Motivating Software-Driven Current Balancing in Flexible Voltage-Stacked Multicore Processors

A thesis presented

by

Svilen Kanev

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Bachelor of Arts

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May 2012

Thesis advisor
David M. Brooks and Gu-Yeon Wei

Author
Svilen Kanev

Motivating Software-Driven Current Balancing in Flexible Voltage-Stacked Multicore Processors

Abstract

Computing hardware has been shifting to a multicore paradigm. One extreme of that approach is many-core computing, where a large number of simple processor cores execute in throughput mode. However, this paradigm is not ubiquitous and latency-bound application do have a place in the workloads of the future. We propose a processor organization that allows an array of homogeneous cores to explicitly trade off between throughput- and latency-centric modes. By organizing the cores of a processor in a *flexible voltage stack (FVS)*, one can achieve per-core control over voltage, changing the optimal operation point of cores, but without resorting to per-core voltage regulators, which could incur significant overheads. This work is an initial step in evaluating a flexible voltage-stacked architecture. We motivate such an architecture as an alternative to dynamic voltage and frequency scaling. We show that the high-voltage delivery of power to a voltage stack can eliminate losses in power delivery and lead to significant overall power savings. However, a practical implementation of a FVS architecture requires careful current matching across a single stack. We propose a hardware-guaranteed, software-augmented solution called *current balancing* that intelligently schedules threads to achieve such matching. We draw the methodology for evaluating this solution from *voltage smoothing* – a scheduling technique we have developed to smooth supply voltage fluctuation. Since no systems have been built to implement a FVS architecture, we describe a simulation framework with sufficient level of detail to generate current traces required for fully evaluating a FVS system. To show that this in-order x86 simulator, XIOSim, is representative of current systems, we developed a rigid simulator validation methodology. This leads us one step further to a complete end-to-end evaluation of the FVS paradigm under real workloads.

Contents

Citations to Previously Published Work

Elements of Chapter 3 appear in the following three papers:

> A System-Level View of Voltage Noise in Production Processors
> S. Kanev, V. J. Reddi, W. Kim, S. Campanoni, M. Smith, G.-Y. Wei, D. Brooks
> ACM Transactions on Architecture and Code Optimization (TACO). Under submission.

> Voltage Noise in Production Processors
> V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. Smith, G.-Y. Wei, D. Brooks
> IEEE Micro, 2011

> Voltage Smoothing: Characterizing and Mitigating Voltage Noise in Production Processors via Software-Guided Thread Scheduling
> V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. Smith, G.-Y. Wei, D. Brooks
> 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010

Most of Chapter 4 is under submission as:

> XIOSim: Detailed Integrated Power-Performance Modeling of Mobile x86 Cores
> S. Kanev, G.-Y. Wei, D. Brooks

Chapter 1

Introduction

Over the last decade, processor designs have undeniably shifted towards multiple cores, mostly in order to be able to improve performance after frequency has stopped scaling according do Dennard's predictions [10]. There has been an observable shift toward throughput-oriented computing, where multiple cores are able to execute the same workload with higher power efficiency than a single monolithic core. However, not all programs are easily ameanable to parallelization or to a throughput-centric paradigm. Thus, single-threaded performance is still relevant and future processors would most likely need a mechanism to adjust to the tradeoffs between single- and multi-threaded operation in different applications.

The most common mechanism for such adaptation is dynamic voltage and frequency scaling (DVFS). Since power and operating frequency scale differently with supply voltage, dynamically adjusting voltage and frequency allows for operation at different points of the power/performance tradeoff space. However, in current systems DVFS is performed most commonly on the operating system layer, paying a high latency cost. Furthermore, while per-core voltage adjustment has been proposed [18], it suffers lower conversion effectiveness and can be costly in terms of on-chip area. Therefore, per-core DVFS still has not received adoption and systems can only adjust frequncy on a per-core basis, which has much smaller potential gains.

In this work, we look into an alternative mechanism that could allow automatic performance/power adjustement based on the workload being executed. The defining feature of this approach is connecting a fraction of the cores in a processor to the voltage planes in a serial configuration that we refer to as a *voltage stack*. Due to the fixed voltage supply

and charge conservation, such a configuration can achieve automatic voltage regulation that operates on a per-core granularity and on a fine timescale. In the most general case, when cores' connection topology to the voltage plane can be reconfigured, we call this technique *flexible voltage stacking (FVS)*.

In addition to being an alternative to DVFS mechanisms, flexible voltage stacking has the effect of decreasing chip current consumption at constant power by a factor of several times because the chip is being supplied at a higher voltage. The reduction in current alleviates pressure on the processor package, significantly reducing I^2R power losses from the power delivery network, or allowing the use of fewer package pins for power delivery. This could reduce the burden on pin-limited designs, such a current server processors or future 3D-stacked systems.

On the other hand, a voltage-stacked configuration can suffer in the presence of too much workload variability. Without a current compensation mechanism, significant diversity in the current consumption of the different levels of a single stack can cause too low or too high supply voltage for cores in the stack, leading to issues with correctness or to transistor degradation. In the event that such a compensation mechanism is present, however, it is most likely to waste excessive power. Therefore, there is an efficiency limit on the amount of workload difference that a voltage-stacked system can tolerate.

This work is an initial effort in characterizing the system-level implications of a future multiprocessor that is organized as a voltage stack. It aims to evaluate the potential of using software to control the amount of workload heterogeneity in terms of current consumption – a term we refer to as *current balancing*. Being preliminary work, it pays significant attention to developing methodology – it looks into the general approach of exposing electrical events to the software layer, as well as into simulating the buildling blocks of a future voltage-stacked system.

We begin by defining the characteristics of such a system in Chapter 2. We illustrate its potential in regulating core operating voltage and eliminating power delivery losses. We point out the limits of workload variability that such a system can handle. We then move on to show how on a real current system software can be used to mitigate workload variability in the context of a different electrical phenomenon. Chapter 3 demonstrates *voltage smoothing*, a technique where a thread scheduler intelligently chooses thread combinations to be executed in order to minimize supply voltage variation. If we are to apply a similar approach to balance current consumption in a future voltage-stacked system, we

need a way to evaluate such a core configuration. Chapter 4 presents a detailed processor power/performance simualation environment we have developed that has a sufficient level of detail to generate current consumption traces. We also present an extensive validation methodology and apply it to our simulator, which proves that the models we developed are well-grounded with current systems.

The contributionis of this work are as follows:

- A preliminary assessment of a multicore where cores are connected to the power supply planes in a flexible voltage stack. We show that such a configuration can introduce performance heterogeneity with homogeneous building blocks, and can potentially have chip-level power and cost savings. However, it is inherently fragile with respect to current consumption variation among cores.

- A hardware-software approach of dealing with circuit-level details that allows us to match software activity with voltage and current events. We validate this approach with respect to voltage noise in a conventional real system and show that a noise-aware operating system scheduler can alleviate some amount of voltage noise.

- A highly-detailed simulation environment for performance and power estimation of multicore x86 systems that focuses on simple in-order cores exhibiting smaller current variation. To the best of our knowledge, it is the only publicly available simulator that models such cores in sufficient level of detail for estimating current traces and voltage events.

Chapter 2

Flexible voltage stacking

We start this work by assessing the need for performance and power adjustment in future many-core systems. We show that workload heterogeneity is likely to require a technique that allows different cores in such a system to operate at different points in the power/performance tradeoff space. We move on to present *flexible voltage stacking* as a novel multicore design pattern that allows such adjustment. It also has the potential to decrease power losses and manufacturing costs associated with the processor package. However, due to its inherent feedback nature, a potential voltage-stacked system is vulnerable to workload variability, which could cause significant reliability problems. This motivates our search for a software technique that smooths cores' current consumption.

2.1 A need for fine-grained power control

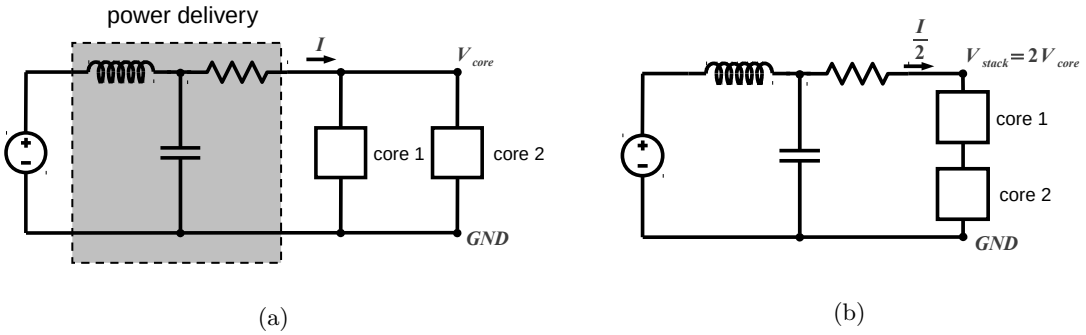
After power has established itself as a significant metric in microprocessor design, there has been a tendency to produce simpler cores because of their better performance per watt. Simple, in-order cores have been the norm for mobile applications, but recent industrial designs [26, 34, 35] explore the opportunity of replacing traditional high-end processor designs by using a multitude of small cores for parallel workloads found in databases, multimedia or networking. The common feature among the workloads that drive this transition to many-core designs is the focus on multi-threaded throughput, as opposed to single-threaded latency. In a datacenter context, typical workloads that lend themselves to the throughput-centric approach are web servers, database systems or explicit latency-hiding programming models such as MapReduce.

However, emerging datacenter workloads are increasingly latency-sensitive. This effect is due to increased computational intensity in novel applications like search, machine learning and natural language processing, as well as to the quality of service (QoS) requirements that client-facing online services must have. Mars et al. [25] demonstrate this effect while considering a typical Google production datacenter workload mix. From the 17 workloads that they characterize, only 5 are considered latency-insensitive. Furthermore, Reddi and Lee [15] show that running such emerging applications on simple processor cores, while undoubtedly more power efficient, results in significantly degraded quality of service metrics.

The presence of such workload diversity, combined with the "power wall" that processor designers are facing, suggests that, if a single platform is to drive the datacenter infrastructure of the future, it needs to support some form of heterogeneity in order to be energy-efficient. One such form of heterogeneity is integrating structurally different cores on the same die and using them at the appropriate time. We will not consider this approach mostly due to the high design and complexity costs of creating specialized hardware.

Dynamic voltage and frequency scaling (DVFS) is another classical approach that tailors structurally identical hardware to the workload characteristics. Since core frequency has an approximately linear relationship with supply voltage and power consumption is approximately quadratic with voltage, a change in voltage can change the energy-optimal point of operation for a core, effectively allowing for a single knob that trades off power consumption and performance. If DVFS is applied separately for cores in the system, this could provide the heterogeneity that future applications require, with each core being near energy-optimal operation for the thread it is executing.

Current systems make heavy use of DVFS, but on the chip level, which has a significantly smaller efficiency potential since it requires coordinated activity among all cores in the system. Implementing per-core DVFS has proven a challenge. If the different voltages that such a scheme requires are generated off-chip, this would create area pressure from multiple board-level regulators, which tend to be bulky, as well as pressure on the power delivery subsystem of the processor package, which has to supply a large number of independent voltage levels. While multiple on-chip voltage regulators can alleviate those problems, such designs are still in their infancy and reported conversion efficiencies are significantly lower than those of off-chip designs [17].



(a) (b)

Figure 2.1: Difference in organizing the cores in a dual-core system (a) conventionally in parallel; and (b) as a voltage stack.

2.2 Organizing cores in a flexible voltage stack

Alternatively to DVFS, heterogeneity in core supply voltage can be introduced by rethinking the way in which cores are connected to the supply voltage and ground grids. Figure 2.1 sketches out the difference between a conventional dual-core system and one where the two cores are connected in series between the power and ground plane, forming a *voltage stack*[1]. In the simplest case, the middle node between the two cores, V_{mid}, is left floating and settles to a value between GND and the supply voltage V_{stack}. Due to charge conservation, the current trough the cores in a single stack is the same. Since for correct operation the voltage difference across each core should remain approximately the same as in a conventional design, this requires that the new supply voltage $V_{stack} = V_{core} \times N$, where V_{core} is the supply voltage in the conventional design and N is the height of the stack. This implies that, for the same power, a voltage-stacked chip consumes N times smaller current, but is supplied at N times lower voltage than a traditional design.

