AN INFRASTRUCTURE TO INSTRUMENT APPLICATIONS AND MEASURE PERFORMANCE IN SELF-ADAPTIVE COMPUTING

Davide B. Bartolini [†], *Filippo Sironi* [†‡], *Donatella Sciuto* [†], *Marco D. Santambrogio* [†‡]

[†] Dipartimento di Elettronica e Informazione, Politecnico di Milano

[‡] Computer Science and Artificial Intelligence Laboratory, Massachussets Institute of Technology

{bartolini,sironi,sciuto,santambrogio}@elet.polimi.it, {sironi,santa}@csail.mit.edu

ABSTRACT

The shift of mainstream computing architectures to the parallel paradigm, together with the increasing demand for functional and non-functional requirements for modern applications result in a heavy burden for developers and administrators when trying to design and tune computing systems. One of the possibilities for lightening this burden comes from research on runtime self-management and adaptation, which aims at automatizing the runtime management of computing infrastructures. The availability of accurate and appropriate system status information (i.e., self-awareness) is crucial for self-adaptive systems to be functional and useful, and can be achieved through runtime measures provided by *monitors*.

This paper illustrates recent advances in the development of an infrastructure for monitoring applications' throughput called *Heart Rate Monitor* (HRM). Its design and structure are illustrated and a showcase of its capabilities is provided. HRM introduces novel features for a runtime monitor, allowing versatile instrumentation and exposing rich runtime information. These characteristics make HRM an enabling technology for advanced adaptation techniques.

1. INTRODCUTION

The turn of computer architectures from the well understood, single-core structure to multiple (possibly heterogeneous) processing elements is pervasive. This change has been dictated by physical (i.e., inability to increase the clock frequency) and architectural (i.e., diminishing performance returns from efforts in further optimizing the individual processors' internal architecture) constraints [1]. To survive its commitment to exponential performance improvements, the computer industry changed its strategy, leading to the multicore era. However, new issues are arising, as modern multicore processors hit the power wall, being thus constrained to make use of (i.e., switch) only a part of their transistors at the same time and leading to the phenomenon called *dark silicon* [2].

In the single-core era, faster processors provided software performance improvements and applications experienced the so-called "free lunch", with free-of-charge speedups just by switching to the next-generation CPU. The new parallel course in computer architectures, despite being due to architectural causes, carries the side effect of ending the "free lunch", posing a considerable burden of improving performance on software developers. The demands for efficient and reliable parallel software sums up to the already considerable bulk of expertise software developers need to successfully cope with requirements for computing performance, functionality, reliability, and constraints satisfaction due to today's IT. Moreover, computational resources must be carefully managed to avoid hitting power and thermal limits, while respecting Service Lever Agreements (SLAs). This situation leads to an increased need of pushing as much of the system management as possible into computing systems themselves, making autonomic computing a possible breakthrough for IT success [3].

Respecting SLAs employing the least amount of resources is one of the goals of *Autonomic Computing* [3]. Such systems are required to monitor themselves and the environment, detect significant changes, decide a chain of actions, and actuate them [4]. The activity of gathering runtime information (referred to as either *observe* or *monitor* phase) is crucial, and the availability of accurate and appropriate status information can determine the efficacy of the system.

This paper focuses on this phase, presenting an active monitoring [5] infrastructure to observe applications' throughput: the *Heart Rate Monitor* (HRM). HRM is designed to be versatile and provide rich information accounting for simplicity, usability, and functionality. HRM has been successfully employed as a building block in a previous work: the *Metronome* [6] framework, demonstrating its utility in gathering relevant runtime information used to provide performance-awareness in an experimental autonomic operating system. The present paper shifts the focus from the actuation phase to the observation phase and on HRM, providing more details regarding its design principles and implementation, describing novel features which were missing in early revisions of HRM.

