Voltage Smoothing: Characterizing and Mitigating Voltage Noise in Production Processors via Software-Guided Thread Scheduling

Vijay Janapa Reddi[†‡], Svilen Kanev, Wonyoung Kim, Simone Campanoni, Michael D. Smith, Gu-Yeon Wei, David Brooks

Advanced Micro Devices (AMD) Research Labs[†], Harvard University

vijay.reddi@amd.com, {skanev, wonyoung, xan, smith, guyeon, dbrooks}@eecs.harvard.edu

Abstract—Parameter variations have become a dominant challenge in microprocessor design. Voltage variation is especially daunting because it happens so rapidly. We measure and characterize voltage variation in a running Intel® Core™2 Duo processor. By sensing on-die voltage as the processor runs single-threaded, multi-threaded, and multi-program workloads, we determine the average supply voltage swing of the processor to be only 4%, far from the processor's 14% worst-case operating voltage margin. While such large margins guarantee correctness, they penalize performance and power efficiency. We investigate and quantify the benefits of designing a processor for typical-case (rather than worst-case) voltage swings, assuming that a fail-safe mechanism protects it from infrequently occurring large voltage fluctuations. With today's processors, such *resilient* designs could yield 15% to 20% performance improvements. But we also show that in future systems, these gains could be lost as increasing voltage swings intensify the frequency of fail-safe recoveries. After characterizing microarchitectural activity that leads to voltage swings within multi-core systems, we show that a voltage-noise-aware thread scheduler in software can co-schedule phases of different programs to mitigate error recovery overheads in future resilient processor designs.

Keywords-dI/dt, inductive noise, error resiliency, voltage droop, hw/sw co-design, thread scheduling, hardware reliability

I. INTRODUCTION

As device feature sizes scale, microprocessor operation under strict power and performance constraints is becoming challenging in the presence of parameter variations. Process, thermal, and voltage variations require the processor to operate with large operating margins (or guardbands) to guarantee correctness under corner-case conditions that rarely occur. This level of robustness comes at the cost of lower processor performance and power efficiency. In the era of power-constrained processor design, supply voltage variation is emerging as a dominant problem as designers aggressively use clock gating techniques to reduce energy consumption. Non-zero impedance in the power delivery network combined with sudden current fluctuations due to clock gating, along with workload activity changes, lead to large and hard-to-predict changes in supply voltage at run time. Voltage fluctuations beyond the operating margin can lead to timing violations. If the processor must always avoid such *voltage emergencies*, its operating margin must be large enough to tolerate the absolute worst-case voltage swing.

Today's production processors use operating voltage margins that are nearly 20% of nominal supply voltage [1], but
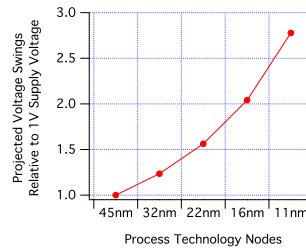

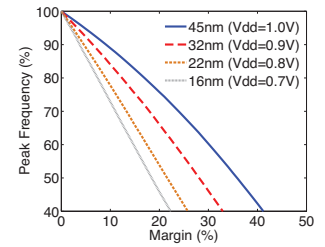Fig. 1: Voltage noise is increasing in future generations.

Fig. 2: Worst-case margins needed for noise are inefficient.

trends indicate that margins will need to grow in order to accommodate worsening peak-to-peak voltage swings. Both processor performance and power efficiency will suffer to an even greater extent than in today's systems. Fig. 1 shows the worst-case peak-to-peak swing in future generations relative to today's 45nm process technology.[1] Voltage swing doubles by the 16nm technology node. Fig. 2 summarizes the performance degradation associated with margins, showing that a 20% voltage margin in today's 45nm node translates to ∼25% loss in peak clock frequency.[2] A doubling in voltage swing by 16nm implies more than 50% loss in peak clock frequency, owing to increasing circuit sensitivity at lower voltages. Therefore, worst-case operating voltage margins are not sustainable in the long run.

Industry recognizes these trends and is moving towards *resilient* processor designs. Rather than setting the operating voltage margin according to the extreme activity of a power virus, designers relax the operating voltage margin to a more typical level of voltage swing. emergency, error-detection and error-recovery circuits dynamically detect and correct timing violations. In this way, designers use aggressive margins to maximize processor performance and power efficiency. Bowman et al. show that removing a 10% operating voltage margin leads to a 15% improvement in clock frequency [5]. Abundant recent work in architecture [6]–[10] and more recent circuit prototyping efforts [5], [11], [12] reflect this impending paradigm shift in architecture design.

As resilient processor architecture designs are still in their infancy, this paper focuses on understanding the benefits

---

[1]Based on simulations of a Pentium 4 power delivery package [2], assuming Vdd gradually scales according to ITRS projections from 1V in 45nm to 0.6V in 11nm [3]. To study package response, current stimulus goes from 50A-100A in 45nm. Subsequent stimuli in newer generations is inversely proportional to Vdd for the same power budget.

[2]Based on detailed circuit-level simulations of an 11-stage ring oscillator that consists of fanout-of-4 inverters from PTM [4] technology nodes.

and caveats of typical-case design. Using only off-the-shelf components to sense on-die silicon voltage fluctuations of an Core^{TM}2 Duo processor, we perform full-length program analysis, characterizing voltage noise in this production processor. Our findings indicate that aggressive margins could enable performance gains from 15% up to 20%. However, these gains are sensitive to three critical factors: the cost of error-recovery, aggressive margin settings, and program workload characteristics. Improperly setting the first two machine parameters leads to degraded workload performance, sometimes even beyond the baseline conservative worst-case design. We characterize the design space to illuminate the tradeoffs.

Future resilient processor microarchitectures will need very fine-grained error-recovery logic to maintain the benefits of resiliency. Building such recovery schemes will require intrusive changes to traditional architectural structures that add on-chip die area and cost overheads, in addition to complicating design, testing or validation. In order to alleviate this complexity, we propose *voltage-noise-aware thread scheduling* at the software layer, which allows designers to leverage more coarse-grained, cost-effective recovery schemes. The software enables this by reducing error-recovery rates, while assuming the hardware provides a fail-stop. To study the efficacy of thread scheduling in the anticipation of large voltage swings, we project future voltage noise trends by reducing decoupling capacitance of an existing processor.

Developing a software solution for mitigating voltage noise begins with understanding activity within the processor that leads to voltage swings. Studying voltage noise in a real processor enables some observations not revealed by published simulation efforts. Using microbenchmarks that stimulate the processor with highly specific events such as TLB misses and branch mispredictions, we examine and quantify the effect of various stall events on voltage noise. For instance, the pipeline flush caused by a single branch misprediction causes a voltage swing 1.7 times larger than that of an idling machine.

