Software Knows Best: A Case for Hardware Transparency and Measurability

by Sarah Bird

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor K. Asanović
Research Advisor

(Date)

* * * * * * *

Professor D. Patterson
Second Reader

(Date)

Abstract

Future gains in computer system performance will only come from increased parallelism and efficient execution on specialized hardware. However parallel programming is more difficult than sequential programming, which means parallel programming will only be used if it leads to improved performance or energy efficiency. Because portability and reusability are required to reduce software costs, parallel software must become performance-portable to become mainstream. In this paper, we evaluate the state of performance-portability across several current platforms and explore approaches to realizing performance-portability for future applications.

We find that appropriate hardware measurements are crucial to all our techniques, but existing detailed microarchitectural performance counters were not designed for use by application software. We provide examples that show how they fail to support the needs of an adaptive parallel software stack making the creation of performance-portable software nigh unto impossible.

We propose SHOT (Standardized Hardware Operation Tracker), which provides a standardized architecture to access to a few high-level system measurements. We argue a standardized hardware measurement system will contribute more to the success of the parallel revolution than many other proposed hardware mechanisms by enabling software to adapt to underlying hardware resources.

1 Introduction

Failure to provide substantial improvements in single-thread performance through technology scaling and microarchitectural innovation, combined with power and energy constraints, has forced an industry-wide shift to multicore architectures. It is clear that applications will only see a performance benefit from future computers if they are parallel programs — sequential programs will be slow programs. However, writing parallel programs is considerably more difficult than writing sequential programs, so the *only* reason to write parallel programs is to improve performance or energy efficiency.

Parallel programming has been mostly confined to niches in the supercomputing and embedded spaces, where hardware cost or performance constraints justify labor-intensive custom parallel application development. For parallel programming to be mainstream, application code has to be portable across different platforms, or else it will be difficult to justify development costs. Furthermore, library code has to be reusable across different applications as well as across different platforms, otherwise each application has to be written from scratch — a prohibitively expensive proposition. Since parallel programming only makes sense if it improves performance, and because portability and reusability are the best ways to reduce software costs, hardware platforms and software stacks must evolve to support *performance-portable parallel software*. Absent this, parallel programming will struggle to be economically viable for mainstream computing.

If anything, the diversity of platforms and execution environments is increasing, making performance-portability even more of a challenge. Hardware platforms range from handheld mobile systems with a few cores to server-class machines with hundreds of cores. Even within machines of the same class, differences in microarchitecture and memory hierarchy cause the "best" code to vary widely across platforms. For example, in our performance-portability experiment, we found that code tuned natively was $2.5\times$ faster than code tuned on another machine.

Platform diversity is not the only challenge for performance-portable software: the dynamic runtime environment varies widely too. What other applications are running concurrently, whether running on battery power or line power, and even the current ambient temperature, can all affect the resources available to an application. Some applications will want to adapt result quality to fit available runtime resources, but even then will want to do this portably across many platforms and runtime scenarios. However, most application developers won't want to know low-level details of any given hardware platform or be bothered with contemplating environmental effects, so the underlying operating system and runtime libraries should be responsible for adapting resources given to an application to make it perform well or use less energy. The operating system also faces a new challenge when co-scheduling multiple parallel applications, each of which can run with varying amounts of resources: how best to allocate resources across concurrently running applications?

Parallel programming will only be widely adopted if writing performance-portable parallel software becomes routine and operating systems are capable of scheduling the resulting code. In this paper, we argue that a critical step in making this a reality is for hardware platforms to implement a standardized interface to simple hardware performance statistics. We term our approach *Standardized Hardware Operation Trackers (SHOT)*. Surprisingly, despite decades of deployment, current performance counters are both ill-suited to this task and considerably more complex than what we believe is required.

2 Performance-Portability of a Single Kernel

As described in the previous section, creating efficient portable software is vital for the success of parallel programming. However, developers face many challenges to designing performance-portable software including concurrently running applications, dynamic runtime environments, and most notably a growing diversity of platforms.

We begin with a case study of a common application kernel to study the interaction between performance optimization and platform diversity. The purpose of our study is to determine how realizable performance-portability is with current platforms, so that we can better understand what is necessary for future systems.

We have chosen to use two stencil codes (27-point and 7-point) for this experiment because stencil is a common computational pattern in applications and parallel libraries across domains ranging from computational photography to blood flow simulation [6]. Additionally, its performance is not dependent upon specific inputs — only the problem size — making it an easier application to tune.

We first review the large improvements possible with hand-optimizations of the stencil code to understand what is necessary to optimize the code and the performance penalty for using unoptimized code. We next describe a performance-portability experiment that demonstrates how poorly optimizations for one platform translate to another, even for successive processors from the same manufacturer.

2.1 Importance of Performance Optimizations

A stencil calculates a function over neighbors in a multi-dimensional space. A 27-point stencil computes over all the points in a $3 \times 3 \times 3$ cube.

Figure 1 shows the impact of various optimizations when tuning a 27-point stencil. This optimization was done by a performance optimization expert. For 8 cores, the optimized code achieves nearly a $3\times$ improvement in performance over the untuned code entirely as a result of software transformations such as register blocking and padding data structures to avoid memory interference.

While it is possible to perform a few of the optimizations (such as prefetching) in hardware, general-purpose hardware cannot know the intended function of the application, ruling out such transformations as cache blocking and memory layout optimization. Conventional compilers are also ineffective at generating highly tuned code for a specific library routine compared to a performance expert, who will explore new variants of the code that are not semantically equivalent at the source code level (*e.g.,* use a completely

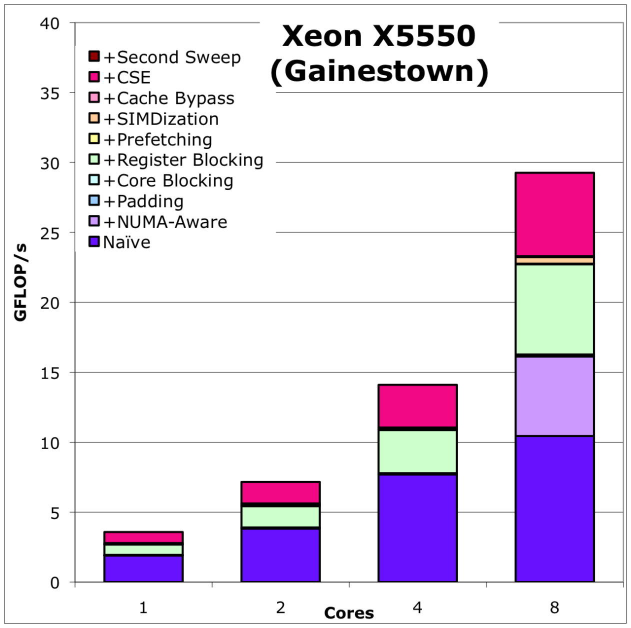


Figure 1: Impact of Various Performance Optimizations of a 27-point Stencil on Nehalem.

different algorithm or data structure) even though they produce the same final result.

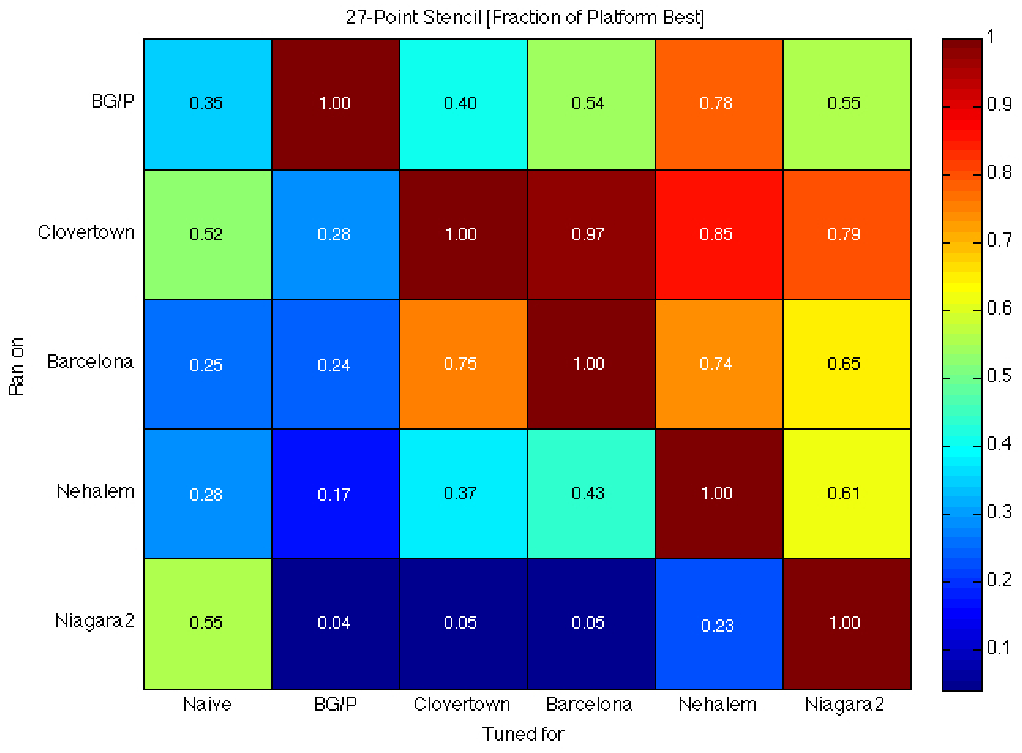
Our experience with the stencil code optimization shows that performance optimization will become increasingly important for application developers as good performance translates to efficient execution on mobile devices. Consumers will not be satisfied with applications that use up 3× the battery life or have only 1/3 the features. In the world of the iTunes Store or the Android Market, where there is rich application competition, another developer will surely develop a better implementation and gain market share. As a result, it could prove very expensive (in terms of lost revenue) to not optimize applications.

