Application Heartbeats

A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments

Henry Hoffmann[1] hank@csail.mit.edu
Jonathan Eastep[1] eastepjm@csail.mit.edu
Marco D. Santambrogio[1,2] santa@csail.mit.edu
Jason E. Miller[1] jasonm@csail.mit.edu
Anant Agarwal[1] agarwal@csail.mit.edu

[1]Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory

[2]Politecnico di Milano
Dipartimento di Elettronica e Informazione

ABSTRACT

The rise of multicore computing has greatly increased system complexity and created an additional burden for software developers. This burden is especially troublesome when it comes to optimizing software on modern computing systems. Autonomic or adaptive computing has been proposed as one method to help application programmers handle this complexity. In an autonomic computing environment, system services monitor applications and automatically adapt their behavior to increase the performance of the applications they support. Unfortunately, applications often run as performance black-boxes and adaptive services must infer application performance from low-level information or rely on system-specific ad hoc methods. This paper proposes a standard framework, Application Heartbeats, which applications can use to communicate both their current and target performance and which autonomic services can use to query these values.

The Application Heartbeats framework is designed around the well-known idea of a heartbeat. At important points in the program, the application registers a heartbeat. In addition, the interface allows applications to express their performance in terms of a desired heart rate and/or a desired latency between specially tagged heartbeats. Thus, the interface provides a standard method for an application to directly communicate its performance and goals while allowing autonomic services access to this information. Thus, Heartbeat-enabled applications are no longer performance black-boxes. This paper presents the Applications Heartbeats interface, characterizes two reference implementations (one suitable for clusters and one for multicore), and illustrates the use of Heartbeats with several examples of systems adapting behavior based on feedback from heartbeats.

ICAC'10, June 7–11, 2010, Washington, DC, USA.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; D.4.8 [**Operating Systems**]: PerformanceMonitors

General Terms

Performance, Measurement, Standardization

Keywords

Adaptive Computing, Self-Tuning Systems

1. INTRODUCTION

As multicore processors become increasingly prevalent, system complexities are skyrocketing. Modern systems are composed of increasing numbers of possibly heterogeneous processors with special-purpose functional units backed up with distributed, deep memory hierarchies. Furthermore, modern systems are faced with new physical constraints on power and energy. It is no longer practical for an average applications programmer to be an expert in their application domain, have the systems knowledge necessary to manage these additional 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 *autonomic*, or *adaptive* hardware and software. Autonomic system services take some of the burden off of programmers by taking over the task of monitoring the environment and adapting their behavior as necessary to help optimize the application's performance.

Widespread use of autonomic systems in system software and hardware serves to modularize the task of performance optimization. In such an environment, application developers focus on the tasks in their realm of expertise: selecting optimal algorithms and making application-specific optimizations. At the same time, autonomic system services (OS, compiler, hardware, etc.) handle tasks like optimizing compilation, adapting scheduling policies, managing memory hierarchies, and making resource allocation decisions to optimize for performance or power. In short, autonomic systems can potentially take over all other tasks that are beyond the comfort zone of most applications programmers.

To achieve this vision, autonomic system services must be able to monitor their environment and make informed decisions about how to alter their behavior as described in [23].

Unfortunately, most applications run as performance black-boxes, providing little information about their current performance and no information about their performance goals. Thus, autonomic systems are left to infer high-level performance information from low-level details obtained from performance counters; however, such approaches frequently do not capture the actual performance 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. Furthermore, measuring performance counters provides absolutely no information about the application's desired performance. This lack of information on the target performance makes it difficult for autonomic system services to perform some types of optimization like minimizing power given a real-time performance constraint. What is needed is a portable, universal method of monitoring an application's actual as well as its target performance.

This paper presents a software interface, called *Application Heartbeats* (or just *Heartbeats* for short), that provides a simple, standardized way for applications to report their performance and goals to external observers. 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). Having a simple, standardized interface makes it easy for programmers to add Heartbeats to their applications. A standard interface, or API, is also crucial for portability and inter-operability between different applications, runtime systems, and operating systems. Registering an application's goals with external systems enables autonomic systems to make optimization decisions while monitoring the program's performance directly rather than having to infer that performance from low-level details.

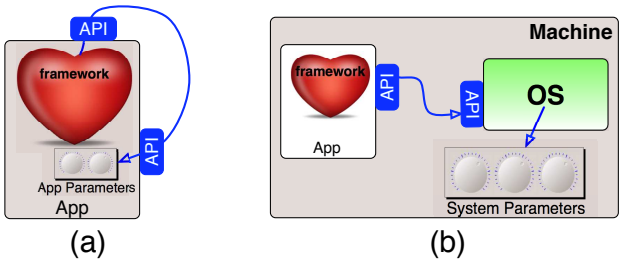


Figure 1: (a) Self-optimizing application using the Application Heartbeats framework. (b) Optimization of machine parameters by an external observer.

