An Event-Guided Approach to Reducing Voltage Noise in Processors

Meeta S. Gupta, Vijay Janapa Reddi, Glenn Holloway, Gu-Yeon Wei and David M. Brooks
School of Engineering and Applied Sciences, Harvard University, Cambridge, MA
{meeta,vj,holloway,wei,dbrooks}@eecs.harvard.edu

***Abstract*—Supply voltage fluctuations that result from inductive noise are increasingly troublesome in modern microprocessors. A voltage "emergency", i.e., a swing beyond tolerable operating margins, jeopardizes the safe and correct operation of the processor. Techniques aimed at reducing power consumption, e.g., by clock gating or by reducing nominal supply voltage, exacerbate this noise problem, requiring ever-wider operating margins. We propose an *event-guided*, adaptive method for avoiding voltage emergencies, which exploits the fact that most emergencies are correlated with unique microarchitectural events, such as cache misses or the pipeline flushes that follow branch mispredictions. Using checkpoint and rollback to handle unavoidable emergencies, our method adapts dynamically by learning to trigger avoidance mechanisms when emergency-prone events recur. After tightening supply voltage margins to increase clock frequency and accounting for all costs, the net result is a performance improvement of 8% across a suite of fifteen SPEC CPU2000 benchmarks.**

I. Introduction

Power-constrained CMOS designs are making it increasingly difficult for microprocessor designers to cope with power supply noise. As current draw increases and operating voltage decreases, inductive noise threatens the robustness and limits the clock frequency of high-performance processors. Large current swings over small time scales cause large voltage swings in the power-delivery subsystem due to parasitic inductance. A significant drop in supply voltage can cause timing margin violations by slowing logic circuits. For reliable and correct operation of the processor, *voltage emergencies*, i.e., large voltage swings that violate noise margins, must be avoided.

The traditional way to deal with inductive noise is to over-design the processor to allow for worst-case fluctuations. Unfortunately, the gap between nominal and worst-case operating conditions in modern microprocessor designs is growing. A recent paper on supply-noise analysis for a POWER6 processor [11] shows the need for timing margins that accommodate voltage fluctuations of more than 18% of nominal voltage (200mV dips at a nominal voltage of 1.1V). Conservative design of processors using large timing margins ensures robustness, but it lowers the maximum achievable operating frequency.

Another way to handle inductive noise is to design the processor for typical-case operating conditions and add a fail-safe mechanism that guarantees correctness despite noise margin violations. This strategy can improve performance, but only if the cost of using the fail-safe mechanism is not too high. In practice, active emergency prevention (e.g., performance throttling techniques) is needed if aggressive operating margins are sought to push performance.

A number of throttling mechanisms have been proposed to dampen sudden current swings, including frequency throttling, pipeline freezing, pipeline firing, issue ramping, and changing the number of the available memory ports [8], [12], [15], [16]. However, such mechanisms require a tight feedback loop that detects an imminent violation and then activates a throttling mechanism to avoid the violation. The detectors are either current sensors or voltage sensors that trigger when a soft threshold is crossed, indicating a violation is likely to occur. Unfortunately, the delay inherent in such feedback loops limits effectiveness and necessitates margins sufficiently large to allow time for the loop to respond [10].

Another approach is to eliminate the feedback loop associated with previous proposals and instead monitor specific microarchitectural events as indicators of processor activity that can lead to voltage emergencies. We have previously shown that there is a strong correlation between voltage violations and microarchitectural events [9]. A variety of events can lead to emergencies such as cache misses, pipeline flushes due to branch misprediction, TLB misses, and long-latency events. For example, a sudden increase in processor activity after a long L2 cache miss can cause a sudden drop in voltage. Figure 1 shows the result of applying the categorization algorithm we previously used [9] to our suite of benchmarks from SPEC CPU2000. Most voltage emergencies in the benchmarks are associated either with L2 cache misses or with pipeline flushes due to branch mispredictions.