However, optimizing an application is a labor-intensive, and therefore expensive, process requiring an expert. Code reuse is necessary to keep development costs down, and so it is imperative that performance optimization effort be reusable across a variety of platforms.

2.2 Performance Portability Experiment

To test the portability of performance optimizations, our expert ran a simple performance-portability experiment using our stencil codes on five current multicore platforms.

A performance-programming expert tuned a 27-point stencil code for three x86 architectures (Intel Clovertown, Intel Nehalem, and AMD Barcelona), an IBM Blue Gene/P node, and a Sun Niagara 2. These computers had between 4 and 8 cores on a single node, with several being multithreaded. A variety of op-



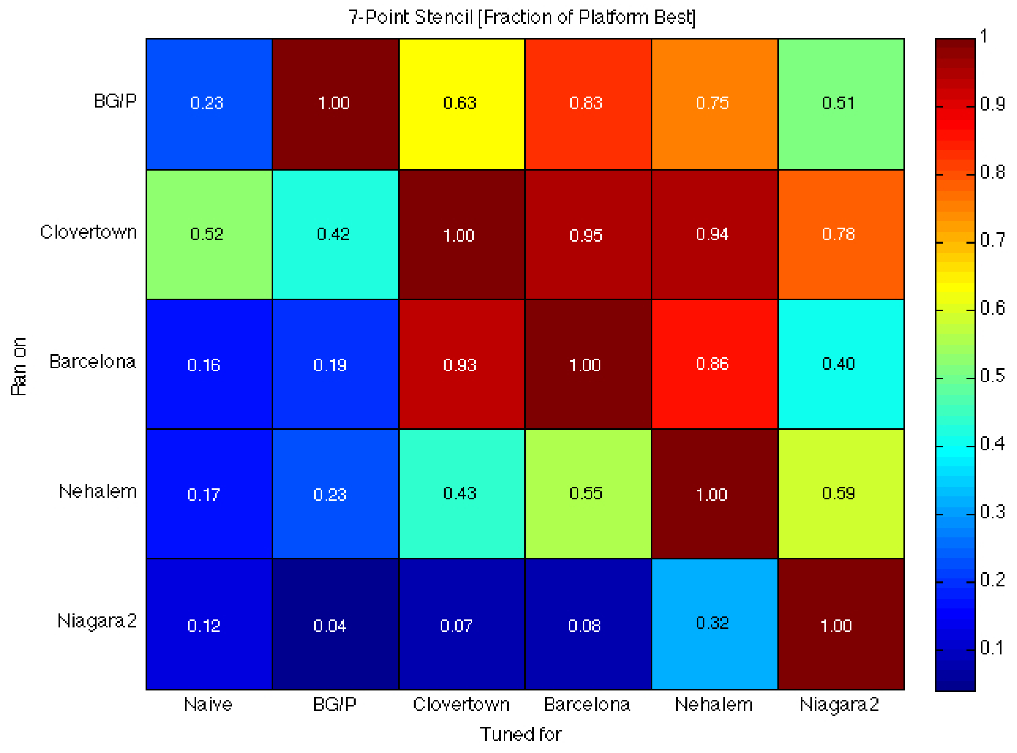
Ran on \ Tuned for	Naive	BG/P	Clovertown	Barcelona	Nehalem	Niagara2
BG/P	0.35	1.00	0.40	0.54	0.78	0.55
Clovertown	0.52	0.28	1.00	0.97	0.85	0.79
Barcelona	0.25	0.24	0.75	1.00	0.74	0.65
Nehalem	0.28	0.17	0.37	0.43	1.00	0.61
Niagara2	0.55	0.04	0.05	0.05	0.23	1.00

Figure 2: Performance Portability of a 27-Point Stencil

timizations (SIMD-ization, cache bypass, software prefetching, padding, and blocking) were tried on each platform, and the best-performing configuration for each platform was found using techniques similar to those described in [16]. Each configuration was then run on all of the other machines and compared to the best-known configuration for that machine and an unoptimized version of the program (“Naive”). Figure 2 shows the results. The experiment was repeated with a 7-point stencil; Figure 3 shows those results.

The figures are sorted in order of parallelism available. Blue Gene P has 4 threads, Clovertown and Barcelona 8, Nehalem 16 and Niagara2 128. Code is tuned to match the parallelism in the platform, which means that code tuned for BG/P will leave 1/2 the thread contexts idle when running on Clovertown and Barcelona, 3/4 on Nehalem and 31/32 on Niagara2. Clearly, this provides an upper bound on the performance available for these configurations. This bound can be observed by looking at the second column of Figure 2 where the BG/P code does not achieve above 30% of the best configuration on any machine. The naive configuration has 128 threads.

There are many observations that can be made from the figure. The leftmost column in each table shows



Ran on \ Tuned for	Naive	BG/P	Clovertown	Barcelona	Nehalem	Niagara2
BG/P	0.23	1.00	0.63	0.83	0.75	0.51
Clovertown	0.52	0.42	1.00	0.95	0.94	0.78
Barcelona	0.16	0.19	0.93	1.00	0.86	0.40
Nehalem	0.17	0.23	0.43	0.55	1.00	0.59
Niagara2	0.12	0.04	0.07	0.08	0.32	1.00

Figure 3: Performance Portability of a 7-Point Stencil

importance of optimization. Unoptimized code would run about 2× to 4× slower on the 27-point stencil and 2× to 8× slower on the 7-point stencil; the average slowdown was 3× and 5×, respectively. Clearly, that is far too much performance to give up in the name of portability.

The experiment also measured energy efficiency (floating-point operations per Joule) and found that the numbers tracked performance results closely. This 3× efficiency gap means that a user could do a third as much computation on his mobile device before the battery runs out, or the machine would be three times as expensive to power in a datacenter, compared with running the tuned application.

Squares above and to the right of the "1.00" diagonal (plus the Clovertown/Barcelona squares) represent configurations where the available parallelism in the code is equal to or greater than the number of thread contexts on the machine. Therefore these performance numbers are not a result of underutilizing hardware thread contexts. We can see that the results range from 40% to 97% of the best known configuration. The average is 69% — a sharp contrast to the 39% from the untuned naive implementation — proving that any optimization can have significant impact even when its for a different platform.

While the average improvement from optimization is good, the figure shows that careful parameter selection can still make a big difference. Code tuned for Niagara, for example, only averages 65% on other machines. However, code tuned for Nehalem averages nearly 80% on all of the other machines except Niagara where it doesn't have enough threads. These results illustrate the importance of using a natural amount of parallelism and not over decompose problems. For future applications to adjust to varying number of cores, simply parallelizing the application as much as possible may not be an efficient solution due to communication and context switching overheads.

Additionally, the practice of buying a new machine and expecting good performance out of the box will need to be reevaluated. Even an expert programmer implementing a well-understood algorithm could not obtain portable parallel performance across two platforms with subsequent generations of the same CPU family: the optimal 27-point Intel Clovertown configuration achieved only 37% of the possible performance for that application on its successor, the Intel Nehalem. This means that even simple machine upgrades will require retuning of libraries in the multicore era. This problem becomes more acute in a dynamic environment typical of a general-purpose computing system, where tuning might be required for each resource configuration (*e.g.,* number of cores) that might be encountered at run time.

Given that this is only one application kernel, it is unwise to make sweeping generalizations from these results. However, they show the importance of performance optimization and indicate a potential for creating performance-portable code.

3 Realizing Performance-Portability

Our experiments with the stencil kernel illustrated that performance-portable software may be possible with some optimization effort and careful selection of the configuration parameters. However, the stencil experiment is just a single kernel running in an isolated environment, which is still a long way from whole applications running concurrently in a dynamic execution environment. In the following sections we discuss some approaches we are considering for libraries, applications, and system software to bridge this gap between traditional performance optimization and future manycore environments. A small standard set of hardware performance measurements are common among all of the approaches discussed, and we believe they are a vital component for obtaining performance portability in general.

3.1 Performance-Portable Libraries

We begin by examining techniques to make libraries performance-portable. Most code in an application usually comes from pre-existing libraries, so making libraries performance-portable is a big step towards making whole applications performance-portable.

