A Cross-Layer Approach to Heterogeneity and Reliability

Daniel Williams Aprotim Sanyal[1] Dan Upton Jason Mars Sudeep Ghosh Kim Hazelwood

Department of Computer Science
University of Virginia
www.tortolaproject.com

Abstract

As modern hardware becomes increasingly complex, it becomes more difficult to create efficient software for common computing workloads. One way to manage this complexity is to employ holistic solutions that consider multiple layers of hardware and software in conjunction, allowing software to adapt and react to changing conditions at run time. This paper focuses on lightweight modifications to commodity hardware that enable virtual execution environments to help solve problems in the areas of power, reliability, security, and performance. We present our experimental simulation framework, which enables us to explore the design space of hardware/software collaboration, and we demonstrate its ability to produce simplified, reactive solutions to two emerging computing problems. First, we improve heterogeneous process migration with hardware feedback, and second, we use hardware information to respond to voltage emergencies (di/dt) in software. These symbiotic design approaches illustrate the simple nature yet significant potential of cross-layer, reactive solutions.

1. Introduction

Research efforts in optimizing computer systems have historically targeted a single logical layer in the system stack, be it application code, operating systems, virtual machines, microarchitecture, or circuits. Solutions that target a single layer in isolation are unfortunately reaching the point of diminishing returns, given the complex nature of modern processors, and the correspondingly complex challenges that have arisen regarding power, reliability, and security.

Over the years, various researchers have explored co-designed hardware-software systems that introduced revolutionary hardware while providing software-compatibility for existing applications [14, 16]. Such approaches have potential that extends beyond performance, but revolutionary approaches often preclude a thorough understanding of the benefits and drawbacks of each individual design choice; not to mention that these approaches experience market resistance. Instead, we take an evolutionary approach to exploring the potential for collaboration between various system layers. We look for opportunities to make major impact with minor hardware and OS feedback channels. A virtual-execution environment (VEE) then orchestrates this feedback information to provide a cohesive solution. This approach permits the design of *adaptive* solutions that adjust to changing conditions of the operating environment. Adaptation can be triggered by a variety of factors, including hardware or OS feedback, program behavior, or user preference, among others. This paper focuses on a subset of adaptive solutions, *reactive* solutions, where the hardware can detect a problem and the VEE can take action to correct the problem. For many problems, reactive solutions are the correct approach because, as we later show, it is much easier to react to a particular problem than it is to predict hardware-specific problems before they occur.

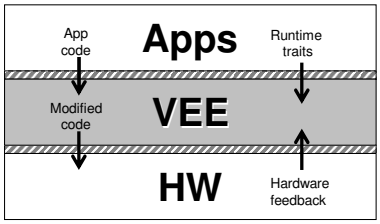


Figure 1. Abstraction of our target system. A virtual layer supports well-defined HW-SW communication channels.

As Figure 1 illustrates, our vision for the future standard in system architecture consists of a virtualization layer (VEE) interjected between the application software and the underlying machine architecture. This virtual layer communicates bidirectionally with the microprocessor and other software layers via well-defined channels. The VEE can use hardware feedback to detect various machine-specific events, such as voltage fluctuations in the power supply, temperature problems, as well as performance-related events, such as cache misses or resource contention. The VEE can then factor in its global knowledge about the executing workload, such as the specific instructions selected by the compiler, the instruction schedule, and the control-flow graph. Finally, the VEE can orchestrate a comprehensive solution to the problem which accounts for both hardware and software inputs.

In this paper, we focus on two specific challenges that are well suited for such collaborative design solutions. Our first example centers on the problem of heterogeneous multicore scheduling. Current trends in architecture design indicate that heterogeneous multicore processors will soon become ubiquitous, in large part due to the heterogeneity that naturally arises during the fabrication process. We show that the problem of process-to-core scheduling, which is currently relegated to the operating system alone, can be better solved using hardware and application feedback in conjunction with operating system input. This turns out to be particularly true for heterogeneous cores, where the OS scheduling algorithms can become needlessly complex and specific to the hardware. Rather than solving these complex application-to-core mappings statically, a simpler (and more effective) approach involves detecting and reacting to performance issues in the current schedule.

Our second example of an opportunity for effective collaborative solutions involves power supply voltage stability[2], which, when left unchecked, can cause timing errors or reliability prob-

[1] Aprotim Sanyal is now employed by Google; this work was completed while he was a student at the University of Virginia.

[2] The International Technology Roadmap for Semiconductors (ITRS) has cited this phenomenon as one of the emerging *Grand Challenges*.

lems due to electromigration. In modern systems, voltage stability is guaranteed by adding decoupling capacitors, or by throttling resources [23]. Yet these solutions are not particularly scalable – specialized capacitors are expensive, and resource throttling harms performance. We provide a lightweight and complementary solution, letting the VEE react and rewrite application code to dynamically respond to power emergencies that may arise.

While we focus on two specific motivating examples in this paper, we feel that the potential for cross-layer design extends well beyond reliability and performance into numerous other areas including security, power, and temperature. The specific contributions of this paper include:

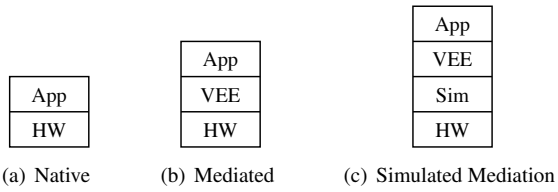
- The introduction of several lightweight changes to existing commodity systems that will enable reactive, cross-layer solutions to emerging computing challenges.
- The presentation of our new simulation framework that tightly integrates two well-known tools: SimpleScalar [4] and Pin [28]. This framework allows researchers to explore lightweight extensions to existing hardware, and the potential of cross-layer collaboration.
- Demonstration of the potential of a reactive, cross-layer solution to the heterogeneous multicore scheduling problem – using hardware-based performance counter feedback to drive process-to-core rescheduling.
- Demonstration of the potential of a reactive, cross-layer solution to the di/dt problem – using novel hardware feedback channels to drive code rewriting to avoid future power emergencies.

The remainder of this paper is organized as follows. Section 2 motivates the notion of cross-layer design solutions. Section 3 then goes on to provide the details of our simulation approach. Next, Section 4 describes our two sample problems that are well-suited for symbiotic solutions, as well as our proposed solutions and evaluation. Section 5 then discusses other collaborative systems and related work. Section 6 presents our ideas for follow-on work, and Section 7 concludes.

2. Lightweight Collaboration

Significant hardware changes often require long lead times before they can be realized in the market. However, minor changes to existing hardware, such as exposing additional hardware performance monitors or other diagnostic information to the ISA, can be added to the hardware relatively quickly. In fact, hardware designers have historically been very willing to expose additional performance monitors if a case is made for their benefit. From the software side, there have been a few success stories in terms of communication to lower layers, and one key example is the `prefetch` instruction. Other software-based *hints* have been less successful (such as the `register` keyword in `C`) because certain design challenges require input from multiple layers in the design stack. Nevertheless, isolated changes that increase hardware-software communication can provide numerous benefits, without the overhead of testing, validating, and tuning completely new hardware.

