Towards a Software Approach to Mitigate Voltage Emergencies

Meeta S. Gupta, Krishna K. Rangan, Michael D. Smith, Gu-Yeon Wei and David Brooks

School of Engineering and Applied Sciences
Harvard University
{meeta,kkrangan,smith,guyeon,dbrooks}@eecs.harvard.edu

ABSTRACT

Increases in peak current draw and reductions in the operating voltages of processors continue to amplify the importance of dealing with voltage fluctuations in processors. One approach suggested has been to not only react to these fluctuations but also attempt to eliminate future occurrences of these fluctuations by dynamically modifying the executing program. This paper investigates the potential of a very simple dynamic scheme to appreciably reduce the number of run-time voltage emergencies. It shows that we can map many of the voltage emergencies in the execution of the SPEC2000 benchmarks on an aggressive superscalar design to a few static loops, categorize the microarchitectural cause of the emergencies in each important loop through simple observations and a simple priority function, and finally apply straightforward software optimization strategies to mitigate up to 70% of the future voltage swings.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces*

General Terms

Reliability, Design

Keywords

di/dt, voltage emergencies, dynamic optimization framework, power-aware computing, hardware-software codesign

1. INTRODUCTION

Inductive noise has been a long-standing problem in processor design further exacerbated by low-power design techniques. With increased interest in reduced-power microarchitectures, this problem will continue to gain in significance, especially as operating voltages decrease and peak current draws increase [1]. Large swings in current over small time intervals cause large swings in voltage due to the parasitic inductance in the processor's power-delivery subsystem. *Voltage emergencies* are voltage swings of magnitudes sufficient to cause transient timing faults or long-term reliability problems. Such emergencies may be addressed by larger design margins at the cost of higher power consumption. This paper presents an alternative solution based on the understanding and categorization of voltage emergencies and presents simple optimization techniques to mitigate them.

Though it is possible to handle voltage emergencies with circuit- and architecture-level solutions [2, 3, 4], Hazelwood and Brooks [5] showed that voltage emergencies are correlated with an application's dynamic code stream and not just the underlying architecture and power-delivery subsystem. As such, a holistic hardware/software approach to handling voltage emergencies has the potential to provide additional advantages beyond the "fail-safe" capabilities of hardware-only solutions. In particular, voltage-monitoring hardware coupled with a dynamic optimization system could be used to sense voltage emergencies, modify the problematic code sequences, and avoid future voltage emergencies in those code sequences.

To be worthwhile, a holistic hardware/software approach should incur a hardware cost that is not much greater than the "fail-safe" circuitry currently employed and a runtime cost that is significantly less than the performance gained from avoiding future emergencies. Historically, dynamic optimization systems have been successfully employed when a small amount of an application's static code base accounts for a majority of the application's execution. We find a similar relationship with voltage emergencies: a small amount of an application's static code accounts for a majority of the voltage emergencies encountered.

Being able to focus a dynamic optimizer's effort, however, is only part of what is needed. We must also be able to understand the seemingly complex interaction among the power-delivery subsystem, the microarchitecture, and the runtime behavior of applications, so that we can effectively categorize the cause of each voltage emergency and understand what kind of code change will avoid the emergency in later executions. For results in this paper, we focused our analysis on the interval around the most frequently executed of the code sequences causing emergencies. We examined the detailed current and voltage fluctuations in these intervals and developed a simple model for the primary microarchitectural causes of the emergencies. Understanding the cause and contexts of the emergencies led us to a cou-

ISLPED'07, August 27–29, 2007, Portland, Oregon, USA.

ple of straightforward optimizations which we simulated to estimate the potential benefits of a holistic approach.

The main contributions of this paper are as follows:

1. This paper identifies loops or code sequences in the SPEC benchmarks that induce voltage emergencies. We show that only a handful of code sequences are responsible for nearly all the emergencies in an application run.

2. Microarchitectural reasons for voltage emergencies are analyzed through detailed simulation.

3. Finally, this paper shows that simple optimization techniques based on the characterization of possible events leading to emergencies are beneficial in eliminating them.

The rest of the paper is organized as follows. Section 2 reviews prior research related to inductive noise. We discuss the characterization framework in Section 3 and present detailed analysis of the various loops in different benchmarks that cause emergencies. Section 4 details the various microarchitectural events that can lead to noise-margin violations and provides a distribution of the different causes for each benchmark. A collection of optimization techniques based on the categorization of emergencies is presented in Section 5. Finally, Section 6 concludes and presents possible future research.

