Voltage Noise: Why It's Bad, and What To Do About It

Vijay Janapa Reddi, Meeta S. Gupta, Krishna K. Rangan, Simone Campanoni, Glenn Holloway,
Michael D. Smith, Gu-Yeon Wei, David Brooks

School of Engineering and Applied Sciences, Harvard University

***Abstract*—Power constrained designs are becoming increasingly sensitive to supply voltage noise. We propose hardware-software collaboration to enable aggressive voltage margins: a fail-safe hardware mechanism tolerates margin violations in order to train a run-time software layer that reschedules instructions to avoid recurring violations. Additionally, the software controls an emergency signature-based predictor that throttles to suppress emergencies that code rescheduling cannot eliminate.**

I. Introduction

Continuing technology advances amplify the importance of reliability in modern high-performance processors. Shrinking feature size and diminishing supply voltage make circuits more sensitive to transient errors such as soft errors caused by radiation and to supply noise caused by current fluctuations.

While soft errors caused by alpha particles or energetic cosmic radiation are still relatively rare, voltage noise occurs more often because of interactions between the run-time behavior of code and characteristics of the underlying power delivery subsystem and microarchitecture. Parasitic inductance in the power system causes voltage to fluctuate as current draw changes with processor activity. A significant dip in supply voltage slows logic, which can cause timing margin violations. Significant voltage overshoots can cause long-term degradation in transistor characteristics. For reliable and correct operation of the processor, voltage swings that violate noise margins, which we call *voltage emergencies*, must be avoided.

The traditional way of dealing with voltage emergencies has been to over-design the system to accommodate worst-case voltage swings. A recent paper analyzing supply noise in a Power6 processor [1] shows the need for operating margins greater than 20% of the nominal voltage. Conservative processor designs with large margins ensure robustness. However, conservative designs either lower the operating frequency or sacrifice power efficiency.

Nominal supply voltage is gradually scaling down with newer technologies, but threshold voltage scaling has all but stopped. Consequently, circuit delay sensitivity to margins is increasing with each technology node. Figure 1 plots peak frequency at different voltage margins across four PTM [2] technology nodes (45nm, 32nm, 22nm, and 16nm) based on detailed circuit-level simulations of an 11-stage ring oscillator consisting of fanout-of-4 inverters. The plot shows that at today's 32nm node, a 20% voltage margin translates to a 33% frequency degradation, and at future technology nodes the situation gets much worse. Practical limitations on reducing power delivery impedance combined with large current fluctuations make margin-based solutions unsustainable.

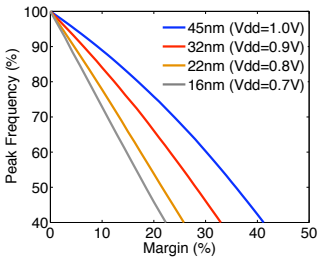


Fig. 1. Worst-case margins limit frequency, and the problem gets worse as technology trends scale.

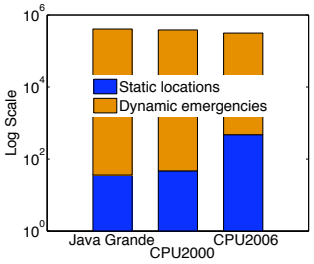


Fig. 2. Few hundred static program locations cause hundreds of thousands of emergencies.

In recent years, researchers have proposed tightening noise margins by adding mechanisms to the hardware that guarantee correctness in the presence of voltage emergencies. Such mechanisms enable more aggressive voltage margins, but at the expense of some run-time performance penalty. Architecture- and circuit-level techniques either proactively reduce activity when an emergency appears imminent [3]–[8], or they reactively restore a correct processor state after an emergency compromises execution correctness [9].

Those prior techniques do not take advantage of the repetitive nature of emergencies. Figure 2 shows the number of distinct instructions responsible for emergencies, and the total number of emergencies they cause over the lifetime of a program across three different benchmark suites. In each case, an average of only a few hundred instructions are responsible for hundreds of thousands of emergencies. So a few emergency-prone code regions are responsible for almost the whole emergency problem. Prior techniques must enable their mechanisms at least once per dynamic emergency. We, in turn, are using the fact that there are so few emergency-prone locations, and that emergency-prone behavior is so repetitive. By using the history of activity that leads to emergencies at these locations, we can both throttle more intelligently and even reshape code to eliminate recurring emergencies altogether.