Obtaining the high performance shown for the stencil experiments described above took months of optimization effort by an expert. Autotuning [10, 51, 17] is a well-known technique to allow a performance expert to encapsulate their knowledge and make it portable across machines by writing a library code generator and measuring the performance of each variant. The autotuner can be parameterized to generate any instance of a whole class of library routines, *e.g.,* differing numbers of stencil points. Autotuners generate many variants of code using various combinations of optimizations known to be effective for the given library routine, then empirically measure the performance of each variant on the desired configuration of target machine to select the best approach. The primary drawback with autotuning is the time taken to search the space of code variants.

Other work [20] has shown that machine learning (ML) can substantially reduce the time taken for autotuning while still providing good results, but this technique requires support for hardware measurements. We had a machine learning expert run an experiment with a machine learning approach to see how hardware measurement could be used to lower the overhead of autotuning. Our expert programmer wrote a portable configurable stencil 27-pt and 7-pt application in C, and then, using techniques discussed in [20], ran several sample points on the Barcelona and Clovertown platforms and used a machine learning algorithm (KCCA) to select the best configuration. For the 7-pt Stencil on Barcelona and the 27-pt Stencil on Clovertown, the ML-selected configuration actually beat the best-known solution created by the expert by 1% and 15%, respectively. The other two configurations were within 3% of the expert optimized application. By combining runtime data with ML, it took less than 12 minutes to find a nearly optimal result, whereas an exhaustive search for the same space would take more than 180 days of machine time and hand-tuning by an expert can take easily as long. While 12 minutes may still be impractical to tune some libraries, it is certainly reasonable for applications that will be run repeatedly on a platform, and tuning can progress incrementally by first searching a subset of the possible space.

The experiment made use of the existing performance counters on our hardware platforms, but the type, number, and API is highly non-standard; thus, the ML-based autotuner is itself not portable. In addition, many of the events available to the counters were not useful for autotuning: ML needed only 6 and 8 of the hundreds of counters found on Barcelona and Clovertown respectively to get the highest performance. Additionally, we found that only using a time counter (cycles) was not sufficient. On Clovertown, for example, using just time as an input achieved only 64% of the best solution, whereas by adding 2 or 3 counters, ML was able to reach the best-known solution. Moreover, we believe some missing counters would have been more effective in reducing the search space since most of them represent resources that are strongly correlated with performance. Hence, a small set of standard counters could enable creation of portable autotuners that deliver high performance for portable libraries to act as the foundation of parallel applications of the future.

3.2 Adaptive Applications

In the previous section, we discussed how to write autotuned libraries that can attain high performance for a fixed computation on any platform. A different type of performance portability, common in real-time applications, is where an application adapts the computation it performs to the resources available. For example, a music application could choose to lower the quality of audio synthesis to ensure that there are no clicks in the music. Or a game could reduce the quality of rendered graphics or the realism of physics simulations to ensure a steady frame rate. Or a web server could reduce the complexity of web pages served to ensure acceptable response times under heavy load. Alternatively, if an application determines it has more resources than necessary, it may release these resources back to the operating system to preserve battery life. Our view is that such adaptivity will become more popular in mobile applications, which reward conservation with greater battery life, and for cloud applications, which reward conservation with lower power costs.

At the very least, applications will need to be able to adjust to varying core counts. The results of the performance portability experiment show that over decomposing applications is not the most efficient method. Therefore, we believe that applications should be designed to break work up into logical sized pieces where the cost of fetching a piece and communication is still small with respect to the amount of work done per piece. This size may change based on the current performance of a platform. For example, if an application is receiving very little bandwidth to DRAM, then it may need to use larger pieces of work to overlap prefetching the next section. Given that resource behavior changes dynamically based on what else is running concurrently, measurement hardware is necessary for applications to gain knowledge of the current system performance and adjust accordingly.

The application logic that controls adaptation policy will usually reside in top-level application-specific code, not in individual library modules, and will make use of global application context information. The adaptation logic executes its chosen strategy by changing the computation requested from each application component. To make adaptation portable, adaptation logic should be isolated from implementation details of each library, yet must be able to reason about the resource demands of a library routine performing a requested computation. A standardized interface for hardware performance measurement, as we propose in Section 5, can provide the necessary support for effective application-level adaptation.

3.3 Operating System Resource Allocation

Although some applications will want to adapt to available resources as discussed in the previous section, we imagine most applications will simply want to run as well as possible given the current platform and the mix of other jobs running on the system. This is the task of the operating system scheduler, which is responsible for allocating appropriate resources to each application. Even if only one application is running, a new responsibility for the OS in the manycore era is to maximize performance and energy-efficiency of that sole app by only allocating the appropriate resources. For example, allocating too many cores may cause the application to slow down, or consume additional power with no additional performance gain. While an adaptive application could introspect on its own behavior, this requires additional application developer effort, and an application does not have the global view of what else has to run in the system. In this section, we discuss a scheduling framework that relies only on a small set of performance counters, which can help operating systems allocate the appropriate resources for an application even when the application developer is oblivious to resource usage.

The operating system scheduler sees a changing mix of concurrent applications with diverse resource requirements that might vary throughout execution. Naively dividing the machine using "fair" sharing may be a poor solution: some applications may not scale well enough to utilize a given resource while other applications may fail to meet user-driven deadlines given too few resources. The scheduler's task is complicated by the fact that concurrently running applications might interfere with each other through shared resources, causing applications to experience unpredictable performance degradations. For example, if two compute-bound threads are simultaneously scheduled on two different cores of a multicore processor, there may be no degradation versus running each in isolation. However, if the two threads are memory-bandwidth constrained, simultaneous scheduling could dramatically impair performance. Even more complex behaviors may occur if cores or hardware thread contexts share functional units, caches, or TLBs.

Previous work has examined how runtime measurements of application behavior can help the operating system scheduler allocate resources. Shen et al. showed that using hardware measurement information for resource-aware scheduling resulted in a 15-70% reduction in request latency over default Linux for RUBiS, TPC-C, and TPC-H [40]. Another line of work investigated techniques to co-schedule applications with disjoint resource requirements to minimize interference, for example, executing a compute-bound and memory-bound application concurrently [43, 13, 40, 52].

To address the scheduling and resource allocation challenges, we have been exploring a framework that combines hardware partitioning mechanisms with application modeling to predict the optimal resource allocation for a set of applications. The approach leverages partitioning mechanisms in hardware and the operating system, which create isolation between applications. Isolation allows us to virtualize the performance of a machine so that given a subsection of the machine (*e.g.,* 2 cores and 3 cache slices) the application will behave as if it was on a separate machine of that size. The operating system collects sample points of the application running with different resource sets using measurement hardware and then uses them to build a predictive model of the application performance for a given set of resources.

The scheduling framework decides the best resource allocation for a set of applications by performing an optimization of an objective function involving all of the models. The objective function represents a system–level goal such as minimize total energy or maximize throughput. Appendix A describes the different components of this scheduling framework in more detail.

We performed some of our own resource allocation experiments to confirm that runtime hardware measurements can help improve OS scheduling decisions. Using an FPGA-based multiprocessor emulator RAMP Gold [7, 44, 45] and a simple research operating system ROS [28, 33, 14], we ran experiments with the Parsec 2.0 Benchmark Suite [9] and some handcrafted microbenchmarks on a 64-core target machine. The OS uses page coloring to allocate sections of the cache to an application, and the simulator implements a simple form of bandwidth partitioning by limiting the number of requests that can be sent to memory over a time interval. Our system executes each of the applications several times, each time varying the number of cores, and cache and bandwidth allocations. We collect application runtime performance data using performance measurement hardware (similar to that we describe later in Section 5) to create models of the performance for that application given a particular resource allocation. We created a quadratic model and

