View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Politecnico di Milano

Application Heartbeats

A Generic Interface for Expressing Performance Goals and Progress in Self-Tuning Systems

Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal

Massachusetts Institute of Technology
{hank,eastepjm,santa,jasonm,agarwal}@csail.mit.edu

Abstract. Self-tuning, self-aware, or adaptive computing has been proposed as one method to help application programmers confront the growing complexity of multicore software development. Such systems have been proposed for architectures, compilers, and operating systems to ease the application programmer's burden by providing services that automatically customize to meet the needs of the application. However, these systems often rely on ad hoc methods for understanding and monitoring an application and thus struggle to incorporate the true performance goals of the applications they are designed to support. This paper presents the Application Heartbeats API which addresses the need to provide a standardized interface for applications to communicate with supportive adaptive systems. The Application Heartbeats framework provides a simple, standard programming interface that applications can use to indicate their performance and which system software can use to query that performance. Several experiments demonstrate the simplicity and efficacy of the Application Heartbeat approach.

1 Introduction

As multicore processors become increasingly prevalent, system complexities are skyrocketing. It is no longer practical for an average programmer to balance all of the system constraints and produce an application that performs well on a variety of machines, in a variety of situations. One approach to simplifying the programmer's task is the use of *self-tuning*, or *adpative* hardware and software. Self-tuning systems take some of the burden off of programmers by monitoring themselves and optimizing or adapting as necessary to meet their goals.

As described in [1], adaptive systems must be able to *monitor* their environment as well as *detect* significant changes. Despite this need, there is no standardized, general approach for applications and systems to measure how well they are meeting their goals. Existing approaches are largely ad hoc: either handcrafted for a particular system or reliant on architecture-specific performance counters. Not only are these approaches fragile and unlikely to be portable to other systems, they frequently do not capture the actual goal of the application. For example, measuring the number of instructions executed in a period

of time does not tell you whether those instructions were doing useful work or spinning on a lock; reliance on CPU utilization or cache miss rates has similar drawbacks. The problem with mechanisms such as performance counters is that they attempt to infer high-level application performance from low-level machine performance. What is needed is a portable, universal method of monitoring an application's actual progress towards its goals.

This paper introduces a software framework called *Application Heartbeats* (or just *Heartbeats* for short) that provides a simple, standardized way for applications to monitor their performance and make that information available to external observers. The framework allows programmers to express their application's goals and the progress that it is making using a simple API. As shown in Figure 1, this progress can then be observed by either the application itself or an external system (such as the OS or another application) so that the application or system can tune its behavior to make sure the goals are met. Application-specific goals may include throughput, power, latency, quality-of-service, or combinations thereof. Application Heartbeats can also help provide fault tolerance by providing information that can be used to predict or quickly detect failures.

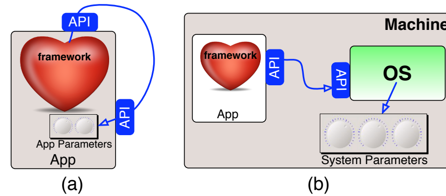


Fig. 1. (a) Self-optimizing application using the Application Heartbeats framework. (b) Optimization of machine parameters by an external observer.

This paper makes the following contributions:

1. A simple, standardized Heartbeats API for specifying and monitoring application-specific performance metrics.
2. Examples of ways that the framework can be used, both within an application and by external services, to develop self-optimizing applications. Experimental results demonstrate the effectiveness of the Application Heartbeats approach.

The rest of this paper is organized as follows. Section 2 identifies key system components that will benefit from the Application Heartbeats framework. Section 3 describes the Application Heartbeats API in greater detail. Section 4 presents our experimental results. Section 5 compares Application Heartbeats to related work. Finally, Section 6 concludes.

2 Heartbeat Usage in System Software and Hardware

The Application Heartbeats framework is a simple end-to-end feedback mechanism that can potentially have a large impact on the design of adaptive and self-tuning computer systems. This section explores ideas for novel computer architectures, operating systems, and compilers which may exploit the Heartbeats framework as a feedback mechanism to enable self-tuning.

Self-tuning Architectures. We envision a multicore microarchitecture that can adapt properties of its TLB, L1 cache, and L2 cache structures such as associativity, size, replacement policy, etc. to improve performance or minimize energy for a given performance level. We envision a multicore microarchitecture that can adapt its execution pipeline in a way similar to the heterogeneous multicores proposed in [2]. Lastly, we envision a multicore microarchitecture where decisions about dynamic frequency and voltage scaling are driven by the performance measurements and target heart rate mechanisms of the Heartbeats framework. [3, 4] are examples of frequency and voltage scaling to reduce power. Driving these new microarchitectures with an end-to-end mechanism such as a heartbeat, as opposed to indicators such as cache misses or utilization, ensures that microarchitectural optimizations focus on aspects of execution most important to meeting application goals.

