Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2009-035 August 7, 2009

Application Heartbeats for Software Performance and Health

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

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Application Heartbeats for Software Performance and Health

Henry Hoffmann Jonathan Eastep Marco Santambrogio Jason Miller Anant Agarwal

Massachusetts Institute of Technology, Cambridge, MA

{hank,eastepjm,santa,jasonm,agarwal}@csail.mit.edu

Abstract

Adaptive, or self-aware, computing has been proposed as one method to help application programmers confront the growing complexity of multicore software development. However, existing approaches to adaptive systems are largely ad hoc and often do not manage to incorporate the true performance goals of the applications they are designed to support. This paper presents an enabling technology for adaptive computing systems: Application Heartbeats. The Application Heartbeats framework provides a simple, standard programming interface that applications can use to indicate their performance and system software (and hardware) can use to query an application's performance. Several experiments demonstrate the simplicity and efficacy of the Application Heartbeat approach. First the PARSEC benchmark suite is instrumented with Application Heartbeats to show the broad applicability of the interface. Then, an adaptive H.264 encoder is developed to show how applications might use Application Heartbeats internally. Next, an external resource scheduler is developed which assigns cores to an application based on its performance as specified with Application Heartbeats. Finally, the adaptive H.264 encoder is used to illustrate how Application Heartbeats can aid fault tolerance.

1. Introduction

As multicore processors become increasing 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-aware* hardware and software. Self-aware systems take some of the burden off of programmers by monitoring themselves and optimizing or adapting as necessary to meet their goals. Interest in such systems has been increasing recently and they have been variously called adaptive, self-tuning, self-optimizing, autonomic, and organic. Self-aware techniques apply to a broad range of systems including: embedded, real-time, desktop, server, and even cloud systems.

Unfortunately, there exist no standardized, universal ways for applications and systems to measure how well they are meeting their goals. Existing approaches are largely ad hoc: either hand-crafted for a particular computing platform 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 just spinning on a lock. Measuring 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 be adapted to make sure the goals are met. Application-specific goals may include throughput, power, latency, output quality, or combinations thereof. Application Heartbeats can also help provide fault tolerance by providing information that can be used to predict or quickly detect failures.

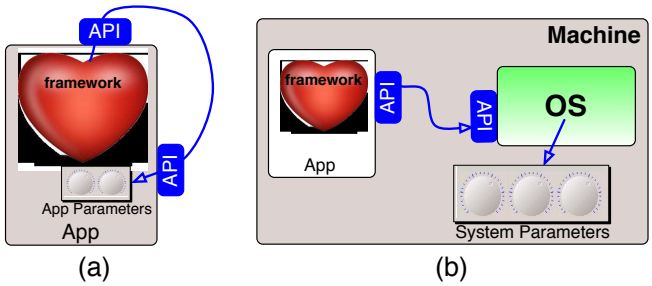

Figure 1. (a) Self-optimizing application using the Application Heartbeats framework. (b) Optimization of machine parameters by an external observer.

This paper makes the following contributions:

1. A simple, standardized Heartbeats API for monitoring application-specific performance metrics.
2. A basic reference implementation of the API which is available as open-source software.
3. 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.

Having a simple, standardized API makes it easy for programmers to add Heartbeats to their applications. It also crucial for portability and interoperability between different applications, runtime systems, and operating systems. Registering goals with external systems enables optimizations that are unavailable within an application such as modifying scheduling decisions or adjusting hardware parameters. When running multiple Heartbeat-enabled applications, it also allows system resources (such as cores, memory, and I/O bandwidth) to be reallocated to provide the best global outcome.

The Application Heartbeats framework measures application progress toward goals using a simple 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 externally-driven optimization. The API allows applications to communicate their goals by setting a target heart rate (*i.e.,*number of heartbeats per second). The operating system, runtime system, hardware, or the application itself monitors progress through additional API calls and takes appropriate actions to help the application achieve those goals.