The Application Heartbeats framework measures application progress toward goals using a simple and well-known abstraction: a heartbeat. At significant points, applications make an API call to signify a heartbeat. Over time, the intervals between heartbeats provide key information about progress for the purpose of application auto-tuning and/or externally-driven optimization. The Heartbeats API allows applications to communicate their goals by setting a target heart rate (*i.e.,*number of heartbeats per second) and a target latency between specially tagged heartbeats. Autonomic system services, such as the operating system, runtime systems, hardware, or the application itself, monitor progress through additional API calls and can then use this information to change their behavior and help the application achieve the specified performance goals. As an example, a video codec application could use Application Heartbeats to specify a target throughput of 30 video frames a second. In the encoder example, an autonomic scheduler could ensure that the encoder meets this goal while using the least number of cores, thus saving power or allowing extra cores to be assigned to other purposes.

This paper makes the following contributions:

1. A simple, standardized Heartbeats API for specifying and monitoring application-specific performance metrics.

2. A basic reference implementation of the API which is available as open-source software[1].

3. A description of two reference implementations of the API, one suitable for performance on multi-machine environments like clusters and clouds, and one suitable for multi-core environments.

4. A characterization of the performance and overheads of each of the two reference implementations.

5. Examples of ways the Heartbeats API can be used to augment autonomic systems, both within an application and by external system services. Experimental results demonstrate the effectiveness of the Application Heartbeats approach in the PARSEC benchmarks [4].

The rest of this paper is organized as follows. Section 2 describes the Application Heartbeats API in greater detail. Section Section 3 discusses the implementation of two reference implementations of the Application Heartbeats API. Section 4 characterizes the performance of these reference implementations and presents several examples showing how to use Heartbeats in applications and autonomic system services. Section 5 compares Application Heartbeats to related work. Finally, Section 6 concludes.

2. 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 server application may use one tag to denote the arrival of a request and another tag to signal its completion. Tags can also be used as sequence numbers in situations where some heartbeats may be dropped or reordered. Finally, tags can be used to determine

[1]See `http://groups.csail.mit.edu/carbon/heartbeats` for details on obtaining the code.

Table 1: Heartbeat API functions

Function Name	Arguments	Description
HB_initialize	window[int], buffer_size[int]	Initialize the Heartbeat runtime system and specify how many heartbeats will be used to calculate the default average heart rate and how many heartbeats to buffer
HB_heartbeat	tag[int]	Generate a heartbeat to indicate progress
HB_current_rate		Returns the average heart rate calculated from the last *window* heartbeats
HB_get_current_heartbeat		Returns the tag, time-stamp, and current heart rate measured the last time a heartbeat was generated
HB_set_target_rate	min[float], max[float]	Called by the application to indicate to an external observer the average heart rate it wants to maintain
HB_get_target_min_rate		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_rate		Called by the application or an external observer to retrieve the maximum target heart rate set by HB_set_target_rate
HB_set_target_latency	min[float], max[float], tag1[int], tag2[int]	Called by the application to indicate to an external observer the average latency it wants to achieve between two heartbeats with the given tags
HB_get_target_min_latency	tag1[int], tag2[int]	Called by the application or an external observer to retrieve the minimum target latency set by HB_set_target_latency
HB_get_target_max_latency	tag1[int], tag2[int]	Called by the application or an external observer to retrieve the maximum target latency set by HB_set_target_latency
HB_get_history	n[int]	Returns the time-stamp, tag, and thread ID for the last n heartbeats

the latency between events, for example, the time between a request arriving at an application and the completion of the task.

We anticipate that many applications will generate heartbeats in a regular pattern. For example, a video encoder 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.

Different applications and observers may be concerned with either long- or short-term trends. Therefore, it is possible to specify the number of heartbeats (or *window*) used to calculate the average heart rate. 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, so the API allows the application to set the window size and this size is the default used whenever an external system requests the current heart rate. A system service that wants to calculate a windowed average using a different window size can make use of the *HB_get_history* function discussed in greater detail below.

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, the value in the Heartbeats approach lies in communicating the performance and goals of an application to external systems which can also adapt their behavior and increase performance. To enable this communication, the API provides the *HB_set_target_rate* function which allows the application to specify a target heart rate range. If a system service sees that an application is not meeting its goals, it can adapt its behavior and to assist the application. Alternatively, if an application is achieving greater performance than it requires, services can reclaim some resources from that application.