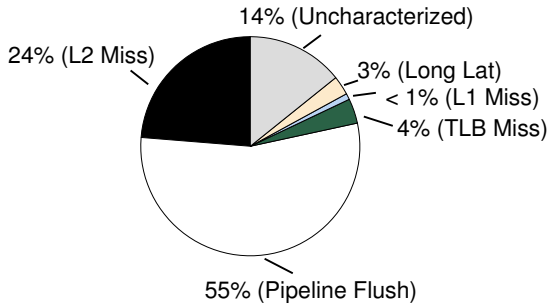

Fig. 1: Distribution of the events that trigger emergencies for our SPEC CPU2000 benchmarks.

Our event-driven mechanism triggers corrective action when it detects certain emergency-prone events (L2 cache misses and branch flushes, as they are the events associated with most of the emergencies). A naïve implementation might be to take preventive measures at every such event (for example, to activate a throttling mechanism at every L2 miss). That

978-3-9810801-5-5/DATE09 © 2009 EDAA

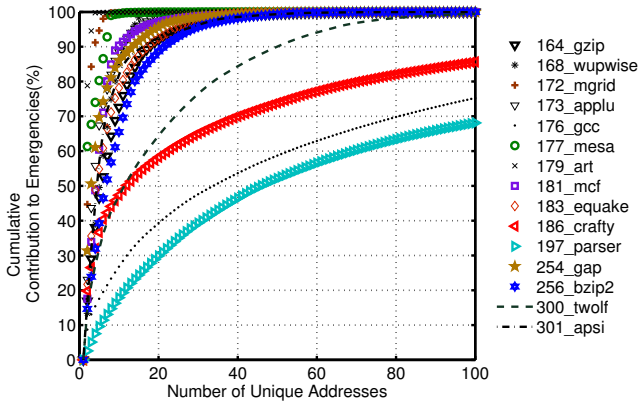

Fig. 2: The number of unique instructions causing emergencies and their corresponding contribution to the total number of emergencies.

would be overly conservative, however, since most such events don't give rise to emergencies. Our analysis shows a false alarm rate of 71% for such a naïve mechanism. Instead, we track specific instructions associated with events (L2 misses or pipeline flushes) that have caused emergencies, and we maintain contextual information for each event and emergency. Reacting only to events associated with emergencies results in much less overhead than the naïve implementation.

Figure 2 shows a cumulative distribution graph plotting the number of unique program addresses that trigger emergencies and their contribution to the total number of emergencies during execution. Each benchmark except for *parser*, *gcc*, *twolf* and *crafty* has fewer than fifteen unique program addresses that cause over 90% of run-time emergencies. Thus, the state that an *instruction-specific* event-guided mechanism needs to maintain can be stored in just a few bytes.

The main contributions of this paper are that:

1) We propose and evaluate a method that learns to avoid recurring voltage emergencies by triggering preventive action on the microarchitectural events that cause them.
2) We combine our voltage emergency avoidance method with a reusable, non-intrusive fail-safe mechanism and demonstrate net performance benefits while guaranteeing safe and correct operation.
3) We show that current-shunt regulation controlled at the microarchitectural level can help to reduce inductive noise with minimal increase in power.

In Section II we describe our proposed event-guided approach. Section III describes the experimental setup that we designed to test our approach through simulation. Section IV describes and evaluates our results. Section V discusses related work on handling inductive noise in processors at the architectural level. Section VI concludes with a brief summary and plans for future work.

II. An Event-Guided Approach

Figure 3 shows the operational flow of our event-guided method for detecting and suppressing voltage emergencies. The parts of the diagram connected by solid arrows detect and respond to actual voltage emergencies. The parts connected by dashed arrows are responsible for learning to recognize impending violations and using this training to suppress future

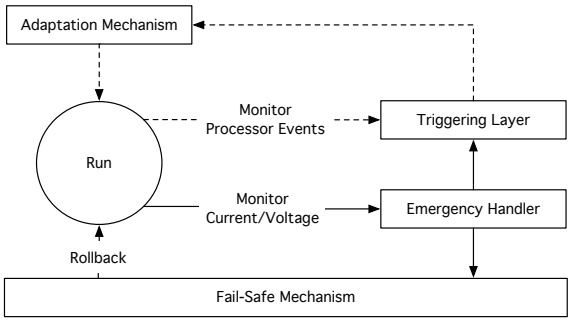

Fig. 3: Overview of the proposed event-guided architecture for handling voltage emergencies.