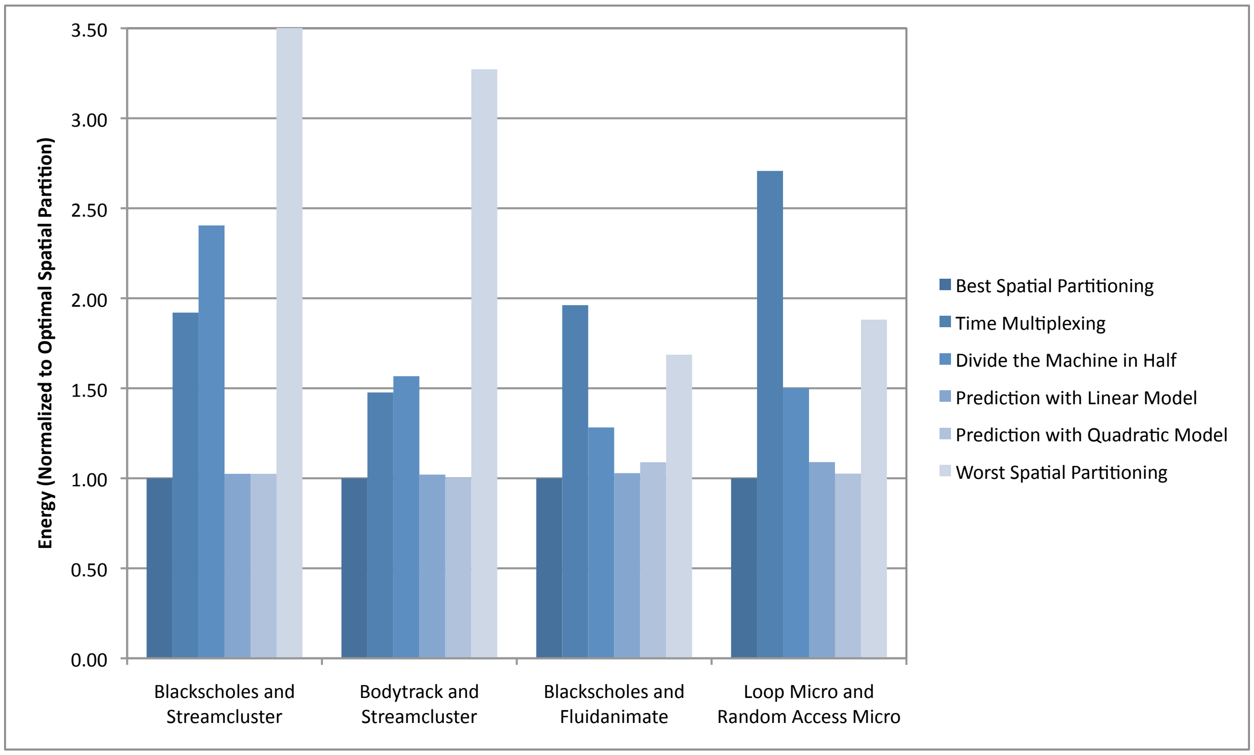


Figure 4: Comparison of the effectiveness of different scheduling techniques normalized to our quadratic model-based approach. The metric (sum of cycles on all cores + 10× sum of off-chip accesses) is a proxy for energy, so lower numbers are better.

linear model with multivariate regression techniques, a GPRS model, and a KCCA model using 20% of the possible allocations. Using these models, our scheduler decides how best to run a mix of two applications by trying to optimize a given objective function. For these results, we chose a simple proxy for total energy consumed, the total number of cycles run (the sum of the cycles on each core) + 10× the total number of off-chip accesses, as our example objective function.

Figure 4 shows the results of our scheduler's decisions based on the runtime performance data as compared with several alternatives: the optimal allocation, naively giving each application half of the machine, or time-multiplexing each application across the entire machine. The time-multiplexing scheme first runs the first application to completion and then runs the second application to completion. More fined-grained time-multiplexing could lead to longer runtimes due to cache interference effects and other context swap overheads.

We do no show results for the GPRS and KCCA as we found that they were extremely non-convex and as a result often produced very poor results when paired with our optimizer. The simple linear and quadratic models perform much better with only a small set of sample points and have significantly lower overhead.

The results show that using runtime data to perform more intelligent scheduling can lead to a significant savings in time and energy. Our approach beats naively dividing the machine by 65% and time multiplexing by 100% on average. Furthermore, it is within a few percent of optimal every time. The worst-case results show that the penalty for poor decision making can be quite large, with an energy cost 3.25$\times$ greater than our allocation on average.

As the number of cores and other shared resources grows on chip, this will increase the number of possible allocations, thereby widening the performance gap between good and bad schedules. Additionally, we expect that much more code will be tuned for parallel execution, and exhibit a greater variety of behaviors. As a result, there will more likely be disjoint resource requirements among applications available to co-schedule. We believe hardware measurement-based scheduling will thus prove even more effective.

4 Hardware Measurement Infrastructure Isn't Evolving to Help Software

Although hardware measurement facilities have been common in commercial platforms for many years, their capabilities fall far short of enabling software adaption of the kind we have described above. Furthermore, our opinion is that future roadmaps indicate their development will not add the necessary features.

We believe this disconnect stems from the fact that current hardware performance counters have been developed with a focus on improving chip engineering, not in helping with application software. This is not too surprising, since the emphasis for the last 20 years has been on inventing complex hardware mechanisms to deliver the increased performance potential of Moore's Law "under the hood" and out of view of software. Hence, performance counters are primarily added to help designers analyze the benefits of new hardware innovations at the microarchitecture level. Because it has not been a goal, measuring simple aspects of application performance, on the other hand, is often complex or impossible. We next list several common shortcomings we have uncovered when surveying existing commercial machines:

Shortcoming #1: Essential metrics are not measurable. Counters vital for composing important metrics are simply not implemented on some architectures. We were not able to compute memory traffic of an application on an Operton or POWER5 because a significant portion of the traffic, prefetches, is not measurable by an accessible counter. In fact, reported cache misses were impossibly low. Others have had similar experiences as they have attempted to use performance counters for more aggressive system scheduling [13].

Shortcoming #2: Many metrics are strongly tied to microarchitectual details. A lot of the counters lack contextual information crucial to application programmers. SiCortex, for example, has performance counters for stalls in each pipeline stage [41]. However, it is difficult to find detailed documentation to explain what is happening in each pipeline stage. Even with the documentation, it is unclear what an application or operating system would do with this information for many of the pipeline stages. In some cases (POWER5), there are meta-counters which explain the values of other counters, but they do not address the wide range of performance questions asked by the application programmer [25].

Shortcoming #3: High access overheads. The overhead required to obtain counter values is so high that subroutine-level or loop-level performance analysis becomes difficult. Some systems require serialization of the pipeline in order to access counter information. Without low-overhead measurement, it can be too expensive for software to support adaptation on the fly.

Shortcoming #4: Limited number of counters that can be used simultaneously. There is often a strong limitation of the number of counters that can be measured at the same time. For instance, on the IBM Blue Gene architecture, one can measure additions and subtractions, or multiplies and divides, but not both at the same time [24]. This limitation might have little impact on a chip designer who can rerun exactly the same test multiple times, but it makes parallel application profiling significantly more difficult. In other cases (PMAPI), performance counters come in groups that are pre-selected and thus inflexible [1].

Shortcoming #5: No support for multiple applications. Not all counter systems are designed to allow multiple applications to get data about shared resources at once. For example, on AMD's Barcelona, the L3 cache is shared between four cores. There is a set of counters that tracks usage of the shared resource, but these counters are programmed and accessed transparently through the private per-core counters. One core's programming of shared counters can be over-ridden at any time by another core's programming, and there is no mechanism to prohibit re-programming [5].

Shortcoming #6: Not standardized. Counter access is not uniform and individual counters are not consistently available on enough architectures for applications and operating systems to rely on them. Although hardware performance counters have been a common feature in machines, they are given low priority in the design and usually provide obscure functionality [29]. Software developers will continue to ignore the existence of hardware measurement mechanisms if there isn't a standardized interface to rely on.

Shortcoming #7: Not functional. Even when counters are provided, they are often simply incorrect or nonfunctional [29]. Because counters are only intended for use by chip engineers, they are not considered a critical component to verify.

While hardware engineers are certainly working to improve the current state of counters, we do not believe that they are moving towards what is needed to support an adaptive software stack. The majority of development has been further improvements in tools to help profile applications during development time. While this is certainly important for some applications, performance counters are typically only used by a small set of people for application development. We believe many more applications can benefit from deployment use of counters either directly or indirectly through libraries and the operating system. Although there is a demand for better runtime counters, the community is small since most developers rarely think about hardware measurement or even realize it is available. We think that hardware developers need to lead the way in providing the right measurement hardware, and it will be quickly adopted by many operating system, library, and application developers that have never attempted to use counters in the past.

5 SHOT Proposed

The current state-of-the-art and future direction of hardware measurement is at best insufficient to help parallel software run efficiently on a variety of parallel platforms. In this section, we describe our proposal for a standardized hardware measurement system we call "SHOT", for *Standardized Hardware Operation Tracker*.

Perhaps the most controversial part of our proposal is simply that it must be a *standard*. We are proposing a set of top-down measurements that help portable parallel applications adapt to the platform and operating systems manage the resources of the machine to achieve performance goals while minimizing energy costs. As we expect applications of the future to run on a wide variety of platforms, from mobile devices to cloud computing servers, software portability has to be a top priority. Hence, these top-down measurements must be available on all platforms to be useful, and this measurement hardware must survive across multiple generations of microprocessors to justify development of applications and operating systems to use these facilities. We believe that for the multicore revolution to succeed at providing performance portability, microprocessor manufacturers will eventually have to embrace a standard just as they embraced the IEEE 754 Floating-Point Standard in the past.

5.1 Basic SHOT Architecture

We now describe the overall architecture of a baseline SHOT system including the first few basic trackers that any SHOT system should provide, together with the system implications of providing these trackers.

The most important measurement is time, and we require that the system provide a globally accessible real-time clock. In earlier machines, a clock cycle counter could be used as a high-resolution proxy for real time. On future manycore designs, cores will be able to operate independently and have individual dynamic voltage and frequency scaling (DVFS). A basic form of this can already be seen with the Turbo Mode on Intel's Nehalem [26]. As a result, a single application could have instruction streams running with several different and time-varying clock rates. In order to provide a common time base for events, there must be a real-time clock available in some form. This real-time clock will have many other uses beyond performance measurement, including support for real-time applications [32]. The clock can be implemented at a much slower frequency than the core frequencies to reduce energy dissipation. Nevertheless, it should be fast enough so that events can be measured at a small enough granularity for demanding real-time applications, so we envision a real-time clock frequency of around 100 MHz.