Some applications may be more interested in reducing latency than increasing throughput. For example, a server might want to minimize the latency of processing a given type of request rather than strictly maximizing the number of requests serviced. In this case, applications can specify a target latency desired between two heartbeat tags using the function *HB_set_target_latency*. In the server example, one tag value would be used to indicate receiving a request while a distinct value indicates request completion. The *HB_set_target_latency* function allows the server to specify the desired latency between these two events. Other functions allow external observers to determine the desired latency and then make their own decisions as to how to help the application meet this goal.

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. The maximum value of n is determined by the *buffer_depth* parameter passed to the heartbeat initialization function.

Some systems may contain hardware that can automatically adapt using heartbeat information. For example, hardware could automatically adjust its own supply voltage 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, an additional standard must be established specifying the components and layout of the heartbeat data structures in memory. We leave the establishment of this standard and the design of hardware that uses it to future work.

3. IMPLEMENTATION

We provide two reference implementations of the Heartbeats API, both implemented for Linux in C and callable from C and C++ programs. We note that both implementations share the same interface and differ only in their internals. The first implementation uses files to communicate heartbeat data and is appropriate for sharing heartbeat information among different machines in a cluster or cloud environment. The second implementation uses POSIX shared memory to communicate heartbeat information and is appropriate for use among separate processes or threads on the same computer.

Both implementations adhere to several shared goals. First, although these are reference implementations, they are designed to minimize overhead given the targeted communication paradigm. Second, both implementations are thread safe with respect to the application registering heartbeats; multi-threaded applications can have each thread signaling heartbeats independently. Third, the small performance overhead is designed to be predictable because as we assume that applications making use of the Heartbeat API are those for whom performance is critical.

There are three central pieces of the Heartbeats implementation. The first is the way the implementation initializes a Heartbeat-enabled application and signals its existence to the rest of the world. The second is the way that applications register a heartbeat and make it available to external systems. The third is the method by which external system services read heartbeat data. While the two implementations each use different communication mechanisms, they share a great deal of overlap, so we discuss the implementation of Heartbeats in general and only highlight areas where the two implementations differ.

3.1 Initializing Heartbeat Applications

When an application calls *HB_initialize*, this function registers the calling application as one that produces a Heartbeat. It does so by writing a file in the directory pointed to by an environment variable, *HEARTBEAT_ENABLED_DIR*. Note that both implementations register themselves by writing a file with the application's process id to this directory. Autonomous services can then query this directory to determine if any Heartbeat-enabled applications are active in the system.

After signaling the existence of the application, the initialization function then allocates memory for maintaining heartbeat state. In the file-based implementation this memory takes the form of a binary file. The desired heart rate and latency between heartbeats are written into the top of this file. The remainder of the file stores the data associated with each individual heartbeat. In the shared-memory implementation the initialization function allocates a buffer capable of storing *buffer_size* heartbeats. This data is allocated in POSIX shared memory using the `shmget` and `shmat` functions. The `shmget` function associates a key with a shared memory region and processes that want to read this memory pass the key to `shmat` to map the memory into their address space. For the shared memory implementation, the initialization function writes the key into the file stored in *HEARTBEAT_ENABLED_DIR* so that other programs can map this memory into their address space and read the heartbeat data stored there.

3.2 Registering Heartbeats

The *heartbeat* function first records a time-stamp using the `clock_gettime` function and the `CLOCK_REALTIME` clock which provides nanosecond timing resolution. Next, the *heartbeat* function updates the average heart rate and stores this information in the shared memory buffer or file as appropriate. This function currently computes and stores three separate heart rates. The first is the heart rate measured over the entire lifetime of the program. The second is the instantaneous heart rate measured since the last heartbeat. The final value is the heart rate computed as a sliding window average using a window-size specified by the Heartbeat API. Each of these values is calculated using an $O(1)$ algorithm so that the performance is constant as a function of the window size. In this way, users can feel free to select the window size that best suits their application without worrying about how that window size impacts performance. After computing these values, the heartbeat function stores the tag, thread id of the caller, the time-stamp and all three heart rate values in either the shared memory buffer or the file depending on the implementation. All updates to internal state are done in a critical section guarded by a mutex to prevent threads in the same application from interfering with each other.

3.3 Reading Heartbeat State

An autonomic service that wants to read heartbeat information from a separate application first looks in the *HEARTBEAT_ENABLED_DIR* to determine if there are any applications in the system that are Heartbeat-enabled. If there are, then these services can use the accessor functions from Table 1 to read the goals and performance of any heartbeat enabled application in the system. In the file-based implementation, these accessor functions simply open the file and reading it. In the shared memory implementation, the initialization method first attaches the shared memory segment using the key provided in the file, then the accessor functions can read the desired values directly from this memory.