occurrences of violations. Current and voltage are monitored by a sensor, and an *emergency handler* determines when the supply voltage exceeds operating margins. On detecting a violation, the handler invokes the *fail-safe mechanism* to recover from any deleterious effects of the emergency. The handler also signals the *triggering layer* to learn from this emergency, in order to recognize future emergencies. Later, when the triggering layer detects an emergency-prone situation, it invokes the *adaptation mechanism* to take appropriate preventive action. In the following paragraphs, we describe the components of Figure 3 in more detail.

Emergency handler. When it detects a voltage swing that violates the microprocessor's voltage margins, the emergency handler invokes the fail-safe mechanism to recover an uncorrupted execution state. After recovery, the handler invokes the triggering layer to train it to prevent voltage emergencies proactively, collecting details (such as code location and event type) about the emergency just encountered to guide its analysis.

Fail-safe mechanism. Emergencies can occur either because they are not anticipated by the triggering layer or because event-guided adaptations prove ineffective. We use a recovery mechanism similar to the reactive mechanisms for processor error detection and correction that have been proposed for handling soft errors [3], [18]. These mechanisms are based on checkpoint and rollback. We evaluate two checkpoint-rollback mechanisms, one that makes checkpoints explicitly [14], [13], [1] and one that saves them implicitly [10]. Each is *fine-grained*—the interval between checkpoints is just tens of cycles. Explicit-checkpoint mechanisms periodically save the architectural state of the processor, i.e., the architectural registers and updated memory state. Restoring the register state incurs substantial overhead, and there are additional cache misses at the time of recovery. (We assume a buffered memory update, and cache lines updated between checkpoints are marked as volatile.) Moreover, a robust explicit-checkpoint mechanism for noise margin violations requires the maintenance of two checkpoints (since any checkpoint falling after a violation but before its subsequent detection must be considered corrupt).

We previously used an implicit-checkpoint-rollback scheme based on delayed commit and rollback that speculatively buffers updates to the machine state for long enough to be sure that no emergency occurred while they were being computed [10]. Completed results wait in the reorder buffer (ROB) or store queue (STQ) for an interval set by the sensor delay

of the emergency detector. After that interval, if there have been no emergencies, results are committed to the retirement register file or the L1 data cache.

Explicit checkpointing is a less intrusive addition to existing processor designs, and it is more likely to be useful for purposes other than voltage emergencies. But because of the high frequency of noise margin violations, the performance cost of an explicit-checkpoint mechanism could be much greater than that of the implicit mechanism. Our event-guided emergency suppression method brings the overhead of explicit checkpointing into line with that of implicit checkpointing, giving us the best of both approaches.

Triggering layer. The triggering mechanism tries to guide the hardware through safe execution using adaptation mechanisms that prevent emergencies (*Adaptation Mechanism* in Figure 3). The triggering layer waits for event notifications from the emergency handler. When it receives one, it caches the time and recent microarchitectural event history, and it updates a frequency counter for the particular emergency. This information determines when to locate and adapt the source instruction that caused an emergency. Once an emergency has been flagged as frequently occurring, the triggering layer uses an *event characterization* algorithm [9] to determine the root cause of the emergency. It targets subsequent occurrences of the emergency for prevention, using microarchitectural events as cues to activate an adaptation mechanism.

In order to provide pertinent information to the triggering layer of the event-guided approach, the processor maintains two circular structures similar to those found in existing architectures like IPF and PowerPC. The first is a *branch trace buffer (BTB)*, which maintains information about the most recent branch instructions, their predictions, and their resolved targets. The second is a *data event address register (D-EAR)*, which tracks recent memory instruction addresses and their corresponding effective addresses for all cache and TLB misses. The triggering layer reads this information at every voltage emergency and uses it as described in Section IV to suppress future emergencies.

