A System-Level View of Voltage Noise in Production Processors

SVILEN KANEV, Harvard University
VIJAY JANAPA REDDI, The University of Texas at Austin
TIMOTHY M. JONES, The University of Cambridge
WONYOUNG KIM, SIMONE CAMPANONI, MICHAEL D. SMITH, GU-YEON WEI, DAVID BROOKS, Harvard University

Parameter variations have become a dominant challenge in microprocessor design. Voltage variation is especially daunting because it happens 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 the investigated 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 two software techniques that have the potential to mitigate such voltage emergencies. A voltage-aware compiler can choose to de-optimize for performance in favor of better noise behavior, while a thread scheduler can co-schedule phases of different programs to mitigate error recovery overheads in future resilient processor designs.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance

General Terms: Performance, Reliability

Additional Key Words and Phrases: dI/dt, inductive noise, error resiliency, voltage droop, hw/sw co-design, thread scheduling, hardware reliability

1. INTRODUCTION

With device feature sizes scaling, 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 and power 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, also referred to as dI/dt *fluctuations*. Such changes that go beyond the operating margin can lead to timing violations. If the processor must always avoid these *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 [James et al. 2007], but 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

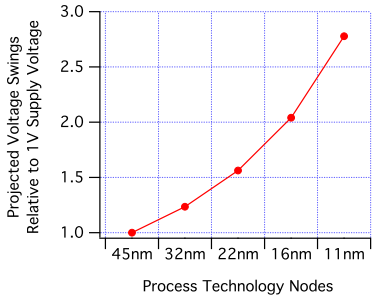


Fig. 1: Voltage noise is increasing in future generations.

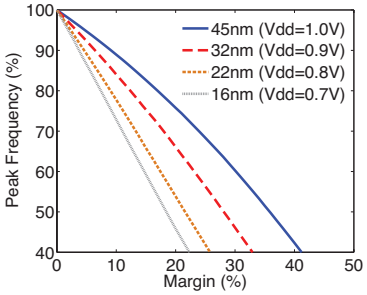


Fig. 2: Worst-case margins needed for noise are inefficient.

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. Such designs rely on typical-case levels of voltage swing for the operating voltage margin, rather than on worst-case activity, as seen in a power virus. In the rare event of a voltage 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 [Bowman et al. 2008]. Abundant recent work in architecture [de Kruijf et al. 2010; Ernst et al. 2003; Gupta et al. 2008; Greskamp et al. 2009; Powell et al. 2009] and more recent circuit prototyping efforts [Bowman et al. 2008; Bull et al. 2009; Tschanz et al. 2009] 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 and caveats of typical-case design. Using only off-the-shelf components to sense on-die silicon voltage fluctuations of an Core™2 Duo processor, we perform full-length program analysis, characterizing voltage noise in this production processor, both droops below and overshoots above the nominal supply voltage. 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

[1]Based on simulations of a Pentium 4 power delivery package [Gupta et al. 2007], assuming Vdd gradually scales according to ITRS projections from 1V in 45nm to 0.6V in 11nm [International Technology Roadmap for Semiconductors 2007]. 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 [Zhao et al. 2006] technology nodes.

die area and cost overheads, in addition to complicating design, testing and validation. In order to alleviate this complexity, we propose software-level solutions, which allow designers to leverage more coarse-grained, cost-effective recovery schemes. Software, such as a voltage-noise-aware compiler, or an operating system thread scheduler, enables this by reducing error-recovery rates, while assuming the hardware provides a fail-stop. To study the efficacy of high-level solutions in the anticipation of large voltage swings, we project future voltage noise trends by reducing decoupling capacitance of an existing processor.

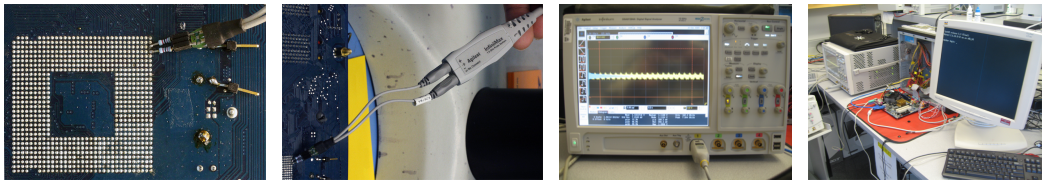
Developing a software solution for mitigating voltage noise begins with understanding activity within the processor that leads to voltage swings. Studying voltage noise in a real processor enables some observations not revealed by published simulation efforts. Using microbenchmarks that stimulate the 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.

Observing voltage noise in a production processor allows us to reveal new findings previously unknown from prior simulator effort. It also helps us verify and validate prior work. For instance, we show measurement-based correlation of voltage droop activity to microarchitectural stalls and processor throughput. We also show how independent microarchitectural activity across separate physical cores leads to inter-thread *interference* that causes peak-to-peak voltage swings to amplify in multi-core chips. Building upon this characterization, we demonstrate the existence of *voltage noise phases*. Similar to program execution phases, programs go through varying levels of recurring voltage droop and overshoot activity. We also quantify the impact of compiler code optimizations on voltage droop counts and droop magnitude. Our measured results indicate that aggressive compiler optimizations can lead to more voltage noise.

Out of the phenomena we observe, voltage noise interference is especially daunting. 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.

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.* We validate prior simulation-based work that uses compiler optimizations to explicitly decrease voltage noise on a single processor core. Furthermore, we propose, investigate, and demonstrate the benefits of a



(a) (b) (c) (d)

Fig. 3: Measurement setup to sense and measure voltage fluctuations within the processor at execution time: (a) Connecting to on-die voltage pins via a low impedance path using VCC_{sense} and VSS_{sense}. (b) Sensing voltage using an InfiniiMax 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.

noise-aware thread scheduler that, by taking advantage of voltage noise phases, smooths 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.

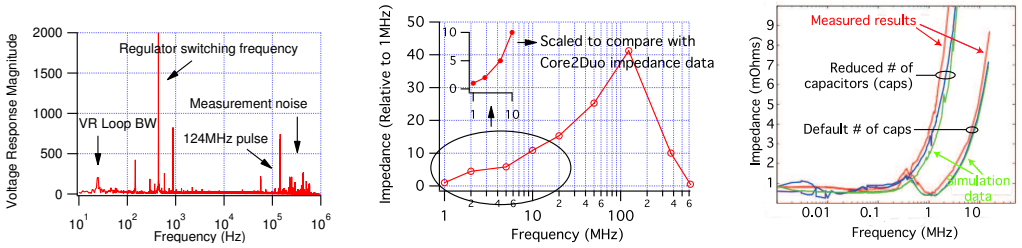
Sec. 2 explains how we sense on-die voltage in a production processor as it is operating in a regular environment. We use this setup to study resilient architecture design under typical-case conditions in Sec. 3. The challenges we identify lead us towards an understanding of how activity within the microprocessor causes voltage to fluctuate (Sec. 4). That in turn motivates us to evaluate software techniques in Sec. 5 as a means of diminishing the causes of emergencies altogether. Finally, we conclude the paper with our remarks in Sec. 6.

2. 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 [Intel 2006; Chickamenahalli et al. 2005], 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.

2.1. Using Off-the-Shelf Components

Previous descriptions of voltage swings have been done primarily by using either custom voltage transient test (VTT) tool kits [web b; Intel 2006] or simulation [Gupta et al. 2008; Gupta et al. 2007; Gupta et al. 2009; Powell et al. 2003; Powell et al. 2004]. 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 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 [Rahal-Arabi et al. 2002; Aygun et al. 2005; Joseph et al.