We note that, in the case of shared memory, functions which read heartbeat state do not acquire the lock. This is done so that a Heartbeat-enabled application's performance is consistent regardless of the number of readers attempting to access its data. If each reader acquired the lock then it might prevent the application from registering a heartbeat with predictable performance, and therefore violates one implementation goal of providing repeatable performance for Heartbeat-enabled applications.

4. EXPERIMENTAL RESULTS

This section presents several examples illustrating the use of the Heartbeats framework. First we evaluate the performance of the reference implementations of the Heartbeats API. Second, we present a brief study using Heartbeats to instrument the PARSEC benchmark suite [4]. Next, an adaptive H.264 encoder is developed to demonstrate how an application can use the Heartbeats framework to modify its own behavior. Finally, an adaptive scheduler is described to illustrate how an external service can use Heartbeats to respond directly to the needs of a running application. All results discussed in this section were collected on Intel x86 servers with dual 3.16 GHz Xeon X5460 quad-core processors using NFS as a shared file system.

Table 2: Performance of Heartbeats

Implementation	Heartbeat Throughput (Kbeat/s)	Heartbeat Latency (microseconds)
Shared Memory	1508.2	1.5
File I/O	0.9	136.2

4.1 Heartbeats API Performance

This section discusses several experiments conducted to measure the performance and overheads of our reference implementations of the Heartbeats API. There are two key metrics needed to evaluate the suitability of the interface for a given application or a system service. The first is the time taken to register a heartbeat, while the second is the delay from when the heartbeat is registered in an application to when it can be read in an external process. We refer to the first metric as the *heartbeat throughput* while the second is called the *heartbeat latency*.

To measure heartbeat throughput, we simply write an application that calls the heartbeat API function repeatedly in a loop. After exiting the loop, we read the global heart rate. The results for both the file-based and shared memory implementations are shown in Table 2. Not surprisingly, the shared memory implementation is significantly faster.

These throughput measures can be used to determine how much overhead the use of heartbeats will add to an application. For example, consider an application that anticipates a heart rate of 200 beats per second. Adding the shared memory based implementation of heartbeats will add 1/1500000s to each beat, for an overhead of approximately .01%. If instead, we used the file-based implementation of the API, we would expect an overhead of approximately 18.5%. Knowing these values allows applications developers to make informed decisions about the placement of Heartbeats within their applications.

We test the heartbeat latency of our implementations using two applications which "ping-pong" heartbeats between each other. Both applications emit heartbeats while reading the other application's heartbeat data. The first application sends a heartbeat and then waits to see the second application register a heartbeat with the same tag. The second application works similarly, waiting for the first application and then emitting a heartbeat. We measure the time taken from when the first application sends its heartbeat until it detects a heartbeat with the same tag from the second application. We measure this value 10000 times and take the average. This average value represents the time taken to transmit two heartbeats (from the first application to the second and from the second back to the first) so we divide the time in half to obtain the heartbeat latency.

The values for heartbeat latency are also shown in Table 2. Knowing these values can aid the development of autonomic system services as heartbeat latency represents the minimum time required for the heartbeat data generated in an application to reach the service that is requesting this data. This also represents the minimum amount of time required for any change in behavior to be reflected in the heartbeat.

For the file-based implementation, there is a tradeoff between heartbeat throughput and heartbeat latency. The throughput could be increased by buffering several heartbeats and writing multiple heartbeats worth of data to the file. This will decrease the overhead of file i/o in the application but will delay the ability of an external process from reading this data. In fact, it would cause multiple heartbeats to appear to an external observer simultaneously. We have therefore chosen a file i/o implementation which minimizes heartbeat latency. Applications are free to reduce heartbeat overhead by registering heartbeats less often and making corresponding adjustments to their desired heart rates.

We have made the programs we use to test heartbeat throughput and latency available with our reference implementations both as examples and so that users can determine these values on their own systems.

Table 3: Heartbeats in PARSEC Benchmarks

Benchmark	Heartbeat Location	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

4.2 Heartbeats in PARSEC Benchmarks

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, determine what constitutes a single input to the benchmark, find a loop in the benchmark that corresponds to these inputs, and insert the heartbeats in this loop. Table 3 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[2]. We note that placement of heartbeats is flexible and can be tailored to the specific needs of the application. Comparing the values in Table 2 and Table 3 indicates that we should expect both of our Heartbeat implementations to be low overhead for most of these benchmarks.