We augment each entry in our BTB and D-EAR structures with a saturating counter that gives the age of the entry in cycles. To determine whether an event described in one of the structures is the likely cause of a subsequent emergency, we need to know the time interval between the two. The age of an entry also helps the hardware to discard entries that are too old to be of use to the triggering mechanism.

The trigger mechanism can either be implemented purely in hardware or as software-assisted hardware. A hardware-only mechanism is practical because monitoring fewer than twenty program points at once typically covers 90% of all emergencies (as discussed in Section I). Information about which instructions to track and the associated emergency-causing events can be maintained in a small content-addressable memory that recognizes when to trigger preventive action. Alternatively, software can be used to add hints to instructions (such as mispredicted branches or loads that have missed in the data cache) that have been associated with noise margin violations, so that the hardware can take preventive measures whenever those instructions are again associated with emergency-prone events. Software has potential benefits that hardware-only mechanisms do not share, since a software layer can perform intelligent code transformations to prevent emergencies. But this and other specifics about the hardware or software implementation are beyond the scope of this paper.

Adaptations. Adaptations are intended to avoid the sudden current increases that lead to large voltage swings. We consider four kinds of adaptations: frequency throttling, using a current-shunt regulator, pseudo-instruction padding, and prefetching. These techniques either stretch out current fluctuations in time or smooth them out in amplitude. In Section IV, we evaluate the effectiveness of several combinations of these techniques.

Throttling mechanism: Throttling mechanisms spread out increases in current by slowing processor activity. Several kinds of throttling have been proposed [8], [12], [15], [16]. We use simple frequency throttling: dividing the frequency of the system in half whenever throttling is turned on. This quickly reduces current draw, but it also reduces performance by slowing down the machine. We use throttling both for mispredicted branches and for loads flagged as missing in the L2 cache.

Current-shunt mechanism: Alon and Horowitz [2] propose a *push-pull* regulator topology that uses an additional higher-than-nominal supply voltage, comparator-based feedback, and a switched-source follower output stage to reduce supply noise. We drive the output stage of this regulator with our triggering mechanism (hardware-event-guided or software-guided) and call it the *current-shunt* mechanism. This technique has an effect similar to throttling, but without the extra performance penalty associated with throttling. We use the current shunt either for a mispredicted branch or an L2 miss. This mechanism suffers from additional power overhead since the extra charge dumped into the power grid comes from a higher supply voltage. To simplify the comparison of schemes, this power overhead can be translated into performance loss. For example, designers might compensate for this power overhead by reducing global supply voltage and clock frequency. We assume a 3% increase in power translates to a 1% decrease in performance [6].

Pseudo-instructions: One possible way to deal with a sudden increase in activity when an L2 cache miss returns is to create a chain of instructions with data dependences that require them to be issued serially, so that processor activity increases gradually. We do so by adding redundant pseudo-instructions, which we call *pseudo-nops*. However, these nop instructions degrade performance by wasting CPU cycles. Hence, we employ a *selective nop* strategy whereby pseudo-nops are discarded unless the L2 miss occurs. Our analysis shows that a single nop is able to achieve the same reduction in emergencies as multiple pseudo-nops with less performance degradation, so we use single pseudo-nop insertion in our evaluation.

Prefetching: Another way to deal with large stalls is by prefetching loads that cause large L2 cache miss penalties. To study the potential of dynamic prefetching adaptations for dealing with emergencies, we assume an *ideal prefetch* mechanism; the system inserts prefetch instructions for delinquent loads, eliminating further cache misses for those loads. This does not capture the complexities of a dynamic prefetching engine, but it gives a measure of its potential to combat

Clock Rate	3.0 GHz	RAS	64 Entries
Inst. Window	128-ROB, 64-LSQ	Branch Penalty	10 cycles
Functional Units	8 Int ALU, 4 FP ALU, 2 Int Mul/Div, 2 FP Mul/Div	Branch Predictor	64-KB bimodal gshare/chooser
		BTB	1K Entries
Fetch Width	8 Instructions	Decode Width	8 Instructions
L1 D-Cache	64 KB 2-way	L1 I-Cache	64 KB 2-way
L2 I/D-Cache	2MB 4-way, 16 cycle latency	Main Memory	300 cycle latency

TABLE I: Processor parameters for SimpleScalar.

emergencies. To be unbiased in our analysis, we omit the performance benefits of prefetching when considering the overall performance of the system.