The next most important tracker is the number of instructions retired on each core, as this provides a basic measure of throughput and can be used, for example, by an operating system scheduler to determine the progress being made by an application without the application's involvement. These counters naturally live in the retirement pipeline of each core.

The third basic metric we require is off-chip memory traffic, since this is of primary importance to both performance and energy consumption. In systems without a shared last-level cache, these counters can be located at the outermost private cache on each core, where they should track prefetch requests along with demand misses and writebacks to memory. In systems with shared last-level caches, the counters would be located on each shared cache bank. A concern with shared caches is how to attribute memory traffic to a particular core's activity and how to expose these counters to multiple concurrently running applications.

Many trackers will be implemented with distributed physical counters, which must be aggregated to discern behavior of a parallel application. To gather the total instructions retired by an application, for example, software must read counters from multiple cores and combine their results. In SHOT, we require that applications can gather these results as if the counters were read atomically on the same edge of the real-time clock, and the real-time clock value is also captured along with the counter snapshot. Implementations might actually implement physical latches that are loaded simultaneously, or use another implementation technique that has the same effect.

SHOT is designed to be used frequently during runtime. Therefore, all measurement data must be accessible with reasonable latency and low overhead to avoid performance losses from accessing measurements.

Unnecessary performance penalties — such as serializing the pipeline or performing a context switch — are not acceptable, because we want SHOT monitoring and analysis to always be enabled in production code. The SHOT system should also be simultaneously usable both by the operating system and by user applications. The counters are designed to be always running, do not cause interrupts for overflows and are accessible from user mode.

For SHOT to be useful with virtual machines, the time to save and restore SHOT information must be minimal so that hypervisors can include them when context swapping and not be involved when a guest OS or application tries to read the counters while running. Virtual Machines are the lynchpin of cloud computing, but Virtual Machine hypervisors ignore performance counters because the time to save and restore the current generation of counters would increase the context switch time significantly.

5.2 Expanded Basic SHOT Trackers

Real time, instructions retired, and memory traffic are the basic metrics we believe all SHOT systems should provide. Without these basic metrics, we believe it will be impossible to provide even a minimal level of software adaptation and intelligent operating system scheduling.

We would have liked to include energy consumption as a fourth basic metric, but energy trackers are still a subject of research. Ideally, systems would provide fine-grained energy tracking at the level of software-visible components (*e.g.,* cores, partitionable cache slices). A SHOT implementation might also synthesize energy trackers from lower-level microarchitectural counters using an energy model.

In addition, there are several further metrics that we believe are useful enough to become standardized, and we list these in Table 1. They support more diagnostic analysis of hardware performance issues and support advanced machine-learning based autotuners, but are still generic enough to be standardized across a variety of hardware architectures.

As this is an initial, we are not claiming that this initial list of trackers is complete or the exact right set. We give this initial list instead to start the conversation between application experts, operating–system programmers, library programmers, architects, and hardware designers on what should be the standard set of hardware trackers. Rather than starting from an abstract set of goals, we seeded our set instead from the measurements that machine-learning algorithms found to be important in getting the best performance on two commercial multicore computers above. We have also implemented several of these in our FPGA-based simulator (shown in bold in Table 1), so believe implementation issues are manageable.

Required	Computation	**Instructions Retired**
	Communication	**Memory Traffic**
	Time	**Real-Time Clock**
		Cycles
Additional	Computation	Instruction by type: **floating point**, **integer**, **load**, **store**, **control**
	Communication	Cache Traffic by category: speculative, compulsory, capacity miss, conflict miss, **write allocate**, **write back**, coherency
	Energy	Energy per task for all components
		Time spent in each power state for each component

Table 1: SHOT Metrics. The counters in bold were implemented in an FPGA prototype RAMP Gold for the machine-learning operating system scheduler experiments.

5.3 Standardization

In order to properly support performance-portability for software, it is imperative that SHOT become a standard. We have found in our conversations with application and library writers, that they are unwilling to add features to an application that are not portable. Performance counters currently vary widely from chip to chip — even for designs in the same architecture family — meaning that software using counters would be very machine dependent. Although SHOT may be implemented differently for different chips, it must always measure the same metrics and have the same access semantics in order to appear standard for software much like the IEEE 754 Floating–Point Standard. If standardized, we believe SHOT could have as big an impact as the IEEE 754 Floating–Point Standard in 1980s.

6 Objections to SHOT

In our discussions with architects about SHOT, some aspects of our proposal have raised eyebrows. In this section, we enumerate these concerns and our responses.

Concern #1: It is impossible to select a useful architecture-independent set of metrics. Many users and designers of current performance counter systems argue it is fundamentally impossible to find a set of architecture-independent metrics that can actually provide enough relevant information to be useful. They claim that meaningful performance analysis with counters requires a detailed knowledge of the microarchitecture and many counters are needed to measure all the possible events in that microarchitecture.

The general concern is that even a seemingly simple counter such as instructions retired is not easily implemented and varies from architecture to architecture. For example, on many Intel platforms, x86 instructions retired, macro–ops, micro–ops, fused macro–ops, and so on all are different forms of instructions retired and are counted by different counters.

However, we believe that this is a case of over-thinking the problem. While someone tuning by hand might care about the difference between macro–ops and fused macro–ops, software adaption environments are not going to change instruction types to increase pipeline utilization. For most of these systems, instructions retired is used simply as a measure of progress for the application.

Detailed microarchitectural runtime information from hundreds of events is the wrong level of performance abstraction for parallel software. All of the software adaptation approaches use methods that abstract the data intentionally. As a result, it is not necessary to have 100% accurate counters that record every minute detail. We believe the software infrastructure will remain robust even if the counters are slightly inaccurate or not completely representative of all parts of the microarchitecture. In any case, the dynamic adaptation algorithms would end up being overly complex and specific if they were designed to do a detailed interpretation of exact counter values and specific microarchitectures.

For example, our ML autotuner was able to beat expert optimizations on both Clovertown and Barcelona using only 8 and 6 metrics respectively with overlap in 5 of the metrics. The non-overlap in counters was due to differences in the memory hierarchy (Clovertown doesn't have an L3) and the non-existence of some coherency counters on Barcelona.

Concern #2: Such measurement hardware is too expensive. A common criticism of performance-measuring hardware is that it is costly in terms of silicon area, design time, and energy consumption. Recent commercial designs indicate that the engineering effort and chip real-estate demands are relatively minor, however. For example, SiCortex's performance counters account for 0.05% of the transistors on chip [42].

We expect the resources consumed by SHOT hardware to be similarly minor. Counters can be made low power and small, and we believe that the energy they dissipate is easily offset by the resulting improvements in application efficiency and programmer productivity. Additional transistors devoted to computational resources are useless if they cannot be fully utilized; SHOT will allow developers to use the resources available to their fullest ability. Few other hardware mechanisms can give such large performance improvements at such a low cost.

Accurately measuring hardware events that are of relevance to application developers does have some impact on hardware design verification effort, however. One reason that performance counters have historically been inaccurate is that their correctness has not been of vital importance to the design. Nevertheless, as was the case with IEEE 754, we believe that the increased design effort in order to comply with standards that are valuable to customers is worthwhile. Moreover, we believe software adaptation to achieve portable parallel performance is valuable to customers.

Concern #3: Exposing power and performance information is a competitive disadvantage. Some chip manufacturers have expressed concern that creating a standardized way to measure hardware performance behavior across different architectures could highlight unfavorable characteristics of the chip. For example, counters could expose to customers that one core runs slower and hotter due to process variation. Additionally, they could give away microarchitectural details that are part of the competitive advantage of the company.

It is our belief that not exposing these metrics will actually be a disadvantage to the company since applications, libraries, frameworks, runtimes and operating systems that make use of them will run more efficiently on a competitor's chip that implements SHOT. Furthermore, the SHOT metrics are by design general and high-level so they do not expose microarchitectural decisions, unlike the hundreds of measurements found in commercial hardware.

Concern #4: Standardization can be done entirely in software. There have been attempts to provide portable performance measurement software layers, but software cannot hide a shaky hardware foundation.

As an illustrative example, the PAPI project is the most widely used interface to access counters today [12]. The PAPI project started in 1999 with the development of a standardized interface to access the native hardware counters across a range of CPU architectures. PAPI preset events were added soon after as an effort to standardize countable events across those systems, by mapping presets to the available set of native events; in some cases, the presets were derived from multiple native events. Alas, there is little portability between platforms, so PAPI often provides no more than common software interface to access raw hardware counters with different semantics on each platform [18]. The situation continues to worsen as architectures shift to multicore designs, due to the increased platform variability of countable events between these new architectures. In our discussion with PAPI developers, they tell us that the situation is getting worse every year [46].

The PAPI example illustrates why it is important to create a new hardware standard since trying to standardize performance counters via software is intractable.

Concern #5: SHOT creates an Information Side Channel that can be a security threat. Previous work has shown that information about the performance of shared resources, such as cache activity, can enable a program to infer the control flow of another application [2, 52, 48]. Much of this information, however, can already be approximated without detailed event counters: for example, an adversarial program can estimate another application's cache or TLB performance by striding through a segment of memory and querying the OS for the execution time of this action.