Organic Operating Systems. Heartbeats provides a framework for novel operating systems with *organic* features such as self-healing and intelligent resource management. Heartbeats allow an OS to determine when applications fail and quickly restart them. Heartbeats provide the feedback necessary to make decisions about how many cores to allocate to an application. An organic OS would be able to automatically and dynamically adjust the number of cores an application uses based on an individual application's changing needs as well as the needs of other applications competing for resources. The OS would adjust the number of cores and observe the effect on the application's heart rate. An organic OS could also take advantage of the Heartbeats framework in the scheduler. Schedulers could be designed to run an application for a specific number of heartbeats (implying a variable amount of time) instead of a fixed time quanta. Schedulers could be designed that prioritize time allocation based on the target heart rate requirements of different applications. Locking mechanisms provided through the OS can be improved using Heartbeats. For example, Smartlocks [5], an adaptive locking framework, uses the Heartbeats API to obtain a direct measure of program performance and adapt locking and scheduling policies to meet the performance goals of the application. Heartbeats provide Smartlocks with a direct measure of application performance as opposed to using statistics gathered within the lock library such as lock contention.

Adaptive Compilation. Adaptive [6] and dynamic [7] compilation techniques have emerged to increase portability and address some challenges that cannot be met by traditional static compilation. Heartbeats could be used to improve the design of these compilers in several ways. First, by providing a standard interface the Heartbeat API allows one program to work with multiple compilers without source-level code changes. Second, by providing a mechanism

Table 1. Heartbeat API functions

Function Name	Arguments	Description
HB_initialize	window[int], local[bool]	Initialize the Heartbeat runtime system and specify how many heartbeats will be used to calculate the default average heart rate
HB_heartbeat	tag[int], local[bool]	Generate a heartbeat to indicate progress
HB_current_rate	window[int], local[bool]	Returns the average heart rate calculated from the last *window* heartbeats
HB_set_target_rate	min[int], max[int], local[bool]	Called by the application to indicate to an external observer the average heart rate it wants to maintain
HB_get_target_min	local[bool]	Called by the application or an external observer to retrieve the minimum target heart rate set by HB_set_target_rate
HB_get_target_max	local[bool]	Called by the application or an external observer to retrieve the maximum target heart rate set by HB_set_target_rate
HB_get_history	n[int], local[bool]	Returns the timestamp, tag, and thread ID for the last n heartbeats

for specifying program goals Heartbeats allow dynamic compilers to know when to stop optimizing, allowing the system to save energy by avoiding unnecessary work. Third, the Heartbeat API allows application code to specify the regions of the application where performance is critical, again allowing the system to avoid unnecessary optimization. As an example, the SpeedPress compiler inserts a runtime system, called SpeedGuard, into an application which uses Heartbeats to detect performance changes. The SpeedGuard runtime can then trade quality-of-service for performance in order to maintain the real-time goals of a system in the face of faults like core-failures and clock frequency changes [8].

3 Heartbeats API

Since heartbeats are meant to reduce programmer effort, they must be easy to insert into applications. The basic Heartbeat API consists of only a few functions (shown in Table 1) that can be called from applications or system software. To maintain a simple, conventional programming style, the Heartbeats API uses only standard function calls and does not rely on complex mechanisms such as OS callbacks.

The key function in the Heartbeat API is *HB_heartbeat*. Calls to *HB_heartbeat* are inserted into the application code at significant points to register the application's progress. Each time *HB_heartbeat* is called, a heartbeat event is logged. Each heartbeat generated is automatically stamped with the current time and thread ID of the caller. In addition, the user may specify a tag that can be used to provide additional information. For example, a video application may wish to indicate the type of frame (I, B or P) to which the heartbeat corresponds. Tags can also be used as sequence numbers in situations where some heartbeats may be dropped or reordered. Using the *local* flag, the user can specify whether

the heartbeat should be counted as a local (per-thread) heartbeat or as a global (per-application) heartbeat.

We anticipate that many applications will generate heartbeats in a regular pattern. For example, the video encoders may generate a heartbeat for every frame of video. For these applications, it is likely that the key metric will be the average frequency of heartbeats or *heart rate*. The *HB_current_rate* function returns the average heart rate for the most recent heartbeats.