III. Experimental Setup

Our evaluation of event-guided voltage emergency suppression compares naïve individual adaptations against different sets of instruction-specific adaptations drawn from the list described in the preceding section. For each adaptation or adaptation set, the evaluation also compares a fine-grained explicit-checkpoint mechanism against the implicit-checkpoint scheme. Table II lists all of the combinations tested. Our objective is to find the most successful combination for minimizing the overhead associated with suppressing or correcting for emergencies. We also want to show that under realistic assumptions, there is a net performance advantage to using our approach.

We use SimpleScalar/$x86$ to simulate a Pentium 4 with the parameters shown in Table I. The modified 8-way superscalar $x86$ SimpleScalar collects detailed cycle-accurate current profiles using Wattch [7]. We calculate voltage variations by convolving the simulated current profiles with an impulse response of the power delivery subsystem [16], [12]. We use a power delivery subsystem model based on the characteristics of the Pentium 4 package [4], which exhibits a mid-frequency resonance at 100MHz with a peak impedance of 5mΩ. This corresponds to a period of 30 clock cycles for a 3GHz machine. Finally, we assume peak current swings of 16-50A.

In our experiments, we assume an aggressive operating margin of 4%, even though with our power delivery package, we see voltage dips as large as 13%. Bowman et al. [5] show that removing a 10% operating voltage guardband leads to a 15% improvement in clock frequency. Using this 1.5 ratio as the scaling factor from operating voltage margin to clock frequency gives an upper bound of 17.5% on the clock frequency increase made possible by reducing a 13% voltage margin down to 4%. Defining performance as $frequency/CPI$, where *CPI* is the average number of cycles per instruction, this frequency increase translates to a potential 17.5% performance increase, provided *CPI* remains constant. Of course, emergency avoidance incurs performance overhead, which we measure in our simulations. We evaluate an adaptation's net impact on performance by using this measured overhead to reduce the top potential performance enabled by tightening to a 4% voltage margin.

We use 20 cycles as the delay required by a voltage sensor to detect a voltage emergency. We also use 20 cycles as the duration of each throttling activation, and 21 cycles as the checkpointing interval in the explicit-checkpoint mechanism.

	Nature of Adaptation	
Naïve	Throttling	Current shunt
Instruction-specific	Throttling + Pseudo-nop	Current shunt + Pseudo-nop
	Throttling + Prefetch	Current shunt + Prefetch

TABLE II: Combinations of adaptation mechanisms studied in our simulations.

Making checkpoints at shorter intervals than the delay time of the voltage sensor would be wasteful. For the current-shunt adaptation, we find that inserting charge equivalent to 10 amperes for 10 cycles is most effective. Total processor current draw is typically 35A.

Baseline fail-safe mechanisms. A checkpoint-rollback mechanism suffers from the penalty of redoing some computation whenever a noise margin violation occurs. The restart penalty is a function of the sensor delay in the system, i.e., the time required to detect a margin violation. An explicit-checkpoint scheme incurs additional overhead associated with restoring the registers (assumed to be 8 cycles, for 32 registers with 4 write ports) and memory state (because volatile lines are flushed, additional misses can occur after rollback). An implicit-checkpoint scheme does not incur a penalty for restoring state, but it suffers from the penalty associated with delaying commits in the ROB/STQ. This penalty is also a function of the sensor delay. With a sensor delay of 20 cycles, the overhead incurred by the fine-grained explicit-checkpoint mechanism is 35%, as compared to only 8% for the implicit-checkpoint scheme (Figure 4(a)).

Event-guided fail-safe execution. Section II provided a detailed description of the various kinds of adaptation in isolation. In this section, we evaluate the effectiveness of individual naïve event-driven adaptations, and we also show the benefits of combining instruction-specific adaptations. The configurations that we analyze are shown in Table II. Each is paired with both fail-safe strategies, i.e., a fine-grained explicit-checkpoint-rollback mechanism and also one using implicit checkpoints.

