0.83	0.03
swaptions	0.98	0.01	0.51	0.02
x264	0.72	0.03	5.48	0.26

3.1 Performance Model Validation

We evaluate the effectiveness of the performance model against a subset of the applications from the PARSEC 2.1 benchmark suite [6] instrumented with *libthroughput*. We run each application 100 times randomly varying the subset of cores allocated and collecting performance measurements through *libthroughput*. We applied the least squares algorithm to regress the performance model's parameters.

We then run each application fixing the subset of cores allocated (from 1 to 6) and computed the coefficient of determination R^2 and the mean absolute percentage error (MAPE) as suggested by Sharifi et al. [22] using the application-specific performance measurements provided by *libthroughput* instead of the IPC.

Table 1 reports the average and the standard deviation of R^2 and MAPE for the applications we analyzed. The averages of the first metric, R^2, is close to 1, which means the performance model is quite accurate; the small standard deviations say the averages holds for most of the applications apart from *dedup* and *x264*. These two applications go through different execution phases, thus benefiting from the online estimation of the performance model's parameters that is not employed for performance model assessment. The averages and standard deviations of the second metric, MAPE, leads to similar considerations with the addition of quantitative information on the percentage error.

The rest of this section focuses on the experiments with two instances of *x264*. Among the subset of the PARSEC 2.1 benchmark suite we employed, *x264* is the most challenging application and provides the opportunity to evaluate complex scenarios, especially when two instances run at the same time.[2]

3.2 Static Resource Allocation: Comparison

We compare the proposed user-space scheduler with two standard (static) resource allocation mechanisms provided by the cgroups subsystem of the Linux kernel: cpuset and bandwidth. cpuset allows a group of threads to use a subset

[2]We show only the experimental results obtained with *x264* for space constraints. Moreover, we limited our study to workloads made up of two applications since our evaluation platform cannot afford running more applications with reasonable performance.

of the available cores. The cpuset subsystem is an example of spatial scheduling solution while the bandwidth is a more classical time-share scheduling approach. The bandwidth subsystem enforces the reservation of a certain quota of the multicore processor computational power over a period of time for a cgroup.

Figures 2a and 2b show the behavior of the first and second instances of *x264*, respectively, with the best static resource allocation to respect the QoS requirements of 8 and 12 frames/s or fps through the cpuset subsystem. The first instance is assigned 2 cores (*i.e.,* 0 and 1) while the second instance gets 3 cores (*i.e.,* 2-4). With these static resource allocations both the applications satisfy the QoS requirements at the end of the execution; however, during the execution the performance measurements vary a lot between 6 and 13 fps for the first instance and between 10 and 18 fps for the second instance. This is due to the different execution phases, which for *x264* are input-dependent, the application goes through. Ideally, one would want to keep the performance measurements as close as possible to the QoS requirements during the whole execution to avoid wasting resources. The take out of this experiment is that: there exists no static resource allocation that can be achieved through the cpuset subsystem such that both the instances of *x264* respect the QoS requirements during the whole execution.

Figures 2c and 2d display the experimental results obtained by replicating the previous experiments exploiting the bandwidth subsystem. With the bandwidth subsystem, applications make use of all the cores available, which are six on our evaluation platform. The first instance of *x264* is assigned 33 % of the bandwidth of the multicore processors, which is approximately equal to the 2 cores assigned in the previous experiment. The second instance gets 47 % of the bandwidth of the multicore processors, which is, once again, approximately equal to the 3 cores assigned in the previous experiment. Even though the bandwidth subsystem is much finer-grained than the cpuset subsystem, the take out of this experiment is the same of the first. It is worth noting that coupling the cpuset and bandwidth subsystems to perform static resource allocation yields equally unsatisfactory experimental results.