Different applications and observers may be concerned with either long- or short-term trends. Therefore, it should be possible to specify the number of heartbeats (or *window*) used to calculate the moving average. Regarding window size there may be some tension between the application registering the heartbeats and the system service reading the heartbeats. We assume that the application knows which window size is most appropriate for the computation it is performing; however, the system service responding to this information may want to override this window if it is trying to make adjustments on a different granularity. Therefore, the API allows the application to set the window size and this size is the default used whenever an external system requests the current heartrate. An additional API call allows system software to override the window size.

Applications with real-time deadlines or performance goals will generally have a target heart rate that they wish to maintain. For example, if a heartbeat is produced at the completion of a task, then this corresponds to completing a certain number of tasks per second. Some applications will observe their own heartbeats and take corrective action if they are not meeting their goals. However, some actions (such as adjusting scheduler priorities or allocated resources) may require help from an external source such as the operating system. In these situations, it is helpful for the application to communicate its goals to an external observer. For this, we provide the *HB_set_target_rate* function which allows the application to specify a target heart rate range. The external observer can then take steps on its own if it sees that the application is not meeting (or is exceeding) its goals.

When more in-depth analysis of heartbeats are required, the *HB_get_history* function can be used to get a complete log of recent heartbeats. It returns an array of the last n heartbeats in the order that they were produced. This allows the user to examine intervals between individual heartbeats or filter heartbeats according to their tags. Most systems will probably place an upper limit on the value of n to simplify bookkeeping and prevent excessive memory usage. This provides the option to efficiently store heartbeats in a circular buffer. When the buffer fills, old heartbeats are simply dropped.

Multithreaded applications may require both per-thread and global heartbeats. For example, if different threads are working on independent objects, they should use separate heartbeats so that the system can optimize them independently. If multiple threads are working together on a single object, they would likely share a global heartbeat. Thus, each thread should have its own private heartbeat history buffer and each application should have a single shared

history buffer. Threads may read and write to their own buffer and the global buffer but not the other threads' buffers.

Some systems may contain hardware that can automatically adapt using heartbeat information. For example, a processor core could automatically adjust its own frequency to maintain a desired heart rate in the application. Therefore, it must be possible for hardware to directly read from the heartbeat buffers. In this case the hardware must be designed to manipulate the buffers' data structures just as software would. To facilitate this, a standard must be established specifying the components and layout of the heartbeat data structures in memory. Hardware within a core should be able to access the private heartbeats for any threads running on that core as well as the global heartbeats for an application. We leave the establishment of this standard and the design of hardware that uses it to future work.

4 Experimental Results

This section presents several examples illustrating the use of the Heartbeats framework. First, a brief study is presented using Heartbeats to instrument the PARSEC benchmark suite [9]. Next, an adaptive H.264 encoder is developed to demonstrate how an application can use the Heartbeats framework to modify its own behavior. Then an adaptive scheduler is described to illustrate how an external service can use Heartbeats to respond directly to the needs of a running application. Finally, the adaptive H.264 encoder is used to show how Heartbeats can help build fault-tolerant applications. All results discussed in this section were collected on an Intel x86 server with dual 3.16 GHz Xeon X5460 quad-core processors.

4.1 Heartbeats in the PARSEC Benchmark Suite

To demonstrate the applicability of the Heartbeats framework across a range of multicore applications, it is applied to the PARSEC benchmark suite (version 1.0). For each benchmark, we read the description of the application, find the outermost loop and insert the heartbeats in this loop. Table 2 shows where the heartbeat was inserted in terms of the application's processing and the average heart rate that the benchmark achieved over the course of its execution running the "native" input data set on the eight-core x86 test platform[1]. We note that placement of heartbeats is flexible and can be tailored to the specific needs of the application.

For all benchmarks presented here, the Heartbeats framework is low-overhead. For eight of the ten benchmarks the overhead of Heartbeats was negligible. For the `blackscholes` benchmark, the overhead is negligible when registering a heartbeat every 25,000 options; however, in the first attempt a heartbeat was registered after every option was processed and this added an order of magnitude

[1] Two benchmarks are missing as neither `freqmine` nor `vips` would compile on the target system due to issues with the installed version of `gcc`.

Table 2. Heartbeats in the PARSEC Benchmark Suite

Benchmark	Heartbeat Location	Average Heart Rate (beat/s)
blackscholes	Every 25000 options	561.03
bodytrack	Every frame	4.31
canneal	Every 1875 moves	1043.76
dedup	Every “chunk”	264.30
facesim	Every frame	0.72
ferret	Every query	40.78
fluidanimate	Every frame	41.25
streamcluster	Every 200000 points	0.02
swaptions	Every “swaption”	2.27
x264	Every frame	11.32

slow-down. For the other benchmark with measurable overhead, `facesim`, the added time due to the use of Heartbeats is less than 5 %.