Moving toward cross-layer solutions means that one layer will be required to collate, analyze, and respond to the various inputs from multiple design layers in order to orchestrate a cohesive solution. We feel that a virtual execution environment is the best place to perform this orchestration. Modern VEEs provide the flexibility required to adapt to changing hardware constraints in a way that is transparent to the application. It is even possible to use VEEs to transparently support changes in the underlying ISA without the multi-year design and adoption time required by significant changes in hardware. Furthermore, if the changes to hardware are minor, we can leverage existing, performance-tuned VEEs to perform these new tasks.



(a) Native (b) Mediated (c) Simulated Mediation

Figure 2. (a) The native execution stack, (b) mediated stack used for a baseline, and (c) simulated mediated stack, used for experimentation.

Using virtual execution environments to improve or extend programs has been a highly active area of research within the architecture and compiler communities. VEEs have been used to perform dynamic optimization [6], security policy enforcement [25], profiling [28], and binary translation [16]. Modern VEEs operate transparently to perform these tasks without otherwise perturbing the application software. While application software (and even operating systems) are designed to be portable and to minimize system-dependent features, VEEs typically possess (and benefit from) direct knowledge of the underlying hardware.

An interesting benefit of using a VEE in the domain of hardware-software collaboration is that it moves us in the direction of decoupling the hardware and software interfaces from a single, fixed ISA. In essence, the VEE can virtualize the ISA by converting any virtualized instructions used in an application with the corresponding instruction(s) for the underlying hardware. This is a powerful approach because it allows hardware designers ultimate design freedom without code compatibility concerns. Meanwhile, any hardware errors that are detected after shipment can be masked by a simple update to the VEE.

The biggest challenge to operating all software under the control of a VEE is the performance overhead. Fortunately, researchers have been extensively investigating ways to reduce this overhead. Meanwhile, other constraints, including power, reliability and security have begun to outweigh the importance of raw performance. We are reaching the point where the 10% performance overhead of modern VEEs becomes much more tolerable if it means that reliability concerns will be addressed, battery life will be extended, or security policies will be enforced.

Given the benefits and challenges of a move toward cross-layer solutions, we have chosen to explore an evolutionary path to our long-term goal. We start with lightweight changes to existing, proven systems. This approach allows us to analyze each hardware and OS change in isolation, comparing the costs and benefits of each technique relative to existing systems.

Developing the best design requires us to throughly prototype and test each hardware or software modification. Yet, simulating our hardware changes can be time consuming because of the need to simulate both the VEE and the application itself. The next section explores simulation methodologies we used in our project.

3. Our Simulation Infrastructure

Ideally a simulation infrastructure is fast, accurate, and easy to modify. In practice these goals are often at odds with each other. To address these trade-offs, we used two experimental *stacks* – configurations of application, virtualization, and hardware layers – to optimize for one particular requirement over the other.

The *native stack*, shown in Figure 2(a), is used to as the baseline for comparison. On-board performance counters are measured using the Performance Application Programming Interface (PAPI) [11]

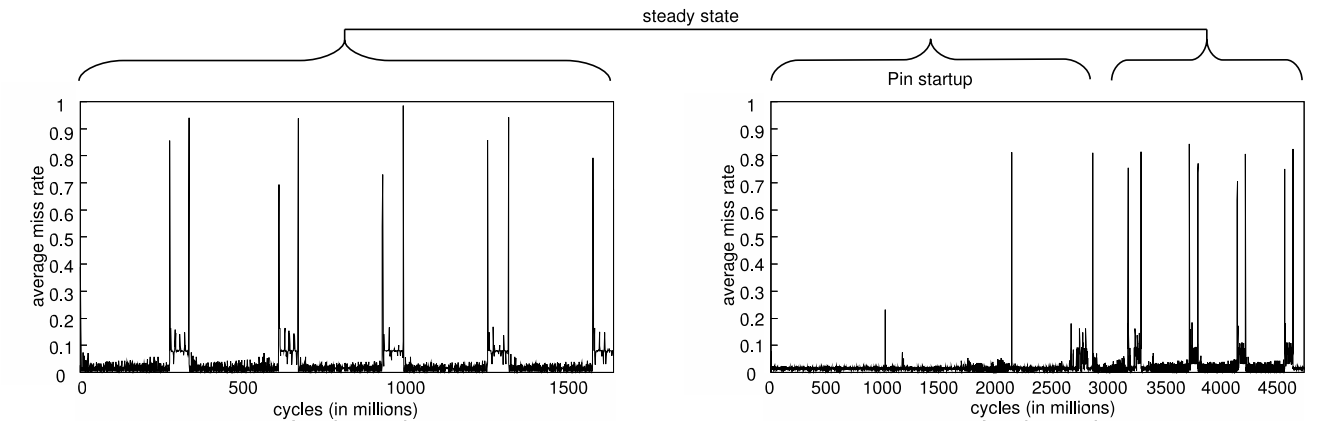


Figure 3. L1 data cache miss rate of `gzip` (a) natively and (b) under control of Pin. After Pin's initialization, the last 1600 datapoints of the Pin graph are similar to those of the native graph.

and the `perfctr` kernel extension for GNU/Linux. The number of performance counters built into modern processors is extensive [3, 22] and useful for evaluating performance for many hardware features of interest (e.g., cache behavior, CPI, etc.) However, the information from this stack only serves to help us understand the impact that introducing software translation has on the system. In order to actually modify the executing application, we need to add a VEE into the stack.

To this end, we use a *mediated stack*, inserting a VEE between the target application and the underlying hardware, as shown in Figure 2(b). This mediated stack allows us to create VEE tools that modify the code based on internal and external control signals. We can, for example, develop a tool that reacts to the performance data being generated by the existing hardware performance counters in real time. Because of the overhead of interjecting a binary transformation layer, this mediated stack results in less than a twofold increase in run time over the native execution stack.

Numerous VEEs are available for the x86 platform, both above and below the OS [9, 12, 28, 35], that are able to orchestrate the interaction between hardware and software. We use the Intel Pin system [28], an extensible run-time binary instrumentation tool. Pin runs in user space, giving it access to binary information on a per-process level. Whole-system implementations need a VEE below the OS, as is the case with PinOS [12]. Once PinOS is publicly released, we plan to investigate its utility for whole-system experimentation.

To enable novel hardware extensions, we created a *simulated mediated stack* (Figure 2(c)), where a processor simulation framework is inserted just above the hardware layer of the corresponding un-simulated stack. Together, the simulator and the hardware form a virtual hardware layer – functionally similar, from the application's or VEE's perspective, to the hardware layer in their un-simulated analogues. However, unlike the actual hardware, we are able to modify and control all aspects of the simulated processor in order to create new control conduits between the virtual hardware layer and the VEE software layer.