The Alarms project at Harvard is working toward an architecture that can tolerate emergencies, and eliminate most recurring emergencies by exploiting patterns in emergency behavior of a running code. We envision a system implemented both in hardware and in software; the software component is at the level of a platform hypervisor. Figure 3 illustrates an overview of the system. The system has a Detector (hardware) that triggers a Restarter (hardware) to rollback execution whenever it detects an emergency. The detector then feeds an emergency Profiler (software) with a signature that represents processor activity leading up to that emergency. The profiler accumulates signatures to guide a code Rescheduler (software)

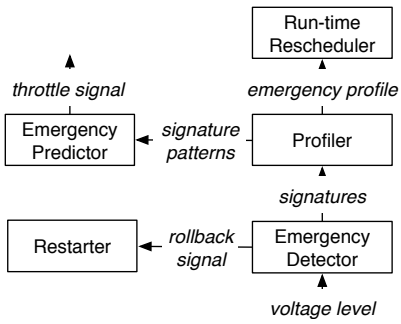


Fig. 3. System overview.

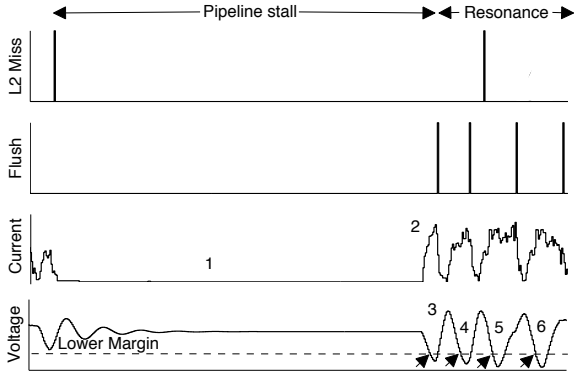


Fig. 4. Snapshot of *art* from the CPU2000 benchmark suite over 430 cycles. The snapshot illustrates how microarchitectural event induced pipeline stalling and resonance activity can lead to emergencies (indicated using arrows).

that eliminates recurrences of that emergency via run-time code transformations. If code reshaping is unsuccessful, the Profiler arms an emergency Predictor (hardware) with signature patterns to instead predict and suppress recurrences of the emergency by throttling processor activity.

We have implemented and tested prototypes of these five components separately in a simulation framework targeting tight margins (around 4%), and have reported on their effectiveness in recent research papers [9]–[13]. In this paper, we summarize that work and describe one way to build an emergency management system from those components.

Our premise is that tolerating emergencies is useful both for tightening margins, and observing the emergency behavior of running code. Section II describes how the detector and restarter tolerate emergencies. In order to eliminate emergencies intelligently, we need to understand, at least empirically, what causes them. Section III describes how to characterize emergencies in a way that is useful for preventing them. Our approach to preventing emergencies uses both software and hardware. Section IV describes how the rescheduler and the predictor complement one another.

II. Tolerating Emergencies

Rather than trying to avoid voltage emergencies, we rely on a hardware mechanism that allows voltage emergencies to occur, but when they do, the architecture has a built-in mechanism to recover processor state. The detector senses supply voltage for margin violations. When the detector detects a voltage dip below the lower voltage margin, it invokes the restarter mechanism to resume execution back at some previously known valid state.

Checkpoint-rollback mechanisms have been proposed for handling soft errors [14]–[16] that support execution rollback. But due to the frequent nature of voltage emergencies as compared to the soft-errors, we discovered that the overhead of explicitly saving the architectural state of the processor is so large that they cannot be directly applied to voltage noise. Combining this fail-safe mechanism with software that eliminates recurring emergencies, however, enables us to tolerate emergencies even at aggressive margins.

Alternatively, we have shown that it is possible to leverage existing store queue and reorder buffers in modern processors to delay commit just long enough to verify whether an emergency has occurred. Performing a rollback in the event of an emergency [9] is akin to flushing the pipeline after a branch misprediction.

III. Characterizing Emergencies

To be able to predict emergencies and reshape code to eliminate them, we need to find leading indicators of dangerous voltage fluctuations. We considered several microarchitectural parameters, such as the number of entries in the re-order buffer, the instruction fetch queue, and the load/store queue, along with microarchitectural events like cache misses and pipeline flushes. In this section, we summarize the perturbation effects of microarchitectural events on processor activity using real program examples, and show they can lead to voltage emergencies. We also discuss patterns in activity that allows us to not only identify emergencies uniquely, but also predict their recurring occurrences [11]–[13].