Multi-core execution leads to *voltage noise interference*. The same processor experiences a 42% increase in peak-to-peak swings when both of its cores are active and running the same microbenchmark. Therefore, either margins will need to be greater in multi-core systems when multiple cores are active simultaneously, or the system will need to tolerate more frequent error recoveries. However, effectively co-scheduling noise compatible events (or threads) together across cores can dampen peak-to-peak swings. Based on our understanding of the relationship between stall events and voltage swings, we construct a metric called *stall ratio* that enables the software layer to infer noise activity using existing hardware performance counters. It also explains the existence of *voltage noise phases* that are like program execution phases. They are recurring patterns of voltage droop and overshoot activity in response to changing microarchitectural behavior.

In summary, this paper (1) characterizes single-core and multi-core noise activity on a real chip, (2) presents a rigorous study that identifies the benefits of a resilient microarchitecture

design for voltage noise, and (3) provides a mechanism by which to dampen voltage noise in future processor generations. The underlying mechanisms that enable these contributions are the following:

- *Measurement and Extrapolation.* By tapping into two unused package pins that sense on-die silicon voltage, we demonstrate and validate the ability to study processor voltage noise activity under real execution scenarios unintrusively. Moreover, by breaking off package capacitors, we amplify the magnitude of voltage swings in the production processor to extrapolate and study voltage noise in future systems.
- *Characterization of Voltage Noise.* Combining the noise measurement setup with hand-crafted microbenchmarks and hardware performance counters, we study microarchitectural activity within the processor that leads to large voltage swings.
- *Mitigating Voltage Noise via Software-Guided Thread Scheduling.* Taking advantage of voltage noise phases, we propose, investigate, and demonstrate the benefits of a noise-aware thread scheduler for smoothing out voltage noise in multi-core chips. Oracle-based simulation results of thread scheduling reveal that software alleviates error recovery penalties in future resilient architecture designs.

Sec. II explains how we sense on-die voltage in a production processor in as it is operating in a regular environment. We use this setup to study resilient architecture design under typical-case conditions in Sec. III. The challenges we identify lead us towards an understanding of how activity within the microprocessor causes voltage to fluctuate. That in turn motivates us to evaluate thread scheduling in Sec. IV as a means of dampening voltage noise in multi-core systems. Finally, we conclude the paper with our remarks in Sec. V.

II. MEASUREMENT AND EXTRAPOLATION

We introduce a new methodology to measure voltage fluctuations in a production processor unintrusively. We explain how we sense the voltage using only off-the-shelf components, rather than relying on specialized equipment. We validate our experimental setup by re-constructing the impedance profile of the system and comparing it with data from Intel [13], [14], as well as past literature. Using the same setup, we describe a new means of extrapolating voltage noise in future systems by removing package capacitors from a working chip. Finally, we show how to determine the worst-case operating margin by undervolting the processor.

### A. Using Off-the-Shelf Components

Previous descriptions of voltage swings have been done primarily by using either custom voltage transient test (VTT) tool kits [13], [15] or simulation [2], [8], [16]–[18]. These approaches have severe limitations that prevent us from observing the voltage noise characteristics of a real processor as it is running full programs and operating under production settings. VTT tools replace both the processor and its encompassing package for platform validation purposes. Such test
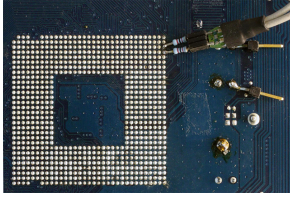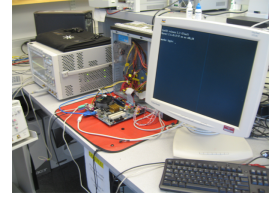
(a) Connecting to on-die voltage pins via a low impedance path using VCC_{sense} and VSS_{sense}.
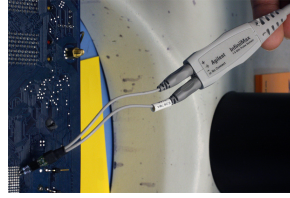
(b) Sensing voltage using an Infini-iMax 1.5GHz 1130A differential probe that has ultra low loading.
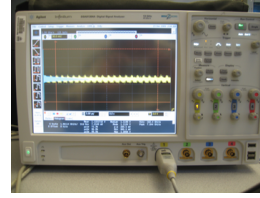
(c) Measuring sensed voltage using an Agilent DSA91304A Infiniium oscilloscope.

(d) Gathering oscilloscope data using an external system that connects over the network.

Fig. 3: Measurement setup to sense and measure voltage fluctuations within the processor at execution time.

harnesses enable only limited characterization of voltage noise phenomena, like resonance, under manual external current stimuli. They require custom hardware that is not publicly available. Since the tools replace the processor, it is impossible to correlate program execution activity to voltage noise on the system. Simulation efforts overcome this limitation by integrating processor package models [19]–[21] with microarchitecture and power consumption models. Simulation has been the primary vehicle for voltage noise research over the past several years. However, analysis via simulation suffers from constraints like the length of program execution one can study, or the extent to which the models are representative of real processors. Moreover, integrated simulation efforts have only focused on single core execution models. In today's multi-core era, it is important to characterize the effect of interactions across cores that lead to voltage noise.

The benefits of the setup we describe here are that it allows us to measure voltage fluctuations within the processor without a special experimental toolkit. Even more importantly, this setup allows us to run through entire suites of real programs to completion, rather than relying on simulation to observe activity over just a few millions of instructions. To the best of our knowledge, nothing is publicly known about the noise characteristics of real benchmarks running on actual production processors, especially in multi-core systems. The processor we study is an Intel® Core™2 Duo Desktop Processor (E6300) on a Gigabyte GA-945GM-S2 chipset platform. The methodology and framework we describe here serve as a basis for all evaluation throughout the rest of the paper. While we constrain our analysis to this processor and motherboard setup, the general methodology is extensible to other system platforms as well.

Methodology. We unintrusively sense voltage near the silicon through isolated low impedance processor pin connections. The processor package exposes two pins for processor power and ground, as VCC_{sense} and VSS_{sense}, respectively. These pins typically exist for validation reasons, allowing designers to sense and test on-die voltage during controlled voltage transitions, such as dynamic voltage and frequency scaling for thermal and power management [22].

Fig. 3 illustrates how we connect, sense, measure and gather data from these pins, going from (a) through (d) in that order. To ensure minimal or no measurement error and to maintain high signal fidelity, we use a InfiniiMax 1.5GHz 1130A

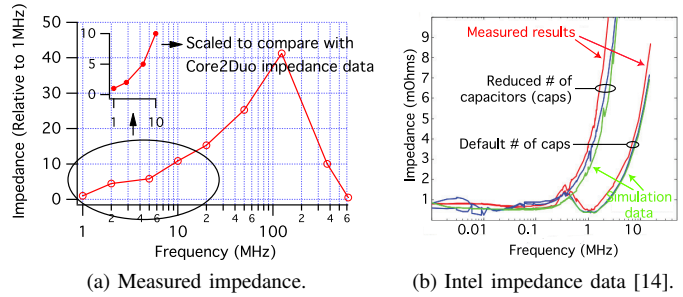
(a) Measured impedance.