In all of our simulations, we used the x86 version of SimpleScalar [4] as our processor simulator. We performed numerous significant extensions to SimpleScalar to enable it to successfully run applications as complex as Pin. For instance, we implemented several missing instructions, such as `pushf`, `popf`, `pushes`, `cpuid`. The time penalty associated with this simulation stack is several orders of magnitude slower than running on native hardware – a thousand-fold or ten-thousand-fold increase in run time, which is on par with the overhead of SimpleScalar alone. We are currently evaluating other simulations alternatives, including PTLSim [40] and Simics [29].

Validation Before implementing solutions using our various stacks, we wanted to verify that the data generated by the tools were trustworthy. Complete validation of two large, complex systems such as Pin and SimpleScalar was outside the scope of this work; however, we did perform some sanity checks on the results reported by Pin and SimpleScalar-x86.

Our first task was to determine whether Pin unreasonably affected the behavior of programs. We found that on many well-behaved programs – those that spend much of their execution time in loops that have already been translated into Pin's code cache – Pin's overhead is small. Figure 3 shows the L1 data cache miss rate collected every one million cycles over the entire execution of `gzip` running natively and on Pin[3]. With the exception of startup time (the first 1.5B cycles) and some dynamic compilation overhead, there is a notable similarity in the run-time behavior of `gzip` executing natively and on Pin. Programs with less code reuse or many indirect branches, however, prevent the VEE from being able to amortize time spent translating or rewriting code. In these cases, the overhead of translation must be taken into account in the design and heuristics.

Figure 4 displays the results of further validation, where we use a log scale so as to clearly display all data on one graph (otherwise, `h264`, `gobmk`, and `astar` would dwarf the other applications). Figure 4(a) compares the instruction counts reported by SimpleScalar, Pin, and PAPI. Pin instruments every dynamic instruction, thus its instruction count should be quite accurate. PAPI accesses the hardware performance counters, but we determined that the calls for starting and stopping the counters also get included in the instruction count, thus there is a fixed error. This fixed overhead becomes negligible, however, for long running programs, such as the SPECint2006 benchmarks [20]. Finally, as the graph demonstrates, SimpleScalar reports an instruction count comparable to those reported by Pin and PAPI. Beyond instruction count, we also looked at performance metrics at the microarchitectural level: clock cycles (Figure 4(b)) and branch mispredictions (Figure 4(c)). Because Pin operates above the OS, only data from SimpleScalar and PAPI are included in these results. SimpleScalar simulates an idealized five-

[3] Previous work [37] has shown correlation between such performance factors as cache miss rates, branch prediction miss rates, energy consumption, and IPC.

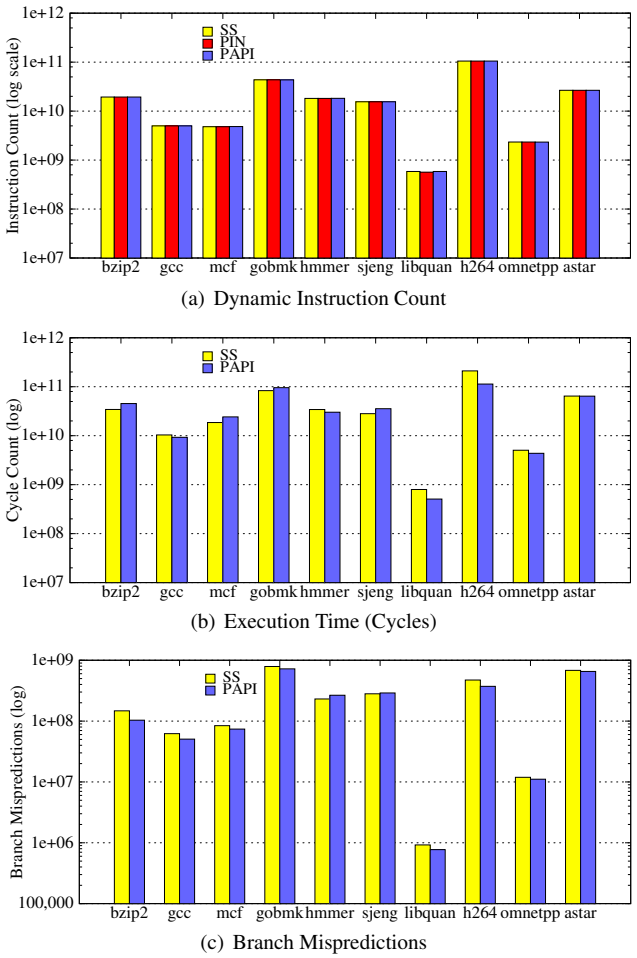


Figure 4. Evaluation of the accuracy of SS-x86 and Pin on instruction count, cycle count, and branch mispredictions, relative to the hardware performance counters as accessed by PAPI.

stage processor, which is unlike most modern processors. Nevertheless, the values reported by SimpleScalar followed similar trends as that reported by PAPI on a real dual-core NetBurst Xeon processor.

4. Symbiotic Optimizations

Collaborative design encompasses a variety of potential challenges within a large design space. We now demonstrate the usefulness of reactive, cross-layer solutions by examining two problems that can be more effectively solved with hardware/software collaboration. The first application – heterogeneous migration – is the task of dynamically scheduling simultaneous processes or threads on cores of differing capabilities in a way that optimizes for resource utilization. The second application – the di/dt problem – is a well-studied issue arising from the current and voltage variations caused by rapidly enabling and disabling processor resources. These are two problems of great interest to the community, and they are problems that are difficult to solve without reactive techniques to handle changes in processor and program state. This section explains our approach and methodologies for using collaborative techniques to solve these two challenges.

4.1 Heterogeneous Migration

We examined the benefits of feedback-based scheduling of multiple programs on heterogeneous cores as an illustrative example of the simplicity and potential of a reactive solution. Processor heterogeneity (sometimes also called processor asymmetry) comes in many forms. Some multicore processors are designed with different features on each core. Heterogeneity can also result from the fabrication process [21]. Regardless of the nature of the heterogeneity, scheduling processes on the available cores becomes a non-trivial challenge, since one core may be better suited than another to execute a particular process. Kumar *et al.* demonstrated that dynamic process migration can outperform the best static process-to-core matching on a many-way heterogeneous Alpha ISA system [27]. In this section, we evaluate a number of migration heuristics in an x86-based environment to determine the system changes necessary to support the solution.