3.3 Dynamic Resource Allocation

We evaluate the proposed performance-aware QoS-driven scheduler for multicore processors in the same scenario of the static resource allocation, with the same multi-programmed workload and the same QoS requirements. *performance managers* run every 50 ms and update the performance model at the same frequency by retrieving performance measurements through *libthroughput*, which is capable of computing fresh information every 10 ms. *performance managers* demand new resource allocations and coordinate through the *resource allocator* every 500 ms. Each of these periods is configurable and different configuration benefit different applications depending on the execution phases they may go through.

Figures 2e and 2f show the experimental results obtained by running the two instances of *x264* with QoS requirements of 8 and 12 fps, respectively. Both the performance profiles track the QoS requirements after an initial settling phase in which the performance model's parameters converge to their "real" values. Figures 2g and 2h display the subset of the available cores assigned to the first and to the second instance of *x264*, respectively. The resource allocations follow the performance profile of the application, decreasing the number of cores when the performance measurements naturally rise because of the lower complexity of the video and increasing the number of cores when the performance measurements fall due to the higher complexity of the video. Moreover, it is important to notice how the sum of the number of cores assigned to the first and second instances of *x264* never exceeds number of available cores thanks to the re-distributing filter inside the *resource allocator*.

3.4 Discussion

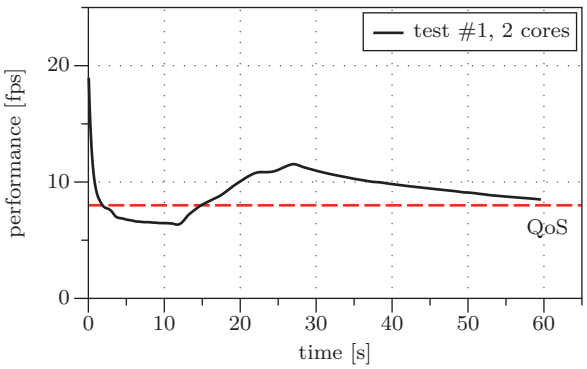
The choice of exploiting the cpuset subsystem instead of the bandwidth subsystem is actually sub-optimal for the proposed approach for two reasons: first, the cpuset subsystem is coarser-grained than the bandwidth subsystem and, second, the cpuset subsystem requires, like the bandwidth subsystem, running each multi-threaded application with a number of threads which is at least as high as the number of available cores. The first drawback is easy to understand since the cpuset subsystem allows partitioning the bandwidth of the hexa-core processors by multiple of $\approx$ 16 %. This issue is simply addressed by increasing the decision/actuation frequency of the *performance managers* and *resource allocator*. The cpuset subsystem enables resource allocation on space axis; variable dynamic resource allocation either requires multi-threaded applications to vary the number of threads accordingly or to run with a number of threads that allows exploiting the full parallelism of the multicore processor. Conversely, the bandwidth subsystem always requires multi-threaded applications to run with an adequate number of threads since resource allocation is performed on the time axis. With the cpuset subsystem multi-threaded applications may end in unbalanced configurations where some cores must handle more threads than others possibly introducing artificial critical paths due to synchronization issues. This is an issue we observe with the proposed approach and it may require the adoption of advance load balancing scheme such as Juggle [12].

4. RELATED WORK

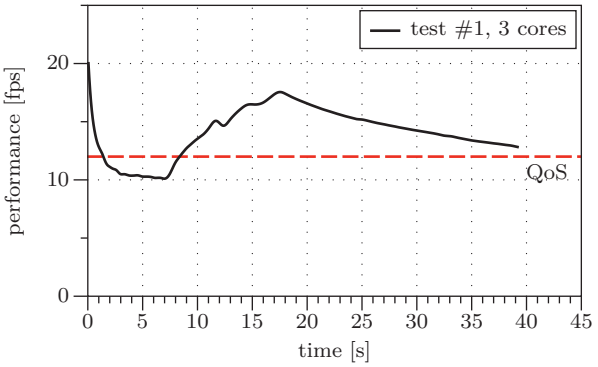
Recently there has been extensive research on solutions to maximize performance and/or respect QoS requirements.