(b) Intel impedance data [14].

Fig. 4: Measurement validation, comparing impedance derived on our machine versus well established data from Intel.

differential probe to sense voltage. A DSA91304A Infiniium oscilloscope measures probe data at a high frequency matching the probe functionalities. The scope stores these measured readings in memory using a highly compressed histogram format that it internally supports. Therefore, it allows us to gather data over several minutes, which corresponds to activity over several hundreds of billions of committed program instructions, well beyond simulation reach. Every 60-second interval, a remote data collection system then transparently gathers all scope data over the network.

Validation. In order to validate our experimental methodology, we construct the impedance profile of our Core™2 Duo system and compare it with Intel data. An impedance plot shows the relationship between current fluctuations and voltage noise (see Fig. 4). It is used to study the noise characteristics of a processor. Typically, voltage regulator module designers and package designers rely on such information to build robust power supplies that can match the needs of a processor.

We follow the methodology that Intel designers prescribe to construct the impedance profile of a chip [14]. However, we replace their step-current generation technique with a current-consuming software loop that runs on the processor. The loop consists of separate high-current draw and low-current draw instruction sequences. Tiwari et al. explain how to determine the amount of current individual instructions draw [23]. We leverage this technique to determine the set of instructions to use within the loop body across each of the paths. By modulating execution activity through these paths, the loop can control the current draw frequency.

Fig. 4a is the impedance profile constructed using this approach on our system. There are two important validation
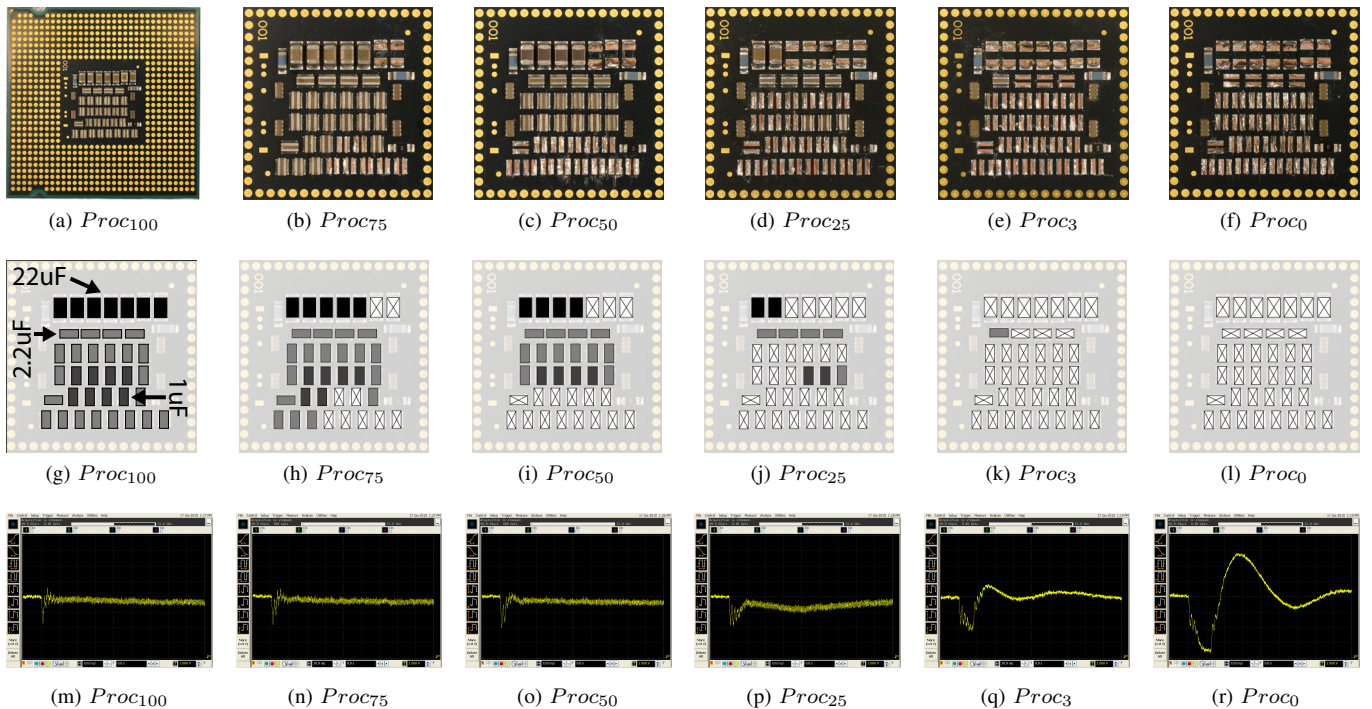
(a) $Proc_{100}$ (b) $Proc_{75}$ (c) $Proc_{50}$ (d) $Proc_{25}$ (e) $Proc_3$ (f) $Proc_0$

(g) $Proc_{100}$ (h) $Proc_{75}$ (i) $Proc_{50}$ (j) $Proc_{25}$ (k) $Proc_3$ (l) $Proc_0$

(m) $Proc_{100}$ (n) $Proc_{75}$ (o) $Proc_{50}$ (p) $Proc_{25}$ (q) $Proc_3$ (r) $Proc_0$

Fig. 5: (a)-(f) Land side of a Core2Duo^{TM} processor, showing its package capacitors as we incrementally remove them to extrapolate voltage noise. (g)-(l) Capacitor values and the manner in which each chip is altered; white boxes with a cross correspond to removed capacitors. (m)-(r) Voltage droop response to the reset signal.

points to observe. First, impedance peaks at around the resonance frequency of 100MHz to 200MHz, which matches a large body of prior work describing typical power delivery network characteristics [1], [2], [8], [24], [25]. Second, the small graph embedded within Fig. 4a corresponds appropriately to previously published Intel data [13], [14]. Between 1MHz and 10MHz, Measured results for the Default # of caps in Fig. 4b closely correspond to our results within the scaled graph. With this we conclude the validation of our experimental setup and utilize it for all other measurements.

B. Studying Future Systems

Voltage swings are growing in future processor generations. To extrapolate and study this effect on resilient architecture designs, we amplify voltage noise in the production processor by reducing package capacitance. As a cautionary note to the reader, the manner in which we remove package capacitance does not translate to an absolute representation of what voltage noise will look like in future nodes. It is a gross estimate. There may be non-linear effects to consider. Nevertheless, this technique suffices as a heuristic that resembles the simulation-based trend line in Fig. 1. This method suffices to approximately study the effects of voltage noise in future systems across a diverse range of workloads and observe full program characteristics using a real processor.