Individual microarchitectural events. We first studied the effect of individual microarchitectural events on current and voltage. Figure 4 shows a snapshot of pipeline activity for benchmark *art* from SPEC CPU2000 over 430 cycles. Microarchitectural events for the cache and branch predictor are illustrated along with current and voltage activity of the processor. In the Pipeline stall part of the figure, we observe an L2 cache miss (illustrated by a vertical bar in the L2 Miss subgraph). During the time it takes to service the L2 miss, pipeline activity ramps down as seen in the current profile (marker 1). However, after the L2 miss data is available, functional units become busy and there is a sudden increase in current activity (marker 2). This steep increase in current causes the voltage to temporarily dip below the voltage margin (marker 3) because of parasitic inductance in the power delivery subsystem.

Additionally, microarchitectural events that cause periodic high and low current activity can cause a resonance build up of voltage, if the period coincides with the resonance frequency of the power delivery subsystem. The Resonance portion of Figure 4 illustrates multiple pipeline flush events occurring in close proximity to one another. Pipeline flush events cause a sudden decrease in activity, and are followed by a rush of activity as instructions are rapidly issued along the correct program. The close distribution of these events leads to a resonating effect that results in rapid fluctuations in current.

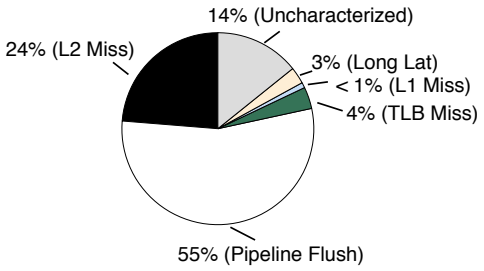


Fig. 5. Distribution of processor events and operations that cause voltage emergencies in the SPEC CPU2000 benchmark suite.

These successive fluctuations not only cause the voltage to swing, but also progressively increase in amplitude from one event to another (markers 4, 5 and 6).

Root-cause identification. Associating a specific microarchitectural event with an emergency requires us to apply our knowledge of how microarchitectural events affect processor activity. Identifying the *root-cause* of an emergency is not as simple as simply looking at the most recent microarchitectural event prior to the emergency. Consider the emergency at marker 5, which occurs slightly past an L2 miss event. But unlike the L2 miss event under Pipeline stall, this L2 miss event is not responsible for the emergency. L2 miss events take a few hundred cycles to complete, and instructions pending on that data are not issued until the event completes. Therefore, the burst of current activity does not correspond to this pending L2 event, rather it corresponds to the pipeline flush preceding the L2 event.

We have devised an algorithm to identify root-causes [11]. The algorithm scans recent processor events in a fixed priority order looking for event completion times that coincide with the time of the emergency. It scans down the list of L2 misses, TLB misses, pipeline flushes, L1 misses and long latency operations in that order. To show the strength of the relationship between these processor events/operations and emergencies, Figure 5 shows the percentage of emergencies caused by different root-causes for the SPEC CPU2000 benchmark suite. A majority of the emergencies are caused by pipeline flushes and L2 misses. The Uncharacterized 14% are because of emergencies that cannot be uniquely attributed to a single root-cause, such as the Resonance case illustrated in Figure 4.

Sensitivity to activity history. Whether or not an event at a particular location causes an emergency depends on activity just before and after the event. Even a small loop like that in Figure 6b, extracted from benchmark *gcc* of SPEC CPU20006, can have behavior phases with markedly different activity patterns. Figure 6a is a snapshot of activity within that loop over 880 cycles. It shows three repeating phases of the loop. Phase A uses paths $1 \rightarrow 4$ and $1 \rightarrow 2 \rightarrow 4$, while phase B uses only path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. The issue rate of phase A is relatively low, while that of phase B is quite high. The flush events labeled 1 are caused by branch mispredictions at the end of basic block 1. Those in phase B where the issue rate is high always cause emergencies, while those in phase A never do. Therefore, tracking program flow and microarchitectural events yields a proxy for the activity leading to emergencies.