Problem Description The problem of heterogeneous process migration is a complication of traditional process scheduling. When an OS schedules processes on homogeneous multicore systems, all cores can be considered equivalent and processor load becomes a good indicator of an effective schedule. By contrast, if the OS assumes equivalence on a heterogeneous system, performance could suffer considerably. To examine heuristics for improved scheduling across heterogeneous cores, we consider two heterogeneous cores and two different processes running simultaneously. On such a system either process could be run on either core at any given time. Therefore, the OS must decide when to migrate processes from one core to another. Yet, integrating very specific hardware details into the OS scheduler is not a scalable approach, as it needlessly complicates the scheduling algorithms to support the seemingly endless amount of heterogeneity possible. An easier solution is to use feedback from the hardware to allow the OS to *recognize* and *react* to poor scheduling decisions.

Consider the relatively simple case of a processor with a single out-of-order superscalar core and a simpler in-order scalar core. Traditionally, the OS does not consider performance in its scheduling decisions, it only considers runnable queues and resource utilization, which works well on today's homogeneous systems. Yet, in our heterogeneous case, simply running two applications to completion on their initially assigned cores may not be the wisest strategy; one program many be able to better take advantage of the out-of-order processor at a particular time than another. Even a single assignment of the programs based on amortized performance may not be sufficient. Phase behavior within the program may cause the proper program-to-core mapping to change while the program executes. We investigate run-time indicators from other design layers that may help us dynamically schedule process execution on the different cores.

4.1.1 Simulating Migration

Table 1 outlines the configuration of our simulated in-order and out-of-order processors. We execute the SPEC2006 integer benchmarks on both configurations. We collect statistics every 1M instructions, checkpointing the current state of the performance counters. These checkpoints are used to drive our migration decisions. After collecting the execution statistics for the programs on both configurations, we can analytically investigate arbitrarily migrating processes at the selected granularity.

Migration Overhead When a migration decision is made, the process state must be transferred from one processor to the other. This transfer is often costly. Thus we must factor this cost into our performance analysis. Multicore processors often share the same physical L2 cache. Assuming such an arrangement, migrating

Parameter	Value
Processor width	8 (outorder), 2 (inorder)
Fetch queue size	16
Branch predictor	Combined predictor w/ 16K-entry meta-table, 2-lev predictor w/ 16K entry L1, 16K entry L2, 14-bit history XORed with address
BTB size	512 sets, 4-way
RAS size	8
RUU size	128
L1 caches	64K, 4-way, 32B blocks
Unified L2 cache	512K, 4-way, 64B blocks
Functional units	6 int ALU, 2 int mult, 4 FP ALU, 2 FP mult

Table 1. Our SimpleScalar-x86 Configuration

processes between cores should incur a penalty similar to that of a context switch invoked by an operating system. The state of the architected registers and the state of memory that is associated with the process must be transferred from one core to the next. Other structures, such as cache contents and on-chip structures designed for speculation such as branch predictors need not be maintained for correctness. However, when this information is lost there is a cost to warm these structures again; for example, the L1 cache contents associated with our process are essentially flushed with the migration. This forces the per-core cache to be repopulated, adding to our migration overhead [32].

To calculate the cost of process migration we used LMBench [30] – a suite of portable benchmarks used to evaluate the performance of Unix machines – to calculate the cost of a typical context switch on our machines. We observed the cost to be approximately 12,000 to 15,000 cycles, depending on the warmth of the L2 cache. Considering that we only consider switching at a granularity of 1M instructions, this overhead proves to have a relatively small impact on the performance gained from migration. In theory, swapping cores should leave the L2 cache perfectly warm, as no data need be marked dirty or invalid. With this in mind, we assumed a relatively warm L2, and thus a penalty of 12,000 cycles for our experiments.

Ideal Scheduling The ideal schedule is the one that minimizes the total cycle count for both processes. Consider the situation where `gcc` takes 500K cycles to execute a given slice of instructions on the out-of-order processor and 1.5M on the in-order processor. Meanwhile, `astar` takes 1M cycles out-of-order and 2.2M cycles in-order. In this case, it is better to schedule `astar` on the out-of-order processor and `gcc` on the in-order processor. The ideal performance in terms of cycle counts for pairs of benchmarks is shown in Table 2. These values were collected by post-processing the performance information to determine the best possible process-to-core mapping for each 1M instruction timeslice. Table 2 shows the sum of the cycle count for the two processors over the run of the two benchmarks. The results shown are the pairwise comparison of the SPEC2006 integer benchmarks. We used the test inputs since they alone required over a week of simulation time. The experiments excluded `perlbench` and `xalancbmk` since SimpleScalar-x86 does not support all of the required system calls.

Throughout this section, we will present the performance of each policy in terms of its distance from ideal, which is defined in Equation 1. The sum cycle count for the schedule is the total cycle count for both processors while both benchmarks continued to run, plus any migration penalty incurred. The ideal sums are the total cycle counts for the ideal weaving of the two benchmarks, which are presented in Table 2.

Benchmark Pair	Cyc(M)	Benchmark Pair	Cyc(M)
gcc + libquant	1298	gcc + omnetpp	6686
gcc + astar	9707	gcc + bzip2	7582
gcc + go	1375	gcc + hmmer	8212
gcc + mcf	1928	gcc + sjeng	8016
libquant + omnetpp	1417	libquant + astar	1659
libquant + bzip2	1225	libquant + go	1173
libquant + hmmer	1345	libquant + mcf	1659
libquant + sjeng	1404	omnetpp + astar	7753
omnetpp + bzip2	6231	omnetpp + go	1520
omnetpp + hmmer	6835	omnetpp + mcf	2092
omnetpp + sjeng	6692	astar + bzip2	12034
astar + go	1611	astar + hmmer	12753
astar + mcf	21623	astar + sjeng	12356
bzip2 + go	1310	bzip2 + hmmer	11275
bzip2 + mcf	1890	bzip2 + sjeng	11464
go + hmmer	1421	go + mcf	1704
go + sjeng	1495	hmmer + mcf	1951
hmmer + sjeng	12208	mcf + sjeng	2022

Table 2. Sum of cycle count (in millions) using ideal weaving at 1M instruction swapping granularity.

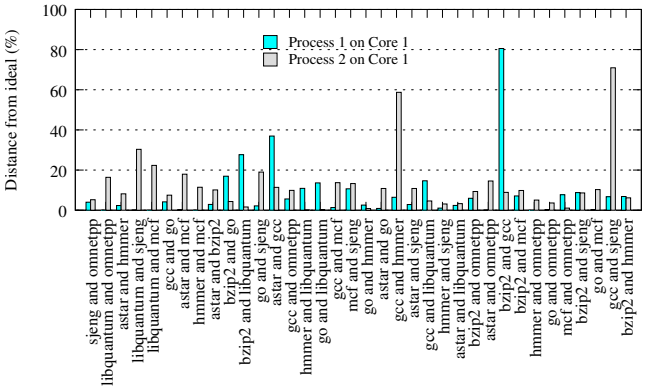


Figure 5. Coarse-grained scheduling performance reported as distance from ideal. (Lower is better.)

$$\text{distance from ideal}\% = \frac{\sum \text{scheduled cycles} - \sum \text{ideal}}{\sum \text{ideal}} \quad (1)$$

Static Scheduling Perhaps the simplest scheduling solution is to choose one core to execute each process in its entirety, never considering migration. Since this policy provides an effective context for comparison, we present its performance in Figure 5. An important observation from this figure is that it is often the case that a particular static mapping can result in a significant performance impact, therefore the scheduling decision becomes paramount. Meanwhile, both static schedules are (often significantly) worse than the ideal schedule.