Basis. Designers ship processors with on-chip and off-chip decoupling capacitance to dampen peak-to-peak voltage swings by reducing impedance of the power delivery network. Off-chip decoupling capacitors are externally visible on the land side of a packaged processor (see Fig. 5a). Since voltage is the product of current and impedance, for a given current stimulus at a particular frequency, the magnitude of voltage swings will be smaller with smaller overall impedance. As package capacitance decreases, impedance increases, causing much larger peak-to-peak voltage swings within the processor for the same magnitude of activity fluctuations.

Fig. 4b quantifies this relationship between package capacitance and impedance. The system experiences much larger impedance across the frequency range with fewer capacitors. See the lines corresponding to Reduced # of capacitors (caps) in the figure. The same system has much smaller impedance with more capacitors (see Default # of caps). At 1MHz, the peak impedance is only 0.5mOhms in a system that is well damped, whereas it is 5 times as much under Reduced # of capacitors (caps).

Decap Removal. By removing decoupling capacitors (or "decaps"), we create a range of five new processors (shown in Figs. 5b-5f) with decreasing package capacitance. We identify the successive processors using a subscript following the word "*Proc*" that describes the amount of package capacitance left behind after decap removal. For instance, $Proc_{100}$ retains all its original capacitors, while $Proc_3$ retains only 3% of its original package capacitance. After decap removal, we verify the operational stability of the processors by subjecting each one to an aggressive run-time test using CPUBurn [26]. This program stresses the processor's execution units while continuously monitoring execution for errors.

The processor package contains different capacitive ele-

ments. After decap removal, we determined their individual values, which are shown in Fig. 5g. Identical values share a color. White boxes in Figs. 5h-5l illustrate the manner in which we altered the processor to lower capacitance. For instance, to eliminate 50% of all capacitors, we remove half of each kind of capacitor.

Effect. To determine the impact of decap removal on voltage swings, we stimulate the processor with a reset signal. Resetting, or turning off and on, the processor, causes a very sharp, large and sudden change in current activity. We reset the processor as it is idling, or running the idle loop of the operating system. Since impedance across $Proc_{100}$ through $Proc_0$ varies because of their differing levels of package capacitance, their magnitude of voltage swings also varies in response to this stimulus.

Oscilloscope screen shots in Figs. 5m-5r correspond to the different processors' core supply voltages at the moment of the reset signal, measured using our experimental setup from the previous section. $Proc_{100}$ in Fig. 5m experiences a sharp 150mV voltage droop for a very brief amount of time, but voltage quickly recovers. As package capacitance progressively decreases going from $Proc_{100}$ to no package capacitance altogether in $Proc_0$, voltage swings not only get incrementally larger, but also extend over a longer amount of time. $Proc_0$ experiences a 350mV drop over several cycles in Fig. 5r. This leads to timing violations that prevent the processor from even booting up. However, it is the only processor that fails stability testing.

Fig. 6 summarizes the peak-to-peak voltage swings across all processors relative to $Proc_{100}$. We can safely normalize this data because differences in the peak-to-peak swing of an idling machine across the processors is negligible. However, their noise characteristics diverge when activity occurs. Fig. 6 shows one instance of such divergence, in response to resetting the processors. The trend in this figure is roughly the same as in Fig. 1. The knee of the curve is around $Proc_{25}$ and $Proc_3$, so from here on we rely on them as our future nodes, while $Proc_{100}$ is representative of today's systems.

C. Worst-Case Margin

The worst-case margin is the voltage guardband that tolerates transient voltage swings. It is $(V_{nominal}\text{-}V_{min})/V_{nominal}$, where V_{min} is the minimum voltage before an execution error can occur. This work discusses performance improvements as a result of utilizing aggressive voltage margins, rather than utilizing worst-case margins. Therefore, we needed to determine the worst-case lower margin.

In the Core^{TM}2 Duo processor, the worst-case margin is approximately 14% below the nominal supply voltage. In order to determine this value, we progressively undervolt the processor while maintaining its clock frequency. This ultimately forces the processor into a functional error, which we detect when the processor fails stress-testing under multiple copies of the power virus.
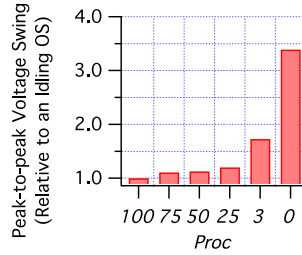

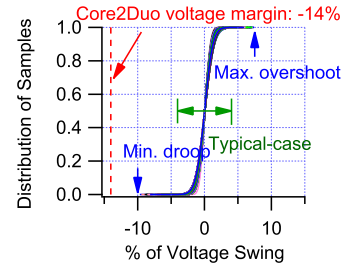Fig. 6: Voltage swings in Figs. 5m-5r across processors shown in Figs. 5a-5f.

Fig. 7: Cumulative distribution of voltage samples across 881 program executions.

III. NOISE IN PRODUCTION PROCESSORS

In this section, we discuss the voltage noise characteristics of real-world programs as they are run to completion, using our experimental measurement setup from Section II. We summarize the noise profiles of single-threaded, multi-threaded and multi-program executions prior to providing more in-depth analysis in later sections. This section covers the extent to which worst-case operating voltage margins are absolutely necessary, followed by motivating and evaluating aggressive voltage margins for typical-case design. Our analysis includes $Proc_{100}$, $Proc_{25}$ and $Proc_3$. Therefore, we characterize typical-case design not only in the context of today's systems, but we also project into the future.

A. Typical-Case Operation

The worst-case operating voltage margin is overly conservative. We determine this from 881 benchmarking runs. The experiments include a spectrum of workload characteristics: 29 single-threaded SPEC CPU2006 workloads, 11 Parsec [27] programs and 29×29 multi-program workload combinations from CPU2006. Consequently, we believe that the conclusions drawn from this comprehensive investigation are representative of production systems and not biased towards a favorable outcome.

Fig. 7 shows a cumulative histogram distribution of voltage samples for $Proc_{100}$. We plot the deviation of each sample relative to the nominal supply voltage. Each line within the graph corresponds to a run. Run-time voltage droops are as large as 9.6% (see Min. droop marker). Therefore, the 14% worst-case margin is necessary. However, they occur very infrequently. Most of the voltage samples are within 4% of the nominal voltage. The Typical-case marker in Fig. 7 identifies this range. Only a small fraction of samples (0.06%) lie beyond this typical-case region. Although the magnitude of overshoots can also be large (see Max. overshoot), they are significantly less frequent, especially in future nodes. Therefore, we will primarily focus on droops.