Several similar designs have been proposed in the literature – for example, Rajapandian et al. [30] demonstrated stacking of multiplier units, while Gu [13] concentrated on the balancing that such a stack requires. More recently, Lee [20] has demonstrated a test chip in which programmable current sources are used to model current consumption of a multicore processor organized as a voltage stack. These early designs demonstrate that creating such systems is, if not easy, at least possible and design efforts exist in resolving

[1]Note that the term *stacking* only refers to stacking the cores with respect to voltage supply and ground – it has no relation to the more established term *3D stacking*, although 3D stacking could be a convenient implementation for the voltage stacking design pattern.

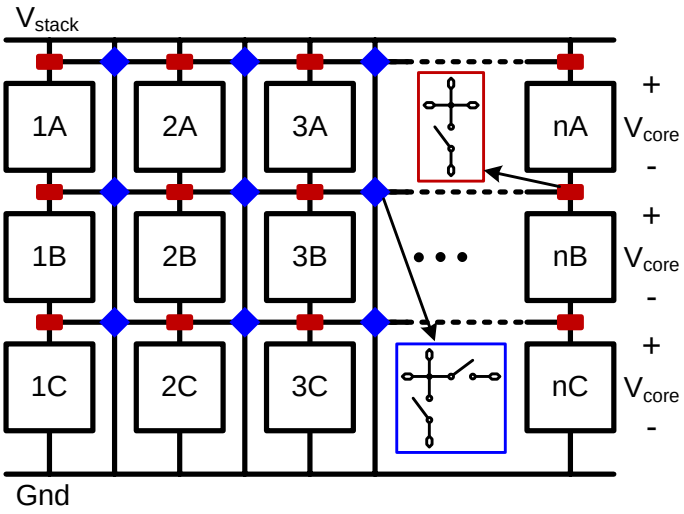


Figure 2.2: A multicore organized as a flexible voltage stack with switches allowing the making and breaking of supply rail connections between different components.

the technical challenges associated with them.

If we believe that future systems can be built using voltage stacking, we can take the concept one step further and introduce programmable heterogeneity in the way cores are connected to the voltage supply network. We illustrate such a *flexible voltage stack (FVS)* in Figure 2.2, where three layers of cores are stacked between the supply voltage and ground. The flexibility in such a system arises from a network of programmable switches that can make or break connections between different layers and cores in the stack. Figure 2.2 shows two types of such switches. The ones illustrated by red rectangles are similar to current power gating transistors and can disconnect a core from the stack. Broadly speaking, there may be two classes of reasons to do that – power savings if the core to be disconnected is idle, or increased single-thread performance if another core in the same stack requires a higher voltage to execute latency-critical workloads. Additionally, the switches represented by blue diamonds can be used to short out disconnected cores such that a higher voltage difference reaches the other cores in stack, or to connect separate stacks together for voltage equalization.

The heterogeneity introduced by an FVS system can produce an alternative to per-core voltage scaling as a way to meet workload variation. Configuring the switch network in different ways can lead to different voltages being supplied to the cores in the voltage

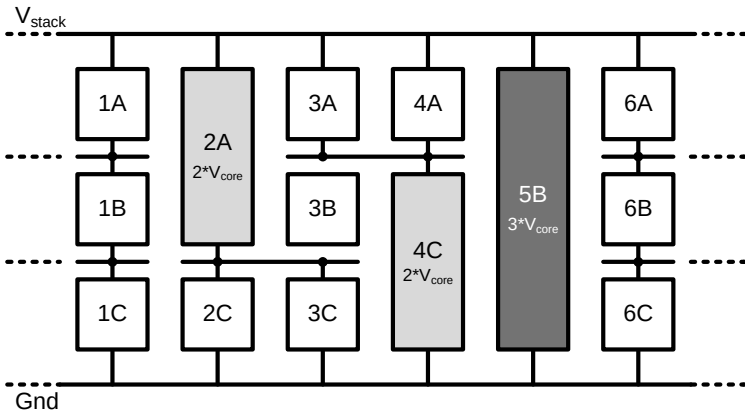


Figure 2.3: Different supply voltage configurations achievable by a FVS system.

stack. Examples of this for a 3-layer system are shown in the stacks of Figure 2.3. In this example, there are three voltage settings available for cores, with corresponding performance differences. In the first column, voltage is distributed evenly and each core runs off the standard voltage $1/3V_{stack}$. On the other extreme, in column 5, both the top and bottom core are shut off and the middle one gets the full supply voltage V_{stack}. Similarly, in columns 2, 3 and 4, all middle cores are gated and different cores in the top and bottom rows operate off of $2/3V_{stack}$ and $1/3V_{stack}$. Notice that some cores need to be shut off in order for others to have higher supply voltage and, consecutively, single-threaded performance. This may seem wasteful in terms of chip area, but what is commonly referred to as "the dark silicon problem" [14] predicts that such behavior will be common for the near future – being able to power a whole chip with each component consuming its peak power is unlikely. Furthermore, in this particular example there are only a few allowed configurations and voltage control is very coarse-grained. One can expect that a higher number of stack layers would allow more combinations and smaller voltage adjustment steps.

Even with large adjustment steps, a FVS approach manages to create heterogeneity in a homogeneous core substrate, and can potentially have fewer overheads than per-core DVFS with on- or off-chip voltage regulators. In a FVS system, only one voltage is delivered to the chip, not incurring the board area overheads of multiple off-chip regulators and the potential cost overheads of delivering multiple voltages on-chip. DVFS with on-chip regulators also does not suffer from this issue. However, on-chip regulators can incur significant chip area overheads [17]. Similarly, a FVS design would also incur area overheads com-

ing from the reconfigurable network of power switches. However, current systems already have power FETs for power gating that have a very similar purpose to the switches and they can probably be re-used. Thus, we are not expecting enormous overheads from the reconfigurable network.

2.3 Decreasing package losses and costs

In addition to providing a platform for energy-efficient execution under application heterogeneity, a FVS system can also be beneficial from a package design point of view. Supplying a significantly smaller current can reduce power loss in the processor package as well as ease pin constraints in possible future power-pin-limited designs.

Power loss The power delivery component of the processor package which is responsible for bringing power to the die has non-zero impedance. At high frequencies, this non-zero impedance manifests itself in the di/dt problem, which we will touch on in Chapter 3. The effect of this impedance at low frequencies is power inefficiency – when constant supply current I flows through the package with apparent resistance R, an I^2R amount of power is dissipated from the package.

Such power delivery loss can be significant in high-end designs that consume large amounts of current. As an example, a current Intel SandyBridge™ server processor uses an LGA1156 package with low frequency resistance of $1.4m\Omega$ [9] and is rated for maximum power consumption of $95W$ at supply voltage of $\approx 1V$. Thus, at theoretical maximum current consumption, the amount of power lost in the package can be as high as $12.6W$, or 13.3% of thermal-design power (TDP).

In the previous section, we showed that in a FVS organization, supply voltage increases by a factor of the number of layers in the stack N, leading to N times smaller current consumption and N^2 times smaller power delivery loss, all else considered equal. In the case of the SandyBridge™ server example, even a two-level stack can achieve up to 10% power savings at maximum load.

Pin count and package cost Based on a survey of processor packages used by current Intel high-end server and desktop processors, we have determined that power and ground pins constitute 50-60% of all package pins. This is mostly done in order to minimize I^R loss

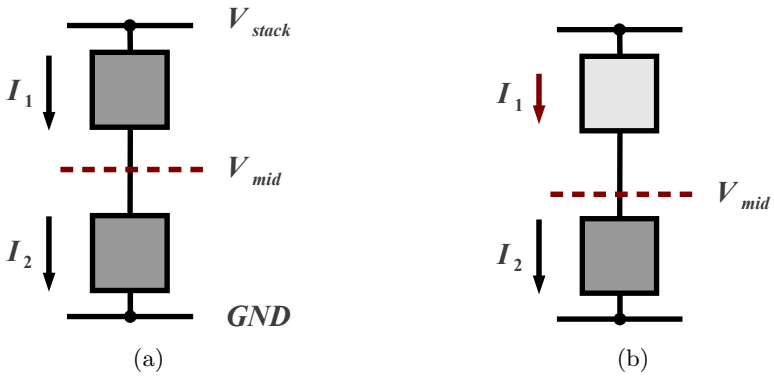


Figure 2.4: Implicit feedback loop in a voltage-stacked system: (a) in a steady state with balanced current consumption; (b) automatically adjusting voltage to current imbalance.

by spreading the current consumed by the whole chip. If by consuming smaller amounts of current, a FVS system decreases the requirement of package pin counts, designers can use cheaper packages with a smaller number of pins and realize overall chip cost savings.

Even if current systems are not power-pin-constrained, this may not be the case in future integrated systems. Higher degrees of integration cause pressure on package power delivery, since off-chip regulators need to be able to provide different and separate supply voltages for the different components. This is further exacerbated in a 3D stack containing separate voltage domains for cores, GPUs and DRAMs. Additionally, such a stack has a smaller area for connecting to the package due to the non-ignorable overhead of through-silicon vias, which have been shown to occupy 10x the area of a standard cell at 45nm [28], which could add to pin pressure.

2.4 Stability implications

While the potential benefits of a FVS organization can be substantial, achieving stable operation in the presence of workload variation can be a challenge. We illustrate this by example in Figure 2.4 for two cores connected in a voltage stack. Since core current consumption is workload dependent, when the two cores are identically loaded the voltage drops across them balance out and the floating potential V_{mid} settles halfway between V_{stack} and GND. However, in case the two cores execute workload that differ substantially, their current consumption may diverge. Because of charge conservation in this closed system,

current cannot change and the voltage difference across both cores is what adjusts. To quantify this relationship, we can assume a typical core dynamic power consumption that is dependent on the workload and the squared core supply voltage:

$$P(\alpha, V) = \frac{\alpha}{2} C V^2 f \tag{2.1}$$

We are assuming that the core frequency f and effective capacitance C do not change and that the activity factor $\alpha \in [0; 1]$ captures workload dependence. In that case, for the current consumption of the two cores in the example:

$$I_1 = \frac{\alpha_1}{2} C (V_{stack} - V_{mid}) f$$
$$I_2 = \frac{\alpha_2}{2} C V_{mid} f$$

Charge conservation implies $I_1 = I_2$, which leaves us with:

$$\begin{aligned} V_1 = V_{stack} - V_{mid} &= \frac{\alpha_2}{\alpha_1 + \alpha_2} V_{stack} \\ V_2 = V_{mid} &= \frac{\alpha_1}{\alpha_1 + \alpha_2} V_{stack} \end{aligned} \tag{2.2}$$

This is an inherent feedback loop in the system. Consider that the top core is executing a less power-intensive section of code and α_1 decreases. This would cause the middle potential V_{mid} to move closer to GND effectively increasing the voltage drop across core 1 and decreasing the one across core 2 (Figure 2.4b). Increased voltage across core 1 signals that it can be run at a higher frequency, but the low activity factor α_1 suggests that this is not necessary. Even worse, the low activity across core 1 causes lower voltage across core 2. If core 2 is running close to the maximum permitted frequency for a balanced V_{mid}, this could even cause correctness errors.

Lee et al.[20] have demonstrated qualitatively similar feedback behavior in a 150nm voltage-stacked test chip. The difference between their measured model and this analytical one is in the exponent of the activity factors, which has to do with the way they model cores and is not important for this discussion.

On a higher level, such feedback behavior is in contrast with the motivation for flexible voltage stacking. The natural feedback loop across a single stack is pushing for higher voltage across cores that require less performance, while we introduced flexibility in the voltage stack with the opposite purpose – to be able to provide a higher voltage for the cores that require more performance. If we are to create a stable and reliable flexible voltage system that allows such adjustments, we need a way to overcome the natural feedback loop.

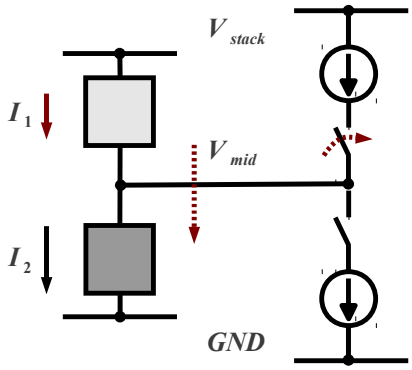


Figure 2.5: Using a shunt regulator to overcome the natural feedback of a voltage stack.