Adding heartbeats to the PARSEC benchmark suite is easy, even for users who are unfamiliar with the benchmarks themselves. The PARSEC documentation describes the inputs for each benchmark. With that information it is simple to find the key loops over the input data set and insert the call to register a heartbeat in this loop. The total amount of code required to add heartbeats to each of the benchmarks is under half-a-dozen lines. The extra code is simply the inclusion of the header file and declaration of a Heartbeat data structure, calls to initialize and finalize the Heartbeats run-time system, and the call to register each heartbeat.

In summary, the Heartbeats framework is easy to insert into a broad array of applications and our reference implementation is low-overhead. The next section provides an example of using the Heartbeats framework to develop an adaptive application.

4.2 Internal Heartbeat Usage

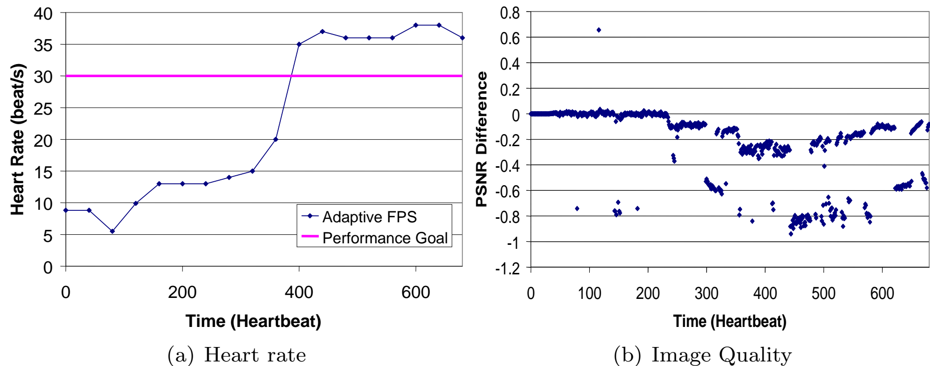
This example shows how Heartbeats can be used within an application to help a real-time H.264 video encoder maintain an acceptable frame rate by adjusting its encoding quality to increase performance. For this experiment the x264 implementation of an H.264 video encoder [10] is augmented so that a heartbeat is registered after each frame is encoded. x264 registers a heartbeat after every frame and checks its heart rate every 40 frames. When the application checks its heart rate, it looks to see if the average over the last forty frames was less than 30 beats per second (corresponding to 30 frames per second). If the heart rate is less than the target, the application adjusts its encoding algorithms to get more performance while possibly sacrificing the quality of the encoded image.

For this experiment, x264 is launched with a computationally demanding set of parameters for Main profile H.264 encoding. Both the input parameters and the video used here are different than the PARSEC inputs; both are chosen to be more computationally demanding and more uniform. The parameters include

the use of exhaustive search techniques for motion estimation, the analysis of all macroblock sub-partitionings, x264's most demanding sub-pixel motion estimation, and the use of up to five reference frames for coding predicted frames. Even on the eight core machine with x264's assembly optimizations enabled, the unmodified x264 code-base achieves only 8.8 heartbeats per second with these inputs.

As the Heartbeat-enabled x264 executes, it reads its heart rate and changes algorithms and other parameters to attempt to reach an encoding speed of 30 heartbeats per second. As these adjustments are made, x264 switches to algorithms which are faster, but may produce lower quality encoded images.

Figures 2(a) and 2(b) illustrate the behavior of this adaptive version of x264 as it attempts to reach its target heart rate of 30 beats per second. The first figure shows the average heart rate over the last 40 frames as a function of time (time is measured in heartbeats or frames). The second figure illustrates how the change in algorithm affects the quality (measured in peak signal to noise ratio) of the encoded frames.



(a) Heart rate (b) Image Quality

Fig. 2. Heart rate and image quality of adaptive x264. (a) shows how the heart rate of x264 changes as the program adapts to meet its goals. (b) shows the difference in PSNR between the unmodified x264 code base and our adaptive version.

As shown in Figure 2(a) the adaptive implementation of x264 gradually increases its speed until frame 400, at which point it makes a decision allowing it to maintain a heart rate over thirty-five beats per second. Given these inputs and the target performance, the adaptive version of x264 tries several search algorithms for motion estimation and finally settles on the computationally light diamond search algorithm. Additionally, this version of x264 stops attempting to use any sub-macroblock partitionings. Finally, the adaptive encoder decides to use a less demanding sub-pixel motion estimation algorithm.