B. Designing for Typical-Case Operation

When a microarchitecture is optimized for typical-case conditions and relies on an error-recovery mechanism to handle emergencies, three critical factors determine its performance: (1) workload characteristics, (2) the operating voltage margin setting, and (3) the cost of rolling back execution. In this
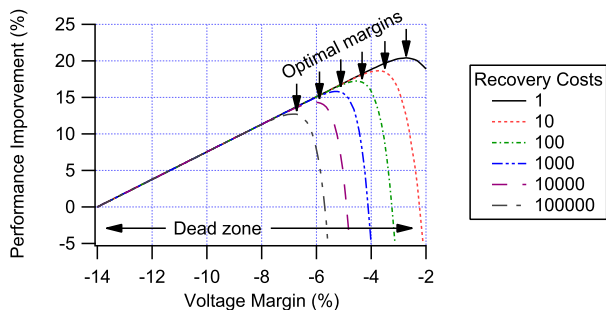
Fig. 8: Typical case improvement across a range of recovery costs on $Proc_{100}$, showing substantial room for tighter voltage margins.

(a) $Proc_{25}$ (b) $Proc_3$

Fig. 9: Typical-case swings in future processors are increasingly more slanted compared to $Proc_{100}$ (see Fig. 7) as voltage noise grow.

section, we evaluate how these factors influence peak performance.

Performance Model. In order to study the relationship between these critical parameters, we inspect performance gains from allowing voltage emergencies at runtime. Since our analysis is based off of a current generation processor 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 [28]. 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 [5]. We assume this $1.5\times$ 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.

Recovery Costs. Fig. 8 shows performance improvement over a range of voltage margins, assuming specific recovery costs. These recovery costs reflect prior work: Razor [7], a very fine-grained pipeline stage-level error detection and recovery mechanism, has a recovery penalty of only a few clock cycles. DeCoR [8] leverages existing load-store queues and reorder buffers in modern out-of-order processors to delay instruction commit just long enough to verify whether an emergency has occurred. Typical delay is around tens of cycles. Reddi et al. [29] propose a scheme that predicts emergencies using program and microarchitectural activity, relying on an optimistic 100-cycle hardware-based checkpoint-recovery mechanism that guarantees correctness. Current production systems typically take thousands of clock cycles to complete recovery [30]. Alternatively, recovery cost-free computing is also emerging where it is possible to exploit the inherently statistical and error-resilient nature of workloads to tolerate
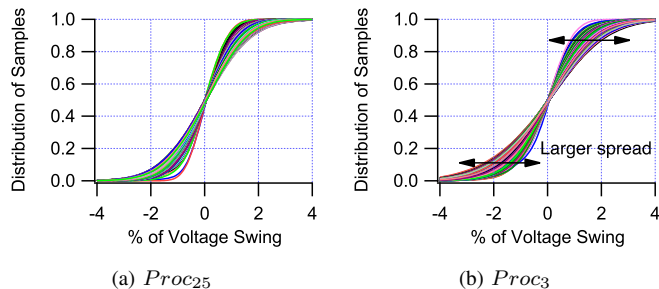
errors without a hardware fail-safe [31]. Our workloads do not fall into this criteria, therefore we target the more general case where hardware robustness is a must.

Optimal Margins. In order for a resilient architecture design to operate successfully under any aggressive margin, an optimal margin must exist. Such a margin is necessary to design the processor towards a specific design point in the presence of workload diversity. Fig. 8 data is an average of all 881 program runs. These include single-threaded, multi-threaded workloads, and an exhaustive multi-program combination sweep that pairs every CPU2006 benchmark with every other benchmark in the suite. Despite this heterogeneous set of execution profiles, we find that it is possible to pick a single static optimal margin. There is only one performance peak per recovery cost. Otherwise, we would see multiple maxima or some other more complicated trend.

Note that each benchmark can have a unique optimal voltage margin. However, we found that the range of optimal margins is small across all executions. So although it is possible to achieve even better results on a per benchmark basis, improvements over our one-design-fits-all methodology are likely to be negligible, at least relative to our gains.

In Fig. 8, every recovery mechanism has its own optimal margin. Depending on the cost of the recovery mechanism, gains vary between 13% and $\sim$21%. Coarser-grained recovery mechanisms have more relaxed optimal margins while finer-grained schemes have more aggressive margins and as a consequence are able to experience better performance improvements. However, being overly aggressive and setting the operating voltage margin beyond the optimal causes rapid performance degradation. At such settings, recoveries occur too frequently and penalize performance, thus the benefits begin to diminish. Recovery penalties can be so high that they can even push losses beyond the original conservative design (i.e., 14% margin on Core^{TM}2 Duo). This corresponds to below 0% improvement, which we refer to as the Dead zone.

Diminishing Gains. As we extrapolate the benefits of resilient microarchitecture designs into future nodes using $Proc_{25}$ and $Proc_3$, we can anticipate an alarming decrease in the corresponding performance gains. These diminishing gains are due to worsening voltage swings. Processors in the future will experience many more emergencies compared to $Proc_{100}$ at identical margins. Fig. 9 shows the distribution of
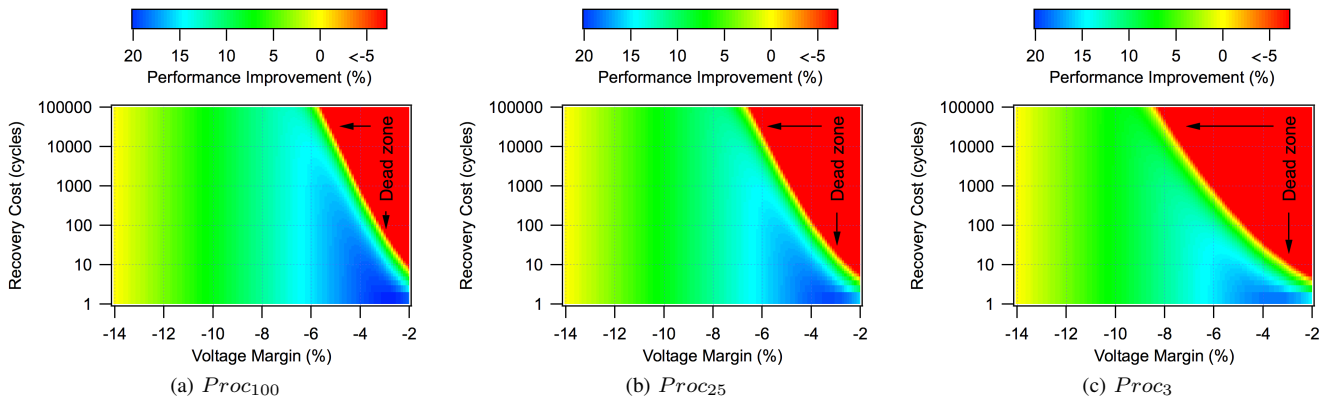
Fig. 10: Performance improvement under typical-case design using various recovery costs and voltage margin settings.

voltage samples around the typical case margin on $Proc_{25}$ and $Proc_3$. Notice how samples for $Proc_{25}$ are packed more tightly around the nominal than for $Proc_3$. Also, the lines are more tightly bound together. In today's $Proc_{100}$ system, only 0.06% of all voltage samples fall below the typical-case -4%. By comparison, over 0.2% and 2.2% of all samples violate the -4% margin in $Proc_{25}$ and $Proc_3$, respectively.