(a) FFT of voltage response. (b) Measured impedance. (c) Intel impedance data.

Fig. 4: (a) Measurement validation, (b)-(c) comparing impedance derived on our machine versus well established data from Intel.

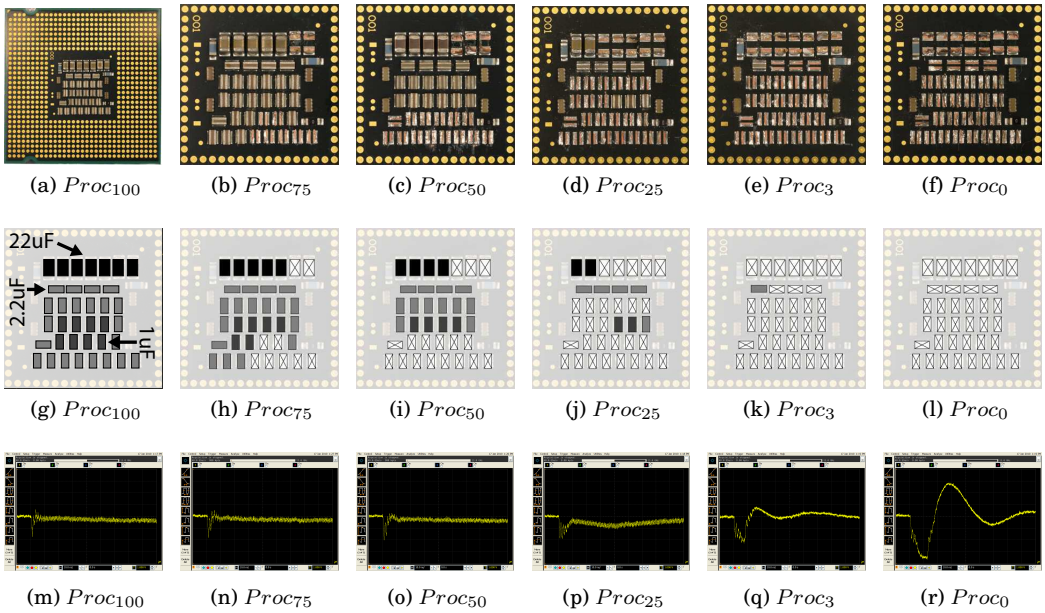
2003] with microarchitecture and power consumption models. Simulation has been the primary vehicle for voltage noise research over the past several years. However, analysis via simulation suffers from constraints like the length of program execution one can study, or the extent to which the models are 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 [Intel 2008].

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 an InfiniiMax 1.5GHz 1130A 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 [Chick-



(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.

amenahalli et al. 2005]. 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 [Chickamenahalli et al. 2005]. Briefly, the processor is subject to square wave current patterns at unique frequencies. For a current stimulus at a particular frequency, the processor experiences a voltage swing of certain magnitude corresponding to the impedance of the system. At this point, we capture a voltage trace. We convert this time domain trace into frequency domain using FFT. Fig. 4a is an example of the frequency domain representation of processor voltage under a 124MHz current pulse stimulus. The figure also shows other components, like VR Loop BW, Regulator switching frequency and background Measurement noise. From this, we observe the magnitude of the frequency component at 124MHz. Since we do not know the amplitude of the current pulse, we cannot determine the absolute impedance value at this frequency. Therefore, we plot values relative to an arbitrary baseline at 1MHz. The complete impedance profile is constructed by sweeping current over a range of frequencies, repeating the above process for each step.

Prior effort requires the use of custom hardware to create the current stimulus [Waizman 2003]. In our effort, we create the current stimuli using software. A user-level assembly program alternates between high power and low power instruction sequences repeatedly in a loop. Instructions for the high power sequence are carefully crafted by hand to ensure high throughput and steady maximum power consumption.

Similar instruction sequences are found in power virus kernels like CPUBurn [Mienik]. The low power sequence consists of pseudo no-operation instructions. The duty cycle alternating between these two extreme current consumption paths is a command line argument to our program. By modulating current activity in this way, we create varying current stimuli at different frequencies. In this way, we continue to use off-the-shelf components.

Fig. 4b is the impedance profile constructed using this approach on our system. There are two important validation 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 [James et al. 2007; Gupta et al. 2008; Aygun et al. 2004; Waizman 2003; Gupta et al. 2007]. Second, the small graph embedded within Fig. 4b corresponds appropriately to previously published Intel data [Intel 2006; Chickamenahalli et al. 2005]. Between 1MHz and 10MHz, Measured results for the Default # of caps in Fig. 4c 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.

2.2. 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. 4c 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 [Mienik]. This program stresses the processor's execution units while continuously monitoring execution for errors.

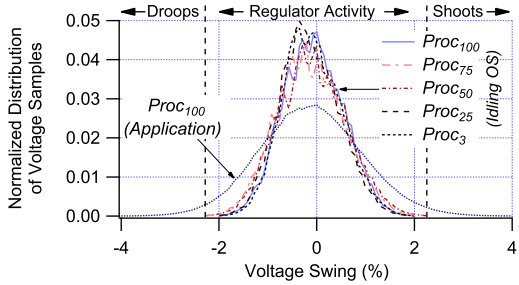


Fig. 6: Histogram of idle noise activity on processors shown in Figs. 5a-5f.

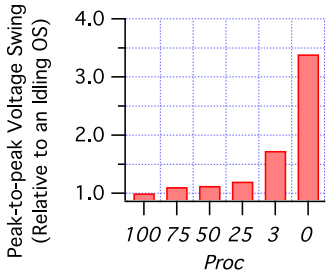


Fig. 7: Voltage swings after reset signal in Figs. 5m-5r.

The processor package contains different capacitive elements. 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 under CPUBurn.

Even though stimulating the different processors with a reset signal triggers very different voltage responses, their idle voltage behavior is very consistent. Fig. 6 shows the distribution of voltage samples of the five functional processors running the operating system idle loop (Idling OS). The differences from $Proc_{100}$ through $Proc_3$ are minimal, especially when compared to a sample chip that is running an application. This proves that removing the package capacitors does not introduce any steady-state error and only amplifies response to transients.

Because steady-state behavior is consistent among the chips, we can safely normalize the peak-to-peak voltage swings after the reset signal. Fig. 7 summarizes these resulting swings across all processors relative to $Proc_{100}$. The trend in this figure is roughly the same as in Fig. 1. The knee of the curve is around $Proc_{25}$ and $Proc_3$. The worst-case swing for them corresponds to simulation data from Fig. 1 for 32nm and 22nm nodes, so from here on we rely on them as our future nodes, while $Proc_{100}$ is representative of today's systems.

2.3. Worst-Case Margin

The worst-case margin is the voltage guardband that tolerates transient voltage swings. It is ($V_{nominal}$-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.

Note that this is a conservative estimate. In reality, the test chip fails when the current, not the nominal voltage reaches the worst-case guardband, so the margin is most likely significantly larger. However, this experiment was performed on a different setup and precisely measuring the minimum runtime voltage was not possible. Therefore, from here on, we assume a 14% margin.

2.4. Limitations and Assumptions

The experimental methodology cannot differentiate voltage droops or overshoots at a per-core level. The Core^{TM}2 Duo contains two physical cores in one processor package and the motherboard supplies power to both processing units together using a single voltage regulator module. Therefore, any transient voltage droop across the supply plane affects both cores.