This paper presents a reference implementation of the Application Heartbeats framework and demonstrates several approaches to adding self-aware optimization to a system. First, we add heartbeats to the PARSEC benchmark suite to show that they are easy to add to existing applications. Second, an H.264 video encoder is used to show an application modifying itself. The encoder automatically adjusts its encoding parameters and algorithms to provide the best quality possible while maintaining a required minimum frame rate. Third, three benchmarks from the PARSEC suite [Bienia et al. 2008] are used to show optimization by an external observer. Here, the OS monitors the application's heart rate (heartbeats per second) and dynamically adjusts the number of cores allocated to maintain the required throughput while minimizing resource usage. Finally, heartbeats are used to keep application performance within a required window in the face of simulated core failures.

The rest of this paper is organized as follows. Section 2 identifies key applications that will benefit from the Application Heartbeats framework. Section 3 describes the Application Heartbeats API in greater detail. Section 4 describes our initial reference implementation of the API. Section 5 presents our experimental results. Section 6 compares Application Heartbeats to related work. Finally, Section 7 concludes.

2. Applications

The Application Heartbeats framework is a simple end-to-end feedback mechanism that can potentially have a far-reaching impact on future computer design, software systems and applications, and service ecosystems. This section explores ideas for novel computer architectures, software libraries, operating systems, and runtime environments built around the Heartbeats framework. Additionally, it identifies existing applications and services that Heartbeats would improve such as system administrative tools and cloud computing.

2.1 Self-tuning Architecture

Heartbeats is designed to enable hardware to inspect application heartbeat statistics. The information within a heartbeat and the information implied by hearbeat trends can be used in the design of self-optimizing multicore microarchitectures. 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 optimizations focus on aspects of execution most important to meeting application goals. Measuring the number of instructions executed per second does not capture whether or not those instructions are progress toward the application goal or just spinning on a lock.

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 [Kumar et al. 2003]. The heartbeat provides a natural mechanism for selecting the most energy-efficient core that meets the required performance. 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. [Govil et al. 1995, Pering et al. 1998] are examples of frequency and voltage scaling to reduce power.

2.2 Self-tuning Software Libraries

Heartbeats can be incorporated in adaptive software libraries, both general-purpose such as STAPL [Thomas et al. 2005] and domain-specific. A library can use the framework to tune its implementation to the host architecture on which it is running. Additionally, it can tune data structure and algorithm choices to high-level behaviors and goals of the applications using the library. For example, consider a place and route application such as those used in CAD tool suites [Betz and Rose 1997, Karro and Cohoon 2001, Cadence Inc. 2009]. Suppose the application uses a domain specific library to approximate an optimal place and route. Because the algorithm is approximate, its data structures and algorithms have a degree of freedom in their internal precision that can be manipulated to maximize performance while meeting a user-defined constraint for how long place and route can run.

2.3 System Administrative Tools

Heartbeats can be incorporated into system administrative tools such as DTrace, a tool for Sun's Solaris operating environment [Sun Microsystems Inc. 2009]. DTrace is an example of a comprehensive dynamic tracing framework that provides infrastructure for examining the behavior of the system and user programs, permitting administrators, developers, and service personnel to diagnose problems or tune parameters to improve performance in the field. In this context, heartbeats might be used to detect application hangs or crashes, and restart the application. Heartbeats also provide a way for an external observer to monitor which phase a program is in for the purposes of profiling or field debugging.

2.4 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.

2.5 Organic Runtime Environments

Heartbeats also finds novel application in runtime environments, giving them "organic" characteristics such as goal-oriented execution, introspection, and self-balancing. Heartbeats can be used as a feedback mechanism for guaranteeing applications like video, audio, wireless, and networking meet real-time deadlines. Heartbeats might be used to notify a runtime system or gateway when an application has consumed data from its input queue. Lastly, heartbeats can be used to mediate a work queue system [Ghosh and Muthukrishnan 1994, Levine and Finkel 1990], providing better load-balancing between workers (especially if workers have asymmetric capabilities). An Organic Runtime Environment would use heartbeats to monitor worker performance and send approximately

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

the right amount of work to its queue, potentially improving upon load-balancing schemes such as work-stealing.