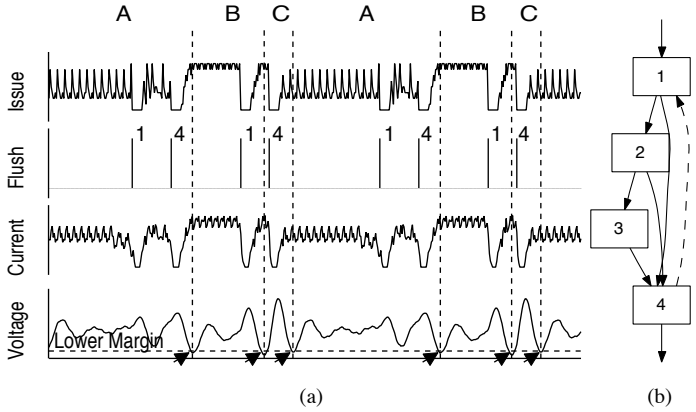


Fig. 6. (a) Voltage emergencies are associated with recurring activity (phases A, B and C) over 880 cycles. The numbers next to the vertical bars in the Flush graph correspond to the basic block number in (b) containing the mispredicted branch. (b) An emergency-prone loop from function `init_regs` in *gcc* from CPU20006 benchmark suite. Its activity snapshot is shown in (a).

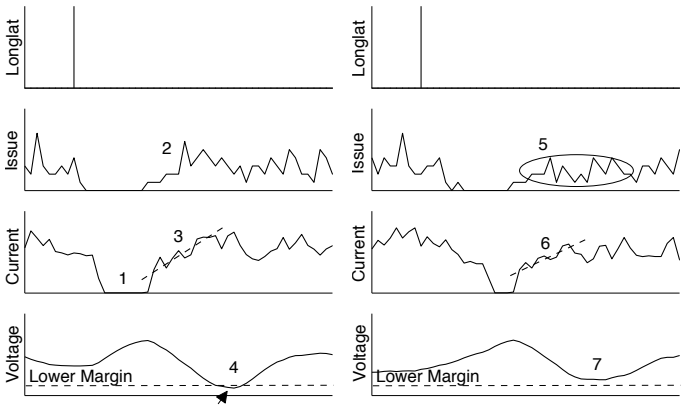
The activity leading up to an emergency is termed as an *emergency signature*, which comprises of program instructions and microarchitectural events. The purpose of tracking the instruction stream is to capture the dynamic path of a program. Consequently, control flow instructions are ideal candidates for tracking a program's dynamic execution path. Capturing the interleaving of program path with events is important to generate a representative snapshot of the corresponding dynamic current and voltage activity resulting from program interactions with the underlying microarchitecture. Emergency signatures can predict emergencies with over 90% accuracy [13] even as far ahead as 16 cycles of an impending emergency.

IV. Preventing Emergencies

To prevent emergencies, our scheme uses cooperating hardware and software. In hardware, the detector constantly monitors voltage to detect emergencies, and when one occurs, it captures a signature. The detector module relies on a shift register to monitor program and processor activity leading up to an emergency, and a signature is a snapshot of this shift register at the time of an emergency. In software, the profiler receives the signatures and builds a profile that it shares with the rescheduler. Some signatures are well handled by the rescheduler. From those that are not, the profiler selects hot signatures to place in the predictor's pattern table.

Rescheduler. Our run-time code rescheduler uses the root-cause identification algorithm discussed in the previous section to decide the kind of code transformation that is best suited for eliminating a specific type of emergency (e.g., pipeline flush, L2 miss). The rescheduler combines this information with control flow extracted from the signature to apply transformations only along certain program paths (if possible). The rescheduler then uses code motion techniques to smooth the original program activity.

For instance, the rescheduler can slow the issue rate following the root-cause instruction using code motion to compress dependent instructions closer together. This reduces the amount of instruction level parallelism available for the machine to exploit when activity resumes, which constrains the issue rate.



(a) Before code rescheduling. (b) After code rescheduling.

Fig. 7. A 50-cycle execution snapshot of *Sieve* from the Java Grande benchmark suite. (a) Shows how a pipeline stall on a long latency operation triggers an emergency (indicated by an arrow) as the issue rate ramps up sharply once the operation completes. (b) Code rescheduling slows the issue rate just enough to prevent the emergency illustrated in (a).

Reducing the issue rate even slightly prevents recurrences of emergencies. Figure 7 shows how the issue-rate smoothing technique works. The plot shows a slice of program activity corresponding to a loop within benchmark *Sieve* from the Java Grande suite. Figure 7a shows that data dependence on a long-latency operation stalls all processor activity, so the Current profile goes flat (marker 1). When the operation completes, the issue rate increases rapidly (marker 2) as several dependent instructions are successively released to functional units. This activity increases draw (marker 3), and as a result the voltage dips below the Lower Margin (marker 4).