2.5 Guaranteeing efficient stability

As we saw in the previous section, current consumption variability in the different layers of a voltage stack can be a cause of inefficiency or even incorrectness for some of the participating cores. In this section, we show a simple circuit-level mechanism that can guarantee that floating voltages in the stack float within bounds at least guarantee correct execution for all cores. Since the mechanism itself can be a source of power inefficiency, we motivate invoking it as infrequently as possible by relying on intelligent thread scheduling.

One example of a simple mechanism that smooths current variability across a voltage stack is the shunt regulator shown in Figure 2.5. It injects or extracts current in the middle layer of the stack based on the difference between the currents I_1 and I_2. A simple control scheme for this regulator could implement a bang-bang controller that senses the difference between V_{mid} and the equilibrium voltage ($V_{stack}/2$ in this case). In case this difference is below a negative threshold, the upper switch is closed and current is injected in the stack. If the difference is above a positive threshold, current is extracted from the stack after closing the bottom switch. These thresholds can be set appropriately, such that there is a bound on the floating node voltage V_{mid}. If cores' frequencies are set while taking into account this bound correct execution can be guaranteed.

The shunt regulator provides us with a correctly executing baseline for examining a voltage-stacked system. However, it is inefficient in terms of consumed power. Depending on the actual implementation of the current sources and the switches that connect them to the voltage stack, the power loss from constantly injecting and extracting current can be significant. In order to minimize these overheads, we can make sure that the feedback loop

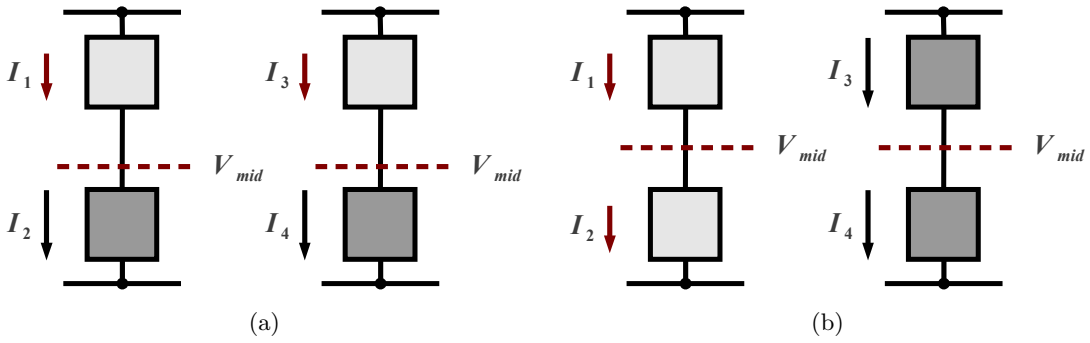


Figure 2.6: Two thread combinations that a current-balancing scheduler can choose from: (a) spreading similarly performing threads across stacks; (b) concentrating them inside stacks.

triggers as fewer times as possible.

This motivates a software scheduling solution that we call *current balancing*. If a thread-to-core scheduler is aware of the underlying voltage stacking processor topology, it can choose to assign threads such that the ones that end up on the same stack have small differences in current consumption. Such a schedule ensures that the stability mechanism is activated as little as possible, minimizing its power loss.

An example of this approach is shown in Figure 2.6. Of the four threads that need to be scheduled to the four cores, two have relatively high current consumption and two are not so computationally intensive. Therefore the scheduler has to make a decision between the two cases illustrated in Figure 2.6a and Figure 2.6b – distributing similarly performing threads across voltage stacks, or keeping them at the same stack. Since the second case results in more balanced stacks, triggering the current compensation mechanism fewer times, it should be preferred.

The rest of this work builds up towards an evaluation of software current balancing. In Chapter 3 we develop the methodology of such an evaluation trough analogy – we look into using thread-scheduling to minimize voltage noise in resilient architectures. The similarities between a voltage-stacked and a noise-resilient architecture allow us to use the same methods as in voltage smoothing. Then, in Chapter 4, we show the development and validation of a detailed power/performance model that can simulate a FVS system and quantify the tradeoffs that we sketched out in this chapter.

Chapter 3

Analogy: software scheduling for voltage noise

In this chapter, we develop the necessary methodology to evaluate a software solution for current balancing by analogy. We examine a different circuit-level phenomenon – di/dt voltage noise – a transient effect from the impedance of the processor package. In short, when a core's current consumption changes rapidly, this can cause a *voltage droop* on the processor's voltage supply. If the voltage droop is larger than the safety margin allocated at design time, a classical architecture has a high probability of a functional error occurring. On the other hand, a resilient architecture has a built-in recovery mechanism that can tolerate such droops, but incurs performance costs.

The specific of voltage noise as a phenomenon are not important for this discussion. We are interested in the analogy between tolerating voltage noise emergencies in a resilient architecture and compensating for current imbalances in voltage stack. In both cases, a novel architecture translates a circuit-level phenomenon related to workload execution in typical architectural performance and power metrics – for a FVS architecture, workload imbalances in a voltage stack cause incur a power cost, and for a voltage-resilient architecture, sudden changes in workload activity impair performance. The power cost in FVS comes from ineffectiveness of shunt regulators, and the performance cost for voltage noise is due to checkpoint-recovery-like mechanisms. In both cases, these mechanisms are put forward so that hardware can guarantee correctness and higher-level software techniques can only be used to improve on this baseline.

In order to apply a software approach to either phenomena, we developed the methodology described though example in the next sections. We first confirm that the event has a significant enough effect on the target metric that it is worth the engineering cost of exposing hardware details to software (Section 3.1). Since software has much higher latency than voltage or current events, we need to demonstrate that the event in question shows large-timescale behavior that a scheduler can exploit (Section 3.3). We also need to demonstrate that scheduling different threads indeed changes the target metric (Section 3.4) and, finally, asses the relationship for our novel scheduler with more traditional ones and aim for integration (Section 3.5).

3.1 Problem characterization

Performance Model In order to study the relationship between the system parameters in a resilient architecture, we inspect performance gains from allowing voltage emergencies at runtime. Since our analysis is based off measurements of a current generation processor[1] that does not support aggressive margins, we have to model the performance under a resilient system. For a given voltage margin, every emergency triggers a recovery, which has some penalty in processor clock cycles. During execution, we record the number of emergencies, which we determine from gathered scope histogram data. After execution, we compute the total number of cycles spent in recovery mode. These cycles are then added to the actual number of program runtime cycles. We gather runtime cycle counts with the aid of hardware performance counters using VTune [1]. The combined number is the performance lost due to emergencies.

While allowing emergencies penalizes performance to some extent, utilizing an aggressive voltage margin boosts processor clock frequency. Therefore, there can be a net gain. Bowman et al. show that an improvement in operating voltage margin by 10% of the nominal voltage translates to a 15% improvement in clock frequency [4]. We assume this 1.5× scaling factor for the performance model as we tighten the voltage margin from 14%. Alternatively, margins could be used to improve (or lower) dynamic power consumption.

[1]The details of how we characterize voltage noise behavior are irrelevant for this work, but can be found in our original paper [31].

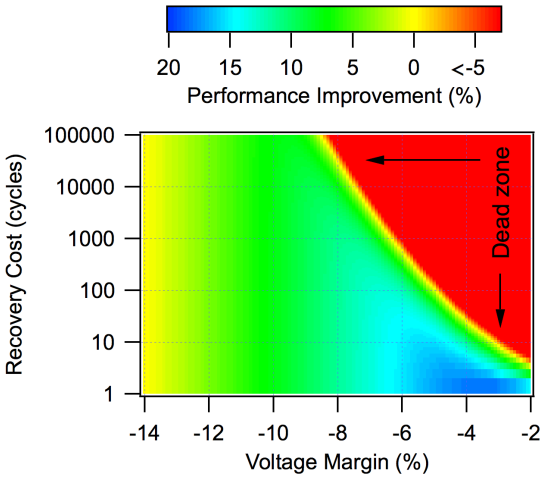
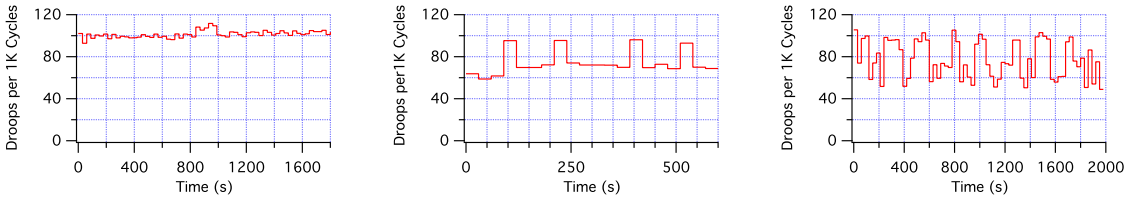


Figure 3.1: Performance characterization under various recovery costs and voltage margin settings.

The cost of recovery Figure 3.1 shows a characterization of our test Core^{TM}2 Duo system using the performance model described above. The workload consists of 881 benchmark executions including the 29 SPEC CPU2006 benchmarks, the Parsec benchmark suite [3] and 29×29 multiprogram CPU2006 workload combinations. For the purposes of this discussion, we need to point out the sensitivity of end performance to the cost of the recovery mechanism. For example, at a 4% margin, changing the cost of recovery from 5 to 20 cycles results in the difference between an overall performance gain of 20% across all benchmarks to a net performance loss.

3.2 Scheduling for voltage noise

In order to smooth voltage noise in multi-core processors and mitigate error recovery overheads, we investigate the potential for a voltage noise-aware thread scheduler. Our technique is hardware-guaranteed and software-assisted. Hardware provides a fail-safe guarantee to recover from errors, while software reduces the frequency of this fail-safe invocation, thereby improving performance by reducing rollback and recovery penalties. In this way, thread scheduling is a complementary solution, not a replacement/alternative for hardware. Due to the lack of existing checkpoint recovery/rollback hardware, we analytically model and investigate this software solution.



(a) *482.sphinx* (b) *416.gamess* (c) *465.tonto*

Figure 3.2: Single-core droop activity until complete execution. Some programs show no phases (e.g., *482.sphinx*). Others, like *416.gamess* and *465.tonto*, experience simple, as well as more complex phases, respectively.

The scheduling policy we motivate is called *Droop*. It focuses on co-scheduling threads across cores to minimize chip-wide droops. This technique exploits *voltage noise phases*, which we introduce and discuss first. We then demonstrate that thread scheduling for voltage noise is different than scheduling threads for performance. Finally, we demonstrate that a noise-aware thread scheduler enables designers to rely on coarse-grained recovery schemes to provide error tolerance, rather than investing in complex fine-grained schemes that are typically suitable for high-end systems, versus commodity processors.

3.3 Voltage noise phases

Similar to program execution phases, we find that the processor experiences varying levels of voltage swing activity during execution. Assuming a 2.3% voltage margin, purely for characterization purposes, Figure 3.2 shows droops below the margin per 1000 cycles across three different benchmarks, plotting averages for each 60-second interval. We use this margin since all activity that corresponds to an idling machine falls within this region. Thus, it allows us to cleanly eliminate background operating system activity and effectively focus only on the voltage noise characteristics of the program under test.

The amount of phase change varies from program to program. Benchmark *482.sphinx* experiences no phase effects. Its noise profile is stable around 100 droops per 1000 clock cycles. In contrast, benchmark *416.gamess* goes through four phase changes where voltage droop activity varies between 60 and 100 per 1000 clock cycles. Lastly, benchmark *465.tonto* goes through more complicated phase changes in Figure 3.2c, oscillating strongly and more frequently between 60 and 100 droops per 1000 cycles every several tens of seconds.