Additionally, our measurement setup is capable of only detecting voltage dips below a certain threshold (i.e., a hypothetical operating voltage margin). The setup cannot determine the absolute magnitude of time voltage was below this hypothetical margin. Since the duration of voltage droop affects whether a timing violation manifests or not, we cannot determine the "true" or effective number of margin violations that lead to timing errors. Instead, we pessimistically assume that every voltage droop leads to a timing violation. This assumption is a tradeoff between doing full program voltage noise analysis on production systems versus using detailed simulations that are slow and provide very limited insight.

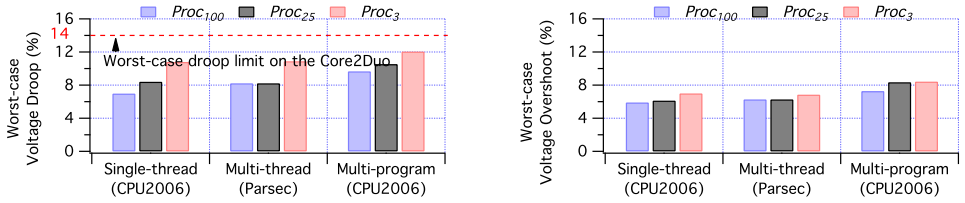
3. NOISE IN PRODUCTION PROCESSORS

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 2. 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.

The benchmark suite that we use consists of 881 benchmarking runs. The experiments include a spectrum of workload characteristics: 29 single-threaded SPEC CPU2006 workloads, 11 Parsec [Bienia et al. 2008] 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.

3.1. Worst-Case Noise Behavior

We first examine the worst-case voltage droops and overshoots of the different classes of workloads in our suite. Worst-case voltage change is a relevant metric, since current architectures must tolerate the worst amount of noise in order to run error-free. We notice that voltage noise is a growing problem in multi-core systems, more so than



(a) Droop summary.

(b) Overshoot summary.

Fig. 8: Dynamic worst-case voltage droop and overshoot corresponding to different workloads.

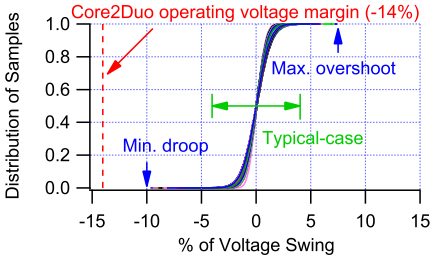


Fig. 9: Cumulative distribution of voltage samples across 881 program executions on $Proc_{100}$.

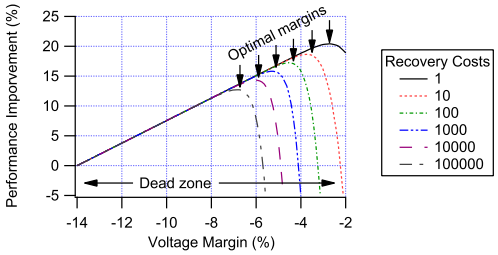


Fig. 10: Typical case improvement across a range of recovery costs on $Proc_{100}$, showing substantial room for tighter voltage margins.

in single-core systems. Relative to single core execution, activity on two cores leads to a larger worst-case overshoot, as well as droop. Multi-program and Multi-thread execution in Fig. 8a and Fig. 8b correspond to execution activity on both cores versus Single-thread execution where the other core of our Core^{TM}2 Duo is idling. This trend is consistent across all three processor categories, $Proc_{100}$, $Proc_{25}$ and $Proc_3$.

Moreover, independent switching activity across cores translates to larger voltage swings than coordinated and synchronized program execution. Fig. 8 shows that Multi-thread execution experiences smaller worst-case dynamic swings compared to Multi-program execution. Multi-threaded execution is more coordinated and synchronized because of producer-consumer like relationships. Multi-program execution is more disjoint and program activity on one core is decoupled from activity on another core. As the industry continues to increase core count per chip, we anticipate larger worst-case voltage swings due to this effect.

Comparing Fig. 8a and Fig. 8b shows that overshoots present a less severe issue than droops. In themselves, both types of events can cause serious correctness problems – droops can cause timing violations, while overshoots strongly accelerate device aging. However, Fig. 8 suggests that for current nodes the worst-case overshoot is significantly smaller than the worst-case droop regardless of the workload. More importantly, the amount of overshoots grows at a slower rate, as we use $Proc_{25}$ and $Proc_3$ to project voltage noise in the future. For this reason, we will focus our subsequent analysis on droops.

3.2. A Case for Typical-Case Operation

The worst-case operating voltage margin is overly conservative. We determine this from the full set of 881 benchmark runs. Fig. 9 shows a cumulative histogram distri-

bution 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, such large droops occur very infrequently. Most of the voltage samples are within 4% of the nominal voltage. The Typical-case marker in Fig. 9 identifies this range. Only a small fraction of samples (0.06%) lie beyond this typical-case region.

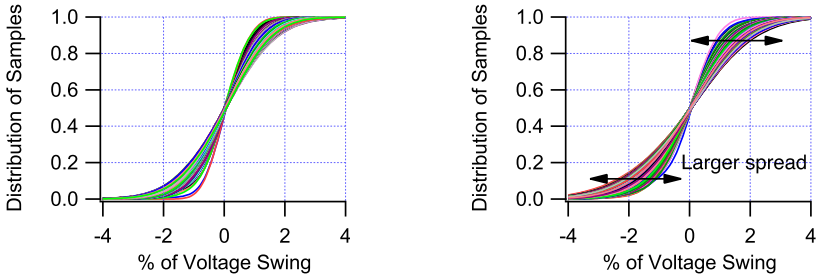
The findings indicate that designing the processor for the infrequent worst-case voltage swing is a bad choice. If alternative strategies are not taken, the industry must continue to expand the worst-case margin even further as voltage swings get larger, which translates to lower performance and a loss in power efficiency.

In an alternative design, when a microarchitecture is optimized for typical-case conditions, it relies on an error-recovery mechanism to handle emergencies. In these case, 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 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 on 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 [web a]. The combined number is the performance lost by the program for the particular recovery cost.

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 [Bowman et al. 2008]. 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. 10 shows performance improvement over a range of voltage margins, assuming specific recovery costs. These recovery costs reflect prior work: Razor [Ernst et al. 2003], a very fine-grained pipeline stage-level error detection and recovery mechanism, has a recovery penalty of only a few clock cycles. DeCoR [Gupta et al. 2008] 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. [Reddi et al. 2009] 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 [Slegel et al. 1999]. 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 [Shanbhag et al. 2010]. Our workloads do not fall into this criteria, therefore we target the more general case where hardware robustness is a must.



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

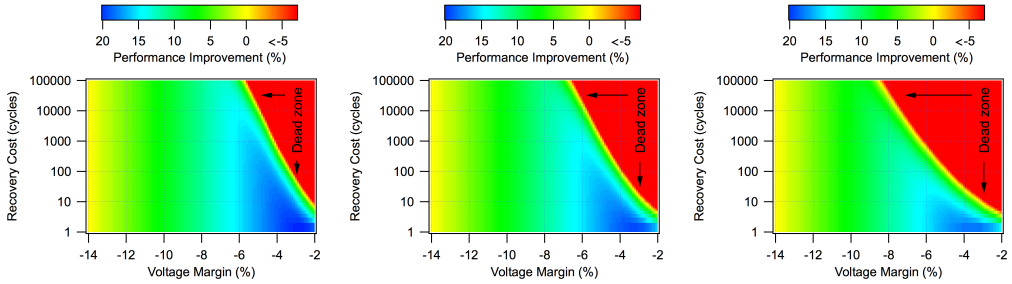
Fig. 11: Typical-case swings are increasingly more slanted than in $Proc_{100}$ (see Fig. 9), since voltage swings will be larger in the future.