Researchers focused on cache partitioning approaches [13, 23], memory bandwidth partitioning solutions [17], cores partitioning algorithms [9, 21], and both time and space-sharing approaches [5, 25]. Moreover, these works exploited different decision-making techniques spanning from heuristics [5, 9, 21, 25] to machine learning [7, 19], control theory [22], a mix of them [11].

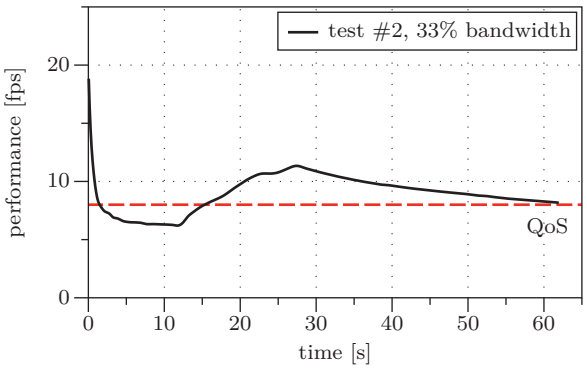
The proposed approach borrows the design of HRM from Metronome [25] to leverage application-specific performance measurements and explores the use of a PI controller instead of the speedup-based approach from Metronome++ [5]. Like METE [22], our user-space scheduler employs an ARMA performance model coupled with a PI controller to implement the *performance manager*. METE handles multiple resources (*i.e.,* cores, cache ways, and memory bandwidth) while the proposed approach partitions a single resource (*i.e.,* cores); our limitation is due to the fact that we implemented the proposed approach on real hardware, while METE cannot be implemented since cache ways and memory bandwidth partitioning is not supported by any com-



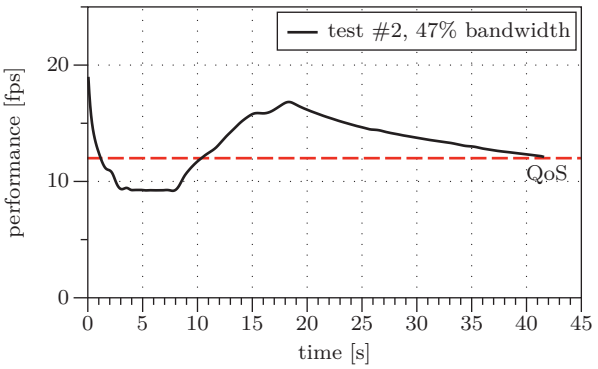
(a) Performance profile of the first instance running with 2 cores.



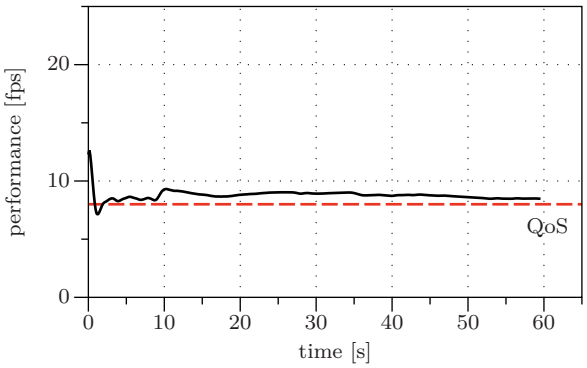
(b) Performance profile of the second instance running with 3 cores.



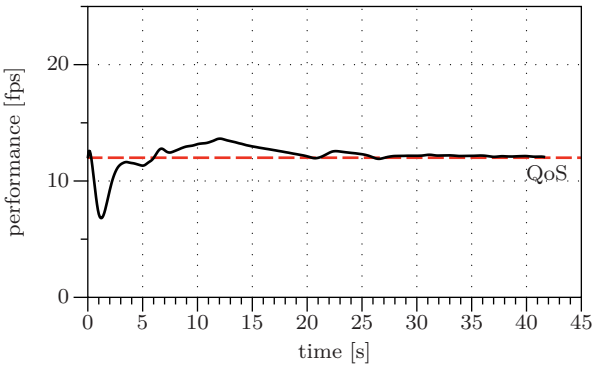
(c) Performance profile of the first instance running with 33 % of the bandwidth.