Figure 7b shows activity after the rescheduler transforms the code slightly to reduce the issue rate. Since dependent instructions are packed more tightly, the issue rate in Figure 7b does not spike as high as in Figure 7a (marker 5). As a result, the processor now draws current less aggressively (the gradient at marker 6 is less steep compared to marker 3). Therefore, the original emergency at marker 4 is now permanently eliminated (marker 7).

Using this one issue-rate constraining technique, the compiler removes over 62% of all emergencies across the Java Grande suite. On average, only 20% of all root-causes had to be rescheduled because they contribute to a large percentage (over 98%) of all emergencies. Our results indicate that issue-rate smoothing works well for isolated emergencies like the cases illustrated in Figure 7a and Figure 4. We are also investigating more aggressive rescheduling techniques like loop unrolling and pipelining to make activity levels rise and fall more gradually, so that current and voltage are more stable [11], [12].

Predictor. In our design, the rescheduler eliminates most emergencies that have clear root causes, and that recur consistently, leaving only the more difficult cases for the hardware predictor. The predictor matches the patterns in its table against current events. On a match, it throttles execution to suppress the predicted emergency.

In our simulation of the SPEC CPU2006 suite, a predictor with no resource constraints suppresses 99% of all emergencies. Remaining emergencies are tolerated using a restarter mechanism (e.g. checkpoint-rollback discussed in Section II). To show the feasibility of the predictor in hardware, we simulated an implementation with realistic resource constraints. With an 8KB pattern table, our predictor suppresses 92% of all emergencies [13].

Our emergency predictor outperforms previously proposed architecture-centric techniques [5]–[8] at aggressive margins [9], [13]. Moreover, these prior mechanisms depend heavily on assumptions about the underlying power delivery subsystem, processor architecture, and sensor behavior. These assumptions complicate retargetability. In contrast, the emergency predictor operates independently of sensor delays, package characteristics, and details of the microarchitecture.

V. Conclusion

Enabling aggressive operating margins is critical as voltage noise increasingly limits performance-per-watt in microprocessor designs. In this paper, we summarize our contributions toward a hardware-software combination that enables aggressive margins: we allow voltage emergencies to occur at the expense of rolling back execution while relying on a feedback-driven software layer to permanently eliminate recurring emergencies. We base this design on characterizing emergencies in terms of code behavior, which enables us to predict them intelligently, or even eliminate them completely.

References

[1] N. James *et al*., "Comparison of split-versus connected-core supplies in the POWER6 microprocessor," in *ISSCC*, 2007.
[2] W. Zhao and Y. Cao, "Predictive technology model for sub-45nm early design exploration," *ACM JETC*.
[3] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *MICRO-32*, 1999.
[4] D. Ernst *et al*., "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," in *MICRO-36*, 2003.
[5] E. Grochowski, D. Ayers, and V. Tiwari, "Microarchitectural simulation and control of di/dt-induced power supply voltage variation," in *HPCA-8*, 2002.
[6] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," in *HPCA-9*, 2003.
[7] M. D. Powell and T. N. Vijaykumar, "Pipeline muffling and a priori current ramping: architectural techniques to reduce high-frequency inductive noise," in *ISLPED*, 2003.
[8] ——, "Exploiting resonant behavior to reduce inductive noise," in *ISCA*, 2004.
[9] M. S. Gupta, K. K. Rangan, M. D. Smith, G. yeon Wei, and D. Brooks, "DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors," in *HPCA-14*, 2008.
[10] M. S. Gupta, V. J. Reddi, M. D. Smith, G.-Y. Wei, and D. M. Brooks, "An event-guided approach to handling inductive noise in processors." in *DATE*, 2009.
[11] M. S. Gupta, K. K. Rangan, M. D. Smith, G. yeon Wei, and D. Brooks, "Towards a software approach to mitigate voltage emergencies," in *ISLPED*, 2007.
[12] K. Hazelwood and D. Brooks, "Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization," in *ISPLED*, 2004.
[13] V. J. Reddi, M. S. Gupta, G. Holloway, G.-Y. Wei, M. D. Smith, and D. Brooks, "Voltage emergency prediction: Using signatures to reduce operating margins," in *HPCA-15*, 2009.
[14] H. Ando *et al*., "A 1.3 GHz fifth generation SPARC64 microprocessor," *IEEE JSSC*, vol. 38, 2003.
[15] N. J. Wang and S. J. Patel, "ReStore: Symptom-based soft error detection in microprocessors," *TDSC*., 2006.
[16] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *ICS*, 2004.