This result implies that feedback-based scheduling may have a very positive effect, even in light of the migration overhead occurred. We next examined heuristics that allow the system to dynamically adapt to changing behavior in the programs as indicated by our hardware feedback mechanism.

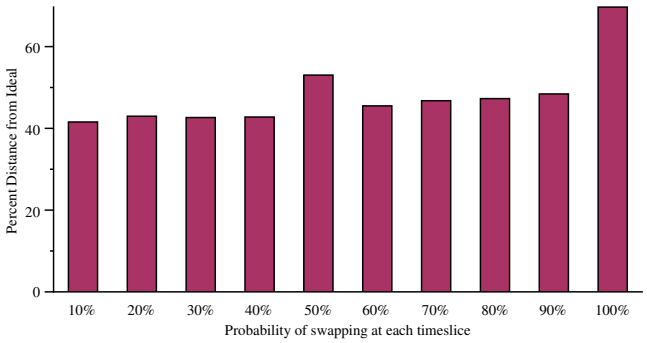


Figure 6. Performance of random fine-grained scheduling, averaged over SPECint2006.

4.1.2 Scheduling Heuristics

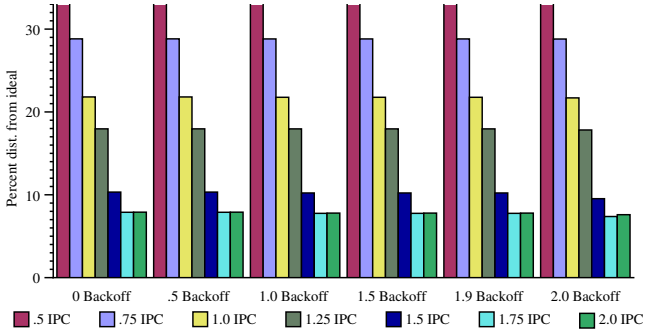
The most naïve migration-based scheduling approach is to randomly migrate programs at run time. Figure 6 shows the performance of random scheduling, taking the average of five runs. As expected, random scheduling performs worse than static scheduling due to unnecessary migrations. Armed with various extreme cases – ideal, static, and random – we now explore more sophisticated heuristics.

IPC-Based Scheduling Our first feedback-based heuristic is to use the instructions per cycle (IPC) of the out-of-order processor to gauge the effectiveness of the current executing program. The core tenet of this heuristic is that the out-of-order processor is the more valuable of the two cores, and therefore a low IPC on the out-of-order processor may indicate that another program can make better use of that resource. Using this heuristic, the system chooses whether to switch the program by looking at the IPC on the out-of-order processor for the last million cycles. If the IPC falls below the threshold, the system swaps the programs between the two cores.

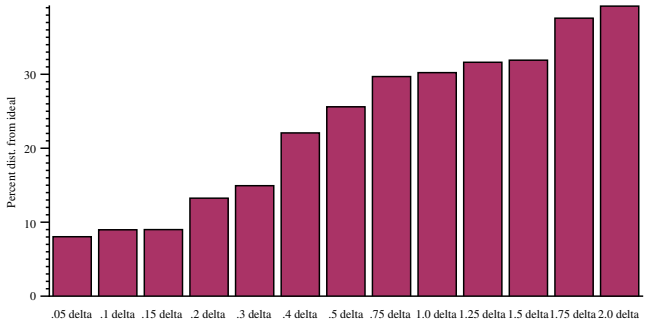
To guard against the thrashing case where both programs have low IPC on the out-of-order core, this heuristic implements an exponential backoff technique, so that after a switch, it doubles the amount of time it waits before attempting to switch again. Figure 7(a) shows the results of using IPC based scheduling as the percent distance from the ideal schedule. The graph is grouped by backoff rate, though performance is similar in most cases, which means that the programs do not quickly change IPC.

IPC-Delta Scheduling The next heuristic we examined is IPC-delta scheduling, which again uses the IPC of the out-of-order processor to decide whether the programs should be swapped. However, instead of waiting until the IPC falls below a certain threshold, this heuristic examines the last two timeslices and swaps if the IPC decreases by a certain amount between the two timeslices. Figure 7(b) shows the performance of the IPC-delta heuristic, with very small deltas performing the best, achieving within 10% of ideal on average. It would be possible to do exponential backoff with this heuristic as well, however given the small effect of the backoff with basic IPC scheduling, we chose not to examine it further.

Other Heuristics The heuristics presented in this section relate to the case of one high-performance out-of-order processor and one simpler in-order processor. However, heterogeneous systems with different parameters may require different triggering mechanisms. For example, a heterogeneous system with differing cache structures might require heuristics that gauge cache misses, and ensure that the program with the larger working set runs on the processor



(a) IPC-threshold performance



(b) IPC-delta performance

Figure 7. IPC-delta and IPC-threshold performance on SPECint2006. Larger deltas lead to a performance degradation.

with the larger cache. A similar case may occur for a pair of processors supporting slightly different instruction sets, i.e., one processor emulates floating point computations while the other handles floating point natively. In this situation, the scheduler must identify the frequency of floating-point instructions and reschedule accordingly.

4.2 Dynamic Response to dI/dt Problems

A second class of problems that is well-suited for a reactive solution is the case where problematic code regions can be identified by hardware and resolved by a VEE. As a representative example, we created a reactive software solution to enhance voltage stability, modifying only the code that the hardware flags as problematic. In this section, we describe the di/dt problem; we then present and analyze our reactive cross-layer solution.

Problem Description The di/dt problem occurs when there are dramatic periodic changes in supply current over short durations of time. These changes can greatly affect the reliability of a processor [5, 34]. Hardware designers have taken a conservative approach when designing the power supply systems and CPUs, therefore voltage emergencies are not yet a critical problem. However, these conservative approaches are more expensive to build. The International Technology Roadmap for Semiconductors (ITRS) lists noise management (di/dt, ground bounce, etc.) as one of their grand challenges for 2010 and beyond [1].

Modern processor designs complicate the di/dt problem because they regularly adjust power consumption by deactivating idle portions of the processor. However, enabling and disabling hardware features causes fluctuations in the current pulled by the processor, which can, in turn, lead to supply voltage variations. The di/dt problem can result in voltage threshold violations for the processor,

which can lead to timing problems, incorrect calculations, or even reliability problems if not corrected.