To quantify the impact and to illustrate diminishing gains in performance better, we rely on the heatmaps in Fig. 10. These heatmaps include additional and more comprehensive sweeps of error recovery costs versus operating voltage margins. The intensity of the heatmaps corresponds to the amount of performance improvement. We see that the large pocket of performance improvement in $Proc_{100}$ between -6% and -2% voltage margin quickly diminishes as we go to $Proc_{25}$ and $Proc_3$. Compare the blue region in Fig. 10a with that in Fig. 10b and Fig. 10c, respectively.

Long-term Implications. Retaining the same level of performance improvement as in today's $Proc_{100}$ will require future processors to make use of more fine-grained recovery mechanisms. For instance, in Fig. 10, designers could use a 1000-cycle recovery mechanism in $Proc_{100}$ to reap a 15% improvement in performance. But in $Proc_{25}$, they would have to achieve a ten-fold reduction in recovery cost implementation, to just 100-cycles. $Proc_3$ requires even further reductions to ∼10 cycles per recovery to maintain the 15% improvement.

The problem with implementing fine-grained recovery is that they are severely intrusive. Razor- and DeCoR-like schemes require invasive changes to traditional microarchitectural structures. Moreover, they add area and cost overheads, making design and validation even more complicated than they already are in today's systems.

Therefore, we advocate mitigating error-recovery costs of coarser-grained mechanisms, by investigating software-assistance to hardware guarantees. A major benefit of coarser recovery mechanisms is that some form of checkpoint-recovery is already shipping in today's systems [30], [32] for soft-error tolerance. Moreover, newer applications are emerging that leverage and re-use this general-purpose hardware for tasks such as debugging, testing, etc. [33], [34]. In this way, software aids efficient and cost-effective typical case design.

C. Understanding Droops and Overshoots

The first step towards mitigating error recovery overheads via software is to understand microarchitectural-level activity that leads to voltage noise. By stimulating just one core within the $Proc_{100}$ Core™2 Duo processor in highly specific ways using microbenchmarks, we quantify the perturbation effects of independent and individual microarchitectural events on the processor's nominal supply voltage. Subsequently, we extend our analysis to multi-core. These microarchitectural events, even when acting in isolation within cores, interfere across cores, leading to much larger chip-wide voltage swings.

Single Core. Microarchitectural events that cause stalls lead to voltage swings. To quantify this effect, we hand-crafted the following microbenchmarks that cause the processor to stall: L1 (only) and L2 cache misses, translation lookaside buffer (TLB) misses, branch mispredictions (BR) and exceptions (EXCP). Each microbenchmark is run in a loop, so that activity recurs long enough to measure its effect on core voltage.

To demonstrate that the microbenchmarks exhibit steady and repeatable behavior for measurements, Fig. 11 is a snapshot of core voltage as the processor is experiencing TLB misses. The sawtooth-like waveform is the switching frequency of the voltage regulator module (VRM). This is background activity. Embedded within that waveform are recurring voltage spikes. These correspond to the TLB microbenchmark. A TLB miss causes voltage within the processor to swing because it stalls execution momentarily, causing a large drop in current draw. As a result of finite impedance in the power supply network, voltage shoots above the nominal value.

The processor may also experience a correspondingly strong voltage droop following an initial overshoot. Consider an L1 cache miss event. During the time it takes to service the miss, pipeline activity ramps down. Current drops and voltage overshoots. But after the miss data becomes available, functional units become busy and there is a surge in current activity. This steep increase in current causes voltage to droop.

The magnitude of the voltage swing varies depending on the type of event. We summarize this across all events relative to an idling system in Fig. 12. On an idling system, we observe only the VRM ripple. Therefore, voltage overshoots and droops are distinctly noticeable and measurable in microbenchmark cases. Fig. 12 shows that branch mispredictions
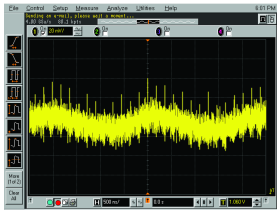
Fig. 11: TLB misses causing overshoots.

Fig. 12: Effect of microarchitectural events on supply voltage.

Fig. 13: Impact of microarchitectural event interference across cores.

cause the largest amount of voltage swing compared to other events. The maximum peak-to-peak voltage swing is over 1.7 times that of our baseline.

Multi-Core. Microarchitectural activity across cores causes interference that leads to chip-wide transient voltage swings that are much larger than their single-core counterparts. Using the same set of microbenchmarks as above, we characterize the effect of simultaneously running multiple events on the processor. Each core runs just one specific microbenchmark. We then capture the magnitude of the peak-to-peak swing across the entire chip. Both cores share the same power supply source. The heatmap in Fig. 13 is the effect of interference across the two cores. Once again we normalize the magnitude of the swings relative to an idling machine. The y-axis corresponds to Core 0 and the x-axis to Core 1.

When both cores are active, the peak-to-peak voltage swing worsens. The maximum peak-to-peak swing in Fig. 13 is 2.42×, whereas in the single-core test it is only 1.7×, a 42% increase. In the context of conservative worst-case design, where designers allocate sufficiently large margins to tolerate the absolute worst-case swing, this increase implies proportionally larger margins are necessary to compensate for amplified voltage swings when multiple cores are active. As the number of cores per processor increases, this problem can worsen. Within the context of resilient microarchitecture designs, error-recovery rates will go up.

However, the maximum voltage swing varies significantly depending on the coupling of events across the cores. Depending on this pair, the chip may experience either *constructive interference* or *destructive interference*. In the context of this paper, constructive interference is the amplification of voltage noise and destructive interference is the dampening of voltage noise relative to noise when only one core is active. The worst-case peak-to-peak swing discussed above is an example of constructive interference. It occurs when both cores are running EXCP. But pairing this event with any other event than itself always leads to a smaller peak-to-peak swing. Sec. IV-A shows an example of destructive interference, where the swing when two cores are simultaneously active is smaller than during single core execution.

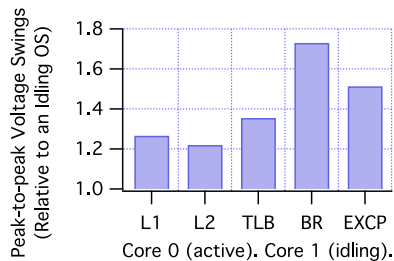Constructive interference is a problem in multi-core systems, since individual cores within the processor typically share a single power supply source.[3] Therefore, a transient voltage droop anywhere on the shared power grid could inadvertently affect all cores. If the droop is sufficient to cause an emergency, the processor must initiate a global recovery across all cores. Such recovery comes at the hefty price of system-wide performance degradation. Therefore, mitigating voltage noise in multi-core systems is especially important.
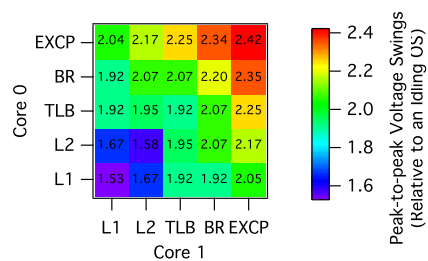
IV. MITIGATING VOLTAGE NOISE

In order to smooth voltage noise in multi-core processors and mitigate error recovery overheads, this section investigates 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.