2.6 Cloud Computing

Cloud computing environments can benefit from the Heartbeats framework in several ways. First, applications can use heartbeat information to automatically add or subtract resources from their pool. Companies such as RightScale [RightScale Inc. 2009] already provide solutions that automatically vary the number of web servers running a website. However, adjustment decisions are based solely on machine load. Using Heartbeats would allow this application to adapt based on more relevant metrics such as average response latency. As the heart rate descreases, the load balancer would shift traffic to a different server to start up a new one. Second, heartbeats can be used to quickly detect failed (or failing) machines and fail-over to working machines. A lack of heartbeats from a particular node would indicate that it has failed, and slow or erratic heartbeats could indicate that a machine is about to fail. This early warning could allow the application to perform an orderly transfer to a new node rather than waiting for complete failure. Finally, heartbeats can be used by cloud providers to conserve resources and reduce costs. Many virtual machines allocated in clouds do not need a full machine or core. As long as their heart rates are meeting their goals, these "light" VMs can be consolidated onto a smaller number of physical machines to save energy and free up resources. Only when an application's demands go up and its heart rate drops, will it need to be migrated to dedicated resources.

3. Heartbeats API

Since heartbeats are meant to reduce programmer effort, it must be easy to insert them into programs. 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 encoder discussed previously 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. Our initial version of the API left the window entirely up to the observer, specifying it only when asking for the current heart rate. This would allow the observer to adjust the window size based on the frequency with which it can make its optimizations. For example, the application itself may want a short window if it is adjusting a parameter within its algorithms. On the other hand, a cloud manager might want a very large window if it is considering migrating the application to a faster machine.

However, we realized that, in some cases, the application has a natural window size that may not be known by an external observer. Therefore, the API also allows the application to specify a default window size when it calls *HB_initialize*. Calls to *HB_current_rate* with a value of zero passed for the window will use the default window. Implementations of the Heartbeat API may wish to restrict the maximum window size to limit the resources used to store heartbeat history. However, whenever possible, they should store at least as much history as the default window size requested by the application. If window values larger than the default are passed to *HB_current_rate* they may be silently clipped to the default value.

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. Therefore the private buffers may be stored anywhere but the global buffer must be in a universally accessible location such as coherent shared memory or a disk file.

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. Implementation

This section describes our initial reference implementation of the Heartbeats API. This implementation is intended to provide the basic functionality of the Application Heartbeats framework but is not necessarily optimized for performance or resource utilization. It is written in C and is callable from both C and C++ programs. Our implementation supports all functions listed in Table 1.

When the *HB_heartbeat* function is called, a new entry containing a timestamp, tag and thread ID is written into a file. One file is used to store global heartbeats. When per-thread heartbeats are used, each thread writes to its own individual file. A mutex is used to guarantee mutual exclusion and ordering when multiple threads attempt to register a global heartbeat at the same time. When an external service wants to get information on a Heartbeat-enabled program, the corresponding file is read. The target heart rates are also written into the appropriate file so that the external service can access them. This implementation does not support changing the target heart rates from an external application. The *HB_get_history* function can support any value for n because the entire heartbeat history is kept in the file; however, implementations that are more concerned with performance and storage utilization may restrict the number of heartbeats that can be returned by this function.

5. 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 [Bienia et al. 2008]. 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.

Table 2. Heartbeats in the PARSEC Benchmark Suite

Benchmark	Heartbeat Location	Average Heart Rate
`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

5.1 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). Table 2 shows the summary of this work. For each benchmark the table shows where the heartbeat was inserted 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].

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 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 only exception is the `blackscholes` benchmark which processes ten million options. For this benchmark, a conditional statement was added so that the heartbeat is registered every 25,000 options. This reduces overhead and makes the output data processing more manageable.

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.