Recently proposed solutions throttle on-chip resources to reduce di/dt to the allowed operating range of the processor. This has the unwanted side-effect of decreasing performance of the application as a whole, when it may be a single loop causing the voltage emergency. We present a more lightweight yet complementary solution that only affects the section of the code causing the voltage emergency. By enabling processor feedback whenever the processor throttling solutions engage, the VEE can identify the section of code causing the voltage emergency. The VEE can then permanently modify the offending code. This removes the per-iteration performance penalty, but still provides the guarantees of the hardware-only approach.

dI/dt Implementation To collaboratively solve the problem of di/dt, the hardware must be modified to report the existence of voltage problems to the software. This is a relatively minor change because processors are already able to report high-level voltage information. With hardware that will generate a hardware trap in the case of a voltage emergency, a VEE can then take program state information from when the trap was generated, and use it to take reactive measures, rewriting the application code to mitigate the change in voltage. There are a number of ways to mitigate the voltage change, ranging from straight-forward `nop` insertion, to reimplementing instruction scheduling. For our experiments we chose the simplest approach of inserting `nops`. This solution is based on one that was proposed but was never implemented online [19].

For our experiments, we modified the SimpleScalar-x86 architectural simulator with Wattch [10] power monitoring extensions to report voltage emergencies to Pin. In order to trap to Pin when a voltage emergency occurs, SimpleScalar would have to signal Pin; however, the current version of SimpleScalar-x86 does not support signals. To solve this problem, we have Pin poll the hardware to see if an emergency has occurred. Because SimpleScalar controls the application's memory space, and therefore Pin's as well, SimpleScalar simply sets a flag at a predetermined memory location in the event of a voltage emergency. Pin then checks this memory location, and if a voltage emergency has occurred, Pin invalidates the current trace and rebuilds it. When the trace is rebuilt, it is built differently, using *instruction padding* to disrupt the voltage profile of the trace and mitigate the problematic voltage swings.

To test the effectiveness of our collaborative di/dt solution, we developed a synthetic "power virus" designed specifically to cause voltage emergencies. The program is shown in Figure 8 and is based on a similar virus presented by Joseph *et al.* for Alpha processors [23]. It consists of a single tight loop containing instructions carefully chosen to cause the processor's voltage to swing dramatically and periodically. Figure 10(a) then shows the voltage behavior representative portion of the power virus running on SimpleScalar. As the graph shows, the virus causes the voltage to repeatedly rise above and below the critical actuation thresholds (+/- 3% of the nominal voltage). In the case of previous hardware-only solutions, the actuation mechanism would have engaged on each and every loop iteration, significantly degrading performance.

To identify and alleviate the voltage problems, our VEE layer must instrument the code to monitor hardware indicators for power emergencies. The code for the plug-in Pin tool is shown in Figure 9. The Pin tool instruments backedges of the application to check the designated flag `ssEmergency`. When the hardware signals an emergency, the flag is set, and once Pin detects the change on the next backedge it invalidates all traces associated with that address. The next time that code region is translated, a `nop` is inserted between each instruction in the trace, disrupting the voltage profile.

Problem: Power Virus

```
BITS32

section .data
prime:  dq 100711433.0 ; constant large prime
three:  dq 3.0         ; constant large prime

section .text

global _start
_start:
  mov eax, [1000] ; loopcount
  mov ebx, 0xaaaaaaaa
  mov ecx, 0xdeadbeef
  fld qword [prime]

.loop:
  ; series of parallelizable load operations
  ; (high power section)
  mov ebx, [ecx]
  mov ebx, [ecx]
  mov ebx, [ecx]
  . . .
  ; series of sequential arithmetic operations
  ; (low power section)
  imul ebx, 2
  imul ebx, 2
  imul ebx, 2
  . . .

  sub eax, 1
  jne .loop

  mov eax, 1
  mov ebx, 0
  int 0x80
  hlt
```

Figure 8. The NASM assembly code that induces voltage emergencies.

However, even without the insertion of `nop` instructions, the addition of the instrumentation code could possibly disrupt the power virus. To determine these effects, we plotted the behavior of a representative portion (after Pin startup costs) of the power virus running with Pin instrumentation without any reactive response in Figure 10(b). The graph shows that the instrumentation has slightly affected the period of the virus, however, its magnitude is unaffected. That is, the virus is still causing voltage emergencies. Figure 10(c) shows a portion of same virus, running with reactive techniques enabled. The reactive `nop` insertion mechanism eliminates the voltage emergencies.

The benefit of our reactive approach is that it only impacts problematic code regions that exist and it permanently cures the problem. The cost of the extra padding instructions is much less than the overhead of hardware-only alternatives, where many of the resources are scaled back on every loop iteration.

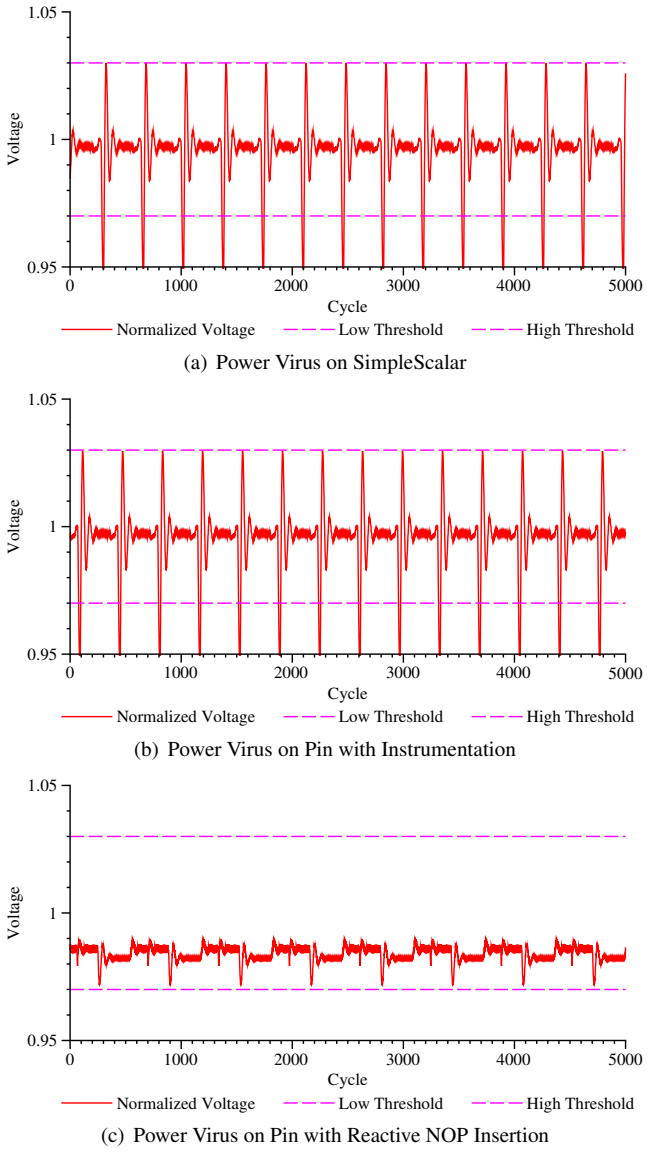
Though these two examples may seem straightforward, they clearly demonstrate the potential of using simple hardware feedback channels and a hardware-aware mediation layer. Additionally, we were able to test our design iteratively, both quantitatively and qualitatively, without the burden of hardware design and fabrication. Though these techniques obviously cannot supplant testing with new hardware, we are able to test and refine a variety of small hardware/software changes to isolate those changes most likely to elicit the most benefit, so that valuable chip design time and cost can be saved.