Adding heartbeats to the PARSEC benchmark suite is easy, even for the authors who were initially unfamiliar with the benchmark implementations. 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 implementations are low-overhead for the variety of different computations represented by the PARSEC benchmarks. The next section provides an example of using the Heartbeats framework to develop an adaptive application.

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

4.3 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 [28] 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.

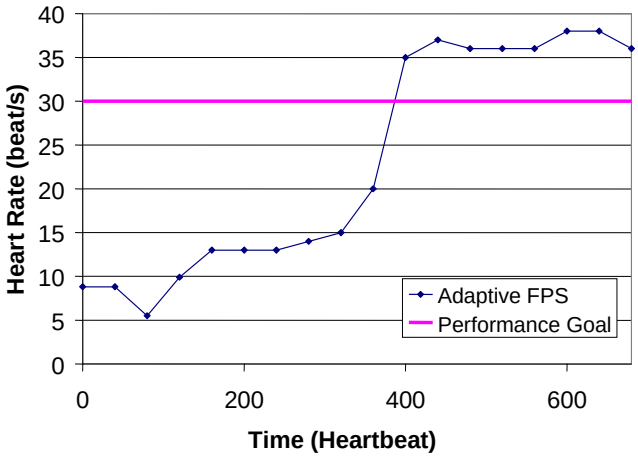
For this experiment, x264 is launched with a computationally demanding set of parameters for Main profile H.264 encoding. Both the input parameters and the video used here are different than the PARSEC inputs; both are chosen to be more computationally demanding and more uniform. The parameters include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

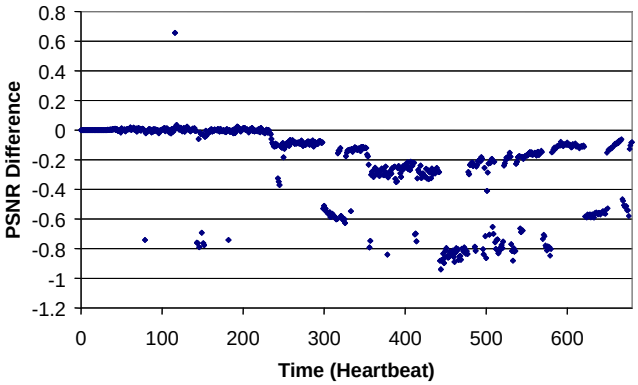
Figures 2(a) and 2(b) illustrate the behavior of this adaptive version of x264 as it attempts to reach its target heart rate of 30 beats per second. The first figure shows the average heart rate over the last 40 frames as a function of time (time is measured in heartbeats or frames). The second figure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

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.



(a) Heart rate



(b) Image Quality

Figure 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.

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. The second video shows the heart rate of the encoder with adaptation[3].

[3]Both videos are available by following the YouTube link from our website: http://groups.csail.mit.edu/carbon/heartbeats.

4.4 External Heartbeat Usage

In this example, Heartbeats are used to help an external system allocate resources while maintaining required application performance, demonstrating how the Heartbeats interface can be used to perform constrained optimization. In this case, as external resource allocator tries to minimize the number of cores used by an application while maintaining the applications minimum desired heartrate. The additional cores can either be turned off, using clock gating to save power, or allocated to other computations. 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`.

4.4.1 *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.

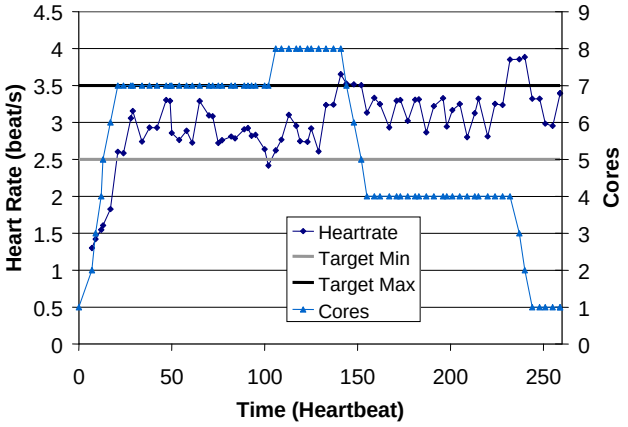


Figure 3: Behavior of `bodytrack` coupled with an external scheduler.

The behavior of `bodytrack` under the external scheduler is illustrated in Figure 3. 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 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.

4.4.2 *streamcluster*

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.