Using the Heartbeats interface can provide additional insight into the performance of these benchmarks beyond that provided by just measuring execution time. For example, Figure 2 shows a moving average of heart rate for the `x264` benchmark using a 20 beat window (a heartbeat is registered as each frame is processed). The chart shows that `x264` has several distinct regions of performance when run on the PARSEC native input. The first occurs in the first 100 frames where the heart rate tends to the range of 12-14 beats per second. Then, between frames 100 and 330 the heart rate jumps to the range of 23-29 beats per second. Finally, the heart

[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`.

rate settles back down to its original range of 12-14 beats per second. In this example, the use of Heartbeats shows distinct regions of performance for the `x264` benchmark with the native input size. This information can be useful for understanding the performance of certain benchmarks and optimizing these benchmarks on a given architecture. Such regions would be especially important to detect in an adaptive system.

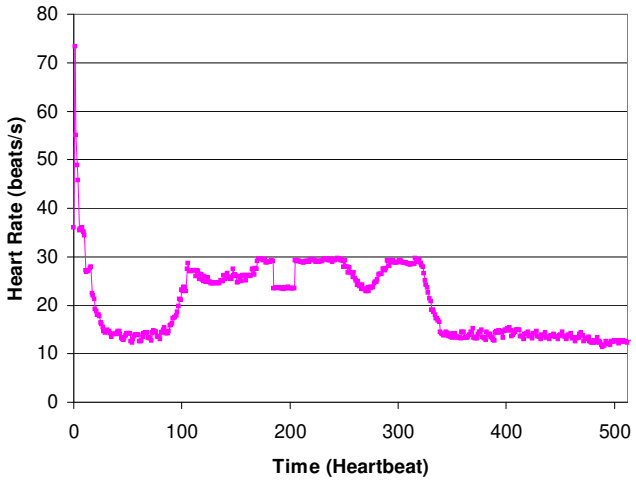

Figure 2. Heart rate of the `x264` PARSEC benchmark executing native input on an eight-core x86 server.

In summary, the Heartbeats framework is easy to insert into a broad array of applications. The reference implementation is low-overhead. Finally, heartbeats can provide much more information about the performance of a benchmark than end-to-end execution time. The next section provides an example of using the Heartbeats framework to develop an adaptive application.

5.2 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 [x264] 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 codebase 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 3 and 4 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.

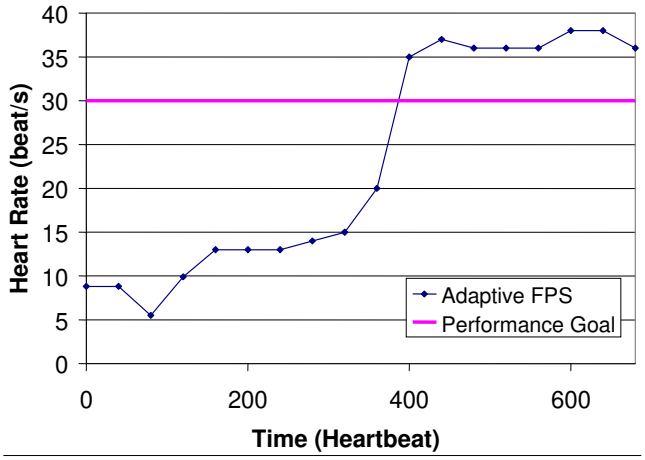

Figure 3. Heart rate of adaptive x264.

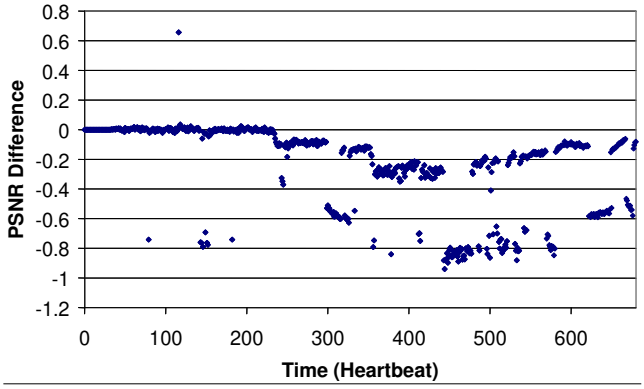

Figure 4. Image quality (PSNR) of adaptive x264. The chart shows the difference in PSNR between the unmodified x264 code base and our adaptive version.