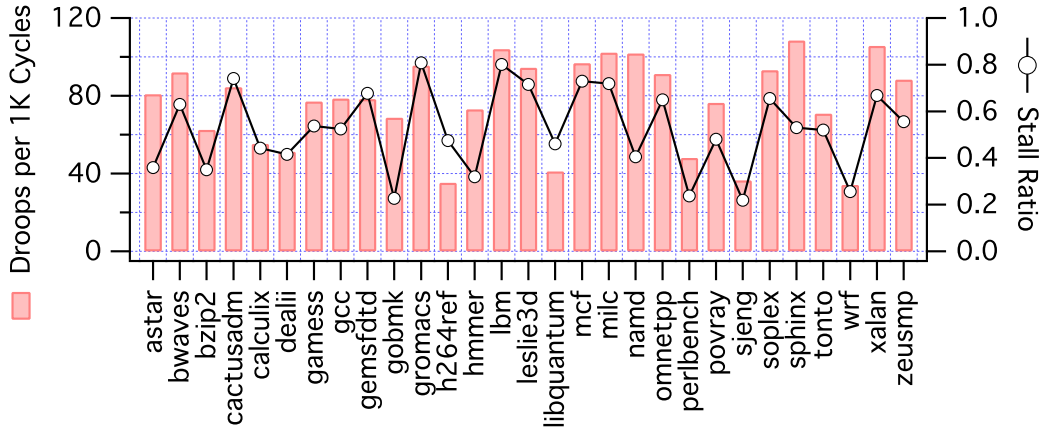
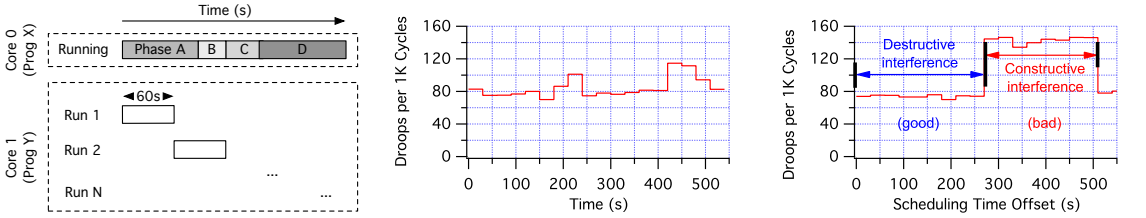


Figure 3.3: Single-core droop activity, showing a heterogeneous mix of noise levels along with correlation to stalls.

Voltage noise phases result from changing microarchitectural stall activity during program execution. To quantify why voltage noise varies over time, we use a metric called *stall ratio* to help us understand the relationship between processor resource utilization and voltage noise. Stall ratio is computed from counters that measure the numbers of cycles the pipeline is waiting (or stalled), such as when the reorder buffer or reservation station usage drops due to long latency operations, L2 cache misses, or even branch misprediction events. VTune provides an elegant stall ratio event for tracking such activity.

Figure 3.3 shows the relationship between voltage droops and microarchitectural stalls for a 60-second execution window across each CPU2006 benchmark. The window starts from the beginning of program execution. Droop counts vary noticeably across programs, indicating a heterogeneous mix of voltage noise characteristics in CPU2006. But even more interestingly, droops are strongly correlated to stall ratio. We visually observe a relationship between voltage droop activity and stalls when we overlay stall ratio over each benchmark's droops per 1000 cycles. Quantitatively, the linear correlation coefficient between droops and stall ratio is 0.97.

Such a high correlation between coarse-grained performance counter data (on the order of billions of instructions) and very fine-grained voltage noise measurements implies that high-latency software solutions are applicable to voltage noise.



(a) Setup for co-scheduling experiment. (b) *473.astar* single-core noise profile. (c) *473.astar* co-scheduled noise profile.

Figure 3.4: (a) Setup for studying co-scheduling of voltage noise phases. (b) Voltage noise profile of *473.astar* as it running by itself on a single core while the other core is idling. (c) Noise profile of co-scheduled instances of *473.astar* as per the setup in (a).

3.4 Co-scheduling of noise phases

A software-level thread scheduler mitigates voltage noise by combining different noise phases together. The scheduler's goal is to generate destructive interference. However, it must do this carefully, since co-scheduling could also create constructive interference. To demonstrate this effect, we setup the sliding window experiment depicted in Figure 3.4a. It resembles a convolution of two execution windows. One program, called Prog. X, is tied to Core 0. It runs uninterrupted until program completion. During its execution, we spawn a second program called Prog. Y onto Core 1. However, this program is not allowed to run to completion. Instead, we prematurely terminate its execution after 60 seconds. We immediately re-launch a new instance. This corresponds to Run 1, Run 2, ..., Run N in Figure 3.4a. We repeat this process until Prog. X completes execution. In this way, we capture the interaction between the first 60 seconds of program Prog. Y and all voltage noise phases within Prog. X. To periodically analyze effects, we take measurements after each Prog. Y instantiation completes. As our system only has two cores, Prog. X and Prog. Y together maximize the running thread count, keeping all cores busy.

We evaluate the above setup using benchmark *473.astar*. Figure 3.4b shows that when the benchmark runs by itself (i.e., the second core is idling), it has a relatively flat noise profile. However, constructive interference occurs when we slide one instance of *473.astar* over another instance of *473.astar* (see Constructive interference in Figure 3.4c). During this time frame, droop count nearly doubles from around 80 to 160 per 1000 cycles. But there is destructive interference as well. Between the start of execution and 250 seconds

into execution, the number of droops is the same as in the single-core version, even though both cores are now actively running.

We expanded this co-scheduling analysis to the entire SPEC CPU2006 benchmark suite, finding that the same destructive and constructive interference behavior exists over other schedules as well. Figure 3.5 is a boxplot that illustrates the range of droops as each program is co-scheduled with every other program. The circular markers represent voltage droops per 1000 cycles when only one instance of the benchmark is running (i.e., single-core noise activity). The triangular markers correspond to droop counts when two instances of the same benchmark are running together simultaneously, or more commonly known as SPECrate.

Destructive interference is present, with some boxplot data even falling below single-core noise activity. With the exception of benchmark *libquantum*, both destructive and constructive interference can be observed across the entire suite. If we relax the definition of destructive interference from single-core to multi-core, then room for co-scheduling improvement expands. SPECrate triangles become the baseline for comparison. In over half the co-schedules there is opportunity to perform better than the baseline.

Destructive interference in Figure 3.5 confirms that there is room to dampen peak-to-peak swings, sometimes even enough to surpass single-core noise activity. From a processor operational viewpoint, this means that designers can run the processor utilizing aggressive margins even in multi-core systems.

3.5 Scheduling for noise versus performance

Co-scheduling is an active area of research and development in multi-core systems to manage shared resources like the processor cache. Most of the prior work in this area focuses on optimizing resource access to the shared L2 or L3 cache structure [33, 11, 24, 19, 37, 8, 7], since it is performance-critical.

Similarly, processor supply voltage is a shared resource. In a multi-core system where multiple cores share a common power supply source, a voltage emergency due to any one core's activity penalizes performance across all cores. A global rollback/recovery is necessary. Therefore, the power supply is on the critical-path for performance improvement as well.

The intuition behind thread scheduling for voltage noise is that when activity on

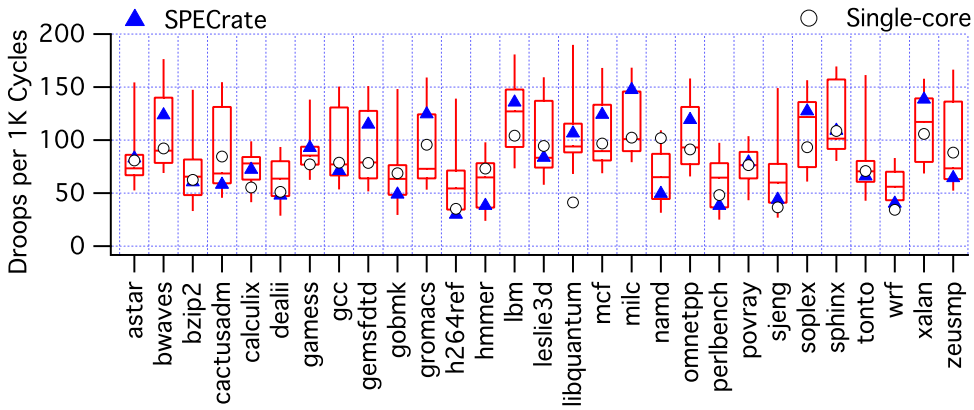


Figure 3.5: Droop variance across single- and multi-core.

one core stalls, voltage swings because of a sharp and large drop in current draw. By maintaining continuous current-drawing activity on an adjacent core also connected to the same power supply, thread scheduling dampens the magnitude of that current swing. In this way, co-scheduling prevents an emergency when either core stalls.

Scheduling for voltage noise is different than scheduling for performance. Scheduling for performance typically involves improving miss rates or reducing cache stalls. Since stalls and voltage noise are correlated, one might expect cache-aware performance scheduling to mitigate voltage noise as well. Inter-thread interference data [31] points out that the interactions between un-core (L2 only) and in-die events (all others) lead to varying magnitudes of voltage swings. Additional interactions must be taken into account.

Therefore, we propose a new scheduling policy called *Droop*. It focuses on mitigating voltage noise explicitly by reducing the number of times the hardware recovery mechanism triggers. By doing that it decreases the number of emergencies, and thus reduces the associated performance penalties.

Due to the lack of resilient hardware, we perform a limit study on the scheduling approaches, assuming oracle information about droop counts and simulating all recoveries. We compare a Droop-based scheduling policy with instructions per cycle (IPC) based scheduling. We use SPECrate as our baseline. It is a sensible baseline to use with IPC scheduling, since SPECrate is a measure of system throughput and IPC maximizes throughput. Moreover, SPECrate in Figure 3.5 shows no apparent preferential bias towards either minimizing or maximizing droops. Droop activity is spread uniformly over the entire work-

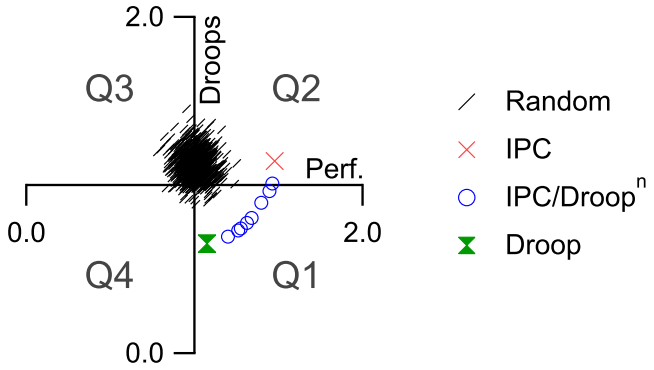


Figure 3.6: Impact of scheduling policies on droop versus performance relative to SPECrate.

load suite, further making it a suitable baseline for comparison. We normalize and analyze results relative to SPECrate for both droop counts and IPC, since this removes any inherent IPC differences between benchmarks and focuses only on the benefits of co-scheduling.

To evaluate the different policies, we setup a batch scheduling experiment where the job pool consists of pairs of CPU2006 programs, enough to saturate our dual core system. From this pool, during each scheduling interval, the scheduler chooses a combination of programs to run together, based on the active policy. In order to avoid preferential behavior, we constrain the number of times a program is repeatedly chosen. 50 such combinations constitute one batch schedule. In addition to deterministic Droop- and IPC-based scheduling policies, we also evaluate 100 random schedules.

The scheduling experiment is oracle-based, requiring knowledge of all runs a priori. During a pre-run phase we gather all the data necessary across 29×29 CPU2006 program combinations. For Droop, we continue using the hypothetical 2.3% voltage margin, tracking the number of emergency recoveries that occur during execution. For IPC, we use VTune's ratio feature to gather data for each program combination.

Figure 3.6 shows the results of our test. Each marker corresponds to one batch schedule. The four quadrants in Figure 3.6 are helpful for drawing conclusions. Ideally, we want results in Q1, since that implies fewer droops and better performance. Q2 is good for performance only. Q3 is bad for both performance and droops. Q4 is good for droops and bad for performance.

Random scheduling unsurprisingly does worst. Random runs cluster close to the center of the graph. Effectively, it is mimicking SPECrate, which is also uniformly dis-

Recovery Cost (cycles)	Optimal Margin (%)	Expected Improvement (%)	# of Schedules That Pass
1	5.3	15.7	28
10	5.6	15.1	28
100	6.4	13.7	15
1000	7.4	12.2	12
10000	8.2	10.8	9
100000	8.6	9.7	9

Table 3.1: SPECrate typical-case design analysis at optimal margins.

tributed with respect to noise. Compared to the baseline SPECrate, IPC scheduling gives better performance. However, since it is completely unaware of droop activity, the IPC marker is at approximately the same average droop value as most random schedules. The Droop policy is aware of voltage noise, therefore it is able to minimize droops. In this case, we also see a slight improvement in performance.