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. 10 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. 10, every recovery mechanism has its own optimal margin. Depending on the cost of the recovery mechanism, gains vary between 13% and $\sim$21%. Coarser-grained recovery mechanisms have more relaxed optimal margins while finer-grained schemes have more aggressive margins and as a consequence are able to experience better performance improvements. However, being overly aggressive and setting the operating voltage margin beyond the optimal causes rapid performance degradation. At such settings, recoveries occur too frequently and penalize performance, thus the benefits begin to diminish. Recovery penalties can be so high that they can even push losses beyond the original conservative design (i.e., 14% margin on Core$^{\text{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. 11 shows the distribution of 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 (0% of Voltage Swing) 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.



(a) $Proc_{100}$ (b) $Proc_{25}$ (c) $Proc_{3}$

Fig. 12: Performance improvement under typical-case design using various recovery costs and voltage margin settings.

To quantify the impact and to illustrate diminishing gains in performance better, we rely on the heatmaps in Fig. 12. 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. 12a with that in Fig. 12b and Fig. 12c, 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. 12, 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 $\sim$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 [Slegel et al. 1999; Ando et al. 2003] 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. [Wang et al. 2006; Narayanasamy et al. 2005]. In this way, software aids efficient and cost-effective typical case design.

4. 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$^{\text{TM}}$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.

Microbenchmarks	Legend	Description

L1 Cache Miss	L1	Generates L1 cache misses using back-to-back-loads and pointer-chasing. This serializes processor execution by repeatedly accessing cache lines that always miss in the L1 cache, but hit in the L2 cache. Pre-initializes an array larger than the cache to form a circular linked list of memory addresses, such that accessing one memory location yields the next memory location to load from. The offset between subsequent accesses determines the frequency of cache misses, and thus ideally matches the cache's line size. Such back-to-back loading of memory locations serializes execution and renders out-of-order execution and memory disambiguation ineffective, thus allowing clean noise measurement of individual cache miss events:
		`register char **ptr = &array[0];`
		`while (*ptr != NULL)`
		`  ptr = (char **)*ptr;`

L2 Cache Miss	L2	Uses the same back-to-back loading technique that the L1 cache miss microbenchmark uses to force second level cache misses. The array size and memory access offset are adjusted to the second level cache's size and its corresponding line size, respectively. Alternatively, certain x86 processors support the `CLFLUSH` instruction. However, this does not guarantee serialization.

TLB Invalidation	TLB	Re-loads the `cr3` register to purge the virtual to physical address mappings from the TLB:
		`asm ("movl %%cr3, %0;`
		`  movl %0, %%cr3;`
		`  : "=r" (tmpreg) :: "memory");`
		An alternative approach is to use the `INVLD` instruction to purge the entry corresponding to the page containing the instruction. Re-executing the instruction forces a stall as a result of the TLB miss.

Hardware Exception	EXCP	Executes an illegal instruction to cause an exception. The following sequence causes the machine to raise an illegal instruction exception when executed:
		`unsigned char insn[4] = {0xff, 0xff, 0xff, 0xff};`
		`void (*func)(void) = (void (*)(void)) insn;`
		`func();`

Branch Misprediction	BR	An if-then-else statement whose condition variable depends upon the output of a randomization function (e.g., `rand()` from standard `C` library) to thwart the branch predictor from successfully knowing the outcome of a branch, thus forcing pipeline flushes. To avoid function call perturbations during measurement, we pre-initialize an array that is larger than the predictor's history register with values from `rand()`. As the static recurring pattern does not fit in the history register, performance counter data indicates 50.59% branch misprediction on the if-then edge and the remainder on the else path.
		`if (array[i] & 0x1)`
		`  ...`
		`else`
		`  ...`

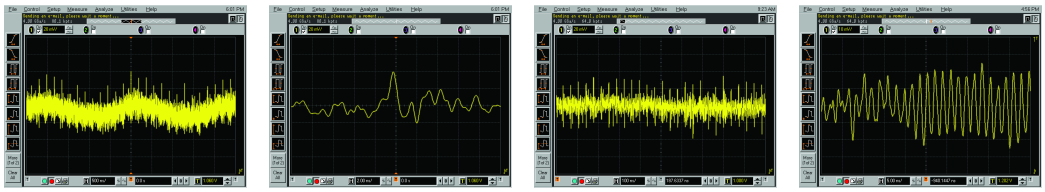
Table I: Descriptions for the microbenchmarks we use to test stall effects on voltage noise.

4.1. Single Core Events

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). Tab. I provides a detailed description of how we construct these microbenchmarks. Each of them 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. 13a 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 (Fig. 13b).

The processor may also experience a correspondingly strong voltage droop following an initial overshoot. Consider an L1 cache miss event, as seen in Fig. 13c. During



(a) Voltage overshoots due to a TLB miss on a single core while the other core is idle.

(b) Zoomed in version of one of the TLB overshoot spikes in subfigure (a).

(c) Overshoot followed by a successive droop due to consecutive L1 misses on one core.

(d) Cache accesses interference across cores causes voltage to resonate.

Fig. 13: $Proc_{100}$ oscilloscope screenshots showing microarchitectural event effects on core voltage.

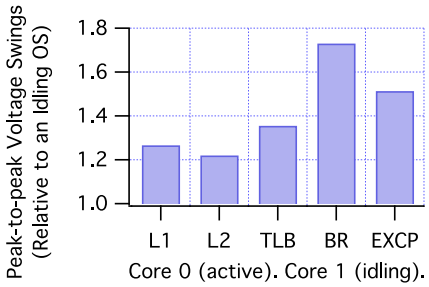


Fig. 14: Effect of microarchitectural events on supply voltage.

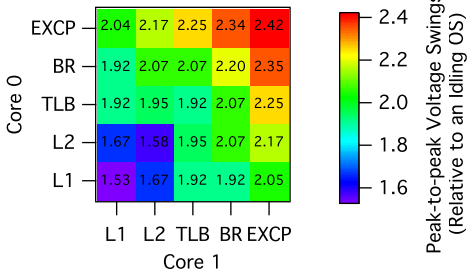


Fig. 15: Impact of microarchitectural event interference across cores.

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.

Activity across two cores can also lead to problems. Fig. 13d shows how interfering cache access patterns across cores cause voltage to droop and shoot repeatedly. We re-visit inter-core interference in more depth in the next section.

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. 14. On an idling system, we observe only the VRM ripple. Therefore, voltage overshoots and droops are distinctly noticeable and measurable in microbenchmark cases. Fig. 14 shows that branch mispredictions 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.

4.2. Multi-Core Event Interference

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. 15 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. 15 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. 5.1 shows an example of destructive interference, where the swing when two cores are simultaneously active is smaller than during single core execution.