As shown in Figure 3 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 4, 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. The next example demonstrates how the Heartbeats API can be used by an external application.

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]. [Note to reviewers: we will move these videos from YouTube and make them available through our project web site after the anonymous review process.]

5.3 External Heartbeat Usage

In this example Heartbeats is used to help an external system allocate system 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`.

5.3.1 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.

The behavior of `bodytrack` under the external scheduler is illustrated in Figure 5. 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.

5.3.2 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. 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 6. 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.

5.3.3 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.

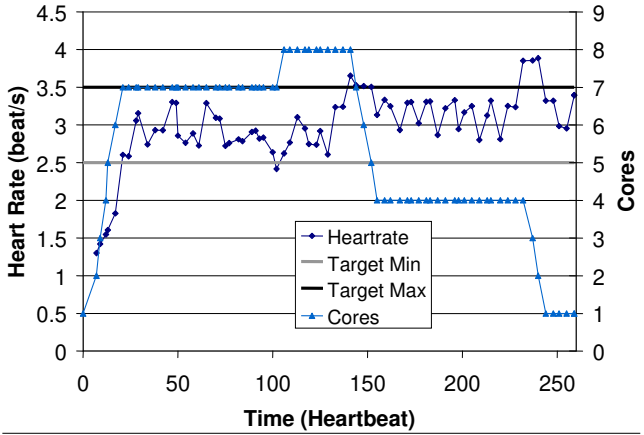

Figure 5. Behavior of `bodytrack` coupled with an external scheduler.

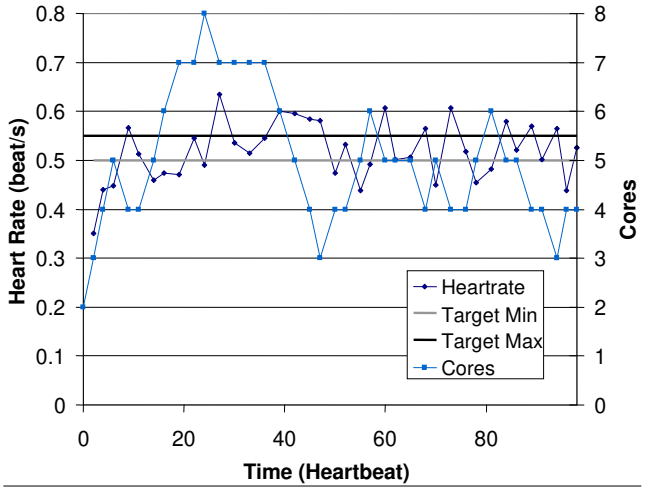

Figure 6. Behavior of `streamcluster` coupled with an external scheduler.

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

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

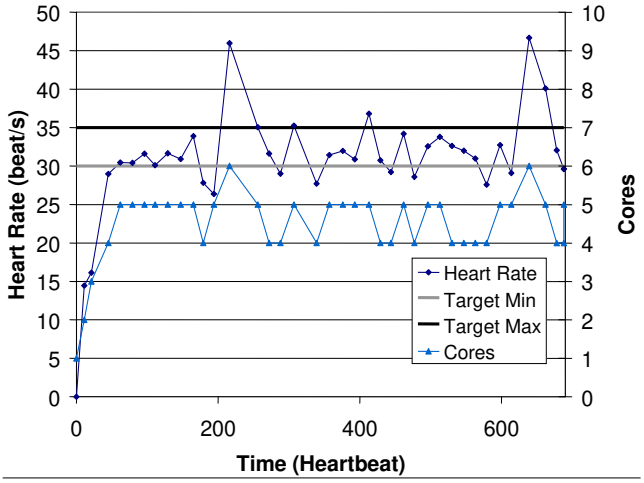

Figure 7. Behavior of `x264` coupled with an external scheduler.