2. DESIGN AND IMPLEMENTATION

Throughput is one of the most used metrics for characterizing applications' performance. For instance, the performance of a web server can be characterized in terms of requests served within each time unit (i.e. $\frac{requests}{second}$) while a video encoder can express its performance using the encoding frame rate (i.e., $\frac{frames}{second}$). Being able to access accu-

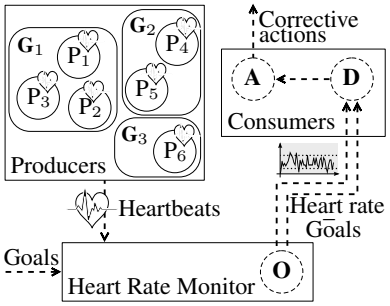


Fig. 1. Black box view of the Heart Rate Monitor. The inputs are heartbeats emitted by instrumented *producers*, organized in *groups*, and goals set by the users. HRM outputs heart rate measures to *consumers*, creating an ODA adaptation loop.

rate and comprehensive information about the throughput of mission-critical applications and to set meaningful performance goals in terms of high-level well understood metrics can enable the system to enact adaptation of resources allocation in order to match SLAs. HRM is designed exactly for this reason: it lets software developers instrument the resource-demanding section (called the *kernel*, or *hotspot*) of the application to emit a *heartbeat* per unit of work done and provides throughput measures in terms of a *heart rate* [6]. Moreover, HRM allows expressing goals in terms of a min–max heart rate window, which directly maps to an application-specific goal.

The advantages of HRM compared to similar solutions (e.g., Application Heartbeats [7, 8]) lie in functionality and efficiency. HRM is functionally superior since it grants both the user and the kernel-space the permission to access information (supporting both the Linux kernel [9] and the FreeBSD operating system [10]). Moreover, HRM supports any kind of parallelization model (i.e., multi-threading and multi-processing, spawning/waiting, pooling, pipelining, etc.). In terms of efficiency, HRM is very low-overhead thanks to its distributed and asynchronous design. In addition, HRM has been recently extended, without sensibly increasing its overhead, with the ability to provide heart rate measurements on multiple windows at the same time. The availability of such information poses a challenge for research, calling for more intelligent adaptation policies able to understand the correct time scale to consider and to take proactive actions to match applications' goals.

2.1. Black Box View

We can consider HRM as a black box implementing a producer/consumer model similar to the one employed by the *Performance and Environment Monitoring* (PEM) [11]: *producers* emit *heartbeats* for signaling progress and *consumers* access the *heart rates* computed by the monitor. HRM acts as an interface, collecting heartbeats, transforming them in throughput measurements, and making them available, realizing the *observation* phase of the *Observe, Decide, Act* (ODA) adaptation loop. Figure 1 represents this black box view, highlighting the flow of information from producers to consumers through, going through HRM, which enables the realization of the ODA adaptation loop.

To provide flexibility and be useful in current and future parallel systems, HRM must support monitoring any kind of parallel workload (i.e., multi-threaded, multi-processed, or any feasible mix of the two); this is attained by defining *monitoring groups* (marked as $\mathbf{G}_i$ in Figure 1). A *group* is a set of tasks cooperating for a certain activity (e.g., encoding video frames) and it constitutes the atomic monitoring entity.

Throughput measurements are computed as heart rates, i.e., for each group, the summation of the heartbeats emitted by all the producers over the elapsed time. Clearly, for such a measurements, the considered time horizon matters: considering the whole execution time provides a smoothed average, while considering a shorter time span discards the old history and allows to better highlight short-term trends. For this reason, HRM provides both a *global* and a *window* heart rate, allowing tuning the focus on longer or shorter-term trends as required by the specific monitoring context. Moreover, several measures on windows of different size (i.e. moving averages on different horizons) can be accessed at the same time to highlight different trends and providing richer information to consumers.

HRM allows for a simple yet general way of setting a desired value for the heart rate of a group through two parameters: a *minimum* and a *maximum* heart rate, defining the desired throughput range; moreover, it is possible to tie the goal to a specific window heart rate. For instance, dealing with a video encoder, the minimum heart rate could be set to the minimum frame rate to guarantee the desired QoS (e.g., 30 $\frac{frames}{second}$), the maximum could be set to a value over which no sensible benefit would be achieved and the goal could be tied to a certain time horizon according to how much buffering space is available for the encoded video.