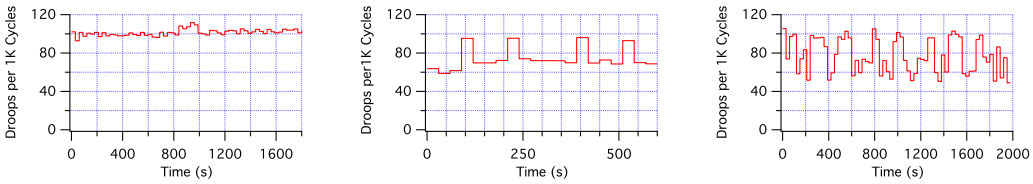
Constructive interference is a problem in multi-core systems, since individual cores within the processor typically share a single power supply source.[3] Therefore, a transient voltage droop anywhere on the shared power grid could inadvertently affect all cores. If the droop is sufficient to cause an 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.

5. MITIGATING VOLTAGE NOISE

In order to smooth voltage noise in resilient architectures, this section investigates the potential for software solutions. Our techniques are 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. We look into leveraging existing compiler optimization levels and tuning their aggressiveness with respect to noise behavior. In a multi-core scenario, where noise interference can further exacerbate performance, we investigate a thread scheduler that is explicitly noise-aware. Note that both solutions are complementary, not a replacement/alternative for hardware. Due to the lack of existing checkpoint recovery/rollback hardware, we analytically model and investigate any performance estimates, as well as any policies that rely on a feedback path from recovery-enabled hardware.

We first introduce and discuss the existence of *voltage noise phases* in order to motivate that software solutions can be applied to voltage noise. In the single-core case, we further examine the connection between microarchitectural stalls, performance and noise behavior. We show that more aggressive compiler optimizations can often lead to a larger amount of noise, which triggers recovery more frequently and ends up hurting final performance. In the multi-core scenario, we motivate a scheduling policy called

[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 [Kim et al. 2007]. 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 [James et al. 2007].



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

Fig. 16: 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.

Droop. It focuses on co-scheduling threads across cores to minimize chip-wide droops. We then demonstrate that thread scheduling for voltage noise is different to scheduling threads for performance. Finally, we show 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.

5.1. 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, Fig. 16 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 (see Fig. 6). Thus, it allows us to cleanly eliminate background operating system effects 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. 16c, 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. Real programs can be stalled not only by explicit pipeline stalls (as we have seen in Sec. 4), but also by more complicated and implicit stalls, such as those caused by data dependencies. For instance, instead of looking only at isolated events that stall the front-end of the pipeline (e.g., branch mispredictions), we must look all activity that leads to front-end stalls.

To better understand the relationship between processor resource utilization and voltage noise, we use a metric called *stall ratio*. 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. We track such activity using VTune. The counters we analyze include the following set: RESOURCE_STALLS.BR_MISS_CLEAR, RESOURCE_STALLS.FPCW, RESOURCE_STALLS.LD_ST, RESOURCE_STALLS.ROB_FULL, and lastly RESOURCE_STALLS.RS_FULL. Tab. II provides a brief summary of these event coun-

Hardware Performance Counter	Stall Description
RESOURCE_STALLS.BR_MISS_CLEAR	Cycles stalled due to branch misprediction.
RESOURCE_STALLS.FPCW	FPU control word write stall cycles. Modifications to the Floating-Point Control Word (FPCW) might cause stalls.
RESOURCE_STALLS.LD_ST	Load/Store unit stall cycles. Cycles during which the pipeline has exceeded load or store limit or it is waiting to commit all pending stores. This prevents new micro-ops from entering the execution pipeline.
RESOURCE_STALLS.ROB_FULL	Cycles during which the ROB is blocked. New instructions can not enter the pipe and start execution. This may happen due to long latency operations in the pipe, possibly load and store operations that miss the last-level cache.
RESOURCE_STALLS.RS_FULL	Reservation station units stall cycles. This occurs when micro-ops that get into the RS cannot execute because they are waiting for their operands to be computed by previous micro-ops.

Table II: Breakdown of resource related stall counters that define our metric *stall ratio*.

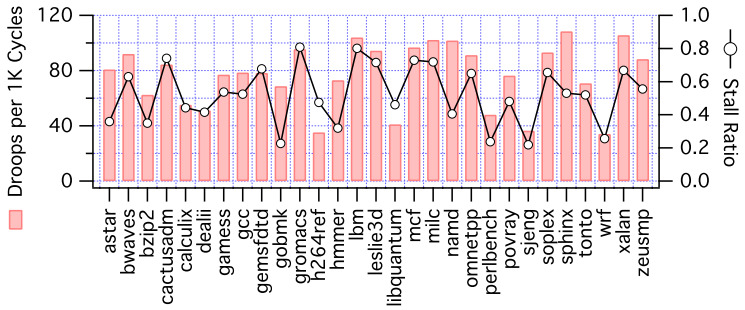


Fig. 17: Single-core droop activity, showing a heterogeneous mix of noise levels along with correlation to stalls.

ters. The VTune manual provides more in-depth explanations for each counter. The stall ratio is the sum of all these counters over the total processor clock cycle time for a particular execution window. We gather raw counter data using VTune and construct the stall ratio off line.

Fig. 17 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.

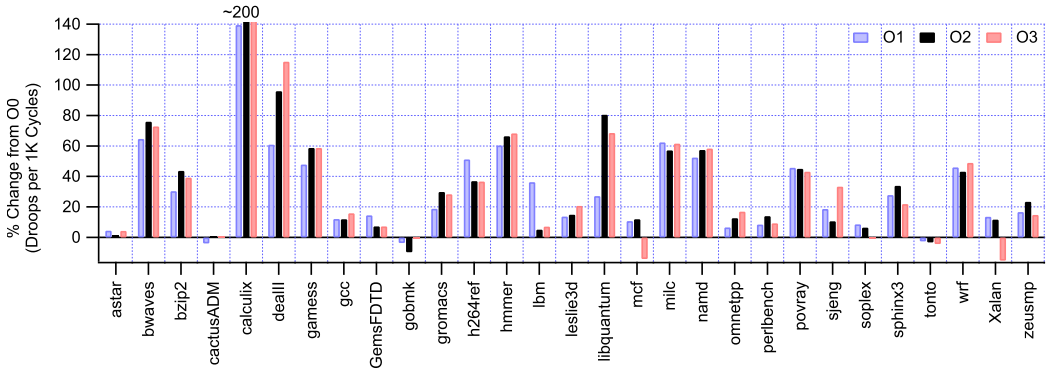


Fig. 18: Noise performance on different degrees of compiler optimizations.

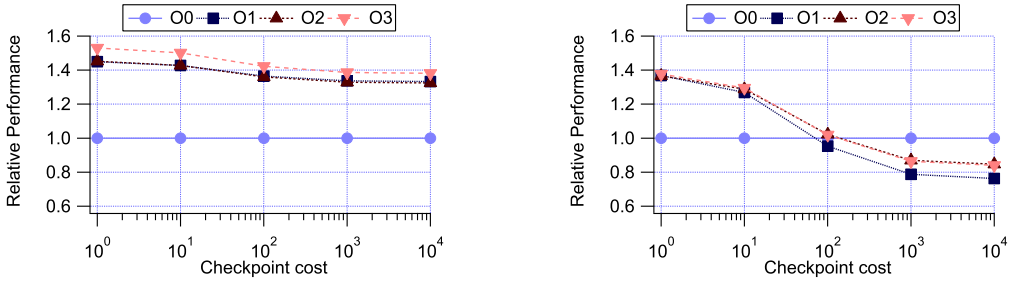
5.2. Compiler Influence on Voltage Noise

The analysis so far strongly suggests a correlation between voltage noise and microarchitectural stalls. This can lead to the false assumption of direct causality – namely, that eliminating stalls directly decreases voltage noise. In order to show why the connection is more complex than that, we evaluated the effects of compiler optimizations on voltage noise. Compiler optimized code experiences a greater number of voltage droops, and in certain cases the magnitude of the droops is also noticeably larger.