Figure 7 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.

5.4 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 5.2. 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 8. 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 "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.

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

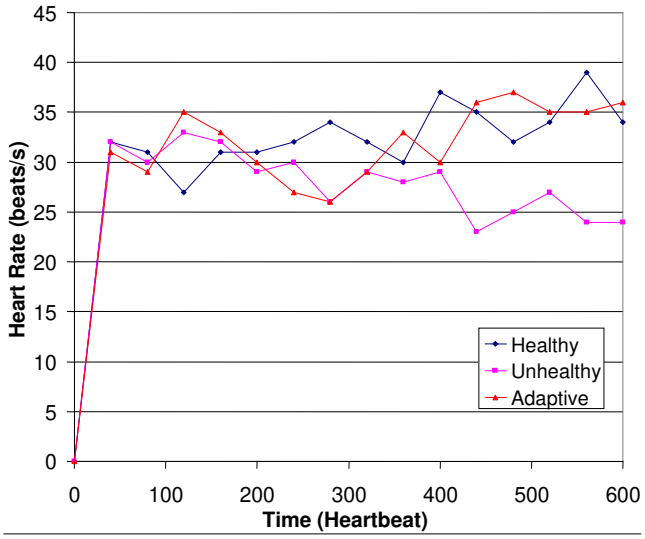

Figure 8. Using Heartbeats in an adaptive video encoder for fault tolerance. The line labeled "Healthy" shows the performance of the encoder under normal circumstances. The line labeled "Unhealthy" shows the performance of the encoder when cores fail. The line labeled "Adaptive" shows the performance of an adaptive encoder that adjusts its algorithm to maintain a target heart rate of greater than 30 beats/s.

Figure 8 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.

6. 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 GRID, cloud computing, and web services [Vadhiyar and Dongarra 2005, Buisson et al. 2005, Reinecke and Wolter 2008], single- and multi-core architectures [Sprunt 2002b, Kumar et al. 2003, Intel Inc. 2006, Azimi et al. 2005], and complex software systems and operating systems [Caporuscio et al. 2005, De Rose and Reed 1999, Cascaval et al. 2006, Krieger et al. 2006, Wisniewski and Rosenburg 2003, Tamches and Miller 1999]. Considerable work has been done using tracing to understand kernel performance

(*e.g.,*K42 [Wisniewski and Rosenburg 2003], KernInst [Tamches and Miller 1999]) and application behavior (*e.g.,*Paradyn [Miller et al. 1995], SvPablo [De Rose and Reed 1999]). Profiling frameworks, such as Oprofile [Levon and Elie 2009], a Linux profiler for system-wide sampling of hardware performance counters, and gprof [Free Software Foundation Inc. 1993],are in wide use today. Most of this work focuses on off-line collection and visualization.

More complex monitoring techniques have been presented in [Schulz et al. 2005, Wisniewski and Rosenburg 2003]. This work represents a shift in approach as the research community moves from using simple aggregate metrics, *i.e.,*cache miss rate, to more advanced statistics such as reuse distance [Ding and Zhong 2001] and predictability [Duesterwald et al. 2003]. 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 [Sprunt 2002a]: too few counters, sampling delay, and lack of address profiling.

Some approaches [Sprunt 2002b] address these deficiencies; however, they still suffer in that they can only be applied to collect aggregate statistics using sampling. It is not possible to react to some rare but high-impact events. The Itanium processor family [Intel Inc. 2006], overcomes this limitation by introducing microarchitectural event data. This data is delivered to the consuming software through an exception triggered by each event, a solution that can cause frequent interrupts on the processor that is consuming the event data. Furthermore these methods are not able to help in detecting and avoiding infinite loops and deadlocks.

A more extensive technique for system monitoring is presented in [Schulz et al. 2005]. Its design is based on the use of reconfigurable logic, *i.e.,*FPGAs, to implement hardware monitors. These monitors are located at different event sources, *e.g.,*memory buses, and update the content of the monitors according to their specific location. This approach has a major drawback in that it places great demands on the underlying architecture. An additional problem with all of these hardware solutions is that they are very architecture specific and it is therefore difficult to write code that is portable between architectures.