3.6 Reducing recovery overheads

As voltage noise grows more severe, it will become harder for resilient systems to meet their typical-case design targets. Table 3.1 illustrates this point. For each recovery cost, the table contains an optimal margin and an associated performance improvement at this aggressive margin. This is the margin at which the system experiences maximum performance improvement over worst-case design (i.e., 14% on the Core^{TM}2 Duo processor). This margin is determined from analyzing the performance improvements across all 881 workloads in our test setup. But not all workloads can meet this target because when voltage swings get larger, the frequency of emergencies increases, leading to more error recoveries, which in turn penalizes performance. When we constrain our analysis to the multi-core (or multi-program) workload subset, this trend becomes apparent. We examine our SPECrate baseline across the different recovery costs. At 1-cycle recovery cost, 28 out of all 29 SPECrate schedules meet the expected performance criteria. In other words, they *pass* the expectations. But as recovery goes up beyond 10 cycles, the number of schedules that pass quickly diminishes.

In a multi-core system, a thread scheduler can improve the number of schedules that pass. To demonstrate this, we run oracle-based Droop and IPC scheduling through

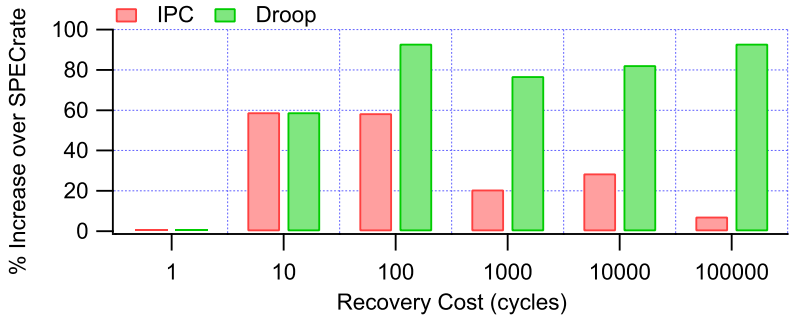


Figure 3.7: Increase over the number of schedules in Table 3.1 that pass.

the performance model described earlier. We then determine the number of schedules that pass across the benchmark suite. Figure 3.7 summarizes these results. At 10-cycle recovery cost, both Droop and IPC scheduling perform well by increasing the number of passing schedules by 60% with respect to SPECrate. IPC scheduling leads to some improvement since reducing the number of cache stalls mitigates some emergency penalties. However, targeting cache events alone is insufficient to eliminate or significantly reduce interference across cores. Therefore, with more coarse-grained schemes, IPC improvements diminish and we see a decreasing trend line. By comparison, Droop scheduling consistently outperforms IPC. At 1000 cycles and beyond, we see an emerging trend line that emphasizes Droop scheduling performs better at larger-recovery costs. This indicates that more intelligent voltage noise-aware thread scheduling is necessary to mitigate recovery overheads, especially at coarser-recovery schemes.

However, it may be beneficial for the noise-aware scheduler to incorporate IPC awareness as well, since co-scheduling for performance has its benefits. Therefore, we propose $\mathsf{IPC/Droop}^n$ to balance performance- and noise-awareness. This metric is sensitive to recovery costs. The value of n is small for fine-grained schemes, since recovery penalty will be a relatively small fraction of other co-scheduling performance bottlenecks. In such cases, weighing IPC more heavily is likely to lead to better performance. In contrast, n should be bigger to compensate for larger recovery penalties under more coarse-grained schemes. In this way, the scheduler can attempt to maximize performance even in the presence of emergencies. The pareto frontier in the lower quadrant of Q1 in Figure 3.6 illustrates this range of improvement. A case where this metric is useful is when designers implement different grades of recovery schemes based on the class of a processor. Server-class or high-performance systems will typically use finer-grained recovery schemes despite

implementation overheads. Therefore, they will have smaller recovery penalty. Cheaper, more cost-effective commodity systems, like workstations and desktop processors, are likely to rely on more coarse-grained solutions. The metric allows the scheduler to dynamically adapt itself to platform-specific recovery costs.

Chapter 4

Modeling simple cores

In Chapter 2, we showed that one of the main characteristics of a voltage-stacked system is the feedback loop between core current consumption disbalances and the supply voltage inside a voltage stack. If we are to evaluate the potential benefits of such a system, we need a detailed enough model that can produce reliable current traces from cores in the system.

The simulator we developed, XIOSim, was created with such a level of detail in mind. It targets the x86 instruction set due to its widespread adoption. Since flexible voltage stacking is an appropriate platform for many-core systems with small per-core current consumption variation, we model small, in-order x86 cores.

Genealogically, XIOSim was derived from the Zesto x86 user-level simulator [22], which in turn originates from SimpleScalar [5]. XIOSim extends the Zesto framework by providing a different, binary instrumentation-based functional execution model, a detailed in-order core model, and by integrating very closely with the McPAT power models [21] for producing power consumption traces.

In order to prove that the current traces produced by XIOSim are indeed reliable, we developed an extensive simulator validation methodology that is based on low-overhead binary instrumentation. It allows us to compare performance and power measurements between the same code sequence running on a simulator and on a real machine even under more than five orders of magnitude execution speed difference. We used this methodology against the XIOSim implementation and showed that the performance models are within 10% of real hardware and the power models are within 5% in terms of average power. To the best of our knowledge, XIOSim is the most detailed publicly available in-order x86

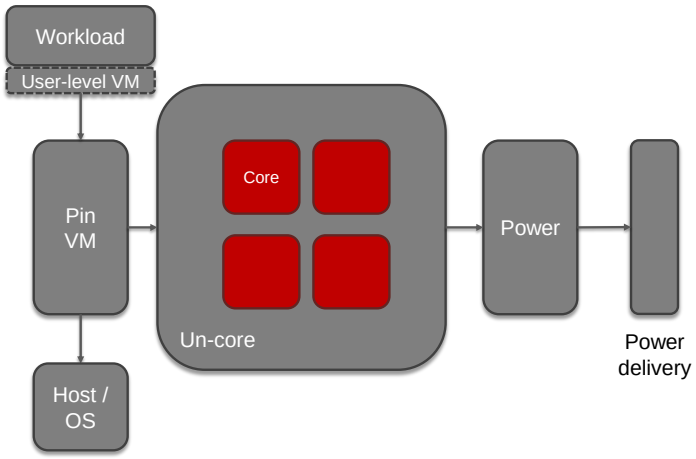


Figure 4.1: XIOSim execution model.

simulator. It can be downloaded at http://xiosim.org.

4.1 Simulator execution model

The simulator execution model we propose is a mixture between traditional user-level simulation and full-system simulation. XIOSim is a user simulator in that it does not model instructions past system call boundaries. However, it runs as a pintool, under the Pin virtual machine [23], which translates application instructions to run natively on the simulation host.

Figure 4.1 sketches out this execution model showing the interactions between functional, performance and power simulation. The user-level workload is run under Pin control, with the Pin VM instrumenting every instruction and capturing the host machine state immediately before it. That instruction context is passed to the performance model, which is expecting it at the fetch stage of the modelled pipeline, effectively achieving execute-at-fetch semantics. At certain cycle intervals, simulated statistics are aggregated and passed to a modified version of McPAT [21] to create a current consumption trace. When evaluating a voltage-stacked system, current traces are fed to a power delivery model that includes the typical feedback loop that adjusts supply voltage.

The closest analog to this execution model is implemented in PTLSim [36] and MARSS [27]. In these simulators, the performance model is run as an extension to a system-level virtual machine monitor or a full-system emulator. In our approach, keeping

the simulator completely in user-level, while limiting the scope of benchmarks whose performance will be accurately modeled, is beneficial from a simplicity and a stability point of view. Full-system simulators need to run a complete OS image, virtual device drivers, etc., all of which affect simulation accuracy and reliability. For microarchitectural studies, it is often beneficial to ignore such behavior and only focus on the characteristics of the benchmark under test in isolation.

Furthermore, this model has benefits over traditional user-level simulation. First, all system code, including system call handling, is directly executed on the host, freeing up the simulator from implementing a pass-through layer and keeping up with relatively fast-changing system call interfaces. This allows simulating a broader range of applications than traditional user-level simulation, where system call support is typically added based on the popularity of the particular call. For example, this has allowed us to simulate workloads under the ILDJIT user-level virtual machine [6] that translates CIL bytecode in IA-32 instructions.

Second, by running under an instrumentation engine, the performance model does not require a complete or correct functional model in order to maintain the correct execution of the workload. Since instructions are executed on native hardware, the performance model can only observe their effects without explicit knowledge of instruction semantics. This allows continuing execution even after encountering esoteric unsupported instructions, which is especially important for simulating a complex and constantly evolving instruction set like IA-32.

Finally, this execution model allows for close-to-native speeds of execution of code outside simulation regions of interest. This enables efficient skipping of non-interesting benchmark phases and integration with statistical sampling tools, such as the SimPoint-based PinPoints [29].

The major disadvantage with such an approach comes from speculative execution. Pin cannot instrument instructions on a speculative path since by definition they are not visible above the architecture level of abstraction. In order to deal with that, XIOSim keeps a legacy functional model that is only used on speculative paths. On simulating a mispredicted branch, this functional model is invoked, and the simulated core continues utilizing it until the branch is resolved in the execute portion of the pipeline. At this point, the functional model is switched back to the instrumentation engine. Note that, to ensure correctness, the legacy functional model does not need to be completely correct or

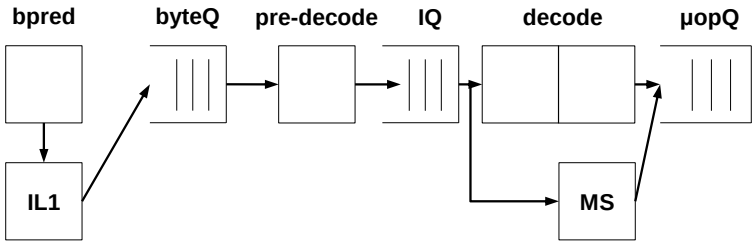


Figure 4.2: Front-end pipeline model for in-order core.

accurate, since the mispredicted branch will get resolved regardless of the instructions on the speculative path.

4.2 Performance model

In this section, we describe the in-order core performance model used in XIOSim. Overall, we model an in-order, multi-issue x86 core, a proxy for Atom-style microarchitectures [12]. Its pipeline is typically deeper than the ones found in mobile cores, partially due to the complexities involved in decoding a variable-length ISA, as well as due to the desire for efficiently handling CISC instructions with memory operands without out-of-order execution.

The front-end model, as seen in Figure 4.2, is typical for an IA-32 architecture. The pipeline starts with branch prediction, which directs accesses to the instruction cache. It continues with explicit pre-decode stages, whose aim is to predict instruction boundaries in the raw byte stream that was fetched. Pre-decoded instructions then propagate to the asymmetrical decoders, which break the possibly complex instructions into RISC-like μ-ops. The decoders can process multiple instructions in parallel, in order to keep the rest of the multiple-issue pipeline occupied. Complex instructions are sent to the micro-sequencer for decoding, which incurs an additional penalty of several cycles.

The decoders make heavy use of μ-op fusion – combining multiple μ-ops from a single instruction, such that they progress together through the pipeline, not incurring additional latency and occupying less space in queue structures. Out-of-order x86 microarchitectures have traditionally used load-op fusion, which combines a load with its dependant subsequent operation. In a deep in-order pipeline not incurring latency costs is even more important and fusion can be applied more aggressively – combining loads, their dependant

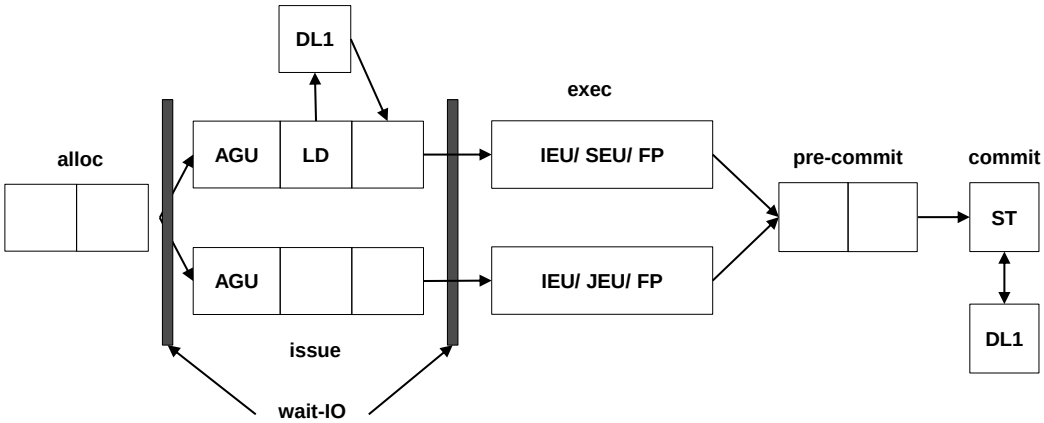


Figure 4.3: Back-end pipeline model for in-order core.

operations and a store from a complex, memory-reference instruction. This results in being able to execute a large fraction of instructions in a CISC-like fashion.