Solution: Dynamic dI/dt Correction

```
UINT32 ssEmergency = 0;  // set by underlying simulator
UINT64 nopCount = 0;     // dynamic count of nop instrs

  // Instruction padding routine for curing emergencies
VOID donop() { nopCount++; }
  // Reads the hardware performance counter in SS-x86
BOOL CheckForEmergency() {
  if (ssEmergency == 0) return FALSE;
  else return TRUE;
}
  // Forces Pin to regenerate the code at given address
VOID UnlinkAndInvalidate(ADDRINT ip) {
  CODECACHE_InvalidateTraceAtProgramAddress(ip);
}
  // Pin calls this function for every new basic block
  // One of two things occur before translating code:
  //  a) NO emergency: inserts detection code
  //        on every loop edge
  //  b) YES emergency: inserts NOPs between
  //        every instruction
VOID Trace(TRACE trace, VOID *v) {
  if (CheckForEmergency() == FALSE) {
    for (BBL bbl=BblHead(trace); BBL_Valid(bbl); bbl++){
      for (INS ins=InsHead(bbl); INS_Valid(ins); ins++){
        if (INS_IsDirectBranchOrCall(ins)) {
            INS_InsertIfCall(ins,IPOINT_TAKEN_BRANCH,
                (AFUNPTR) CheckForEmergency,IARG_END);
            INS_InsertThenCall(ins,IPOINT_TAKEN_BRANCH,
                (AFUNPTR) UnlinkAndInvalidate,
                IARG_INST_PTR, IARG_END);
        }
      }
    }
  }
  else {  // CheckForEmergency() == TRUE
    for (BBL bbl=BblHead(trace); BBL_Valid(bbl); bbl++){
      for (INS ins=InsHead(bbl); INS_Valid(ins); ins++){
        INS_InsertCall(ins,IPOINT_AFTER,
            (AFUNPTR)donop,IARG_END);
      }
    }
    ssEmergency = 0;
  }
}
int main(int argc, char * argv[]) {
  PIN_Init(argc, argv);
    // Register Trace as instrumentation routine
  TRACE_AddInstrumentFunction(Trace, 0);
    // Start the program, never returns
  PIN_StartProgram();
  return 0;
}
```

Figure 9. The C++ code for a Pin plug-in tool that detects and corrects voltage emergencies.



(a) Power Virus on SimpleScalar

(b) Power Virus on Pin with Instrumentation

(c) Power Virus on Pin with Reactive NOP Insertion

Figure 10. Voltage readings for the power virus running (a) natively on SimpleScalar (b) on Pin, and (c) on Pin with reactive NOP insertion.

5. Related Work

Systems such as Transmeta's Code Morphing Software [14] , BOA [2], DAISY [16], and work by Kim and Smith [24] also took the approach of co-designing hardware and virtual machine software. Our approach differs in two major respects: first, we take an evolutionary approach to developing our design by exploring lightweight changes to existing hardware; second, by taking hardware feedback into account, we explore re-translating program code. The Trident system [41] also explores extending hardware to trigger recompilation; their work primarily considers optimizing code for performance, whereas our approach also targets metrics such as power, reliability, and security.

Balakrishnan *et al.* [7] studied the performance effects of the asymmetry in heterogeneous chip multiprocessors (CMP). Their results showed that in some cases, unpredictable performance due to asymmetry could be fixed either by making the operating system scheduler aware of the asymmetry or by rewriting the application. In either case, a collaborative system such as the one presented here would be able to make the necessary changes at run-time without needlessly complicating the OS scheduling algorithm, or developing hardware-specific versions of each OS.

The topic of process migration has been heavily studied in the distributed computing community. They explore the migration of a process from one computer to the next through typically high-latency interfaces such as TCP/IP over Ethernet. Many of the obstacles and challenges that prove prohibitive to their work are absent when considering CMPs. For example, communications overhead is orders of magnitude lower in CMPs, since CMPs can use processor-internal buses, whereas distributed systems must use much higher-latency interconnects. Similarly, the problems of providing consistent shared memory and transferring program context from one node to another are non-trivial in the distributed context; com-

plex invalidation schema and high communication costs mean that both these tasks incur significant overhead. By contrast, CMPs share the same physical memory, and can in fact share a single physical cache, meaning that both these tasks can have orders of magnitude lower overhead in a CMP context. This feature introduces more optimization opportunities. Since the cost of migration is much lower, we can migrate more frequently.

Kumar *et al.* [26, 27] studied assignment and migration of threads across a heterogeneous CMP for improving throughput and reducing power. They proposed heuristics involving large-scale IPC changes either in a single application or absolute value over all currently executing applications, and then required sampling all possible matchings of processes to processors before settling on an ideal configuration. We have shown that the problem also occurs on x86 architecture, meanwhile we did a sensitivity study of the IPC migration thresholds. Work by Constantinou *et al.* [13] suggests that the overhead of register state migration and private cache management is negligible if the period between migrations is at least 160K cycles. Michaud [31] presented a method of distributing cache lines across multiple cores and then migrating execution to improve performance by having a larger total cache size.

The heterogeneous migration case study presented in this paper implicitly attempts to perform some form of phase detection and analysis in deciding on which core to schedule each process. Sherwood *et al.* presented work on automatically detecting and predicting phase changes over large intervals [36, 37]. Dhodapkar and Smith suggested using a hardware mechanism along with a virtual machine monitor to detect phase changes to optimally reconfigure hardware [15]. Our work assumes a higher-level perspective than that of Dhodapkar and Smith in that we schedule across multiple cores rather than altering the configuration of current hardware, which may lead to greater flexibility as the number of available cores increases.

There has been some work suggesting solutions to the di/dt problem. In 1999, Toburen [38] proposed heuristics for reducing the number of bit-flips between successive instructions in the execution core of high performance microprocessors. In 2002, Grochowski *et al.* [17] proposed disabling and enabling functional units to reduce voltage variation based on a complex calculation of the voltage. In 2003, Joseph *et al.* [23] extended this idea to use on-chip voltage sensors rather than online calculations, as part of the voltage control mechanism. Hazelwood and Brooks proposed a dynamic software-based solution to the di/dt problem [19]. Our work, and that of Gupta et al. [18, 33] can be considered an extension and implementation of that effort.