The rise of adaptive and autonomic computing systems (designed to configure, heal, optimize, and protect themselves without human intervention) creates new challenges and demands for system monitoring [Dini 2004]. Major companies such as IBM [IBM Inc. 2009] (IBM Touchpoint Simulator, the K42 Operating System [Krieger et al. 2006]), Oracle (Oracle Automatic Workload Repository [Oracle Corp.]), and Intel (Intel RAS Technologies for Enterprise [Intel Inc. 2005]) have all invested effort in adaptive computing.

An example of an adaptive approach, related to the IBM K42 Operating System [Krieger et al. 2006], is proposed in [Azimi et al. 2004, Cascaval et al. 2006, Baumann et al. 2004]. This work aims to characterize and understand the interactions between hardware and software and to optimize based on those characterizations. This work led to an architecture for continuous program optimization (CPO) [Cascaval et al. 2006] 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 [Baumann et al. 2004]. An agent-based system has been developed to create a competitive scenario where agents negotiate for resources affecting the performance of the applications. To be efficiently used, the CPO framework needs to be instrumented with the information on what data needs to be collected and what events need to be detected. Once the CPO is properly instrumented, it can be used to understand performance events and improve performance prior to, during, and across different executions of the applications.

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 pre-defined 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.

Another example of these emerging adaptive systems can be found in the self-optimizing memory controller described in [Ipek et al. 2008]. 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.

Finally, the Application Heartbeats API should not be confused with the Heartbeat daemon and API which provides cluster management services as part of the High-Availability Linux (Linux-HA) project. Nor should it be confused with the Heartbeat synchronization API used in the LIGO (Laser Interferometer Gravitational-Wave Observatory) diagnostics software. These projects are similar to Application Heartbeats in name only.

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.

7. 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.

Acknowledgements

This work is supported by DARPA, NSF, and Quanta Computer.

References

R. Azimi, C. Cascaval, E. Duesterwald, M. Hauswirth, K. Sudeep, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for whole-system characterization and optimization. pages 15–24, 2004.

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 annual international conference on Super-*

computing, pages 101–110, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: http://doi.acm.org/10.1145/1088149.1088163.

A. Baumann, D. Da Silva, O. Krieger, and R. W. Wisniewski. Improving operating system availability with dynamic update. In *In Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, pages 21–27, 2004.

V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 213–222, London, UK, 1997. Springer-Verlag. ISBN 3-540-63465-7.

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 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2008.

J. Buisson, F. André, and J. L. Pazat. Dynamic adaptation for grid computing. *Lecture Notes in Computer Science. Advances in Grid Computing - EGC*, pages 538–547, 2005.

Cadence Inc. SoC Encounter System, 2009. URL `http://www.cadence.com/products/di/soc_encounter`.

M. Caporuscio, A. Di Marco, and P. Inverardi. Run-time performance management of the siena publish/subscribe middleware. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 65–74, New York, NY, USA, 2005. ACM. ISBN 1-59593-087-6. doi: http://doi.acm.org/10.1145/1071021.1071028.

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. ISSN 0018-8646. doi: http://dx.doi.org/10.1147/rd.502.0239.

L. A. De Rose and D. A. Reed. SvPablo: A multi-language architecture-independent performance analysis system. In *International Conference on Parallel Processing*, pages 311–318, 1999.

C. Ding and Y. Zhong. Reuse distance analysis. Technical Report TR741, University of Rochester, Rochester, NY, USA, 2001.

P. Dini. Internet, GRID, self-adaptability and beyond: Are we ready? pages 782–788, Aug 2004. doi: 10.1109/DEXA.2004.1333571.

E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT-2003: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, Sept 2003. doi: 10.1109/PACT.2003.1238018.

Free Software Foundation Inc. GNU gprof, 1993. URL `http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`.