Decoded and possibly fused μ-ops then enter the allocation stages of the pipeline (see Figure 4.3), where they access the register file, and get assigned to an execution port based on functional unit availability and port loading. Because decoding latencies can vary and different execution ports can stall independently of each other, there is an explicit in-order synchronization point at the end of the allocation pipeline. Its purpose is to enforce program order of μ-ops entering the execution ports, so that data dependencies can be observed without any of the complexities of an out-of-order pipeline.

Once in an execution port, μ-ops go through issue stages, mostly dedicated to memory accesses. These include address generation, load dispatch and return from the data cache. Having dedicated pipeline stages for processing loads is in unison with μ-op fusion, effectively achieving a zero-cycle load-to-use penalty for load-op or load-op-store fusions.

There is another in-order synchronization point at the end of the issue stages. Since all execution latencies after it are deterministic, this point ensures that μ-ops are properly serialized for in-order commit. Once executed and serialized, μ-ops enter a pre-commit pipeline stage which models exception handling. Then they follow to the commit buffer, where stores are sent to the data cache and individual μ-ops are composed back to macro instructions in order to commit atomically, observing instruction boundaries.

4.3 Simulator validation

4.3.1 Validation Methodology

We will show that the performance model described earlier can be configured to match a real design and is therefore useful as a performance exploration tool, as well as a power model driver. We start by presenting the extensive validation methodology that we used for developing XIOSim.

The major difficulty in validating a detailed model against real hardware arises from the discrepancy in speed. While real hardware can often commit instructions at rates of 10s billions instructions per second (BIPS), a detailed performance model typically achieves 10-100s kilo-instructions per second (KIPS). Simulating full real-world benchmarks is therefore prohibitively slow and a sampling approach has to be used. This presents two possible reference points for the sampled benchmark run: (i) either data gathered from the full benchmark execution, or (ii) collected data from a similarly sampled run on a real machine.

The first approach is conveniently simple, but has several issues. First, when doing sampling, the short execution slices are selected with a particular metric in mind. For example, SimPoints [32] targets end performance and the selected slices can be used to estimate the full program's execution time. However, this does not guarantee that SimPoint-derived slices are representative of the full program's branch behavior or power consumption, since these metrics are not always strongly correlated with performance.

Furthermore, even when used with the appropriate metric, the predictions from sampling are within some confidence interval of the full-length values. Patil et al. [29] demonstrate that this error can be as high as 10% for predicting CPI with SimPoints. This makes it impossible to separate simulator errors from sampling errors in a validation scenario.

For these reasons, we choose the second validation approach. Since sampling is inevitable when dealing with the slowdown of detailed simulation, we do sample with end performance in mind for our simulated runs. For the reference runs on a real machine, we use a very similar sampling approach, making sure we gather data only during the same instruction intervals that are simulated. The difficulty in this case lies in gathering data for a short period (measured in millions of instructions) without either modifying the workload source code, or perturbing the experiment with the measurement code.

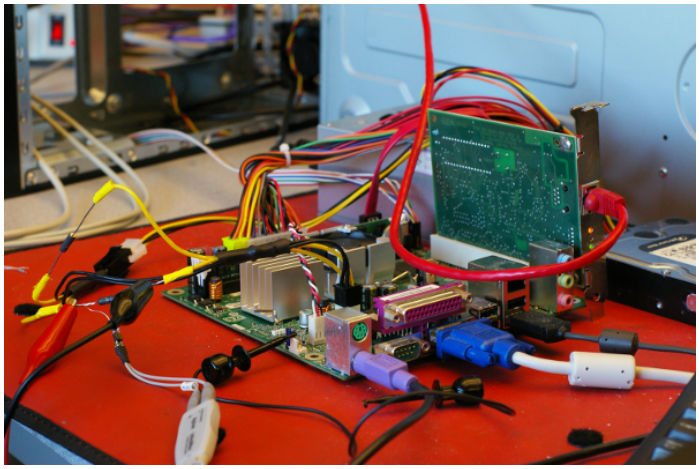


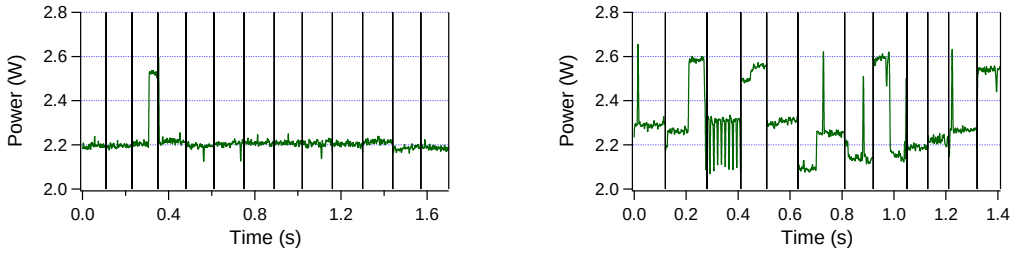
Figure 4.4: Power measurement setup for an Atom™-based system.

Without source modification, one can use binary instrumentation to trigger the appropriate measurements once execution reaches the simulated region. However, the instrumentation must be light enough in order not to introduce errors in measurements on the host machine. For example, in the Pin virtual machine even inserting a simple basic block counting instrumentation routine can have overheads in execution time on average exceeding 100% [23].

In order not to pay this penalty, we use Pin to insert lightweight trampolines that effectively replace routines of interest with versions that have thin instrumentation code inserted. The instrumentation code tracks function call counts and determines when it is appropriate to start or stop data collection. The routines of interest are defined as the closest ones to the beginning and the end of simulation slices, as identified by SimPoints, and the target call counts for them are gathered during an initial profiling run.

The stub code that we insert is a hand-optimized assembly sequence tracking the number of times the routine has been called and toggling data capture when appropriate. The measured average instruction overhead of this scheme over the SPEC CPU2006 suite is 0.7%. For performance validation, the stub code triggers performance counter collection on the host machine through the Linux perfmon2 interface. Its behavior in power validation mode is described in the following section.

Power measurement setup Our particular power capture system taps in the processor 12V power supply line and measures current consumption through a low-impedance high-



(a) *453.povray* (b) *447.dealII*

Figure 4.5: Power traces with qualitatively different behavior, but similar average power.

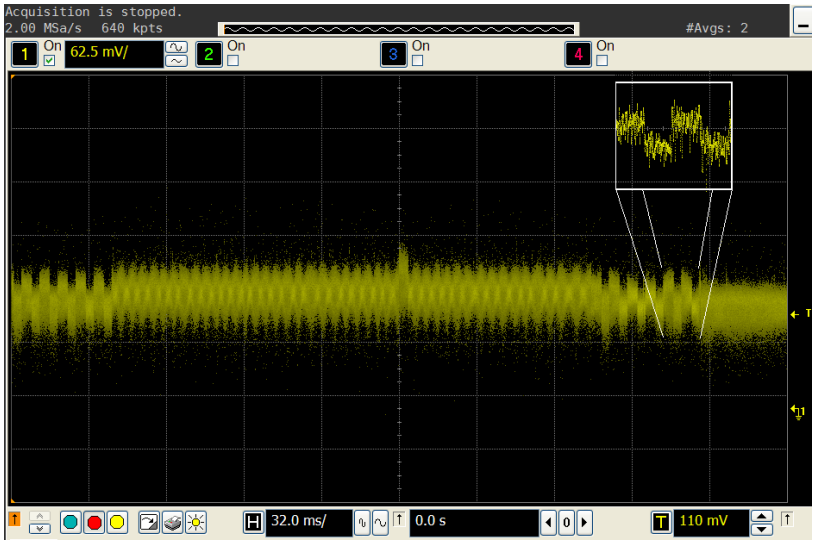


Figure 4.6: Power measurements for a sample execution slice. Inset shows a close-up of the power markers used to signal the start and end of a slice.

precision sense resistor (Figure 4.4). At a fixed nominal supply voltage this provides enough data to capture processor power consumption. The traces are collected using a high-end Aglient DSA-91304A oscilloscope and a DSA-3100 differential probe. Trace collection is triggered remotely over the network. The high-end collection system allows us to gather power traces for a short amount of time with enough resolution to capture intra-slice power behavior. The sampling rate used in the following experiments is set to 160 kHz.

The benefit of being able to capture single execution slices can be seen in Figure 4.5, which shows power traces for samples of the SPEC benchmarks *453.povray* and *447.dealII*. If we average power over all execution slices, the difference between the two benchmarks

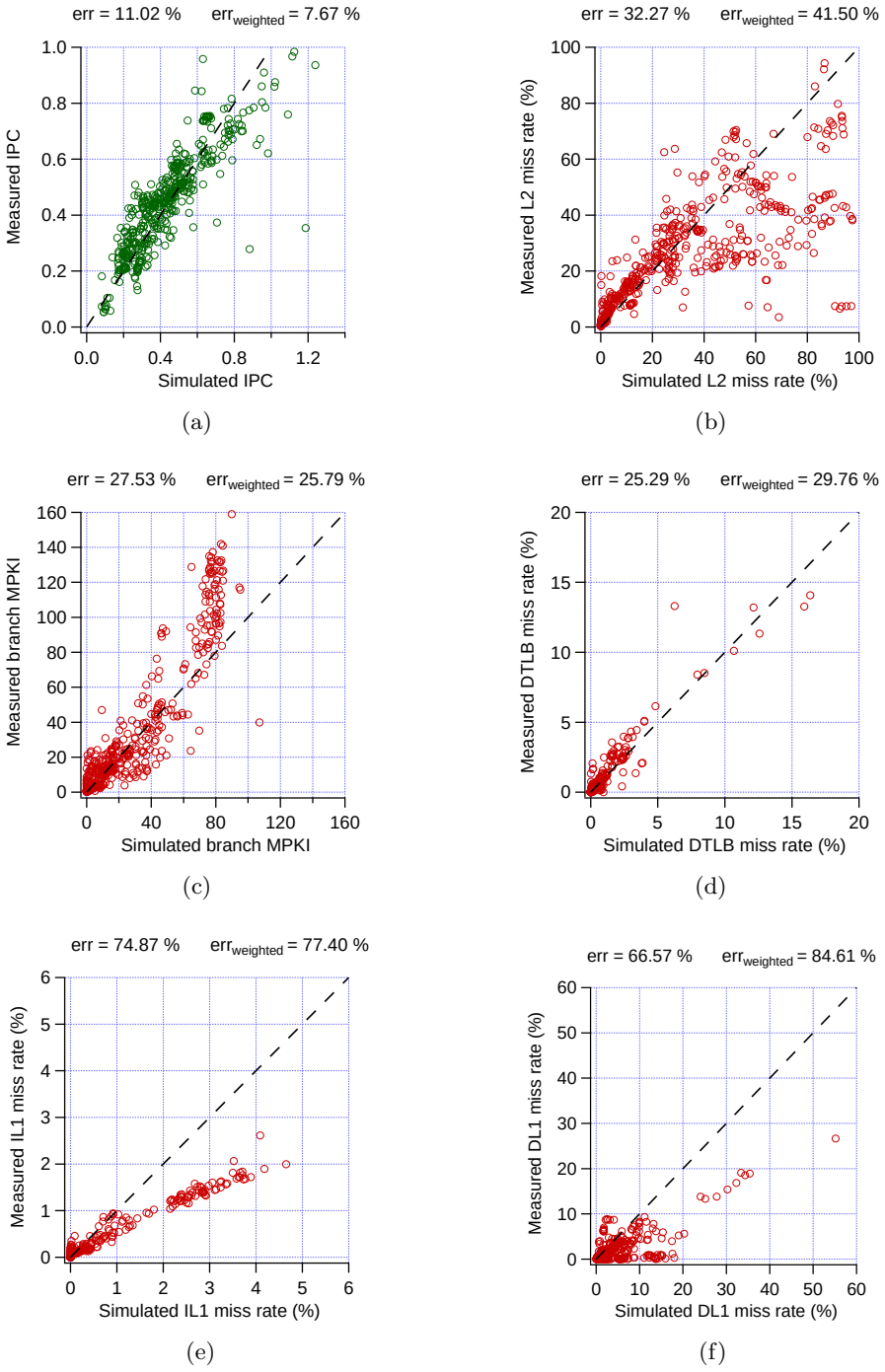


Figure 4.7: Comparison of microarchitectural statistics between performance counter measurements and simulation for execution slices of the SPEC CPU2006 suite.

is only 4%. However, in order to perform any meaningful power validation, we need to be able to capture and compare individual slice behavior for benchmarks like *447.dealII*.