Interaction with HRM is provided through a simple API implemented by *libhrm*: instrumenting an application needs as little as adding a couple of calls for attaching to a group and emitting heartbeats. Consumers are provided with a simple and powerful API, which was extended to support the new features. Details on the API are skipped here due to space constraints.

2.2. Under the Hood

The implementation of HRM has been partitioned between user and kernel-space. Partitioning the implementation lowers the overhead due to heartbeats emission while allowing the design of both user and kernel-space adaptation policies (i.e., consumers). The user-space partition is essentially an implementation of *libhrm*. The management logic, which handles grouping and logging, is implemented in kernel-space. Communication among different address spaces is

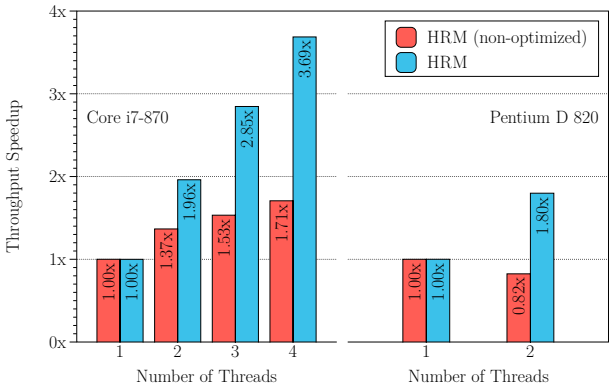


Fig. 2. Throughput speedup in emitting heartbeats when scaling the number of concurrent producers per group.

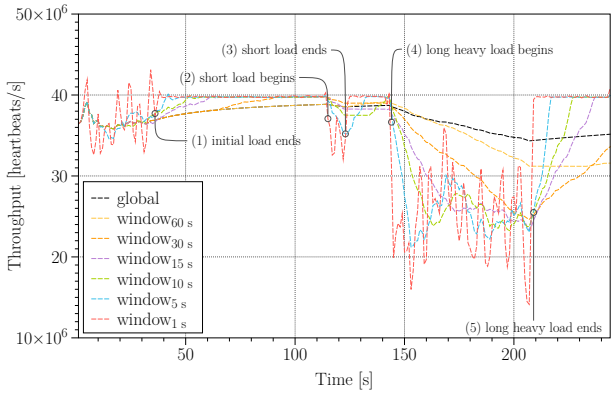


Fig. 3. Global and six different window heart rates of an ad-hoc application showing different performance trends.

enabled through shared memory, which grants low-overhead accesses from each side. For each group, the kernel allocates memory to store the information; memory segments are mapped in the address space of producers and user-space consumers upon group attachment. This way, all the entities of the group can access (with proper read/write privileges) the information. Sharing memory among different address spaces (and even within the same address space) is a delicate practice. Requiring carefully laid out data structure to achieve high efficiency. Since the most frequent operation within a group is emitting a heartbeat, the associated code path must be thoroughly optimized. This is done with a mapreduce-like approach, decoupling heartbeats emission and data computation (i.e., heart rates). Each producer receives counter within group memory to store the amount of heartbeats generated. Heartbeats emission becomes as quick as the increment of an atomic integer. Snapshots of the emitted heartbeat counts for all the active windows are periodically (with tunable period, defaulting at $100ms$) made available to allow on-demand heart rates computations. This way, heart rates are represented as floating point numbers in user-space and integer numbers in kernel-space, where floating point computation is discouraged. When a consumer asks for the global heart rate, such measurement is computed according to Equation 1. The window heart rate is otherwise computed according to Equation 2.