Although SHOT may make it easier to exploit this extant side channel, such attacks are difficult in practice because the adversarial program must also know if the victim application is currently running, what other programs are sharing the resource, and so forth. Moreover, there are so many simpler attacks to our software stack that security experts tell us that side channel attacks are not very high on their list of concerns [39].

However, if SHOT's security implications really prove to be a concern in some environments, then the OS can determine, at the time of application launch, whether or not to allow access to the performance statistics of shared resources. Furthermore, well-designed partitioning mechanisms will prevent access to non-shared statistics of other partitions.

7 Related Work

There has been some other research in the areas of evaluating and augmenting current hardware measurement systems. We believe the existence of so many papers in this area supports the desire to have useable accurate hardware measurement.

Many papers focus on determining the accuracy of performance counters. Korn et al. use microbenchmarks to evaluate the accuracy of performance counters on the MIPS R12000 and determine that the access interface affects the accuracy of the counters [29] finding errors of up to 25% for instructions decoded. However, they rely on understandable microbenchmarks and do not validate their methodology for determining the expected values. Weaver et al. test the accuracy of the instructions retired counter on nine x86 platforms and comparing the results to numbers generated from dynamic binary instrumentation found that instructions retired had 1.07% variance [50]. Maxwell et al. [36] evaluate the accuracy of counters using microbenchmarks on an IBM POWER3, MIPS R10000, and Intel Itanium and found the percent error to range from 2%-500%.

Another class of papers has attempted to use performance counters for scheduling and as result claims that important counters are missing and counters and their access interface should be consistent across machines. Calandrino et al. [13] use counters for cache-aware real-time scheduling but find that Nehalem's counters are insufficient for the task due to the inability to measure prefetch events. Zhang et al. [52] argue for direct management of the hardware counters by the operating system by showing that they can improve scheduling

7% over the Linux Baseline Scheduler. Shen et al. [40] use counters to do resource-aware scheduling with a 15%-70% reduction in request latencies and suggest that standardized interfaces would make their solution more portable.

The Performance API (PAPI) project has worked to standardize the counter access interface [12]. However, they have found that although the interface is common the counters have different semantics on each platform so they often find their users needing to consult detailed architecture manuals to understand them [18].

8 Discussion and Conclusion

Although we have emphasized the problems of the mobile client thus far, we think SHOT would be valuable in the cloud as well. The distributed/cloud computing community has significant performance and scheduling challenges across hundreds of machines. Much effort has gone into debugging high latency requests. These unusually long requests are often difficult to reproduce since they are due to a very specific set of interactions across hundreds or thousands of servers. Some timing faults could be captured by hardware measurements on the machine, which would help to identify slow/faulty components or destructive interference. However, the community has yet to make much use of hardware statistics because the code must work across a diverse set of machines, and it would be difficult to include an architecture specific component into a distributed diagnostic system like X-Trace [3]. With the standardized SHOT system, runtime metrics could easily be integrated to allow per request hardware statistics to be added to the distributed trace reports and thereby help find the long-tail latency events.

In this thesis, we have argued that portability is one the main obstacles to the success of the multicore revolution. The predicted expansion in the diversity of parallel computing environments poses a significant threat to performance portability. How can developers meet application performance goals on this wide variety of possible execution platforms, each of which may dynamically switch between a wide range of operating modes? Developers will continue to face aggressive time-to-market schedules for producing large applications with the rich functionality described above, and it is unrealistic to expect manual tuning of the code for each possible execution scenario. We believe all levels of the hardware and software stack must change to enable applications to be automatically and adaptively tuned to perform well across this great diversity of execution environments.

To backup our beliefs, we performed some simple experiments. To demonstrate the importance of adaptivity to a platform, we showed that autotuning improved performance on two kernels by 3× over untuned code on five multicore platforms. Moreover, these experiments showed the difficulty of high-performance portability: running code optimized for a different platform typically ran 1.5× to 3× slower. Even successive multicore products from the same company exhibited a 3× slowdown by running code optimized for the prior generation. To demonstrate the importance of runtime adaptivity, we showed that using runtime measurements allowed scheduling of pairs of benchmarks that was within 8% of optimal and a factor of 1.25× to 2.4× better than static schedules. We used machine learning for two kernels on two multicore platforms to find which of the hundreds of performance counters were important for autotuning, and found that just 6-8 were enough to automatically autotune the code. This result suggests that SHOT need not be an onerous standard to implement. To further prove that point, we implemented the first version of SHOT in an FPGA multicore simulator.

Although architects will doubtless continue to create designs that have impressive peak performance, we believe many such computers will have unimpressive delivered performance unless they provide mechanisms that enable the hardware to be understandable and measurable to different software layers. Thus, SHOT could have as big an impact on portable parallel programs for mobile clients and cloud computing in the next decade as the IEEE 754 Floating Point Standard had on portable numerical programs in 1980s.

9 Acknowledgements

Thanks to Kaushik Datta and Archana Ganapathi for providing the autotuning experiments in this paper and to Henry Cook for all his help with the model-based scheduling work. I would also like to thank Karl Fuerlinger, Shoaib Kamil, Rajesh Nishtala, David Skinner, Andrew Waterman, Samuel Williams, Krste Asanović, and David Patterson for significant intellectual contributions to this work.

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. This work has also been in part supported by an NSF Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Pmapi. http://www.alphaworks.ibm.com/tech/pmapi.

[2] Side-channel attack. Wikipedia, en.wikipedia.org/wiki/Side-channel_attack.

[3] *X-Trace: A Pervasive Network Tracing Framework*, Cambridge, MA, 04/2007 2007. USENIX Association.

[4] L. Alvarez. *Design Optimization based on Genetic Programming*. PhD thesis, University of Bradford, 2000.

[5] AMD. *BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, Page 332*.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] K. Asanović, D. A. Patterson, Z. Tan, A. Waterman, R. Avizienis, and Y. Lee. RAMP Gold: An FPGA-based architecture simulator for multiprocessors. In *Proc. of the 4th Workshop on Architectural Research Prototyping (WARP-2009)*, 2009.

[8] S. J. Bates, J. Sienz, and D. S. Langley. Formulation of the audze–eglais uniform latin hypercube design of experiments. *Adv. Eng. Softw.*, 34(8):493–506, 2003.

[9] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[10] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.

[11] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proc. ACDC*, 2009.

[12] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. J. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[13] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. *Proceedings of the 21st Euromicro Conference on Real-Time Systems, to appear. IEEE*, 0, 2009.

[14] J. Colmenares et al. Resource Management in the Tessellation Manycore OS. In *HotPar10*, Berkeley, CA, June 2010.

[15] H. Cook and K. Skadron. Predictive design space exploration using genetically programmed response surfaces. In *45th ACM/IEEE Conference on Design Automation (DAC)*, June 2008.

[16] K. Datta, M. Murphy, V. Volkov, S. W. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick. Stencil computation optimization and autotuning on state-of-the-art multicore architectures. In *SC08*, 11/2008 2008.

[17] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, Feb. 2005.

[18] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *International Parallel and Distributed Processing Symposium (IPDPS2003)*, April 2003.

[19] K. Eikland. lpsolve. http://sourceforge.net/projects/lpsolve.

[20] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A case for machine learning to optimize multicore performance. In

HotPar09, Berkeley, CA, 3/2009 2009.

[21] A. Ganapathi, K. Datta, A. Fox, and D. A. Patterson. A case for machine learning to optimize multicore performance. In *HotPar09*, Berkeley, CA, 3/2009 2009.

[22] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Proc. MICRO '07*, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[23] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning, Second Edition.* Springer New York, 2005.

[24] IBM. *IBM SystemBlue Gene Solution: Application Development.* `http://www.redbooks.ibm.com/redbooks/pdfs/sg247179.pdf`.

[25] IBM. *System p Power5 Systems Hardware Information.* `http://publib.boulder.ibm.com/infocenter/systems/scope/hw/index.jsp?topic=/iphc5/9118_575_landing.htm&tocNode=int_4875`.

[26] Intel. *Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors.* `http://download.intel.com/design/processor/applnots/320354.pdf`.

[27] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Memory Systems Performance and Correctness*, San Jose, CA, October 2006.

[28] K. Klues et al. Processes and Resource Management in a Scalable Many-core OS. In *HotPar10*, Berkeley, CA, June 2010.

[29] W. Korn, P. Teller, and G. Castillo. Just how accurate are performance counters? In *Performance, Computing, and Communications, 2001. IEEE International Conference on.*, pages 303–310, Apr 2001.

[30] S. Kounev, R. Nou, and J. Torres. Autonomic qos-aware resource management in grid computing using online performance models. In *Proc. ValueTools '07*, pages 1–10. ICST, 2007.

[31] J. Koza. *Genetic Programming: On the programming of computers by means of natural selection.* MIT Press, 1992.

[32] E. A. Lee. Computing needs time. Technical Report UCB/EECS-2009-30, EECS Department, University of California, Berkeley, Feb 2009. — See also the ¡a href="http://chess.eecs.berkeley.edu/pubs/615.html"¿Published Version¡/a¿, Communications of the ACM, 52(5), pp. 70-79, May 2009 —.