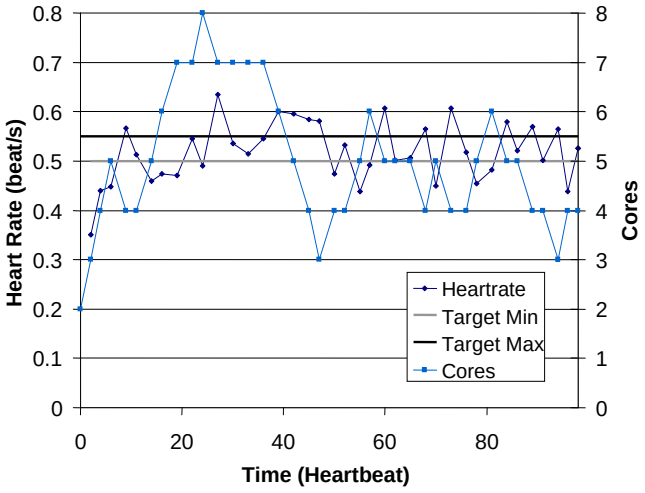


Figure 4: Behavior of `streamcluster` coupled with an external scheduler.

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 4. 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.

4.4.3 *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 5 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

[4]Also available from our website.

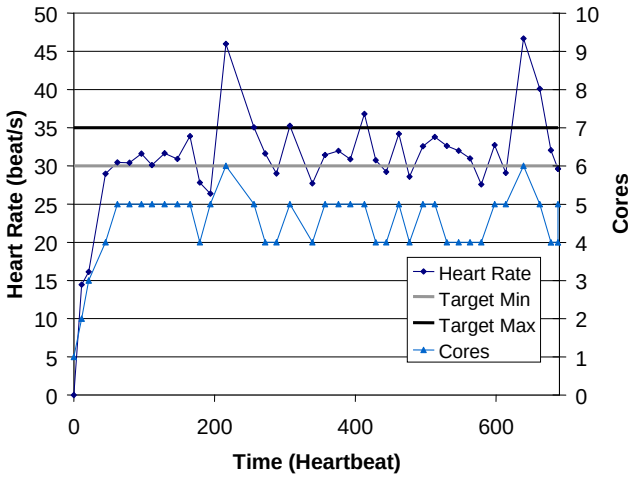


Figure 5: Behavior of x264 coupled with an external scheduler.

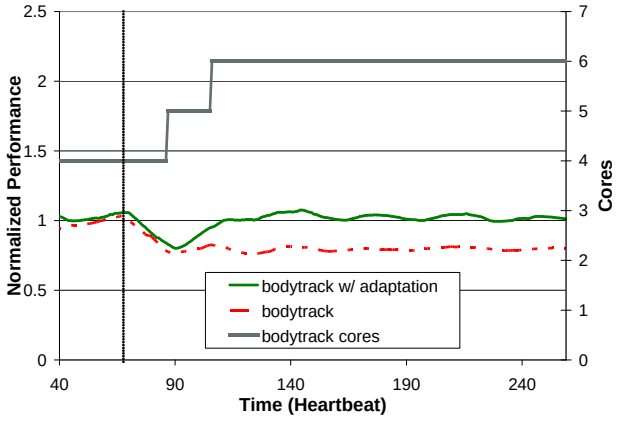
data and adapt their behavior to meet the application's needs without having to infer these needs by measuring low-level hardware events. Furthermore, the Heartbeats interface makes it easy for an external service to quantify its effects on application behavior. In this example, an external scheduler 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.

4.5 Adapting for Multiple Applications

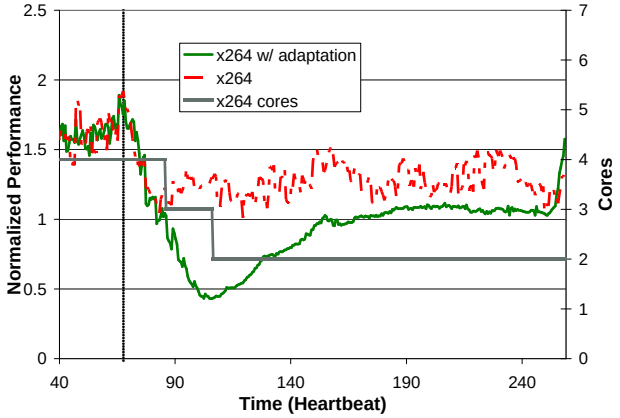
This example illustrates how heartbeats can help autonomic systems make informed decisions when adapting their behavior in the face of a changing computing environment. In this scenario the system must adapt to an abrupt change in the clock frequency of the processor on which it is executing. In addition, this experiment shows how an autonomic system service can balance the needs of multiple Heartbeat-enabled applications. Furthermore, this example demonstrates modularity of optimization as autonomic system services handle adapting resource allocation, while an autonomic application handles adapting its algorithms internally.