As shown in Figure 2(b), as x264 increases speed, the quality, measured in PSNR, of the encoded images decreases. This figure shows the difference in PSNR between the unmodified x264 source code and the Heartbeat-enabled

implementation which adjusts its encoding parameters. In the worst case, the adaptive version of x264 can lose as much as one dB of PSNR, but the average loss is closer to 0.5 dB. This quality loss is just at the threshold of what most people are capable of noticing. However, for a real-time encoder using these parameters on this architecture the alternative would be to drop two out of every three frames. Dropping frames has a much larger negative impact on the perceived quality than losing an average of 0.5 dB of PSNR per frame.

This experiment demonstrates how an application can use the Heartbeats API to monitor itself and adapt to meet its own needs. This allows the programmer to write a single general application that can then be run on different hardware platforms or with different input data streams and automatically maintain its own real-time goals. This saves time and results in more robust applications compared to writing a customized version for each individual situation or tuning the parameters by hand.

Videos demonstrating the adaptive encoder are available online. These videos are designed to capture the experience of watching encoded video in real-time as it is produced. The first video shows the heart rate of the encoder without adaptation[2]. The second video shows the heart rate of the encoder with adaptation[3].

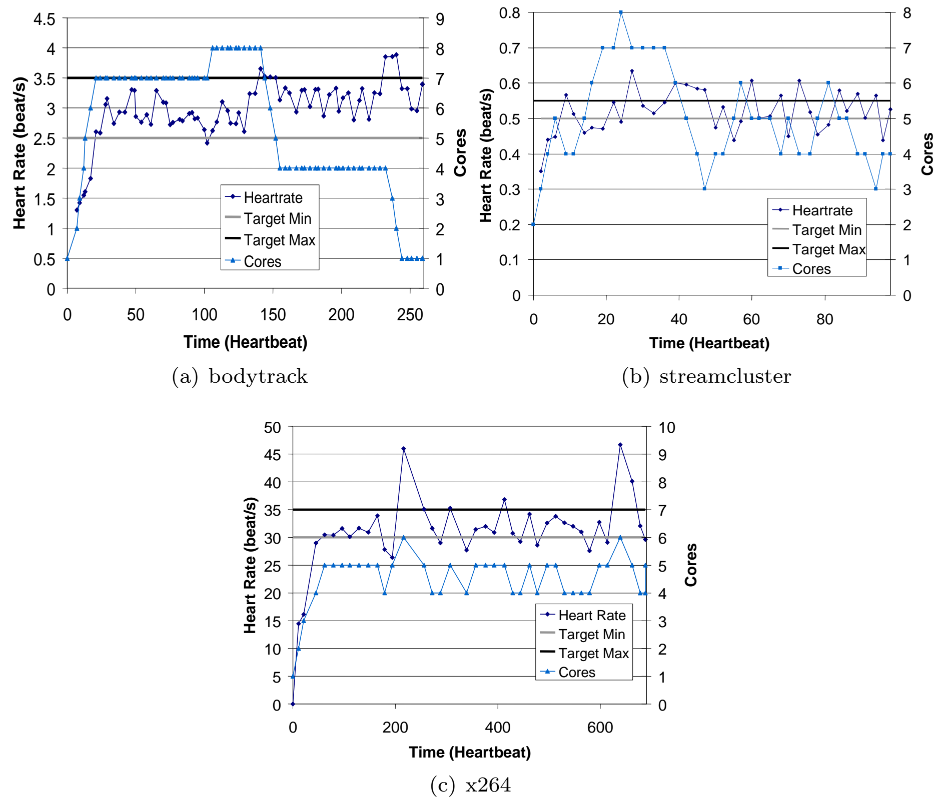
4.3 External Heartbeat Usage

In this example, Heartbeats are used to help an external system allocate resources while maintaining required application performance. The application communicates performance information and goals to an external observer which attempts to keep performance within the specified range using the minimum number of cores possible. Three of the Heartbeat-enabled PARSEC benchmarks are run while an external scheduler reads their heart rates and adjusts the number of cores allocated to them. The applications tested include the PARSEC benchmarks `bodytrack`, `streamcluster`, and `x264`.

bodytrack The `bodytrack` benchmark is a computer vision application that tracks a person's movement through a scene. For this application a heartbeat is registered at every frame. Using all eight cores of the x86 server, the `bodytrack` application maintains an average heart rate of over four beats per second. The external scheduler starts this benchmark on a single core and then adjusts the number of cores assigned to the application in order to keep performance between 2.5 and 3.5 beats per second.