For load instructions, we apply combined adaptations in sequence. For example, for the combination of instruction-specific throttling and pseudo-nops, we use throttling first for a load instruction, but if the throttling at any recurrence of the cache-miss event for that load is unable to avoid a voltage emergency, then we insert a pseudo-nop. Similarly, we try throttling alone before adding prefetching in the instruction-specific-throttle-and-prefetch combination. We maintain the same order for current-shunt adaptations.

IV. Results and Analysis

First, some general observations. In the SPEC CPU2000 benchmarks, instruction-specific throttling is able to eliminate nearly 100% of voltage emergencies associated with pipeline flushes that follow branch mispredictions. But throttling alone is less effective in reducing emergencies after loads that miss in the L2 cache. On the other hand, prefetching alone eliminates nearly all of these L2-miss-related emergencies.

Figure 4 illustrates the performance improvements of our event-guided framework averaged over our CPU2000 benchmark suite. Note in Figure 4(a) that each adaptation scheme

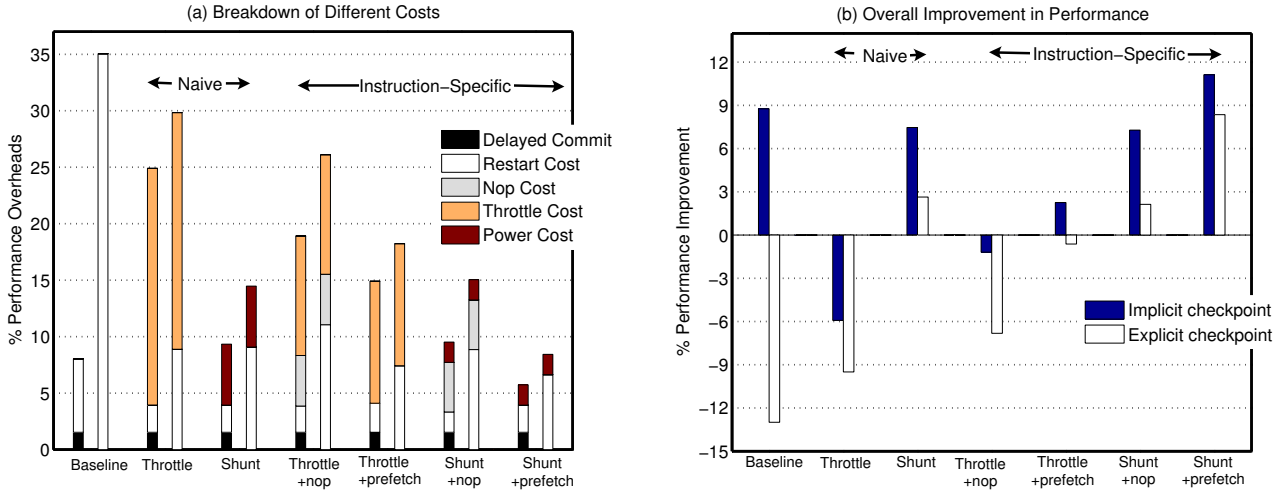

Fig. 4: Comparison of different approaches to handling noise margin violations. Each bar represents an average over our suite of SPEC CPU2000 benchmarks. The left bar for each scheme represents the implicit-checkpoint mechanism; the right bar, the explicit-checkpoint mechanism. Graph (a) shows the distribution of performance overhead associated with each adaptation. Graph (b) shows overall improvements in performance ($frequency/CPI$), with 17.5% as the upper bound.

reduces the total restart cost relative to the fail-safe-only baseline, because the elimination of emergencies reduces the number of rollbacks needed. As expected, naïve event-based adaptations have higher overhead than instruction-specific ones because the penalties for using them (reduced clock frequency in the case of throttling or increased power consumption in the case of current shunt) apply to all mispredicting branch instructions and delinquent loads, not just to the problematic ones.