Aligning code and measurements Since executing a single sampled slice on a real machine can take time on the order of 10ms, which is comparable to the network delay to trigger power capture, explicit measures have to be taken to synchronize the power trace collection with the slice execution. We use the low-overhead instrumentation described in Section 4.3.1 to insert power markers at the beginning and end of a sampled slice. The markers are repeating sequences of a hand-crafted power virus instance (derived from CPUBurn and modified for an in-order pipeline) and suspending the core by executing a `usleep()` system call. By sandwiching the slice execution with long enough markers, we can mitigate the various variable delays and latencies in the measurement system and be also sure that the correct execution slice has been captured. A sample trace captured with this methodology is shown in Figure 4.6, with two iterations of a power marker in the inset.

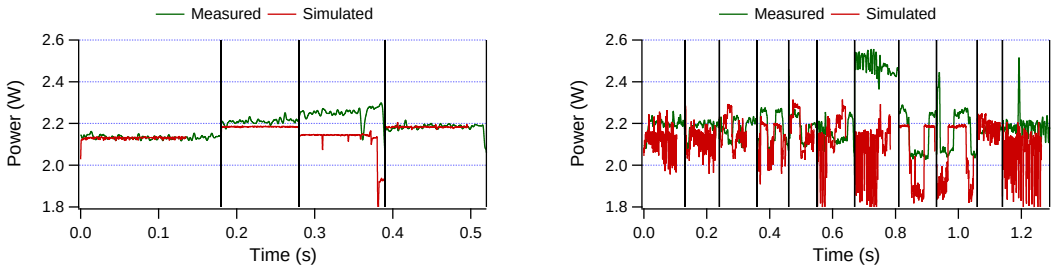
In a post-processing step, the power markers are programmatically removed from the collected trace, so that only the slice region-of-interest is accounted for. The marker removal algorithm operates by searching for the specific frequency footprint of the power markers.

4.3.2 Validation Results

We applied the validation methodology described in the previous section to the XIOSim implementation. Measurements were performed on a real system with a dual-core Intel Atom 330 processor, with only one core active. Hyperthreading was also disabled during all experiments.

Since XIOSim focuses on detailed modeling of the core microarchitecture, the benchmark suite we use is the CPU-centric SPEC CPU2006 [2]. The sampled slices are collected using PinPoints [29], with a slice length of 100M instructions. Over the whole benchmark suite, this results in ≈500 slices, totalling ≈50B simulated instructions.

Performance The in-order performance model is reasonably accurate. This is demonstrated in Figure 4.7. It shows measured versus simulated data for overall IPC, as well as for the major microarchitectural events. The dashed unit lines in each subplot represent



(a) *400.perlbench* (b) *401.bzip2*

Figure 4.8: Measured and simulated power traces for some CPU2006 benchmarks.

the ideal case, and each circle marker is measured from a single 100M instruction execution slice. In an ideal, cycle-accurate model, all markers should be clustered along the unit lines.

Figure 4.7a shows that the performance model tracks end-to-end performance with satisfactory accuracy – the geometric mean of the IPC error over all slices is 11.02%. Furthermore, if we take into account the relative importance of each slice as computed by SimPoints, the mean IPC error decreases to 7.67%. Looking at the distribution of markers in Figure 4.7a, the simulator is not systematically underestimating or overestimating end performance.

The relative accuracy in predicting individual microarchitectural events is lower, as seen by Figures 4.7b-4.7f. For example, the average error in instruction cache miss rates (Figure 4.7e) is as high as 74%. The explanation of this discrepancy comes from the absolute values of these events – they are often small numbers. In that case, the error between a simulated miss rate of 0.1% and a measured rate of 0.2% is 50%, but the end impact on performance is insignificant because both rates are sufficiently small. This is especially true for miss rates in the instruction cache and the TLB, which are, on average 0.3% and 0.7%.

While most of the event errors also appear unbiased toward under-counting or over-counting, there are two distinct clusters of slices in Figure 4.7c and Figure 4.7e, where the simulator underestimates branch mispredictions and overestimates instruction cache misses. Most of those slices can be traced to the benchmark *445.gobmk*, which has been shown to have high branch misprediction rates [16]. On a real machine, the higher number of branch misses can have a prefetcher-like effect explaining the lower miss rates in the instruction cache.

Power It is reasonable to expect that the power model is less accurate than the performance model. We first show that it can qualitatively track benchmark behavior on a real machine. We then go on to validate the quantitative predictions for power consumption, starting with leakage estimates, and continuing with the sum of static and dynamic power.

Workload tracking The methodology described in Section 4.3.1 allows us to demonstrate that the power model can adequately track workload-related activity. Figure 4.8 shows that by comparing measured versus simulated power consumption for the benchmarks *400.perlbench* and *401.bzip2*. Vertical lines denote execution slice boundaries. Even though the two benchmarks show qualitatively very different power profiles, XIOSim's power model tracks the shape of both relatively well. As with any model, the synchronization is not perfect – for example, notice how the first simulated slice of *400.perlbench* finishes significantly before its corresponding measured slice. In this particular instance, the misalignment comes from a large IPC error in the performance model, which predicts execution time wrongly.

Leakage In order to quantify the model accuracy, we start by estimating the leakage of our real machine with a simple experiment. While our test processor is executing an instance of the power virus, we scale the frequency from the maximum supported 1600 MHz to the minimum 200 MHz at 200 MHz increments, measuring average power consumption at each step. Since we are only scaling frequency and not voltage, power consumption scales linearly, and we can estimate the static power as the portion independent of frequency in a linear fit.

Performing this experiment, we estimated that the Atom 330 consumes 1.79 W static power. Our power model leakage estimate is 1.4 W. However, it does not include any overheads associated with simultaneous multi-threading (SMT). Even though we keep SMT disabled through our experiments, it still incurs a leakage cost because it requires additional queues and buffers, or additional access ports to existing structures. Intel data [12] suggests that the power cost of adding SMT to an Atom core is 15%. Adding this overhead to our leakage estimates results in 1.61 W in static power, which is within 10.1% of the measured value. It is logical that the model underestimates static power – a real design includes components that are not explicitly modelled, such as PLLs, bus drivers and memory buses, which could consume a non-ignorable amount of energy.

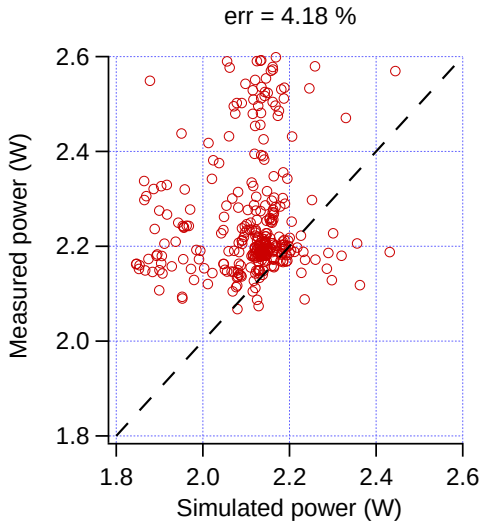


Figure 4.9: Simulated and measured power for execution slices of CPU2006.

Total power After incorporating the leakage correction for SMT, we can look at total processor power across benchmarks. We collapse the dynamic behavior of each execution slice to a single average power value and compare measured and predicted per-slice average power in a manner similar to Section 4.3.2. Figure 4.9 shows the results of this validation procedure. The geometric mean power error is only 4.18% (after accounting for SMT). However, as seen in Figure 4.9, a large fraction of execution slices cluster around an average power consumption of 2.2 W. We attribute this low variability to the fact that the core under test is in-order and relatively simple.

From Figure 4.9, we can clearly notice that the simulation systematically underestimates power consumption. As discussed in the previous paragraph, this is expected behavior, since we do not model all components on a real system. There is a distinctive set of slices with measured power between 2.4 and 2.6 W which correlate very poorly with simulation results. One such example is the 7-th slice in Figure 4.8b. We have not been able to identify the reason for the discrepancy, but possible candidates are the operating system timer triggering on the real system, or the voltage regulator triggering a feedback loop.

Chapter 5

Conclusion and future work

In this work we have proposed *flexible voltage stacking* as one way of achieving heterogeneity in a many-core processor made of homogeneous cores. This approach to organizing a system allows matching each core's optimal energy operation point to the workload. As a side benefit, it significantly reduces power lost in the processor package, resulting in up to 10% power savings. We have shown that the major hurdle to practically implementing such a system is workload heterogeneity across a single stack. We propose a hardware-software solution called *current balancing* that intelligently schedules threads in order to homogenize current consumption across a single stack. We base this solution on voltage smoothing and propose a detailed methodology for evaluating its effectiveness. As a first step in this methodology, we have developed and thoroughly validated a detailed power/performance simulator for small x86 cores.

This is only the first step of a thorough evaluation of flexible voltage stacking. We are planning additional effort in better characterizing hardware balancing mechanisms and analytical models of a voltage stack. We can then fully characterize the behavior of real workloads on a FVS system and assess the benefits and drawbacks of such organization end-to-end. Only then we can fully and comprehensively assess the potential of current balancing.

Appendix A

Detailed performance model validation

This appendix contains detailed validation data for the performance model presented in Chapter 4. In the main text, this dataset is compressed into Figure 4.7. We present detailed data here since a thorough reader may find per-benchmark characterization data for an Atom™-based platform interesting. Figures A.1-A.6 show measured and simulated data on the execution slice granularity for instructions per cycle (IPC), L2 cache miss rates, branch predictor misses, L1 data cache miss rates, instruction cache miss rates, and data TLB miss rates.

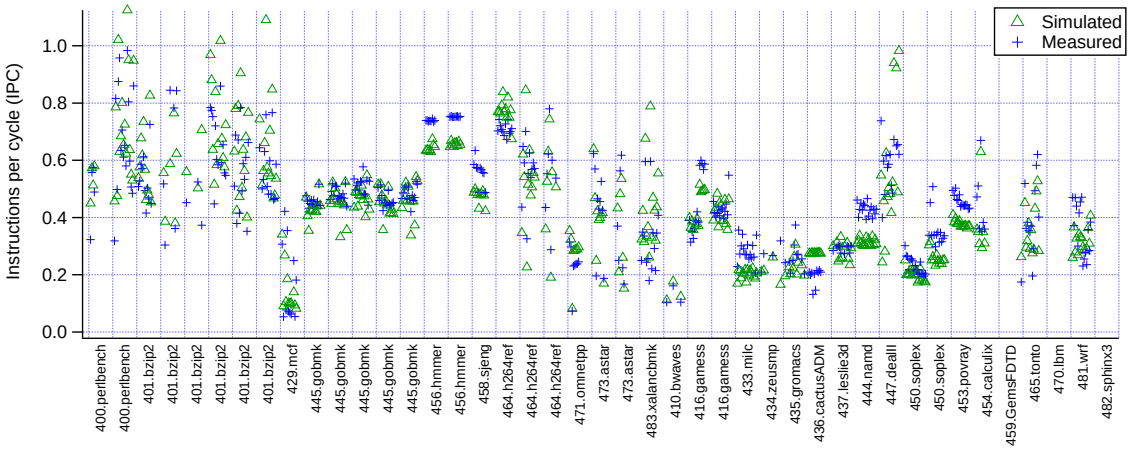


Figure A.1: Measured and simulated instructions per cycle for each execution slice in CPU2006.