$$ghr_g(t) = \frac{\sum_i cnt_i(t)}{t - t_0} \quad (1)$$

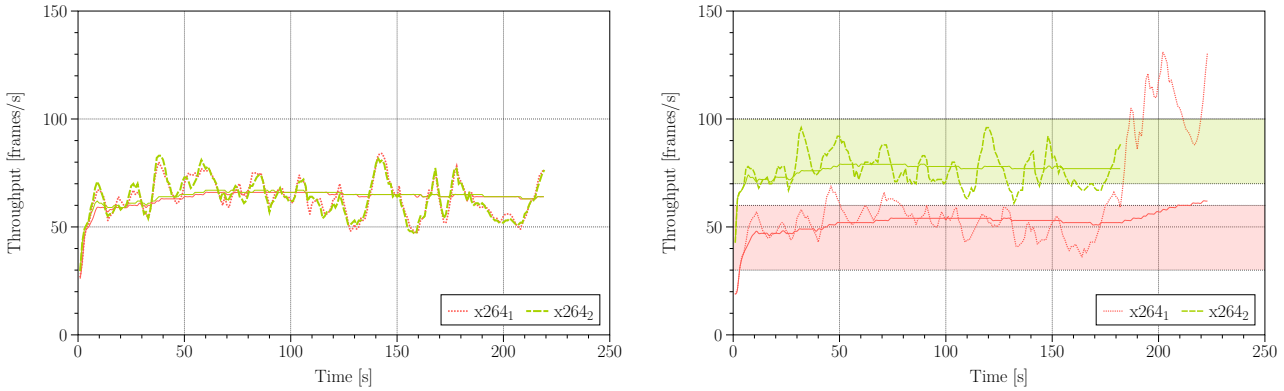
$$whr_g(t) = \frac{\sum_i cnt_i(t) - cnt_i(t - t_w)}{t - t_w} \quad (2)$$

In the formulae, t indicates the current timestamp, t_0 is the time at which the group was created, cnt_i are the counters associated with each of the group's producers, and t_w indicates the timestamp at the beginning of the window. To compute the window heart rates, HRM uses a circular buffer to store, at each accounting period, a snapshot of the current overall heartbeats count for the group and the timestamp.

This approach requires a careful implementation to avoid pitfalls resulting in poor performance. The memory location of each counter must be cache line-aligned to avoid false sharing, which would cause useless cache coherency traffic [12]. Figure 2 shows the speedup on throughput that can be achieved going scaling the number of producers emitting heartbeats in a tight loop for the same group. The test compares the optimized (final) revision with the non-optimized (i.e., non cache-friendly) revision of HRM on different processors. On the left side, the test is run with one to four concurrent producers executing on a quad-core Intel Core i7-870 processor with the Intel Hyper-Threading Technology disabled; both the non-optimized revision and the optimized revision of HRM scale. However, the latter scales almost linearly with the number of producers since it avoids false sharing. The On the right side, the same test is run with one to two concurrent producers executing on a dual-core Intel Pentium D 820 processor; the optimized revision of HRM scales almost linearly while the non-optimized revision of HRM shows a slow down when two producers emit heartbeats together for the same group. The slow down is due to the false sharing problem, which causes a notable performance decrease due to the off-chip (i.e., through the northbridge), inefficient cache coherency protocol of the Intel Pentium D processor.

3. SHOWCASE AND CASE STUDY

HRM has been employed within the *Metronome* [6] framework to measure applications' throughput, information that is later used to adapt process scheduling. Previous work proved the *efficiency* of HRM compared to the reference implementation of Application Heartbeats [13]. This is an incremental work extending the *functionality* of HRM with multiple window heart rates, a novel feature not yet adopted in similar contexts. Figure 3 presents a showcase of this capability: a 4-threaded microbenchmark is run to emit heartbeats as fast as possible on a quad-core Intel Core-i7 870 processor while many workloads differing in both duration and intensity run simultaneously. CPU-bounded workloads are simulated through the *cpuburn* utility. HRM is used with the multi-window capability to highlight performance



(a) Unmanaged instances of x264.

(b) Managed instances of x264; goals set at $30-60$ and $70-100\,\frac{frames}{s}$.