[33] R. Liu et al. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar09*, Berkeley, CA, March 2009.

[34] M. Lourakis. levmar: Levenberg-marquardt nonlinear least squares algorithms in C/C++. http://www.ics.forth.gr/ lourakis/levmar/, 2004.

[35] Mathworks. Matlab 2008b. http://www.mathworks.com/, 2008.

[36] M. Maxwell, P. Teller, L. Salayandia, and S. Moore. Accuracy of performance monitoring hardware. In *Los Alamos Computer Science Institute Symposium (LACSI 2002)*, October 2002.

[37] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05*, pages 229–240, Washington, DC, USA, 2005. IEEE Computer Society.

[38] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, 2008.

[39] V. Paxson and D. Wagner. private communication, April 2009.

[40] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. *SIGOPS Oper. Syst. Rev.*, 42(2):189–200, 2008.

[41] SiCortex. *The SiCortex System Programming Guide*. `http://sicortex.com/var/ezwebin_site/storage/original/application/0b57e366808c27887788cf8a75988126.pdf`.

[42] W. Snyder and P. Mucci. private communication, May 2009.

[43] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 47–58, New York, NY, USA, 2007. ACM.

[44] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, K. Asanović, and D. Patterson. RAMP Gold: An FPGA-based architecture simulator for multiprocessors. In *Proc. of the 47th Design Automation Conference (DAC 2010)*, June 2010.

[45] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson. A case for FAME: FPGA architecture model execution. In *Proc. of the 37th ACM/IEEE Int'l Symposium on Computer Architecture (ISCA 2010)*, June 2010.

[46] D. Terpstra. private communication, March 2009.

[47] G. Tesauro, W. E. Walsh, and J. O. Kephart. Utility-function-driven resource allocation in autonomic systems. In *Proc. ICAC '05*, pages 342–343, Washington, DC, USA, 2005. IEEE Computer Society.

[48] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, New York, NY, USA, 2007. ACM.

[49] L. Wasserman. *All of Nonparametric Statistics (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., Se- caucus, NJ, USA, 2006.

[50] V. Weaver and S. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 141–150, Sept. 2008.

[51] S. W. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[52] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

A. Details of Model-Based Scheduling

A.1 Creation and Use of Predictive Performance Models

Our spatial resource scheduler is an example of *model-based control*, an area which has been investigated in many other contexts [11, 30, 37, 47]. In model-based control, a model is created by generating a mapping between a set of inputs (a perceived system load and a set of system resources) and an output (a measure of system performance relevant to the application). The model is trained by observing the performance metric of interest at a sample set of points in the space of possible variable assignments (*i.e.,* resource allocations). Once trained, the model can then be used to predict the performance of a resource allocation without actually testing the configuration.

We must consider a variety of trade-offs when creating a model for our predictive control system. The model will have to accurately capture the complexity inherent to the true relationship between a set of resource allocations and performance. Complex relationships necessitate complicated models that may be difficult to create. Whether the target system allows for accurate models to be created offline — or whether the system requires a model that can be trained online to adapt to changing conditions — also plays a role in choosing an appropriate modeling technique. We discuss these tradeoffs in the following subsections.

A.1.1 Model Inputs and Outputs

Our models use allocations of machine resources as independent variables (*i.e.,* inputs) and some metric of application performance as the dependent variables (*i.e.,* outputs). The inputs we include in this study are number of cores, off-chip bandwidth allocation, L2 cache ways and L2 cache banks. Since all input parameters are actually allocation sizes (*i.e.,* integers), they are easily represented as terms in the regression analysis.

We use cycles of execution time as a predicted output representative of application performance. To help correlate application performance with resource utilization and not just resource allocation, we also collect data from several performance counters that are usually strongly correlated with performance. These utilization metrics include L2 cache misses, L2 cache requests, and instructions retired. Any metric of interest that is measurable at runtime can be a potential output candidate. All of these models can be referenced by the spatial resource scheduling algorithm.

A.1.2 Modeling Techniques

We use multivariate regression techniques to create explicit statistical models for predicting the performance of an application given a resource allocation of a particular size. We create one regression model per performance metric per application phase.

Linear least-squares regression techniques produce simple models that can be expressed concisely and are therefore more portable. Linear regression techniques can outperform nonlinear ones when training sets are small, the data has a low signal to noise ratio, or sparse sampling is used [23]. These criteria apply in our case. These models are attractive due to their simplicity, but their restricted expressiveness may reduce their accuracy of the underlying system.

Linear models may be realized in varying forms (*i.e.,* it is the combination of terms that is linear, rather than the degree of each term). The simplest models are linear additive models, which take the form:

$$y(x) = a_0 + \sum_{i=1}^{N} a_i x_i \tag{1}$$

Multivariate linear additive models contain one term for each variable (*i.e.,* an allocation, x_i) and an intercept term (a_0). The regression tunes the coefficient associated with each term (a_i) to fit the sample data as accurately as possible. Note that the linear additive model has no way to represent any possible interaction between the variables, implying that all variables are independent—which is expressly not true in our scheduling scenario. We include them in our study as a straightforward baseline for comparison.

More complex multivariate linear regression models often include terms for variable interaction and polynomial terms of degree 2 or more. Such models are commonly termed *response surface models* and have the general form:

$$y(x) = a_0 + \sum_{i=1}^{N} a_i x_i + \sum_{i=1}^{N} \sum_{j=i}^{N} a_{ij} x_i x_j + ... \tag{2}$$

These polynomial models capture more complex dependencies between the input variables. However, we as modelers are still expressing beliefs about the nature of the relationship between input and output in the form we give the polynomial equation.

Selecting the best possible equation form for the data automatically requires the use of nonlinear regression techniques such as local regression, cubic splines, neural networks, or genetic programming [4, 11, 49].

The disadvantage of these techniques is that the models may create difficulties for analytic maximization algorithms.

Genetic programming is a technique, based on evolutionary biology, used to optimize a population of computer programs according to their ability to perform a computational task. In our case, the ‘program’ is an analytic equation whose evaluation embodies a response surface model, and the ‘task’ is to match sample data points obtained from full–scale simulations [4]. The output, termed a *genetically programmed response surface* (GPRS), are nonlinear models that create explicit equations describing the relationship between design variables and performance, and we incorporate them into our framework as an example of a nonlinear modeling alternative. A GPRS is generated automatically, meaning that the modeler does not have to specify the form of the response surface equation in advance. Instead, genetic programming [31] is used to create an equation and tune the coefficients. For more information on GPRS creation, see [4] or [15].

Other statistical machine learning techniques such as clustering can also be used in our framework to make predictions about application behavior. Any such technique must be able to correlate changes in resource allocations to changes in performance in an application–specific way. Ganapathi et al. have had success using machine learning to model application performance and select the best performing configuration in [21]. We use the same technique as Ganapathi et al. called Kernel Canonical Correlation Analysis (KCCA) to build models.

A.1.3 Online versus Offline Model Training

Predictive performance models can be created either offline or online. Offline model creation might occur during compilation or autotuning stages or the models might be encoded within a program binary distributed by the software developer (assuming fixed platform hardware as in mobile devices). Online model creation and modification is also possible, though it requires additional monitoring capabilities and potentially a more complex scheduler. A hybrid approach that starts with a sparse model and adds sample points as the application executes is also possible.

Offline models are more acceptable when robust performance isolation can be guaranteed by hardware partitioning mechanisms, since running applications concurrently will not cause destructive interference. They can be more complex, since it is acceptable for them to take longer to train, and the sample size they use may be larger. Offline models may be inappropriate for applications with heavily data-dependent runtime behavior. If an application has distinct phases, it may be beneficial to build separate models for each phase.

Online models are created at runtime, meaning that they offer increased flexibility and opportunities for adaptation. However, the granularity at which the models are updated and the time spent doing so must be offset against time spent making forward progress in the application. They are more likely to be useful for long–running applications with phases of behavior much larger than the scheduling quanta.

A key metric in deciding when to train models is the likelihood of model reuse. Model creation overheads are prohibitive for applications that are only executed once or never repeat their performance behavior. We do not believe this lack of reusability is the case for many applications. Performance isolation increases the chances that a given model will remain accurate in spite of changing load placed on the system at runtime by other concurrently running applications.

A.1.4 Collecting a Training Sample

In an online system, sample data points used to train the model are collected as the application runs on varying allocations of hardware resources. In an offline system, these data points are collected and a model is built in advance. We collect samples and build models offline to test the feasibility of model-based scheduling, but our framework is straightforward to extend to an online methodology.

We use a design of experiments (DOE) technique known as the Audze-Eglais Uniform Latin Hypercube design of experiments [8] to select the points included in the sample set of performance data used in the creation of the response surface models. Audze-Eglais selects sample points which are as evenly distributed as possible through the design space by formulating the problem as one of gravitational attraction and minimizing the "potential energy" of the system of points. We formulate our Audze-Eglais DOEs using the optimization technique described by Bates et al. [8].