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. As everything in this section builds towards the ultimate goal of improving the efficiency of resiliency-based architectures in the future, we use the $Proc_3$ processor.

A. 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

---

[3]In this study, we only focus on off-chip VRMs, as they are more widespread. Future processors may have on-chip per-core VRMs, but Kim et al. show that such designs can in fact worsen voltage noise [35]. Similarly, designers of the IBM POWER6 processor tested split- versus connected-core power supplies and found that voltage swings are much larger when the cores operate independently [1].
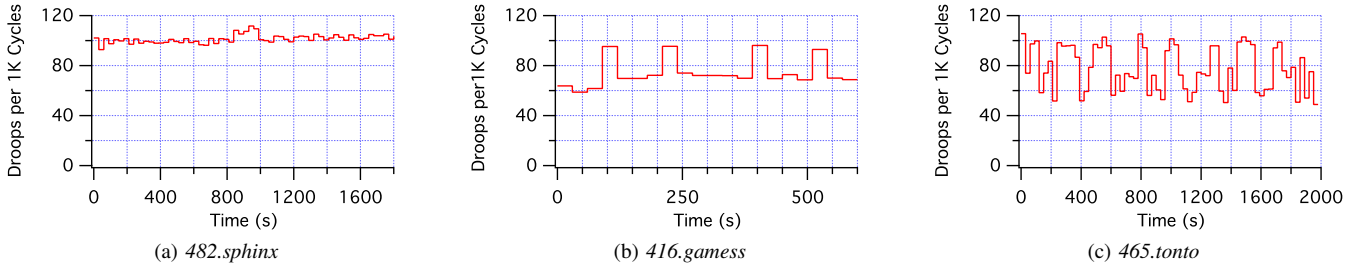
(a) *482.sphinx*

(b) *416.gamess*

(c) *465.tonto*

Fig. 14: 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.
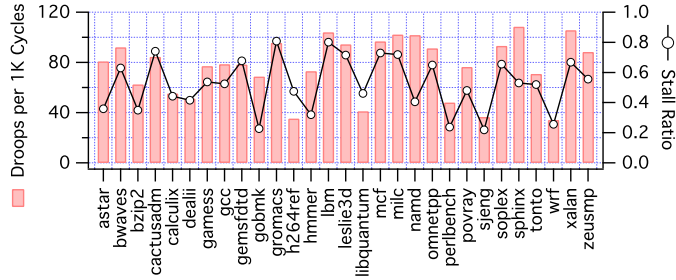

Fig. 15: Single-core droop activity, showing a heterogeneous mix of noise levels along with correlation to stalls.

for characterization purposes, Fig. 14 shows droops 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 Fig. 14c, oscillating strongly and more frequently between 60 and 100 droops per 1000 cycles every several tens of seconds.

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.

Fig. 15 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.

B. 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 Fig. 16a. 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 Fig. 16a. 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*. Fig. 16b 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 Fig. 16c). 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

(a) Setup for co-scheduling experiment.

(b) *473.astar* single-core noise profile.
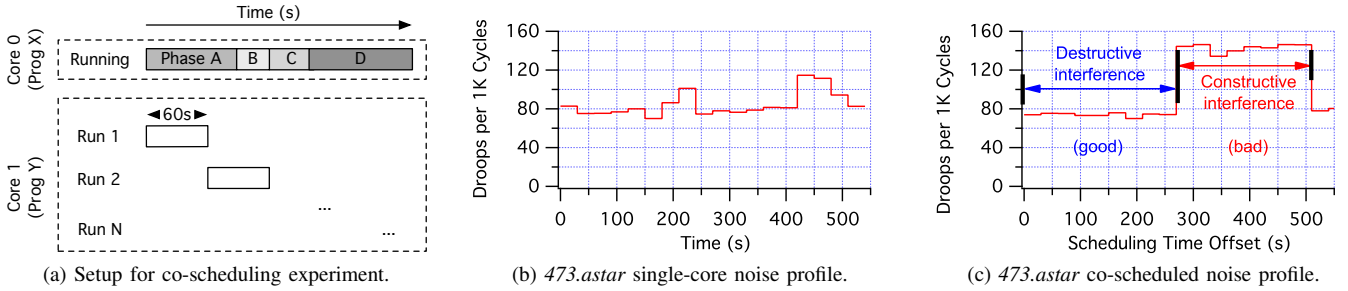
(c) *473.astar* co-scheduled noise profile.

Fig. 16: (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).
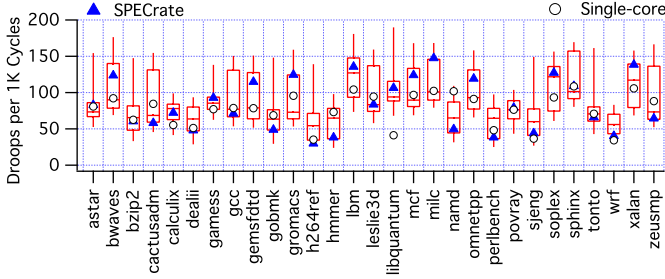

Fig. 17: Droop variance across single core and dual cores.

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

Tab. I: SPECrate typical-case design analysis at optimal margins.

schedules as well. Fig. 17 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 Fig. 17 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. In contrast, if nothing were done to mitigate voltage swings in multi-core systems, microbenchmarking analysis in Sec. III-C indicates that margins will need to grow.

C. 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 [36]–[42], 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 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 in Fig. 13 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 Fig. 17 shows no apparent preferential bias towards either minimizing or maximizing droops. Droop activity is spread uniformly over
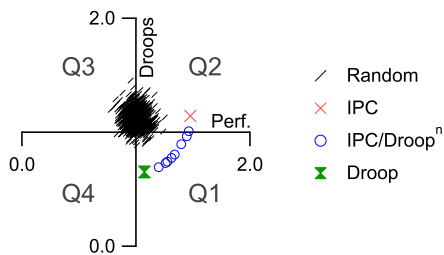
Fig. 18: Policy impact of droop vs. performance relative to SPECrate.

Fig. 19: Increase over the number of schedules in Tab. I that pass.

the entire workload 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.

Fig. 18 shows the results of our test. Each marker corresponds to one batch schedule. The four quadrants in Fig. 18 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 distributed 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.