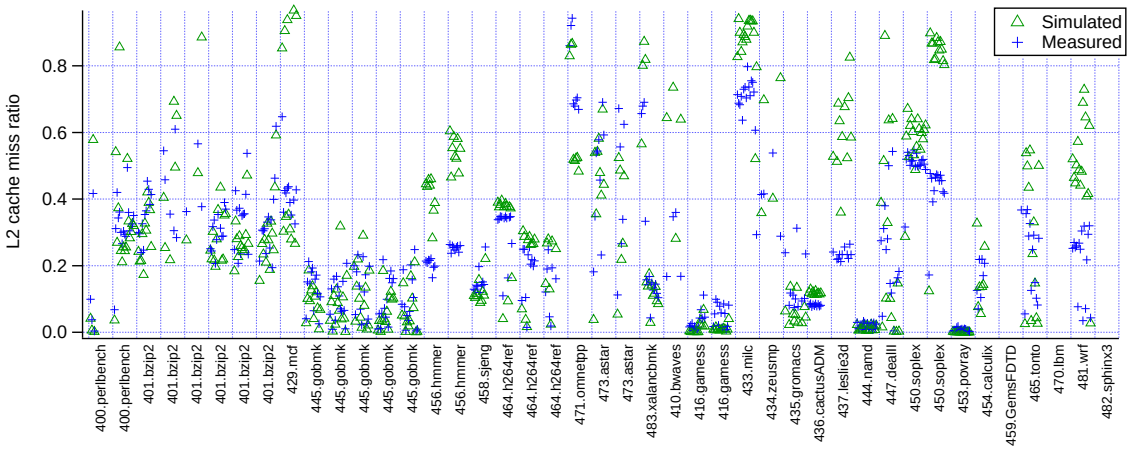


Figure A.2: Measured and simulated L2 cache miss rates for each execution slice in CPU2006.

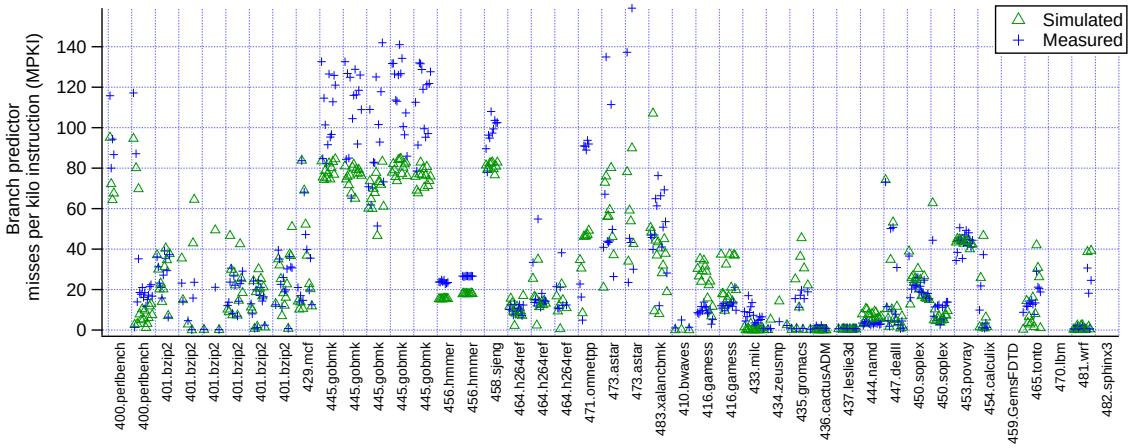


Figure A.3: Measured and simulated branch predictor misses per kilo-instruction (MPKI) for each execution slice in CPU2006.

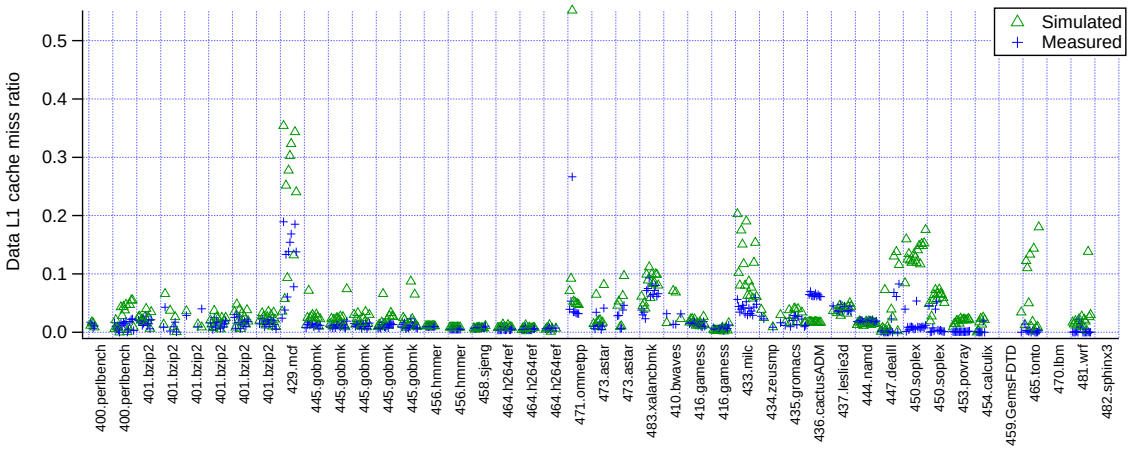


Figure A.4: Measured and simulated data cache miss rates for each execution slice in CPU2006.

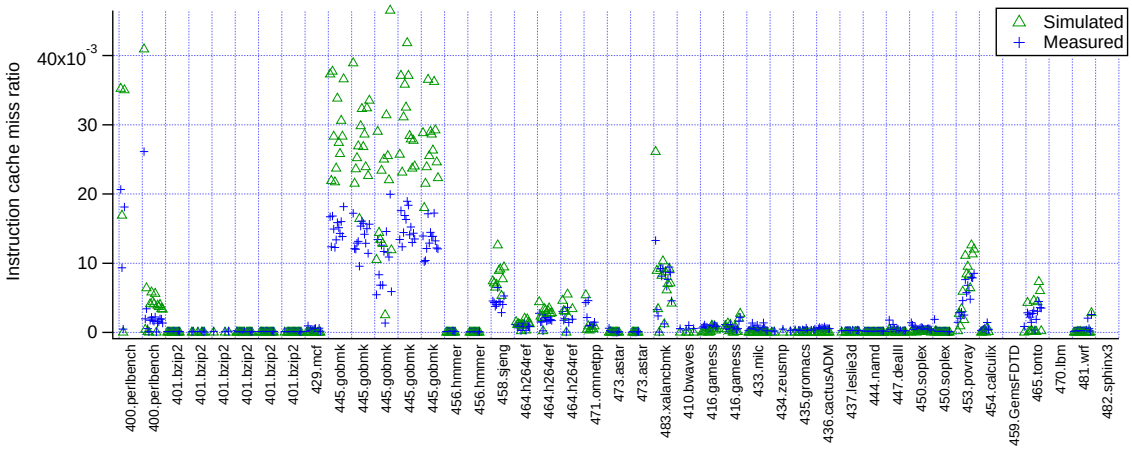


Figure A.5: Measured and simulated instruction cache miss rates for each execution slice in CPU2006.

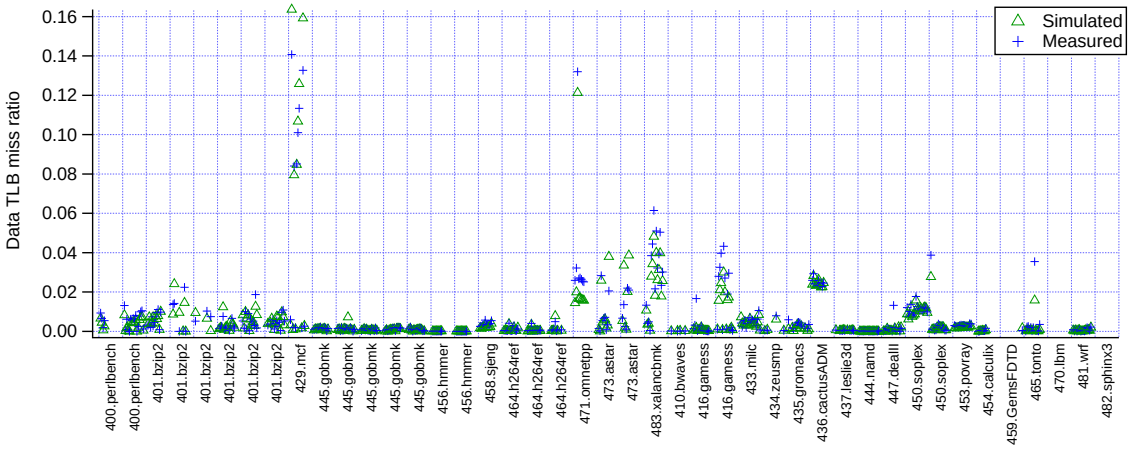


Figure A.6: Measured and simulated DTLB miss rates for each execution slice in CPU2006.

Bibliography

[1] Intel VTune, http://software.intel.com/en-us/intel-vtune/.

[2] SPEC CPU2006, http://www.spec.org/cpu2006/.

[3] Christian Bienia, Sanjeev Kumar, J.P. Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*, 2008.

[4] K. A. Bowman et al. Energy-efficient and metastability-immune timing-error detection and instruction replay-based recovery circuits for dynamic variation tolerance. In *ISSCC*, 2008.

[5] D. Burger and T.M. Austin. The SimpleScalar tool set, v. 2.0. *ACM SIGARCH Computer Architecture News*, 1997.

[6] Simone Campanoni et al. A highly flexible, parallel virtual machine: design and experience of ILDJIT. *Softw., Pract. Exper.*

[7] Francisco J. Cazorla et al. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Trans. Comput.*, 2006.

[8] Dhruba Chandra et al. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *HPCA*, 2005.

[9] Intel Corp. Voltage Regulator-Down 11.1: Processor Power Delivery Design Guide. 2009.

[10] R.H. Dennard, F.H. Gaensslen, VL Rideout, E. Bassous, and AR LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[11] Alexandra Fedorova. *Operating system scheduling for chip multithreaded processors.* PhD thesis, 2006. Adviser-Seltzer, Margo I.

[12] Gianfranco Gerosa et al. A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi- κ Metal Gate CMOS. In *2008 IEEE International Solid-State Circuits Conference.*

[13] J. Gu and C.H. Kim. Multi-story power delivery for supply noise reduction and low voltage operation. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[14] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *Micro, IEEE*, 2011.

[15] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores. *Proceedings of the 37th annual international symposium on Computer architecture (ISCA)*, 2010.

[16] Arun Kejariwal et al. Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel Core2 Duo processor. *2008 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation.*

[17] W. Kim, D.M. Brooks, and G.Y. Wei. A fully-integrated 3-level dc/dc converter for nanosecond-scale dvs with fast shunt regulation. In *2011 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011.

[18] W Kim, M.S. Gupta, G.Y. Wei, and D Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[19] Rob Knauerhase et al. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 2008.

[20] Sae-Kuy Lee, David Brooks, and Gu-Yeon Wei. Evaluation of voltage stacking for near-threshold multicore computing. In *submission*, 2012.

[21] Sheng Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd International Symposium on Microarchitecture (MICRO-42)*, 2010.

[22] Gabriel H. Loh and Samantika Subramaniam. Zesto: A Cycle-level Simulator for Highly Detailed Microarchitecture Exploration. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*.

[23] Chi-Keung Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*.

[24] Jason Mars et al. Contention aware execution: Online contention detection and response. In *CGO*, 2010.

[25] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, 2011.

[26] Harlan McGahn. Niagara 2 Opens the Floodgates. *Microprocessor Report*, 2006.

[27] A. Patel et al. MARSS: A full system simulator for multicore x86 CPUs. In *Design Automation Conference (DAC)*, 2011.

[28] M. Pathak, Y.J. Lee, T. Moon, and S.K. Lim. Through-silicon-via management during 3d physical design: When to add and how many? In *Proceedings of the International Conference on Computer-Aided Design*. IEEE Press, 2010.

[29] H Patil et al. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37)*, 2004.

[30] S. Rajapandian, K.L. Shepard, P. Hazucha, and T. Karnik. High-voltage power delivery through charge recycling. *IEEE Journal of Solid-State Circuits*, 2006.

[31] Vijay Janapa Reddi, Svilen Kanev, Wonyoung Kim, Simone Campanoni, Michael D Smith, Gu-yeon Wei, and David Brooks. Voltage Smoothing: Characterizing and Mitigating Voltage Noise in Production Processors via Software-Guided Thread Scheduling.

In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[32] Timothy Sherwood et al. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[33] Allan Snavely et al. Symbiotic job scheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 2000.

[34] S.R. Vangal et al. An 80-tile Sub-100-W Teraflops processor in 65nm CMOS. *IEEE Journal of Solid-State Circuits*, 2008.

[35] David Wentzlaff et al. On-chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 2007.

[36] M.T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2007.

[37] Sergey Zhuravlev et al. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.