6. Future Work

Our simulation infrastructure allows us to investigate many symbiotic optimizations, thus our next step is to investigate solutions for additional computing problems. For example, we are interested in exploring how code can be dynamically rewritten to more evenly distribute the power and thermal pressure on a processor core. Information about programmatic hot spots can be provided to a VEE by hardware-level temperature and power sensors. With this information, the VEE can take action to relieve some of this pressure, such as performing partial loop unrolling to take pressure off of the branch predictor. This kind of intelligent code transformation requires extensions to both hardware and software to get more accurate readings about which code is adversely affecting the hardware.

Another difficult challenge we intend to explore is how to reduce the simulated mediation overhead by running the VEE and simulation layers independently, rather than atop one another. By running the VEE alongside the simulator and transferring information laterally, we believe that we can make the overhead costs additive, rather than multiplicative.

7. Conclusions

Collaboration between hardware and software can achieve results that neither could realize in isolation. As processors become more complex and as metrics of performance other than execution speed become more important, unanticipated problems will arise which can only be dealt with at run time. To this end, we have created a simulation framework for exploring this design space, which gives us the flexibility to trade off speed and accuracy for environmental control when exploring novel collaborative designs.

We have demonstrated this suitability of reactive solutions to two disparate problems facing system designers today. In particular, we explored process migration on a virtual heterogeneous multicore system. We were able to design and evaluate migration methodologies that outperform the average static decision, even without prior knowledge of program behavior, all by taking advantage of hardware performance counters. We were also able to demonstrate how program knowledge and minimal modifications to commodity hardware could lead to performance and reliability improvements by using a reactive approach to the di/dt problem. By combining hardware feedback with a virtualization layer in software, we were able to target specific regions of concern and modify them to provide a permanent solution to the di/dt problem.

Overall, there is significant potential in the future of cross-layer collaborative design. Armed with our experimental framework, we intend to continue to develop, plan, and test the next generation of hardware and software that can effectively work together to improve overall system performance.

Acknowledgments

This work was made possible by the National Science Foundation (CNS-0747203 and CCF-0811302), the Semiconductor Research Corporation (GRC Task 1790.001), monetary awards from Microsoft, Google, the Woodrow Wilson Foundation, and the T100 Group, and equipment and software donations from Intel.

References

[1] Int'l technology roadmap for semiconductors. *Semiconductor Industry Association*, 2005.

[2] E. R. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritz, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*, Dec 2000.

[3] AMD Corporation. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*. Publication #24593, SepSept 2003.

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, Feb 2002.

[5] D. Ayers. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *8th Int'l Symp. on High-Performance Computer Architecture*, pages 7–16, Cambridge, MA, 2002.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM Conf. on Programming Language Design and Implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000.

[7] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *32nd Int'l Symp. on Computer Architecture*, pages 506–517, Madison, WI, 2005.

[8] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on itanium-based systems. In *36th Int'l Symp. on Microarchitecture*, pages 191–201, San Diego, CA, Dec 2003.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM symposium on Operating Systems Principles*, pages 164–177, New York, NY, 2003. ACM Press.

[10] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Int'l Symp. on Computer Architecture*, Vancouver, British Columbia, Canada, 2000.

[11] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int'l Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[12] P. P. Bungale and C.-K. Luk. Pinos: A programmable framework for whole-system dynamic instrumentation. In *3rd Int'l Conf. on Virtual Execution Environments*, July 2007.

[13] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Computer Architecture News*, 33(4):80–91, 2005.

[14] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization*, pages 15–24, San Francisco, CA, March 2003.

[15] A. Dhodapkar and J. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *Int'l Solid State Circuits Conf.*, San Francisco, CA, Feb 2002.

[16] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Int'l Symp. on Computer Architecture*, pages 26–37, Denver, CO, June 1997.

[17] E. Grochowski, D. Ayers, and V. Tiwari. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *8th Int'l Conf. on High Performance Computer Architecture*, pages 7–16, Boston, MA, 2002.

[18] M. S. Gupta, V. J. Reddi, G. Holloway, G.-Y. Wei, and D. Brooks. An event-guided approach to handling inductive noise in processors. In *Design, Automation, and Test in Europe Conference (DATE-09)*, Nice, France, April 2009.

[19] K. Hazelwood and D. Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *Int'l Symp. on Low-Power Electronics and Design*, pages 326–331, Newport Beach, CA, Aug 2004.

[20] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[21] E. Humenay, D. Tarjan, and K. Skadron. Impact of process variations on multi-core architectures. In *2007 Conf. on Design, Automation, and Test in Europe (DATE)*, Nice, France, April 2007.

[22] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Order #253668-019, March 2006.

[23] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *9th Int'l Symp. on High-Performance Computer Architecture*, pages 79–90, Anaheim, CA, 2003.

[24] H.-S. Kim and J. E. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Code Generation and Optimization*, pages 25–35, San Francisco, CA, March 2003.

[25] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, San Francisco, Aug 2002.

[26] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *36th Int'l Symp. on Microarchitecture*, pages 81–93, San Diego, CA, 2003.

[27] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *31st Int'l Symp. on Computer Architecture*, pages 64–76, München, Germany, 2004.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conf. on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[29] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. 35(2):50–58, 2002.

[30] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conf.*, pages 279–294, San Diego, CA, 1996.

[31] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *10th Int'l Symp. on High Performance Computer Architecture*, pages 186–196, Madrid, Spain, 2004.

[32] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, 1991.

[33] V. J. Reddi, M. S. Gupta, G. Holloway, M. D. Smith, G.-Y. Wei, and D. Brooks. Voltage emergency prediction: A signature-based approach to reducing voltage emergencies. In *International Symposium on High-Performance Computer Architecture (HPCA-15)*, Raleigh, NC, February 2009.

[34] V. J. Reddi, M. S. Gupta, K. K. Rangan, S. Campanoni, G. Holloway, M. D. Smith, G.-Y. Wei, and D. Brooks. Voltage noise: Why it's bad, and what to do about it. In *5th IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, Palo Alto, CA, March 2009.

[35] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *Code Generation and Optimization*, pages 36–47, March 2003.

[36] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, San Jose, CA, 2002.

[37] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Int'l Symp. on Computer Architecture*, pages 336–349, San Diego, CA, 2003.

[38] M. C. Toburen. Power analysis and instruction scheduling for reduced di/dt in the execution core of high-performance microprocessors. Master's thesis, North Carolina State University, 1999.

[39] V. Venkatachalam, C. Probst, and M. Franz. A new way of estimating compute boundedness and its application to dynamic voltage scaling. *Int'l Journal of Embedded Systems*, 1(1/2/3):64–74, 2006.

[40] M. Yourst. PTLsim users guide and reference: The anatomy of an x86-64 out of order microprocessor. Technical report, SUNY Binghamton.

[41] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multi-threaded dynamic optimization framework. In *14th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 87–98, St. Louis, Missouri, 2005.