For this scenario, we run two Heartbeat-enabled applications: bodytrack and the adaptive version of x264 described above. Initially, both applications are assigned four distinct cores on our eight-core processor, and both applications request a minimum heart rate that is 90% of what they average using 4 cores. In addition to these two applications the adaptive scheduler (described above) is also running; however, in this case, bodytrack and x264 are given priorities with bodytrack holding the higher priority. This priority means that the scheduler will favor bodytrack if it determines that it cannot meet the needs of both applications. x264 is assigned lower because priority because it is adaptive and has the ability to alter its own algorithms to increase performance as needed.

During the experiment, the clock frequency of the processor is dropped from 3.16 GHz to 2.33 GHz and the system



(a) bodytrack



(b) x264

Figure 6: Behavior of bodytrack and x264 in the presence of frequency scaling. During execution chip frequency drops from 3.16 GHz to 2.33 GHz (represented by the dotted line. a) shows how the adaptive scheduler reallocates cores to bodytrack to meet target performance. b) shows how cores are deallocated from x264, but x264 adapts its internal algorithms to maintain its performance.

attempts to adapt behavior to maintain the performance of both applications despite the loss of 26% of its compute power. We compare the behavior of the autonomic system, which uses the adaptive scheduler and the adaptive version of x264 to that of a system with no adaptation.

Figures 6(a) and 6(b) represent the behavior of this system in this scenario, where Figure 6(a) illustrates bodytrack's behavior and Figure 6(b) shows that of x264. Both figures show performance (normalized to the average performance recorded for the application using four cores) on the left y-axis and the number of cores allocated to each application on the right y-axis. Time (measured in heartbeats) is shown on the x-axis. The time where frequency changes is shown by the vertical line in each graph. The performance of the applications without adaptation is shown with a dashed line, while the performance of applications with adaptation is shown with a solid line.

Figure 6(a) shows how the adaptive scheduler maintains

bodytrack's performance despite the loss in compute power. When the system detects the performance loss it begins to deallocate cores from x264, because it has lower priority, and allocate them to bodytrack. Once the scheduler has allocated two additional cores to bodytrack, that application is able to meet its target performance and the scheduler stops. As shown in the figure, without adaptation bodytrack would only achieve 80% of its desired performance.

Figure 6(b) shows how the adaptive scheduler sacrifices x264's performance to meet the needs of bodytrack. The adaptive scheduler deallocates cores from x264 because it has a lower priority. Thus, it is up to x264 to adapt its algorithms and sacrifice some encoding quality to maintain performance despite the loss of both cores and processor frequency. As shown in the figure, x264 is able to adapt its behavior to meet its performance goals.

We note that without adaptation, bodytrack would miss its performance goals while x264 would exceed these goals despite the fact that x264 has lower priority. By using Heartbeats to adapt its behavior, the scheduler is able to take cores from the low priority application to meet the needs of the higher priority one. The low priority process is then able to adapt its own behavior to meet its goals despite the loss of resources.

This example illustrates the use of Heartbeats to help an autonomic system maintain performance goals in the face of a changing computing environment. This behavior is possible because the scheduler can directly measure the application's performance goals and its current performance. We note that this system does not detect the clock frequency change directly, but instead detects the change in performance reflected in the change in the application's Heartbeats. Thus this system could respond to any change that altered the performance of the component applications. Finally, the scheduler presented in this scenario uses a heuristic based on priority, but Heartbeats could be used to enable a more sophisticated scheduling policy. For example, the Smartlocks framework uses Heartbeats as a reward function to a machine learning engine that determines a scheduling policy for lock acquisition [10].

5. RELATED WORK

For several applications performance monitoring is a crucial problem whose solution has been addressed by a variety of different approaches. This work includes research on monitoring single- and multi-core architectures [19, 16, 2], networks [27], adaptive [1] and dynamic [5] compilation techniques and operating systems [25, 6, 8, 7, 18, 26, 24]. Considerable work has been done using tracing to understand kernel performance (*e.g.,*K42 [26], KernInst [24]) and profiling frameworks, such as Oprofile [20] and gprof [11],are in wide use today. Most of this work focuses on off-line collection and visualization of performance data, rather than supporting online frameworks capable of making decisions at runtime.

A software approach for application monitoring is proposed in [25]. This work presents 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, 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 external autonomous services to help them meet their goals.