B. Ghosh and S. Muthukrishnan. Dynamic load balancing in parallel and distributed networks by random matchings (extended abstract). In *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 226–235, New York, NY, USA, 1994. ACM. ISBN 0-89791-671-9. doi: http://doi.acm.org/10.1145/181014.181366.

K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM. ISBN 0-89791-814-2. doi: http://doi.acm.org/10.1145/215530.215546.

IBM Inc. IBM autonomic computing website, 2009. URL `http://www.research.ibm.com/autonomic/`.

Intel Inc. Intel itanium architecture software developer's manual, 2006. URL `http://www.intel.com/design/itanium/manuals/iiasdmanual.htm`.

Intel Inc. Reliability, availability, and serviceability for the always-on enterprise, 2005. URL `www.intel.com/assets/pdf/whitepaper/ras.pdf`.

E. Ipek, O. Mutlu, J. F. Martnez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. doi: http://dx.doi.org/10.1109/ISCA.2008.21.

J. Karro and J. P. Cohoon. Gambit: A tool for the simultaneous placement and detailed routing of gate-arrays. In *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*, pages 243–253, London, UK, 2001. Springer-Verlag. ISBN 3-540-42499-7.

O. Krieger, M. Auslander, B. Rosenburg, R. Wisniewski J. W., Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 133–145, New York, NY, USA, 2006. ACM. ISBN 1-59593-322-0. doi: http://doi.acm.org/10.1145/1217935.1217949.

R. Kumar, K. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Processor power reduction via single-isa heterogeneous multi-core architectures. *Computer Architecture Letters*, 2(1):2–2, Jan-Dec 2003. ISSN 1556-6056. doi: 10.1109/L-CA.2003.6.

A. Levine and D. Finkel. Load balancing in a multi-server queuing system. *Comput. Oper. Res.*, 17(1):17–25, 1990. ISSN 0305-0548. doi: http://dx.doi.org/10.1016/0305-0548(90)90024-2.

J. Levon and P. Elie. OProfile - A System Profiler for Linux, 2009. URL `http://oprofile.sourceforge.net`.

B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, Nov 1995. ISSN 0018-9162. doi: 10.1109/2.471178.

Oracle Corp. Automatic Workload Repository (AWR) in Oracle Database 10g. URL `http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php`.

T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81, New York, NY, USA, 1998. ACM. ISBN 1-58113-059-7. doi: http://doi.acm.org/10.1145/280756.280790.

P. Reinecke and K. Wolter. Adaptivity metric and performance for restart strategies in web services reliable messaging. In *WOSP '08: Proceedings of the 7th International Workshop on Software and Performance*, pages 201–212. ACM, 2008.

RightScale Inc. Rightscale website, 2009. URL `http://www.rightscale.com/`.

M. Schulz, B. S. White, S. A. McKee, H. H. S. Lee, and J. Jeitner. Owl: next generation system monitoring. In *CF '05: Proceedings of the 2nd conference on Computing Frontiers*, pages 116–124, New York, NY, USA, 2005. ACM. ISBN 1-59593-019-1. doi: http://doi.acm.org/10.1145/1062261.1062284.

B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, Jul/Aug 2002a. ISSN 0272-1732. doi: 10.1109/MM.2002.1028477.

B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22 (4):72–82, Jul/Aug 2002b. ISSN 0272-1732. doi: 10.1109/MM.2002.1028478.

Sun Microsystems Inc. The Solaris Dynamic Tracing (DTrace), 2009. URL `http://www.sun.com/bigadmin/content/dtrace/`.

A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 117–130, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1.

N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *PPoPP '05: Proceedings of the 10th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 277–288, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: http://doi.acm.org/10.1145/1065944.1065981.

S. S. Vadhiyar and J. J. Dongarra. Self adaptivity in grid computing. *Concurr. Comput. : Pract. Exper.*, 17(2-4):235–257, 2005. ISSN 1532-0626. doi: http://dx.doi.org/10.1002/cpe.v17:2/4.

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

x264. URL `http://www.videolan.org/x264.html`.