Impact on Droop Counts. Typically, the task of an optimizing compiler is to increase instruction throughput through the processor. A large body of well-known optimizations (such as loop unrolling, instruction scheduling, register allocation) achieve that by eliminating various microarchitectural stalls. Thus, if the analysis in Fig. 17 was interpreted as a simple and direct causality relation between the number of stalls and the number of voltage droops, one would expect that higher compiler optimization levels would decrease the amount of voltage noise, while increasing performance.

Data that we gathered for the GCC compiler contradicts with such a notion. Fig. 18 shows the noise behavior of the single-core CPU2006 benchmarks, when compiled with optimization levels ranging from O0 to O3. We can see that increasing the aggressiveness of performance optimization with respect to the O0 baseline leads to a larger number of droops per 1K cycles in the majority of benchmarks. Out of the 29 runs in this experiment, 19 binaries compiled with maximum optimization result in a more than 10% increase in the number of droops, compared to the respective non-optimized versions. *454.calculix* shows the largest increase – at O3 its droop counts more than triple. The fluctuations for the other 10 benchmarks in the experiment are predominantly smaller.

The behavior of the majority of the benchmarks is easily explicable. When better optimized at O3, benchmarks exhibit a higher number of instructions per cycle. At the microarchitectural level, this implies that pipeline utilization is high, and consequently switching factors are larger, therefore the core consumes a relatively larger amount of current. On a stall, the net change in current is larger than in the unoptimized case. Since voltage fluctuations are proportional to such changes in current, each stall is more likely to cause a subsequent voltage droop. This effect leads to a larger aggregate number of droops over the whole execution, even though the number of stalls may be smaller.



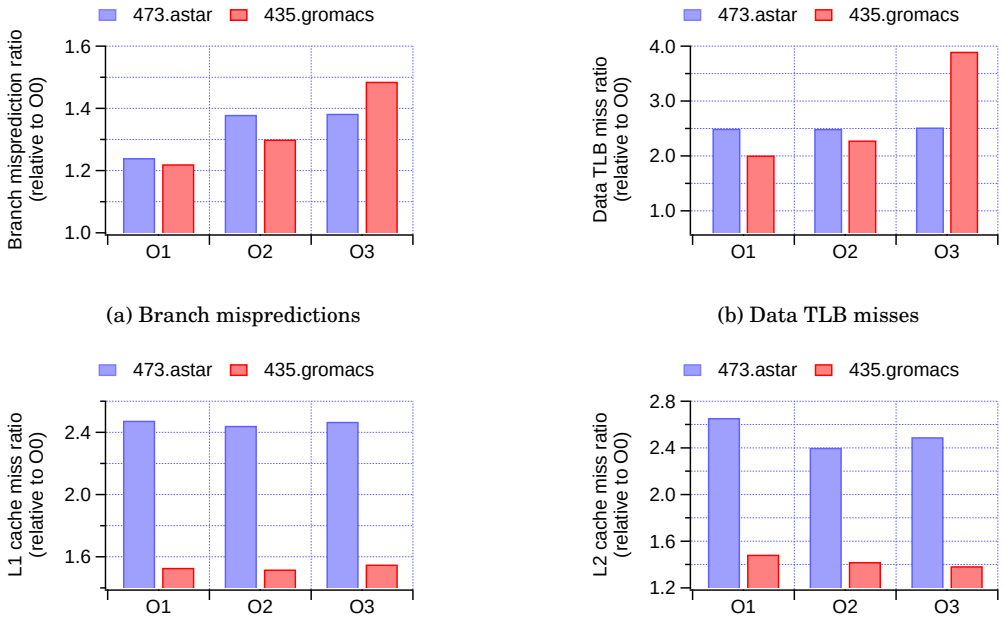
(a) *473.astar* (b) *435.gromacs*

Fig. 19: Influence of compiler optimizations on performance for varying recovery costs, normalized to O0.

Impact on Net Performance. In a resilient design, the increased number of droops at higher optimization levels has a respective performance penalty. As a proof-of-concept that this penalty may be sufficiently large to even offset the initial performance gains from optimizing more aggressively, we analyze the net performance of one benchmark from each of the two groups outlined above (those that experience little change in their noise profiles, represented by *473.astar*, and those that have a significant increase, represented by *435.gromacs*). We account for the recovery cost of each voltage emergency below the hypothetical 2.3% margin using the performance model described in Sec. 3, expanded to include IPC differences among the differently optimized binaries.

Fig. 19 shows performance improvement achieved by these benchmarks for different recovery mechanism costs. Both benchmarks rightfully receive a significant performance gain from higher levels of performance-centric compiler optimizations. For *473.astar*, this gain is sufficient to sustain higher net performance even at very large recovery costs. Even though the gains diminish because of the slightly increased droop counts (and therefore emergency recoveries), in this case net performance is dominated by factors other than noise. For workloads represented by *435.gromacs*, fine-grained recovery presents similar results. However, after a certain recovery cost (100 cycles in this particular case), voltage noise effects begin to dominate over the initial performance gains and less optimized binaries achieve better net performance, after factoring in emergency recovery penalties. Even the modest 30% increase in relative droop counts that *435.gromacs* shows is sufficient to offset the 50% initial performance gains from compiling with O3.

In order to better understand this performance loss, we quantify the differences in behavior between the benchmarks by examining their performance counter data for the different degrees of optimization. In Sec. 4 we identified branch mispredictions as the cause of the largest number of voltage droops and this is the metric we examine first in Fig. 20a. All data in the figure is relative to the case with no optimizations (O0) and optimization aggressiveness grows to the right. For both benchmarks, higher levels of optimization lead to an increasingly higher branch misprediction ratio. Eventually, at O3, *435.gromacs* achieves a higher misprediction ratio than *473.astar*. A similar trend holds for the other noise-critical event identified in Sec. 4 – TLB misses, which we show in Fig. 20b. On the other hand, both L1 and L2 miss ratios increase significantly more for *473.astar* than for *435.gromacs* (Figs. 20c-20d). Since *435.gromacs* indeed shows a larger growth in droop activity, this performance counter evaluation confirms that branch mispredictions and TLB misses are more tightly correlated with voltage droops than cache miss events.



(a) Branch mispredictions

(b) Data TLB misses

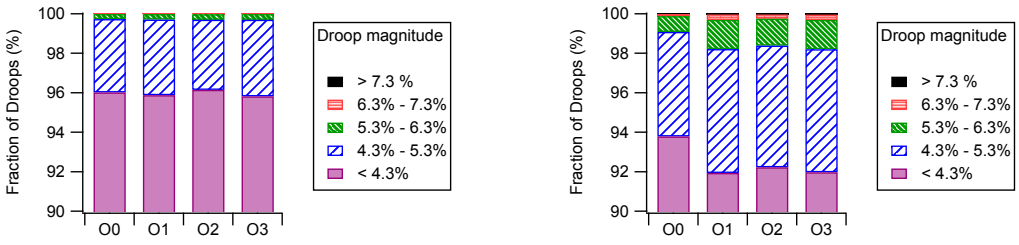
(c) L1 cache misses

(d) L2 cache misses

Fig. 20: Performance counter data for *473.astar* and *435.gromacs* under different optimization levels.