Random sampling, which is the common sample selection alternative to a formal design of experiments, is particularly suboptimal for small sample sizes because it does not guarantee an even sampling distribution across the space of resource allocations.

The size of the sample generally has a significant impact on model accuracy, but some models are more robust than others. Our scheduling framework must balance the sample size and training time required for the model with the applications' runtime behavior.

Any performance isolation that can be provided by the OS or hardware partitioning mechanisms will have a significant impact on model accuracy, which may inform our choice of modeling methodology. Given improved isolation, we can create an individual model for the performance of each application workload separately and yet still be confident that the overall system performance of a multiprogrammed system will be accurately predicted by the amalgamation of these individual models. On a system without robust partitioning, destructive interference between the applications may result in unpredictable performance. While our spatial resource scheduler would still function on such a system, its effectiveness would be tied to the amount of inter-application contention. For this reason, we advocate the adoption of hardware support for on-chip performance isolation, as described in Section A.3.

A.2 Spatial Resource Scheduling

Given a set of models corresponding to the the applications currently running on the system and knowledge of the current spatial resources that have been provided to each application, our scheduler predictively explores the space of potential reallocations. It can combine the performance predicted by each application model to predict overall system performance, and if partitioning mechanisms are present this joint prediction will be extremely accurate.

The critical benefit of predictive model–based control is that after the model is built the scheduler does not actually have to perform a reallocation in order to evaluate its effectiveness. A spatial scheduler based on trial–and–error search through possible allocations (rather than on models) must incur a significant overhead as it requires applications to move around in the physical machine space. Furthermore, the trial–and–error search must be repeated for every scheduling quanta. To avoid starting from scratch every time, a trial–and–error scheduler might store information about allocations it has already seen, at which point it is effectively building an online model. When searching for an optimal system-wide allocation, our model–based scheduler can rapidly evaluate many allocations based on analytic models rather then having to repeatedly reallocate and remeasure performance.

The following subsections detail the design of our spatial resource scheduler.

A.2.1 Scheduling Frequency

The spatial resource scheduler complements a standard time–multiplexing OS scheduler, which is run every scheduling quanta or when certain special events occur. These events may include when applications start or halt execution, request additional threads, or block on I/O. If the system has the capability to detect phases of application behavior with hardware phase detectors, the scheduler should be run when new phases are detected.

A.2.2 Scheduling Decisions

Deciding how to reallocate system resources among applications is now a search over sets of model predictions, rather than a trial–and–error search through actual reallocations. The goal is to optimize the gain from any potential reallocations while minimizing the performance disruption caused by the reallocation. In general, the distance between the current allocation and proposed allocation may decrease the perceived benefit of the proposed allocation, but this disadvantage is amortized across the amount of time that the allocation will persist. Our implementation does not currently take into account reallocation overheads when making decisions. However this extension can be easily incorporated into the objective function, as described below.

If no model currently exists for an application, the scheduler can assign it some default allocation of resources until it can be profiled. This approach avoids the sampling and modeling overhead for applications that are only run a few times.

The decision-making algorithm inherent to our spatial resource scheduler can take into account a variety of concerns in addition to performance that are expressed by either the operating system or individual applications. Such concerns might include Quality–of–Service constraints, hardware–enforced power and thermal constraints, or a variety of performance and energy goals (*e.g.,* system–wide utility, per–application maximums and minimums, elitist priorities, awareness of power source conditions, race to halt). The model–based control approach we propose can incorporate all of the above simply by changing how the individual applications' models' predictions are weighted by the decision–making algorithm and what constraints are placed on the allocation choices available to the algorithm. Goals, weights and constraints are incorporated into the scheduling algorithm's objective function and search heuristic.

A.2.3 Objective Function

The actual decisions made by the scheduler are governed by an objective function, which converts model outputs into a measure of overall decision fitness. This objective function can therefore incorporate weighting model outputs as well as individual configuration choices. Weight can be used to prioritize certain models: for example, we might use an energy model to weight the different predictions of application activity and thereby arrive at an expected application energy usage. The decision algorithm can then search for allocations with the lowest predicted energy. The common basic objective function we use in our experiments is to minimize the maximum runtime of any of the applications available. This objective function is useful for platforms with high platform power, where the goal is to complete the entire workload as quickly as possible and then shutdown.

The instantaneous evaluation (no reallocation necessary) of scheduling decisions provided by models allows for more complicated objective functions, since the searches made over them execute quickly and can be made relatively large and robust.

A.2.4 Searching for Optimal Schedules

Given predictions provided by a set of trained models, and an objective function describing scheduler goals and priorities, we now need to search through the space of possible schedules. The runtime of this process must be short – even with models rather than experiments providing feedback, we cannot exhaustively search through all possible schedules.

If we have used a linear model, we can treat this optimization problem as a constrained linear or quadratic programming problem and can solve it using one of the many solvers available (we use Matlab's `fmincon` function [35], but `lpsolve` [19] or `levmar` [34] are open source alternatives). If the models and objective function are nonlinear, we can treat the optimization as constrained nonlinear programming. Other search heuristics, like hill–climbing or simulated annealing, might be employed to optimize over particularly discontinuous objective functions, but do not provide any guarantees about solution optimality.

In this paper, we evaluate using a medium–scale active–set algorithm and an interior–points algorithm that calculates the Hessian by a dense quasi-Newton approximation. Active–set is a sequential quadratic–programming–based solver. Both methods depend on the convexity of the function to guarantee optimality (our objective functions are generally not convex), and both are gradient-based methods designed to work on problems where both objective and constraint functions are continuous and have continuous first derivatives. For objective functions which are not convex (due to non-monotonicity of the models or its own form) these algorithms may choose local minima. The runtime of the solver will increase proportional to the cube of the number of constraints, which in turn increase proportionally with the number of contending applications. For the problems we consider in this paper, the runtime of these algorithms is acceptable for scheduling purposes.

A.3 Hardware Support

While our scheduling system can be run without additional partitioning hardware support, several architectural features can be added to significantly improve the effectiveness of the modeling and scheduling decisions. The following subsections discuss useful hardware mechanisms to provide performance isolation, better measurement data, and easy detection of reallocation points in applications. Our experimental results include the effects of the following partitioning mechanisms and assume the presence of performance counters.

A.3.1 Partitioning Mechanisms

To achieve performance isolation for each application and improve model accuracy, we implement hardware partitioning mechanisms on several of the most important shared on–chip resources. Shared resources include discrete functional units (*i.e.,* cores), containers with shared capacity (*i.e.,* caches), or communication media with shared bandwidth (*i.e.,* interconnects). There may be many such resources on an individual chip and even more in an entire computer system. In this study, we provide partitioning mechanisms for cores, L2 cache capacity, and interconnect bandwidth to DRAM.

A.3.2 Measurement Hardware

It is difficult to accurately model application behavior without dynamic feedback from the hardware about how the application is performing during the training phase. This requires monitoring hardware that can measure and record important metrics that correlate to the application's performance. For offline models, the hardware can store the data and the modeling system can access it after the sampling has finished. However, systems that create online or hybrid models and use performance data to refine the models/objective functions require low access overhead to the measurement data, so that it can be frequently accessed. For online modeling/refinement, the hardware must be able to measure metrics on a per partition basis and attribute behavior on shared resources to that partition. Current performance counters on some platforms can be used for the offline technique; however, on other platforms important metrics are not measurable (*e.g.,* bandwidth to DRAM) [27][13]. Current platforms do not have the ability to measure enough metrics or applications concurrently (or at a partition–level granularity) to be truly effective with online modeling for more complex objective functions.

Our most basic objective functions require only models capturing runtime cycles, which are relatively easy to track. However, the more complex objective functions use activity metrics to create more accurate performance or energy models. Our current implementation uses L2 caches misses, L2 cache accesses and instructions retired to represent the activity on the interconnect, cache, and cores respectively. We are exploring the effectiveness of incorporating other metrics into our models and objective functions.

A.4 Related Work

Guo et al. [22] point out that most prior work is insufficient for true QoS – merely partitioning hardware is not enough, because there must also be a way to specify performance targets and an admission control policy for jobs. The framework they present incorporates a scheduler that supports multiple execution modes.

Nesbit et al. [38] introduce Virtual Private Machines (VPM), a framework for resource allocation and management in multicore systems. A VPM consists of a set of virtual hardware resources, both spatial (physical allocations) and temporal (scheduled time-slices). Unlike traditional virtual machines that only virtualize resource functionality, VPMs virtualize a system's performance and power characteristics, meaning that a VPM has the same performance and power profile as a real machine with an equivalent set of hardware resources.

They break down the framework components into policies and mechanisms which may be implemented in hardware or software. Critical software components of the VPM framework are VPM *modeling*, which maps high-level application objectives into VPM configurations, and VPM *translation*, which uses VPM models to assign an acceptable VPM to the application while adhering to system-level policies. A VPM scheduler then decides if the system can accommodate all applications or whether resources need to be revoked.

The VPM approach and terminology mesh well with our study, which can be seen as a specific implementation of several key aspects of the type of framework they describe (*i.e.,* VPM modeling and translation). Nesbit et al. did not perform any evaluations of the modeling, translation, or scheduling processes suggested in their paper.