The rise of adaptive computing systems creates new challenges and demands for system monitoring [9]. Major companies such as IBM [14] (IBM Touchpoint Simulator, the K42 Operating System [18, 7]), Oracle (Oracle Automatic Workload Repository [21]), and Intel (Intel RAS Technologies for Enterprise [15]) have all invested effort in adaptive computing. One example of these emerging adaptive systems can be found in the self-optimizing memory controller described in [17]. 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 might 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 both for the application writer who adds heartbeat calls to the program and for those auto-tuning tools which might want to read the heartbeat such as Orio [12], Autopilot [22], and Active Harmony [13]. The fact that these adaptive systems could all be built on top of the Heartbeat interface demonstrates the API's usefulness.

An adaptive approach which aims to characterize and understand the interactions between hardware and software and to optimize them based on those characterizations has been presented in [7, 3]. This work led to an architecture for continuous program optimization (CPO) [7] which can help automate the challenging task of performance tuning a complex system. CPO agents utilize the data provided by the performance and environment monitoring (PEM) infrastructure to detect, diagnose, and eliminate performance problems [3]. The approach taken by the PEM and CPO projects is promising, but differs from the Heartbeats approach in several respects: CPO uses an efficient multi-layer monitoring system, but it is not able to support multiple optimizations and all the measurements are compared to predefined and expected values. This agent-based framework, combined with the performance and environment monitoring infrastructure stands in contrast to the lightweight approach proposed by the Heartbeats solution. Moreover, the CPO and PEM infrastructure requires a separate instrumentation phase which is unnecessary with the Heartbeats approach.

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 several applications of the framework: dynamically reducing output quality (accuracy) as necessary to meet a throughput (performance) goal and optimizing system resource allocation by minimizing the number of cores used to reach a given target output rate. In addition we have shown how Heartbeats can enable an autonomic system to adapt to a changing computing environment. We believe that a unified, portable standard for application performance monitoring, which incorporates application's performance goals, is crucial for a broad range of future applications.

7. REFERENCES

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In *Conf. on Programming Language Design and Implementation*, Jun 2009.

[2] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th Inter. Conf. on Supercomputing*, pages 101–110, 2005.

[3] A. Baumann, D. D. Silva, O. Krieger, and R. W. Wisniewski. Improving operating system availability with dynamic update. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 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.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on code generation and optimization*, 2003.

[6] M. Caporuscio, A. Di Marco, and P. Inverardi. Run-time performance management of the siena publish/subscribe middleware. In *WOSP '05: Proc. of the 5th Inter. Work. on Software and performance*, pages 65–74, 2005.

[7] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM J. Res. Dev.*, 50(2/3):239–248, 2006.

[8] L. A. De Rose and D. A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *Inter. Conf. on Parallel Processing*, 1999.

[9] P. Dini, W. Gentzsch, M. Potts, A. Clemm, M. Yousif, and A. Polze. Internet, GRID, self-adaptability and beyond: Are we ready? In *Proceedings of the 15th International Workshop on Database and Expert Systems Applications* Aug 2004.

[10] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal. Smartlocks: Self-aware synchronization through lock acquisition scheduling. In *SMART 10: 4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion*, Jan 2010.

[11] Free Software Foundation Inc. GNU gprof, 1993.

[12] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IPDPS '09: Proc. of the Inter. Symp. on Parallel&Distributed Processing*, 2009.

[13] J. Hollingsworth and P. Keleher. Prediction and adaptation in active harmony. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 180–188, Jul 1998.

[14] IBM Inc. IBM autonomic computing website, 2009.

[15] Intel Inc. Reliability, availability, and serviceability for the always-on enterprise, 2005.

[16] Intel Inc. Intel itanium architecture software developer's manual, 2006.

[17] E. Ipek, O. Mutlu, J. F. MartŠnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08: Proc. of the 35th Inter. Symp. on Comp. Arch.*, 2008.

[18] O. Krieger, M. Auslander, B. Rosenburg, R. W. J. W., Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *EuroSys '06: Proc. of the 1st ACM SIGOPS/EuroSys Euro. Conf. on Computer Systems*, 2006.

[19] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Processor power reduction via single-isa heterogeneous multi-core architectures. *Computer Architecture Letters*, 2(1):2–2, Jan-Dec 2003.

[20] J. Levon and P. Elie. OProfile - A System Profiler for Linux, 2009.

[21] Oracle Corp. Automatic Workload Repository (AWR) in Oracle Database 10g.

[22] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: adaptive control of distributed applications. In *High Performance Distributed Computing*, Jul 1998.

[23] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[24] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI '99: Proc. of the third symp. on Operating systems design and implementation*, 1999.

[25] J. Vetter and P. Worley. Asserting performance expectations. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 33–33, Nov. 2002.

[26] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC '03: Proc. of the ACM/IEEE conf. on Supercomputing*, Nov 2003.

[27] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[28] x264. Online document, `http://www.videolan.org/x264.html`.