Impact on Droop Magnitude. The magnitude of single droops can also be influenced by varying compiler optimizations. While droop magnitude is a second order effect that our performance model does not include, larger droops require more time for voltage to stabilize (Figs. 5m-5r), which can potentially limit the minimum cycle cost for emergency recovery. We quantify the distribution of droop magnitudes for our two representative benchmarks. Fig. 21 shows the top 10% of droops, broken down by the magnitude of voltage swing. The first benchmark, *473.astar*, does not show a significant change in droop magnitude across increasing compiler optimizations. This behavior is similar to the total number of droops seen in Fig. 18. For benchmark *435.gromacs*, the relative severity of droop magnitude increases (see Fig. 21b), despite the already larger number of total droop count. This can also be correlated with the fact that *435.gromacs* shows a higher number of noise- critical miss events per cycle.

The overall conclusion from analyzing the effect of compiler optimizations is that higher levels of optimization amplify the amount of noise that a benchmark exhibits. An optimal point can be achieved by utilizing a less optimized binary on a resilient architecture. However, constructing a coherent policy for finding this optimal requires either extensive and non-trivial knowledge about the impact of various optimizations on both performance- and noise-critical microarchitectural events, something beyond the scope of this paper; or a runtime component that dynamically adapts compiler optimizations to a changing noise profile, a working prototype of which has already been demonstrated [Reddi et al. 2009].



(a) *473.astar* (b) *435.gromacs*

Fig. 21: Influence of compiler optimizations on the magnitude of droops for benchmarks discussed in Fig. 19.

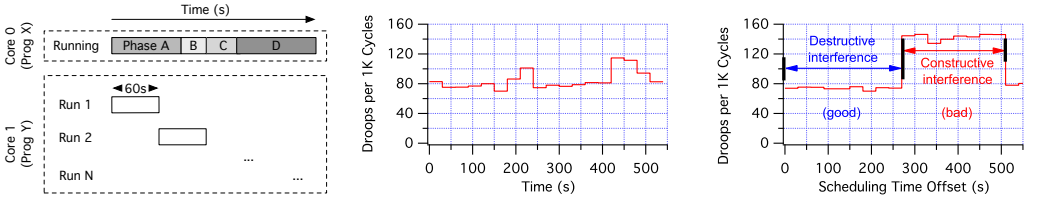
5.3. Co-Scheduling of Noise Phases

Compiler-inspired techniques are useful in a single-core scenario, but it is typically not the compiler's responsibility to handle inter-core interactions. The correlation we demonstrated earlier in Sec. 5.1 between coarse-grained performance counter data (on the order of billions of instructions) and very fine-grained voltage noise measurements implies that higher latency software solutions are applicable to mitigate voltage noise.

We believe designing effective mitigation solutions will be much easier at the operating system-level software layer. At this level the software has a more global view of execution activity across the whole processor. It could leverage recurring voltage noise phases to make it conducive for software to amortize its overheads. Additionally, it could also re-use past activity records to influence and make better decisions in the future. Designing such techniques at the hardware layer is not trivial. Orchestrating hardware-level solutions when there are multiple cores in the chip can be extremely challenging because of the granularity at which voltage noise occurs, which is in the order of a few ten's of clock cycles. One core must know what microarchitectural events are occurring on neighboring cores prior to deducing the possibility of an emergency.

A runtime thread scheduler can mitigate voltage noise by combining different noise phases together in a multicore environment. 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. 22a. 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. 22a. 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. 22b 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. 22c). During this time frame, droop count nearly doubles from around 80 to 160 per 1000



(a) Setup for co-scheduling experiment.

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

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

Fig. 22: (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).

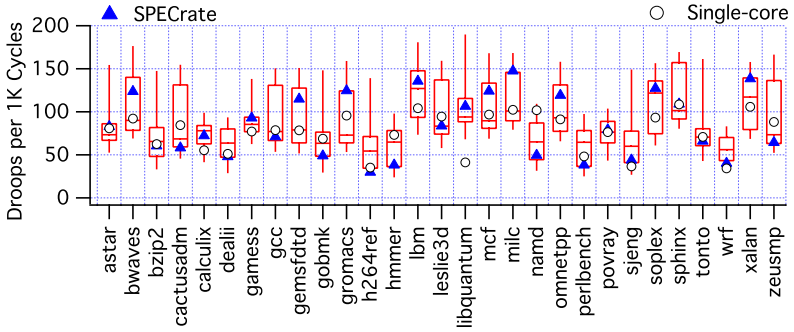


Fig. 23: Droop variance across single- and multi-core.

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. Fig. 23 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 *462.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. 23 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. 4 indicates that margins will need to grow.

5.4. 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 [Snavely et al. 2000; Fedorova 2006; Mars et al. 2010; Knauerhase et al. 2008; Zhuravlev et al. 2010; Chandra et al. 2005; Cazorla et al. 2006], 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 roll-back/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 from scheduling for performance. Scheduling for performance is independent of any noise-related parameters and typically involves improving miss rates or reducing cache stalls to increase performance in a non-resilient context. 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. 15 points out that the interactions between uncore (L2/L3 only) and other events can lead to varying magnitudes of voltage swings. Therefore, additional interactions must be taken into account.

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.

A specific and actual implementation is beyond the scope of this paper, as we are focused on investigating possibilities and setting up a framework for future direction. In reality, this policy would require a hardware feedback path, such as a dedicated performance counter that tracks the number of emergencies occurring during each scheduling interval. Alternatively, a heuristic estimate as the stall ratio could be used. The scheduler would utilize this information to dynamically detect, track and determine the set of threads that would run best together (i.e., cause minimum number of droops). One simple approach would be centered around continuous optimization, where the scheduler dynamically tracks destructive versus constructive interference as a set of co-scheduled threads are running, and uses such past activity records to influence and make better decisions in the future.

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. 23 shows no apparent preferential bias towards either minimizing or maximizing droops. Droop activity is spread uniformly over 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,

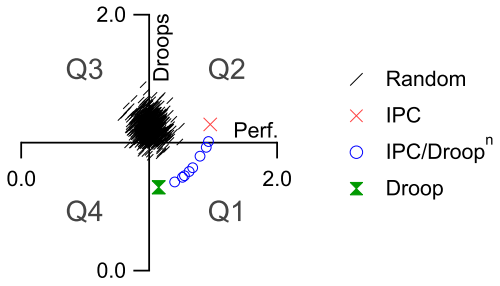


Fig. 24: Impact of scheduling policies on droop versus performance relative to SPECrate.

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 and an IPC/Droop^n technique that we explain further in section 5.5.

Due to the lack of dynamic feedback from hardware, which is necessary for Droop scheduling, we rely on an oracle-based algorithm, requiring knowledge of all runs a priori. During a pre-run phase we gather all the data necessary across 29$\times$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, in order to be fair, we simulate an oracle scheduler as well, using VTune's ratio feature to gather IPC data for each program combination.

Fig. 24 shows the results of our test. Each marker corresponds to one batch schedule. The four quadrants in Fig. 24 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.

5.5. Reducing Recovery Overheads

As voltage noise grows more severe, it will become harder for resilient systems to meet their typical-case design performance targets. Tab. III 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

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 III: SPECrate typical-case design analysis at optimal margins on $Proc_3$ processor.

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 and selecting the one that produces the largest improvement over the most benchmarks. This is representative of how a resilient architecture with a fixed margin would be designed.