The behavior of `bodytrack` under the external scheduler is illustrated in Figure 3(a). This figure shows the average heart rate as a function of time measured in beats. As shown in the figure, the scheduler quickly increases the assigned cores until the application reaches the target range using seven cores. Performance stays within that range until heartbeat 102, when performance dips below 2.5

[2] Available here: http://www.youtube.com/watch?v=c1t30MDcpP0
[3] Available here: http://www.youtube.com/watch?v=Msr22JcmYWA



(a) bodytrack

(b) streamcluster

(c) x264

Fig. 3. Behavior of selected PARSEC applications coupled with an external scheduler.

beats per second and the eighth and final core is assigned to the application. Then, at beat 141 the computational load suddenly decreases and the scheduler is able to reclaim cores while maintaining the desired performance. In fact, the application eventually needs only a single core to meet its goal.

The `streamcluster` benchmark solves the online clustering problem for a stream of input points by finding a number of medians and assigning each point to the closest median. For this application one heartbeat is registered for every 5000 input points. Using all eight cores of the x86 server, the `streamcluster` benchmark maintains an average heart rate of over 0.75 beats per second. The scheduler starts this application on a single core and then attempts to keep performance between 0.5 and 0.55 beats per second.

The behavior of `streamcluster` under the external scheduler is displayed in Figure 3(b). This figure shows the average heart rate as a function of time (measured in heartbeats). The scheduler adds cores to the application to reach the target heart rate by the twenty-second heartbeat. The scheduler then works to keep the application within the narrowly defined performance window. The

figure illustrates that the scheduler is able to quickly react to changes in application performance by using the Heartbeats interface.

x264 The x264 benchmark is the same code base used in the internal optimization experiment described above. Once again, a heartbeat is registered for each frame. However, for this benchmark the input parameters are modified so that x264 can easily maintain an average heart rate of over 40 beats per second using eight cores. The scheduler begins with x264 assigned to a single core and then adjusts the number of cores to keep performance in the range of 30 to 35 beats per second.

Figure 3(c) shows the behavior of x264 under the external scheduler. Again, average heart rate is displayed as a function of time measured in heartbeats. In this case the scheduler is able to keep x264's performance within the specified range while using four to six cores. As shown in the chart the scheduler is able to quickly adapt to two spikes in performance where the encoder is able to briefly achieve over 45 beats per second. A video demonstrating the performance of the encoder running under the adaptive external scheduler has been posted online[4].

These experiments demonstrate a fundamental benefit of using the Heartbeats API for specifying application performance: external services are able to read the heartbeat data and adapt their behavior to meet the application's needs. Furthermore, the Heartbeats interface makes it easy for an external service to quantify its effects on application behavior. In this example, an external scheduler is able to adapt the number of cores assigned to a process based on its heart rate. This allows the scheduler to use the minimum number of cores necessary to meet the application's needs. The decisions the scheduler makes are based directly on the application's performance instead of being based on priority or some other indirect measure.

4.4 Heartbeats for Fault Tolerance

The final example in this section illustrates how the Heartbeats framework can be used to aid in fault tolerance. This example reuses the adaptive H.264 encoder developed above in Section 4.2. The adaptive encoder is initialized with a parameter set that can achieve a heart rate of 30 beat/s on the eight-core testbed. At frames 160, 320, and 480, a core failure is simulated by restricting the scheduler to running x264 on fewer cores. After each core failure the adaptive encoder detects a drop in heart rate and adjusts its algorithm to try to maintain its target performance.

The results of this experiment are shown in Figure 4. This figure shows a moving average of heart rate (using a 20-beat window) as a function of time for three data sets. The first data set, labeled "Healthy," shows the behavior of unmodified x264 for this input running on eight cores with no failures. The second data set, labeled "Unhealthy," shows the behavior of unmodified x264 when cores "die" (at frames 160, 320, and 480). Finally, the data set labeled

[4] Available here: http://www.youtube.com/watch?v=l3sVaGZKgkc

"Adaptive" shows how the adaptive encoder responds to these changes and is able to keep its heart rate above the target even in the presence of core failures.

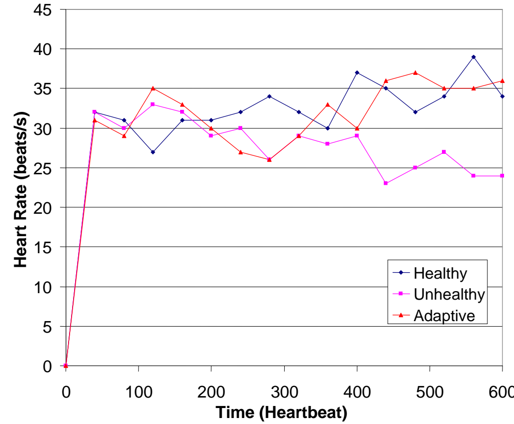


Fig. 4. Using Heartbeats in an adaptive video encoder for fault tolerance. The line labeled "Healthy" shows the performance of the encoder under normal circumstances. The line labeled "Unhealthy" shows the performance of the encoder when cores fail. The line labeled "Adaptive" shows the performance of an adaptive encoder that adjusts its algorithm to maintain a target heart rate of greater than 30 beats/s.