Fig. 4. Global and window heart rates for the x264 instances scheduled by the CFS (a) and by the adaptive scheduler (b).

trends and hence workloads' phases. In absence of additional workloads, the microbenchmark peaks at about $40 \times 10^6\,\frac{heartbeats}{s}$. The traces on the plot track the global heart rate and six different moving averages of size $\{1, 5, 10, 15, 30, 60\}s$[1]. The execution presents six phases: initially, up to the point marked (1), there is a light additional load which then terminates, letting the benchmark reach its peak performance up to point (2), when another external load is started. At point (3) the second load terminates and the microbenchmark goes back to its peak throughput but, at point (4), a heavier and longer-lasting load is applied up to point (5). It can be noticed from the plot how measurements on different time horizons highlight different trends: short windows give a prompt feedback when changes happen; however, they tend to be noisy when the execution is regular.

3.1. Adaptive Performance-Aware Scheduling

We evaluated the enhanced revision of HRM within the *Metronome* [6] framework. HRM monitors two 4-threaded instances of the *x264* application [14] encoding the *Big Buck Bunny* full HD movie [15]. The test platform is a quad-core Intel Core-i7 870 processor with the Intel Hyper-Threading Technology disabled running a modified version of Debian GNU/Linux [9]. The window size has been empirically set to $5s$ after an experimental evaluation using the multi-window capability of the latest revision of HRM. Figure 4 shows the results of this experiment: since the two instances are exactly the same, they have almost overlapping performance when scheduled by the Completely Fair Scheduler (CFS), as shown in Figure 4(a). The experiment consists in setting two different high-level performance goals for the two instances and let the adaptation policy implemented within the *Metronome* framework dynamically allocate processor time to match the performance goals. Figure 4(b) shows the managed case: the performance goals (i.e., the red and green shaded areas), are $30-60\,\frac{frames}{second}$ and $70-100\,\frac{frames}{second}$ and the two instances of *x264* are driven towards meeting their SLAs. The throughput of the slower application receives a sudden speedup when the other terminates, since it is now the only application in execution and the maximum heart rate is considered as a *soft bound* on the QoS, and not as a performance cap.

[1]Note that when there is not enough data to compute a window heart rate over its full size, the measure is still provided using the available data.

4. CONCLUSIONS

HRM proved to be a flexible, efficient, and scalable throughput monitor and was employed for realizing adaptive computing. This paper offers a detailed description of the careful design of HRM, which allows to provide very small overhead, and provides a showcase of a novel feature: the availability of measurements on multiple tunable moving averages. We believe that this feature could be exploited by smarter adaptation policies able to leverage this richer status information in order to take more effective adaptation decisions.

5. REFERENCES

[1] S. H. Fuller *et al.*, "Computing Performance: Game Over or Next Level?" *Computer*, vol. 44, no. 1, 2011.

[2] H. Esmaeilzadeh, *et al.*, "Dark silicon and the end of multicore scaling," in *Proc. of ISCA*, 2011.

[3] J. O. Kephart *et al.*, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003.

[4] M. Salehie *et al.*, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, 2009.

[5] M. C. Huebscher *et al.*, "A survey of Autonomic Computing: Degrees, Models, and Applications," *ACM Comput. Surv.*, vol. 40, no. 3, 2008.

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

[7] H. Hoffmann, *et al.*, "Application Heartbeats for Software Performance and Health," in *Proc. of PPoPP*, 2010.

[8] ——, "Application Heartbeats: a Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *Proc. of the ICAC*, 2010.

[9] "The Linux Kernel," http://www.kernel.org/.

[10] "The FreeBSD Project," http://www.freebsd.org/.

[11] C. Cascaval, *et al.*, "Performance and Environment Monitoring for Continuous Program Optimization," *IBM J. Res. Dev.*, vol. 50, no. 2/3, 2006.

[12] M. M. K. Martin, *et al.*, "Why On-Chip Cache Coherence is Here to Stay," *Commun. ACM*, vol. 55, no. 7, 2012.

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

[14] "x264 - H.264/ACV encoder," http://www.videolan.org/developers/x264.html.

[15] "Big Buck Bunny," http://www.bigbuckbunny.org/.