The current shunt and prefetching combination (last pair in Figure 4) achieves the best result overall. With the implicit-checkpoint mechanism, this combination reduces total overhead from 8% (for the baseline) down to 6%, which increases the performance gain from 8.7% (for the baseline) to 11%. With the fine-grained explicit-checkpoint mechanism, the combination of current shunt and prefetching yields a significant increase in performance (a net performance improvement of 8% for the combined adaptations as compared to 13% degradation for the baseline), primarily because of reduced total restart costs. The overall cost is reduced from 35% for the baseline to around 9%, equivalent to the baseline penalty of the implicit-checkpoint-rollback mechanism. Because our event-guided approach reduces the need to roll back and restart execution, it allows fine-grained explicit checkpointing to be used with relatively little penalty.

Figure 5 presents a closer view of this most successful combined adaptation (current shunt plus prefetching scheme). The adaptation is effective in eliminating emergencies attributed to L2 miss events and flush events for most of the benchmarks. In *mgrid*, however, applying prefetching to the loads that cause L2 misses increases the emergencies (almost doubling them). A detailed analysis shows that removing the stall periods associated with L2 misses increases the fluctuation of current, which causes more voltage swings. Further investigation reveals that the emergencies have a strong correlation with a set of events instead of a single event, leading to misclassification by the event characterization algorithm. It is in such cases that a software-assisted solution could help recognize that the usual handling of events is not working and turn the adaptation off to prevent further aggravation of the problem.

Another anomaly is *apsi*, which does not show a significant reduction in emergencies. The main reason is that few of its emergencies are attributed to branches and L2 miss events, and we focused on handling those events.

V. Related Work

Much of the recent research on the inductive noise problem has addressed only the hardware level. Very little work has been done on software-assisted approaches for handling voltage emergencies. Grochowski et al. [8] describe a voltage simulation capability based on cycle-by-cycle activity levels of each functional block, and they use the simulation result in a feedback mechanism to handle dI/dt emergencies. Joseph et al. [12] present a control-theoretic technique to handle voltage emergencies. Powell and Vijaykumar handle high-frequency inductive noise using a pipeline muffling mechanism [15] and resonance tuning [16]. Most of these works use sensor-triggered mechanisms. Our method uses sensors only to detect unavoidable emergencies, not to trigger adaptations.

As Figure 4 shows, throttling is a poor adaptation mechanism for the event-guided scheme described in this paper. In a separate study, we have shown that collecting activity history as a sequence of control flow events plus microarchitectural events enables throttling to suppress emergencies effectively [17]. However, such a mechanism requires sizable resources (e.g., an 8KB structure) by comparison with tracking a few relevant program addresses.

VI. Conclusion

We have developed and evaluated a method for reducing supply voltage fluctuations that result from inductive noise in microprocessors. Using the fact that most voltage emergencies are correlated with unique events like cache misses and pipeline flushes, our method learns to trigger