Figure 4 shows that in a healthy system, x264 is generally able to maintain a heart rate of greater than 30 beat/s. Furthermore, the performance in the healthy case actually increases slightly towards the end of execution as the input video becomes slightly easier at the end. In the unhealthy system, where cores die, the unmodified x264 is not able to maintain its target heart rate and performance falls below 25 beat/s. However, the adaptive encoder is able to change the algorithm and maintain performance in the face of hardware failures.

The adaptive encoder does not detect a fault or attempt to detect anything about which, or how many, cores are healthy. Instead, the adaptive encoder only attempts to detect changes in performance as reflected in the heart rate. The encoder is then able to adapt its behavior in order to return performance to its target zone.

The generality of this approach means that the encoder can respond to more than just core failures. For example, if a cooling fan failed and the hardware lowered its supply voltage to reduce power consumption, the encoder would detect the loss of performance and respond. Any event that alters performance will be detected by this method and allow the encoder a chance to adapt its behavior in response. Thus, the Heartbeats framework can aid fault tolerance and detection by providing a general way to detect changes in application performance.

5 Related Work

The problem of performance monitoring is fundamental to the development of parallel applications, so it has been addressed by a variety of different approaches. This work includes research on monitoring single- and multi-core architectures [2, 11, 12], networks [13], complex software systems and operating systems [14–20]. Most of this work focuses on off-line collection and visualization of performance data. More complex monitoring techniques have been presented in [21, 19]. This work represents a shift in approach as the research community moves from using simple hardware-based metrics, *i.e.,*cache miss rate, to more advanced statistics. Hardware assistance for system monitoring, often in the form of event counters, is included in most common architectures. However, counter-based techniques suffer common shortcomings [22]: too few counters, sampling delay, and lack of address profiling. In addition, adaptive systems based on hardware event counters must infer application performance from low-level hardware statistics.

A software approach for application monitoring is proposed in [14]. This work proposes an assertion based framework which can be used to verify that the runtime performance meets expected performance. Using the assertions, the programmer specifies performance expectations which the application can use at runtime to adapt itself. This framework allows a rich description of the program performance in terms of hardware specific parameters like the expected rate of floating point operations; however, use of the framework requires extensive code annotation and only allows the application to make internal updates to itself. In contrast, the Heartbeat framework is designed to specify performance in terms of a simple, and general mechanism and directly communicate this performance to external systems which can customize their behavior to meet those goals. We envision the use of the Heartbeat framework within a broader context where multiple applications can be executed in parallel, each using heartbeats to communicate performance and relying on the external services to help them meet their goals.

The rise of adaptive computing systems creates new challenges and demands for system monitoring [23]. One example of these emerging adaptive systems can be found in the self-optimizing memory controller described in [24]. This controller optimizes its scheduling policy using reinforcement learning to estimate the performance impact of each action it takes. As designed, performance is measured in terms of memory bus utilization. The controller optimizes memory bus utilization because that is the only metric available to it, and better bus utilization generally results in better performance. However, it would be preferable for the controller to optimize application performance directly and the Heartbeats API provides a mechanism with which to do so. Furthermore, the Heartbeats API is kept simple, which makes it easy for not only the end-users to get started with the framework but also to hook up third party auto-tuning tools such as Orio [25], Autopilot [26], Active Harmony [27], to make the application adaptation decisions based on the observed heartbeat rates. The fact

that this research could be built on top of the Heartbeat interface demonstrates the API's usefulness.

System monitoring, as described in this section, is a crucial task for several very different goals: performance, security, quality of service, etc. Different ad hoc techniques for self-optimization have been presented in the literature, but the Heartbeats approach is the only one that provides a simple, unified framework for reasoning about and addressing all of these goals.

6 Conclusion

Our prototype results indicate that the Heartbeats framework is a useful tool for both application auto-tuning and externally-driven optimization. Our experimental results demonstrate three useful applications of the framework: dynamically reducing output quality (accuracy) as necessary to meet a throughput (performance) goal, optimizing system resource allocation by minimizing the number of cores used to reach a given target output rate, and tolerating failures by adjusting output quality to compensate for lost computational resources. The authors have identified several important applications that the framework can be applied to: self-optimizing microarchitectures, self-tuning software libraries, smarter system administration tools, novel "Organic" operating systems and runtime environments, and more profitable cloud computing clusters. We believe that a unified, portable standard for application performance monitoring is crucial for a broad range of future applications.