D. Reducing Recovery Overheads

As voltage noise grows more severe, it will become harder for resilient systems to meet their typical-case design targets. Tab. I illustrates this point for the $Proc_3$ processor. For each recovery cost, the table contains an optimal margin and an associated performance improvement at this aggressive
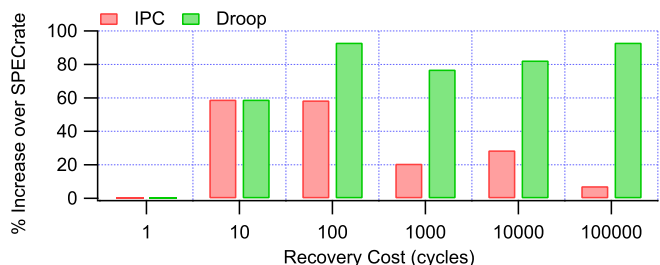
margin. This is the margin at which the system experiences maximum performance improvement over worst-case design (i.e., 14% on the $Core^{TM}2$ Duo processor). This margin is determined from analyzing the performance improvements across all 881 workloads in our test setup. But not all workloads can 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 the performance model (see Sec. III-B). We then determine the number of schedules that pass across the benchmark suite. Fig. 19 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 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 Fig. 18 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.

V. Conclusion

Measurements on a Core^{TM}2 Duo processor show that voltage noise will be a dominant issue in the future because designing processors for worst-case conditions will increasingly compromise performance and/or power efficiency. Resilient microarchitecture designs, optimized for typical-case, rather than worst-case, operation, and backed by error-recovery hardware, will become essential. But while these emerging designs hold great promise in the short term, their long-term outlook is doubtful. Because growing voltage swings lead to frequent voltage emergencies in such architectures, their error-recovery overhead will become a major performance bottleneck. And we show that increasing the number of cores per processor can make voltage noise worse. But not all noise interference among cores is constructive, and voltage noise is not altogether irregular. Destructive interference can smooth supply voltage, and recurring microarchitectural stall behavior can produce periodic voltage noise phases. Based on these observations, we demonstrate via simulation a software thread scheduler for multi-core processors that mitigates error-recovery overheads by co-scheduling threads known to interfere destructively.

References

[1] N. James et al., "Comparison of split-versus connected-core supplies in the POWER6 microprocessor," in ISSCC, February 2007.
[2] M. S. Gupta et al., "Understanding voltage variations in chip multiprocessors using a distributed power-delivery network," in DATE, 2007.
[3] International Technology Roadmap for Semiconductors, "Process integration, devices and structures," 2007.
[4] W. Zhao et al., "Predictive technology model for sub-45nm early design exploration," ACM JETC, 2006.
[5] 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.
[6] M. de Kruijf et al., "Relax: An architectural framework for software recovery of hardware faults," in ISCA, 2010.
[7] D. Ernst et al., "Razor: A low-power pipeline based on circuit-level timing speculation," in MICRO, 2003.
[8] M. S. Gupta et al., "DeCoR: A Delayed Commit and Rollback Mechanism for Handling Inductive Noise in Processors," in HPCA, 2008.
[9] B. Greskamp et al., "Blueshift: Designing processors for timing speculation from the ground up." in HPCA, 2009.
[10] M. D. Powell et al., "Architectural core salvaging in a multi-core processor for hard-error tolerance," in ISCA, 2009.
[11] D. Bull et al., "A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient- error tolerance and adaptation to PVT variation," in ISSCC, 2009.
[12] J. Tschanz et al., "A 45nm resilient and adaptive microprocessor core for dynamic variation tolerance," in ISSCC, 2009.
[13] "Voltage Regulator-Down (VRD) 11.0," Processor Power Delivery Design Guidelines For Desktop LGA775 Socket, 2006.
[14] S. Chickamenahalli et al., "Microprocessor platform impedance characterization using VTT tools," in APEC, 2005.
[15] "http://www.cascadesystems.net/lga775.htm."
[16] M. S. Gupta et al., "An event-guided approach to handling inductive noise in processors." in DATE, 2009.
[17] M. Powell et al., "Pipeline muffling and a priori current ramping:architectural techniques to reduce high-frequency inductive noise," in ISLPED, 2003.
[18] ——, "Exploiting resonant behavior to reduce inductive noise," in ISCA, 2004.
[19] T. Rahal-Arabi et al., "Design and validation of the Pentium 3 and Pentium 4 processors power delivery," in VLSI Circuits Digest of Technical Papers, 2002. Symposium on, 2002.
[20] K. Aygun et al., "Power delivery for high-performance microprocessors," Intel Technology Journal, 2005.
[21] R. Joseph et al., "Control techniques to eliminate voltage emergencies in high performance processors," in HPCA, 2003.
[22] "Intel Core Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 and E4000 Series," Datasheet, 2008.
[23] V. Tiwari et al., "Instruction level power analysis and optimization of software," VLSI Signal Processing, 1996.
[24] K. Aygun et al., "Measurement-to-modeling correlation of the power delivery network impedance of a microprocessor system," in EPEPS, 2004.
[25] A. Waizman, "CPU power supply impedance profile measurement using fft and clock gating," in EPEPS, 2003.
[26] M. Mienik, "http://users.bigpond.net.au/cpuburn."
[27] C. Bienia et al., "The PARSEC benchmark suite: Characterization and architectural implications," in PACT, 2008.
[28] "http://software.intel.com/en-us/intel-vtune/."
[29] V. J. Reddi et al., "Voltage emergency prediction: A signature-based approach to reducing voltage emergencies," in HPCA, 2009.
[30] Slegel et al., "IBM's S/390 G5 microprocessor design," Micro, IEEE, 1999.
[31] N. R. Shanbhag et al., "Stochastic computation," in DAC, 2010.
[32] H. Ando et al., "A 1.3 GHz fifth-generation SPARC64 microprocessor," in DAC, 2003.
[33] N. J. Wang et al., "ReStore: Symptom-Based Soft Error Detection in Microprocessors," IEEE Trans. Dependable Secur. Comput., 2006.
[34] S. Narayanasamy et al., "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," in ISCA, 2005.
[35] W. Kim et al., "System level analysis of fast, per-core DVFS using on-chip switching regulators," in HPCA, 2007.
[36] A. Snavely et al., "Symbiotic job scheduling for a simultaneous mutlithreading processor," SIGPLAN Not., 2000.
[37] A. Fedorova, "Operating system scheduling for chip multithreaded processors," Ph.D. dissertation, 2006, adviser-Seltzer, Margo I.
[38] J. Mars et al., "Contention aware execution: Online contention detection and response," in CGO, 2010.
[39] R. Knauerhase et al., "Using OS observations to improve performance in multicore systems," IEEE Micro, 2008.
[40] S. Zhuravlev et al., "Addressing shared resource contention in multicore processors via scheduling," in ASPLOS, 2010.
[41] D. Chandra et al., "Predicting inter-thread cache contention on a chip multi-processor architecture," in HPCA, 2005.
[42] F. J. Cazorla et al., "Predictable performance in SMT processors: Synergy between the OS and SMTs," IEEE Trans. Comput., 2006.