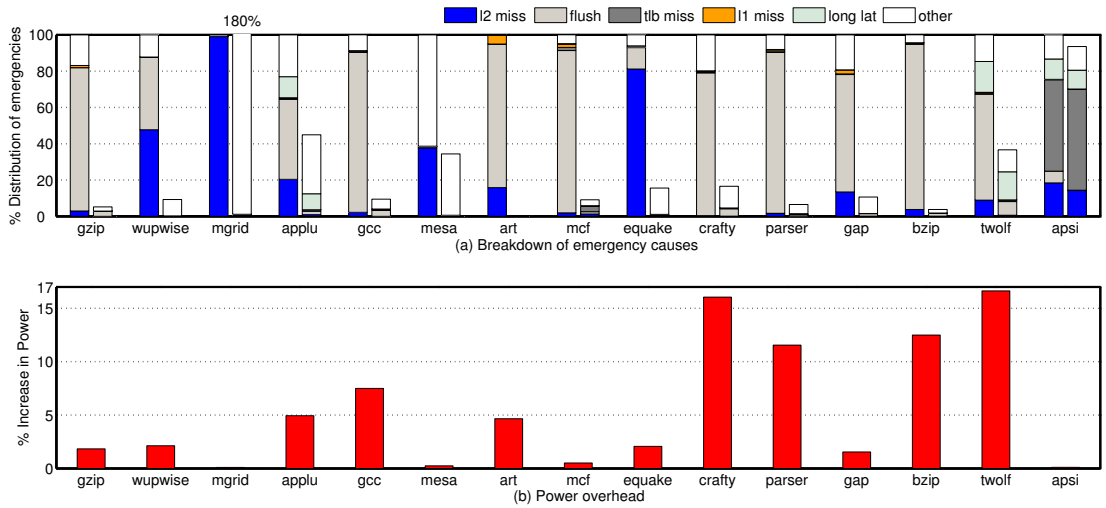

Fig. 5: Overall effect of applying the current-shunt adaptation for branches and loads, followed by prefetching of loads when the current shunt for loads does not eliminate the emergency. The left bar for each benchmark in graph (a) represents the baseline emergency distribution and the right bar represents the distribution of emergencies after applying the adaptation mechanisms. Its height reflects the relative number of emergencies not avoided. Graph (b) shows the increase in power due to the additional charge provided by the current shunt.

preventive adaptations when the recurrence of such an event at a particular code location forecasts a new emergency. We tested several kinds of preventive adaptations, alone and in combination, including a novel application of current-shunt regulation that smoothed fluctuations effectively without consuming much power. We also evaluated two fail-safe mechanisms that guarantee correct operation in the face of unavoidable voltage emergencies. We showed that because our event-guided method reduces emergencies so effectively, a fine-grained explicit-checkpoint and rollback mechanism becomes nearly as low in overhead as an implicit-checkpoint scheme that would be less reusable and more intrusive to add to microprocessor designs. After tightening supply voltage margins to 4% to increase clock frequency and accounting for all costs associated with emergency reduction and fail-safe protection, the net result is an average performance improvement of about 7% across our suite of fifteen SPEC CPU2000 benchmarks.

In future work, we intend to improve our techniques for dynamically identifying the causes of voltage emergencies. We plan to add a firmware or software component to our framework that will both extend its memory for emergency signatures and prevent the hardware from pursuing unproductive adaptations. We will also study the use of run-time code transformations that avoid emergencies.

Acknowledgments

We are grateful to our colleagues in industry and academia for the many discussions that have contributed to this work. We also thank the anonymous reviewers for their comments and suggestions. This work is supported by gifts from Intel Corporation, and National Science Foundation grants CCF-0429782 and CSR-0720566. Any 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

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO-36*, 2003.

[2] E. Alon and M. Horowitz. Integrated regulation for energy-efficient digital circuits. In *CICC*, 2007.

[3] H. Ando et al. A 1.3 GHz fifth generation SPARC64 microprocessor. *IEEE JSSC*, 38, 2003.

[4] K. Aygun et al. Power delivery for high-performance microprocessors. *Intel Technology Journal*, 9, Nov. 2005.

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

[6] D. Brooks et al. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. In *IEEE-MICRO*, 2000.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA-27*, 2000.

[8] E. Grochowski, D. Ayers, and V. Tiwari. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *HPCA-8*, 2002.

[9] M. S. Gupta, K. Rangan, M. D. Smith, G.-Y. Wei, and D. M. Brooks. Towards a software approach to mitigate voltage emergencies. In *ISLPED*, 2007.

[10] M. S. Gupta, K. Rangan, M. D. Smith, G.-Y. Wei, and D. M. Brooks. DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors. In *HPCA-14*, 2008.

[11] N. James et al. Comparison of split-versus connected-core supplies in the POWER6 microprocessor. In *ISSCC*, 2007.

[12] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *HPCA-9*, 2003.

[13] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed early load retirement. In *HPCA-11*, 2005.

[14] J. F. Martínez et al. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO-35*, 2002.

[15] 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.

[16] M. D. Powell and T. N. Vijaykumar. Exploiting resonant behavior to reduce inductive noise. In *ISCA-28*, 2004.

[17] 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. (to appear).

[18] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Trans. Dependable Secur. Comput.*, 3(3):188–201, 2006.