References

1. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2) (2009) 1–42
2. Kumar, R., Farkas, K., Jouppi, N., Ranganathan, P., Tullsen, D.: Processor power reduction via single-isa heterogeneous multi-core architectures. Computer Architecture Letters **2**(1) (Jan-Dec 2003) 2–2
3. Govil, K., Chan, E., Wasserman, H.: Comparing algorithm for dynamic speed-setting of a low-power CPU. In: MobiCom '95: Proceedings of the 1st Annual Inter. Conf. on Mobile Computing and networking. (1995) 13–25
4. Pering, T., Burd, T., Brodersen, R.: The simulation and evaluation of dynamic voltage scaling algorithms. In: ISLPED '98: Proceedings of the 1998 Inter. Symp. on Low Power Electronics and Design. (1998) 76–81
5. Eastep, J., Wingate, D., Santambrogio, M.D., Agarwal, A.: Smartlocks: Self-aware synchronization through lock acquisition scheduling. Technical Report MIT CSAIL, MIT (Nov 2009)
6. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: PetaBricks: A language and compiler for algorithmic choice. In: Conf. on Programming Language Design and Implementation. (Jun 2009)
7. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: Proceedings of the international symposium on code generation and optimization. (2003)

8. Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., Rinard, M.: Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT (September 2009)
9. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: PACT-2008: Proceedings of the 17th Inter. Conf. on Parallel Architectures and Compilation Techniques. (Oct 2008)
10. x264. Online document, `http://www.videolan.org/x264.html`
11. Intel Inc.: Intel itanium architecture software developer's manual (2006)
12. Azimi, R., Stumm, M., Wisniewski, R.W.: Online performance analysis by statistical sampling of microprocessor performance counters. In: ICS '05: Proceedings of the 19th Inter. Conf. on Supercomputing. (2005) 101–110
13. Wolski, R., Spring, N.T., Hayes, J.: The network weather service: a distributed resource performance forecasting service for metacomputing. Future Generation Computer Systems **15**(5–6) (1999) 757–768
14. Vetter, J., Worley, P.: Asserting performance expectations. In: Supercomputing, ACM/IEEE 2002 Conference. (Nov. 2002) 33–33
15. Caporuscio, M., Di Marco, A., Inverardi, P.: Run-time performance management of the siena publish/subscribe middleware. In: WOSP '05: Proc. of the 5th Inter. Work. on Software and performance. (2005) 65–74
16. De Rose, L.A., Reed, D.A.: SvPablo: A multi-language architecture-independent performance analysis system. In: Inter. Conf. on Parallel Processing. (1999)
17. Cascaval, C., Duesterwald, E., Sweeney, P.F., Wisniewski, R.W.: Performance and environment monitoring for continuous program optimization. IBM J. Res. Dev. **50**(2/3) (2006) 239–248
18. Krieger, O., Auslander, M., Rosenburg, B., W., R.W.J., Xenidis, Silva, D.D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., Uhlig, V.: K42: Building a complete operating system. In: EuroSys '06: Proc. of the 1st ACM SIGOPS/EuroSys Euro. Conf. on Computer Systems. (2006)
19. Wisniewski, R.W., Rosenburg, B.: Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In: SC '03: Proc. of the ACM/IEEE conf. on Supercomputing. (Nov 2003)
20. Tamches, A., Miller, B.P.: Fine-grained dynamic instrumentation of commodity operating system kernels. In: OSDI '99: Proc. of the third symp. on Operating systems design and implementation. (1999)
21. Schulz, M., White, B.S., McKee, S.A., Lee, H.H.S., Jeitner, J.: Owl: next generation system monitoring. In: CF '05: Proc. of the 2nd conf. on Computing Frontiers. (2005)
22. Sprunt, B.: The basics of performance-monitoring hardware. IEEE Micro **22**(4) (Jul/Aug 2002) 64–71
23. Dini, P.: Internet, GRID, self-adaptability and beyond: Are we ready? (Aug 2004)
24. Ipek, E., Mutlu, O., Martnez, J.F., Caruana, R.: Self-optimizing memory controllers: A reinforcement learning approach. In: ISCA '08: Proc. of the 35th Inter. Symp. on Comp. Arch. (2008)
25. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using orio. In: IPDPS '09: Proc. of the Inter. Symp. on Parallel&Distributed Processing. (2009)
26. Ribler, R., Vetter, J., Simitci, H., Reed, D.: Autopilot: adaptive control of distributed applications. In: High Performance Distributed Computing. (Jul 1998)
27. Hollingsworth, J., Keleher, P.: Prediction and adaptation in active harmony. In: High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on. (Jul 1998) 180–188