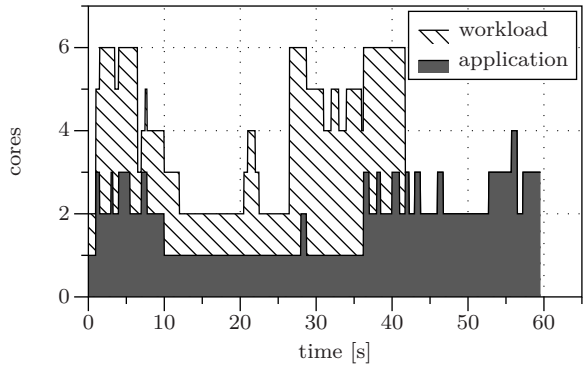
(d) Performance profile of the second instance running with 47 % of the bandwidth.



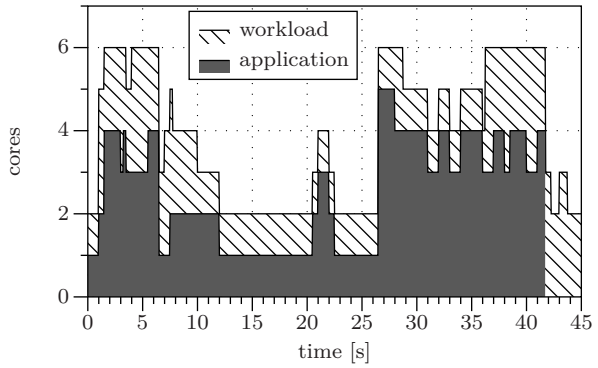
(e) Performance profile of the first instance meeting the 8 fps requirement.



(f) Performance profile of the second instance meeting the 12 fps requirement.



(g) Core allocation for the first instance while meeting the 8 fps requirement.



(h) Core allocation for the second instance while meeting the 12 fps requirement.

Figure 2: Performance profiles of the two instances of *x264* running simultaneously with various static resource allocations obtained through the cpuset and bandwidth subsystems, and performance profiles of the two instances of *x264* running simultaneously with the proposed user-space scheduler.

modity multicore processor.

Orthogonal approaches [15] dynamically adjust the number of threads within multi-threaded applications to optimize the overall efficiency of the system or proactively addresses the load balancing issue [12]. Coupling the proposed approach with these approaches may solve the second issue discussed in Section 3.4. The availability of Scheduler Activations [2] can improve the efficiency of the proposed approach avoiding costly kernel-space thread migrations in favor of user-space thread migrations.

The proposed approach might recall a real-time scheduling infrastructure. However, we believe the two scheduling solutions are different in some of the key aspects. Let us focus on priority-based real-time scheduling infrastructures that are the one resembling more the proposed scheduling approach. The earliest deadline first (EDF) is a priority-based scheduling algorithm; with EDF an application i specifies a relative deadline D_i and a worst-case execution time C_i. Applications programmers may need to overestimate C_i to account of workload variations among the deadlines. The overestimation may in turn lead to a waste of resources (*e.g.,* the scheduling infrastructure implements a hard admission control policy and no application can exploit unused resources). The proposed approach borrows ideas from adaptive systems and accounts for application-specific performance measurements and QoS requirements to gracefully adapt resource allocation shifting from resource-centric solutions like real-time scheduling infrastructures to a goal-oriented one.