However, as emergency frequency is strongly related to workload characteristics (see Sec. 5.1), some schedules continue to meet the expected performance (pass), while others do not (fail). When we constrain our analysis to the multi-program workload subset, and even more specifically only to SPECrate schedules, we find a diminishing number of all schedules pass at higher recovery costs. At 1-cycle recovery cost, nearly all, 28 out of all 29 SPECrate schedules, pass. But as recovery cost goes up beyond 10 cycles, the number of passing SPECrate schedules quickly diminishes to single digits.

Since the expected performance levels were derived from all possible schedules, this implies that, given enough combinations to choose from, a scheduler can do better than choosing the same benchmark for all cores (as in SPECrate) ,thus improving the number of schedules that pass the performance target. To demonstrate this, we run oracle-based Droop and IPC scheduling through the performance model (see Sec. 3.2). We then determine the number of schedules that pass across the benchmark suite. Fig. 25 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

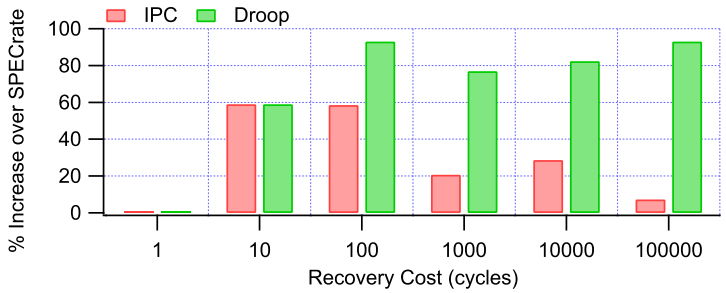


Fig. 25: Improvement over SPECrate schedules in Tab. III that fall below their expected target performance improvement.

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. 24 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.

6. 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. We demonstrate (via measurements) that the compiler can reduce these performance bottlenecks, by calibrating its optimization level according to the cost of error-recovery. We also show that increasing the number of cores per processor can make voltage noise worse. However, not all noise interference among cores is constructive, and voltage noise is not altogether irregular. Destructive interference can smooth supply voltage. Recurring microarchitectural stall behavior can produce periodic voltage noise phases. We exploit these phases and demonstrate (via simulation) a software thread scheduler for multi-core processors that mitigates error-recovery overheads by co-scheduling threads known to interfere destructively.

Acknowledgments

We thank our colleagues in industry and academia, specifically Glenn Holloway, for the many discussions that have contributed to this work. This work was funded by gifts from Intel, National Science Foundation grants CCF-0429782 and CSR-0720566, the Royal Academy of Engineering and EPSRC. Opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

http://software.intel.com/en-us/intel-vtune/.

http://www.cascadesystems.net/lga775.htm.

ANDO, H. ET AL. 2003. A 1.3 GHz fifth-generation SPARC64 microprocessor. In *DAC*.

AYGUN, K. ET AL. 2004. Measurement-to-modeling correlation of the power delivery network impedance of a microprocessor system. In *EPEPS*.

AYGUN, K. ET AL. 2005. Power delivery for high-performance microprocessors. *Intel Technology Journal*.

BIENIA, C. ET AL. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*.

BOWMAN, K. A. ET AL. 2008. Energy-efficient and metastability-immune timing-error detection and instruction replay-based recovery circuits for dynamic variation tolerance. In *ISSCC*.

BULL, D. ET AL. 2009. A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient- error tolerance and adaptation to PVT variation. In *ISSCC*.

CAZORLA, F. J. ET AL. 2006. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Trans. Comput.*.

CHANDRA, D. ET AL. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*.

CHICKAMENAHALLI, S., AYGUN, K., HILL, M., RADHAKRISHNAN, K., EILERT, K., AND STANFORD, E. 2005. Microprocessor platform impedance characterization using vtt tools. *Applied Power Electronics Conference and Exposition (APEC)*.

DE KRUIJF, M. ET AL. 2010. Relax: An architectural framework for software recovery of hardware faults. In *ISCA*.

ERNST, D. ET AL. 2003. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*.

FEDOROVA, A. 2006. Operating system scheduling for chip multithreaded processors. Ph.D. thesis. Adviser-Seltzer, Margo I.

GRESKAMP, B. ET AL. 2009. Blueshift: Designing processors for timing speculation from the ground up. In *HPCA*.

GUPTA, M. S. ET AL. 2007. Understanding voltage variations in chip multiprocessors using a distributed power-delivery network. In *DATE*.

GUPTA, M. S. ET AL. 2008. DeCoR: A Delayed Commit and Rollback Mechanism for Handling Inductive Noise in Processors. In *HPCA*.

GUPTA, M. S. ET AL. 2009. An event-guided approach to handling inductive noise in processors. In *DATE*.

INTEL. 2006. Voltage Regulator-Down (VRD) 11.0. *Processor Power Delivery Design Guidelines For Desktop LGA775 Socket*.

INTEL. 2008. Intel Core Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 and E4000 Series. *Datasheet*.

INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS. 2007. Process integration, devices and structures.

JAMES, N. ET AL. 2007. Comparison of split-versus connected-core supplies in the POWER6 microprocessor. In *ISSCC*.

JOSEPH, R. ET AL. 2003. Control techniques to eliminate voltage emergencies in high performance processors. In *HPCA*.

KIM, W. ET AL. 2007. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA*.

KNAUERHASE, R. ET AL. 2008. Using OS observations to improve performance in multicore systems. *IEEE Micro*.

MARS, J. ET AL. 2010. Contention aware execution: Online contention detection and response. In *CGO*.

MIENIK, M. http://users.bigpond.net.au/cpuburn.

NARAYANASAMY, S. ET AL. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*.

POWELL, M. ET AL. 2003. Pipeline muffling and a priori current ramping:architectural techniques to reduce high-frequency inductive noise. In *ISLPED*.

POWELL, M. ET AL. 2004. Exploiting resonant behavior to reduce inductive noise. In *ISCA*.

POWELL, M. D. ET AL. 2009. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *ISCA*.

RAHAL-ARABI, T. ET AL. 2002. Design and validation of the Pentium 3 and Pentium 4 processors power delivery. In *VLSI Circuits Digest of Technical Papers, 2002. Symposium on*.

REDDI, V., GUPTA, M., SMITH, M., WEI, G., BROOKS, D., AND CAMPANONI, S. 2009. Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack. In *Design Automation Conference, 2009. DAC'09. 46th ACM/IEEE*. IEEE, 788–793.

REDDI, V. J. ET AL. 2009. Voltage emergency prediction: A signature-based approach to reducing voltage emergencies. In *HPCA*.

SHANBHAG, N. R. ET AL. 2010. Stochastic computation. In *DAC*.

SLEGEL ET AL. 1999. IBM's S/390 G5 microprocessor design. *Micro, IEEE*.

SNAVELY, A. ET AL. 2000. Symbiotic job scheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*.

TSCHANZ, J. ET AL. 2009. A 45nm resilient and adaptive microprocessor core for dynamic variation tolerance. In *ISSCC*.

WAIZMAN, A. 2003. CPU power supply impedance profile measurement using fft and clock gating. In *EPEPS*.

WANG, N. J. ET AL. 2006. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Trans. Dependable Secur. Comput.*.

ZHAO, W. ET AL. 2006. Predictive technology model for sub-45nm early design exploration. *ACM JETC*.

ZHURAVLEV, S. ET AL. 2010. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*.