5. CONCLUSIONS

We presented a performance-aware QoS-driven scheduler for multicore processors and multi-programmed workloads that sits on top of well-established resource allocation mechanisms, namely the cgroups subsystem of the Linux kernel. The proposed approach harnesses application-specific performance measurements and QoS requirements provided through *libthroughput*, the user-space dual of HRM to address the impedance-mismatch problem and turn the resource allocation problem into a goal-definition problem. In addition, the proposed approach leverages a discrete-time ARMA performance model and a RLS filter to dynamically establish the relationship between the resource allocation and the performance measurements. A set of PI controllers determine suitable allotments of cores so as applications can respect (if possible) QoS requirements. Experimental results on a commodity multicore processor with emerging real-world applications highlight the effectiveness of the proposed approach that allows applications respecting QoS requirements even in presence of execution phases.

6. REFERENCES

[1] Read how NVIDIA Tegra is redefining mobile performance. URL `http://www.nvidia.com/object/white-papers.html`.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Trans. Comput. Syst.*, volume 10, 1992.

[3] R. Azimi, M. Stumm, and R. W. Wisniewski. Online Performance Analysis by Statistical Sampling of Microprocessor Performance Counters. In *ICS*, 2005.

[4] G. Banga, P. Druschel, and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*, 1999.

[5] D. B. Bartolini, R. Cattaneo, G. C. Durelli, M. Maggio, M. D. Santambrogio, and F. Sironi. The Autonomic Operating System Research Project – Achievements and Future Directions. In *DAC*, 2013.

[6] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[7] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *MICRO*, 2008.

[8] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz. Tessellation: Refactoring the OS around Explicit Resource Containers with Continuous Adaptation. In *DAC*, 2013.

[9] J. Corbalan, X. Martorell, and J. Labarta. Performance-Driven Processor Allocation. In *OSDI*, 2000.

[10] D. L. Eager, J. Zahorjan, and E. D. Lozowska. Speedup Versus Efficiency in Parallel Systems. In *IEEE Trans. Comput.*, volume 38, 1989.

[11] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. E. Miller, S. M. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. P. Chandrakasan, and S. Devadas. Self-aware Computing in the Angstrom Processor. In *DAC*, 2012.

[12] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz. Juggle: Proactive Load Balancing on Multicore Computers. In *HPDC*, 2011.

[13] M. Kandemir, T. Yemliha, and E. Kultursay. A Helper Thread Based Dynamic Cache Partitioning Scheme for Multithreaded Applications. In *DAC*, 2011.

[14] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. In *IEEE Micro*, volume 28, 2008.

[15] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *ISCA*, 2010.

[16] W. S. Levin. *The Control Handbook*. CRC-Press, 1996.

[17] F. Liu and Y. Solihin. Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors. In *SIGMETRICS*, 2011.

[18] H. McIntyre, S. Arekapudi, E. Busta, T. Fischer, M. Golden, A. Horiuchi, T. Meneghini, S. Naffziger, and J. Vinh. Design of the Two-Core x86-64 AMD "Bulldozer" Module in 32 nm SOI CMOS. In *IEEE J. Solid-State Circuits*, volume 47, 2012.

[19] J. Panerati, F. Sironi, M. Carminati, M. Maggio, G. Beltrame, P. J. Gmytrasiewicz, D. Sciuto, and M. D. Santambrogio. On Self-adaptive Resource Allocation through Reinforcement Learning. In *AHS*, 2013.

[20] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora. A 45 nm 8-Core Enterprise Xeon Processor. In *ISSSC*, 2009.

[21] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-Based Manycore Partitioning. In *PACT*, 2012.

[22] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. METE: Meeting End-to-End QoS in Multicores through System-Wide Resource Management. In *SIGMETRICS*, 2011.

[23] A. Sharifi, S. Srikantaiah, M. Kandemir, and M. J. Irwin. Courteous Cache Sharing: Being Nice to Others in Capacity Management. In *DAC*, 2012.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS*, 2002.

[25] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. D. Santambrogio. Metronome: Operating System Level Performance Management via Self-Adaptive Computing. In *DAC*, 2012.

[26] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *ASPLOS*, 2009.