2. RELATED WORK

Much of the recent research addressing the inductive noise problem has done so at either the hardware or circuit level. Joseph et al. [3] present a control-theoretic technique to handle the voltage emergencies. They also developed a *stressmark* that produces emergencies based on alternating periods of high and low processor activity. Powell and Vijaykumar handle high-frequency inductive noise using a pipeline muffling mechanism [4] and resonance tuning [6]. Grochowski et al. [2] describe a voltage simulation capability based on cycle-by-cycle activity levels of each functional block and use the simulation result in a feedback mechanism to handle di/dt emergencies.

A few studies have considered adjustments to application code as a mechanism for addressing the problems of inductive noise. For example, Toburen [7] presents static compilation techniques to reduce di/dt emergencies. Yun and Kim [8] propose *power-aware* modulo scheduling to reduce the step and peak power for VLIW processors. Hazelwood and Brooks [5] propose a collaborative hardware-software framework that would optimize applications to eliminate future di/dt emergencies. They investigate how different runtime code optimization techniques could help reduce the di/dt emergencies in a synthetic benchmark, byt their analysis does not provide an understanding of the interaction between different events leading to emergencies. El-Essaway and Albonesi [9] study thread management techniques to reduce inductive noise in SMT processors.

Our work maps voltage emergencies to looping structures in applications and looks to understand what microarchitectural events are most important in the loops with the most-frequent voltage emergencies. We use complete applications (SPEC2000 benchmarks) and find that simple optimization strategies, an alternative to expensive packaging solutions, are effective in mitigating the inductive noise problem.

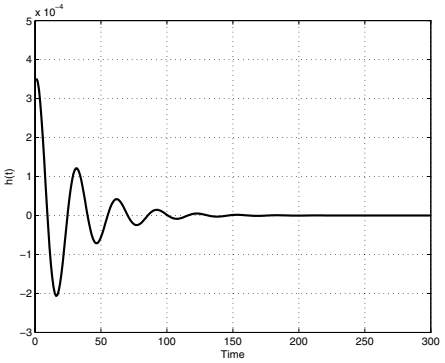

Figure 1: Transient Response of the PDS

Clock Rate	3.0 GHz
Instruction Window	256-RUU, 128-LSQ
Functional Units	8 integer ALU, 4 Floating Point ALU, 2 Integer Mul/Div, 2 FP Mul/Div
Fetch Width	8 Instructions
Decode Width	8 Instructions
Branch Penalty	10 cycles
Branch Predictor	64-KB bimodal/gshare/chooser
BTB	1K Entries
RAS	64 Entries
L1 D-Cache	64 KB 2-way
L1 I-Cache	64 KB 2-way
L2 I/D-Cache	2KB 4-way, 16 cycle latency
Main Memory	300 cycle Latency

Table 1: Processor Parameters for SimpleScalar

3. LOOP BASED CATEGORIZATION OF VOLTAGE EMERGENCIES

We performed a detailed analysis of the SPEC benchmarks to understand the behavior that leads to emergencies. In this section, we illustrate that a few code regions in each benchmark are responsible for most of the emergencies associated with that benchmark.

3.1 Experimental Setup

An 8-way superscalar, out-of-order processor was configured with the parameters shown in Table 1. An identical configuration was used in earlier studies [3, 5]. Voltage emergencies are closely linked to the power delivery subsystem (PDS) of the chip. Figure 1 shows the impulse response of the second-order lumped model used in this paper. Our package model is based on the characteristics of the Pentium IV package [10]. In this model, the dominant resonant frequency of the PDS occurs at 100MHz with a peak impedance of 5mΩ. Finally, we assume peak current swings of 16-50A and noise-margin violations at $\pm$4% of a 1V supply. A modified version of Wattch [11] was used to estimate cycle-level current consumption and the voltage variation was calculated by performing a convolution of the current estimates with the impulse response of the PDS, as detailed in [3].

3.2 Identifying a code region with voltage emergencies

The first step towards understanding the relationship between application runtime behavior and voltage emergencies is to uniquely identify the signature of the emergencies, to understand whether the same signature of emergencies occurs repeatedly, and determine the frequency at which they occur. This code-region identification is required to provide the compiler with hints in order to eliminate the problematic

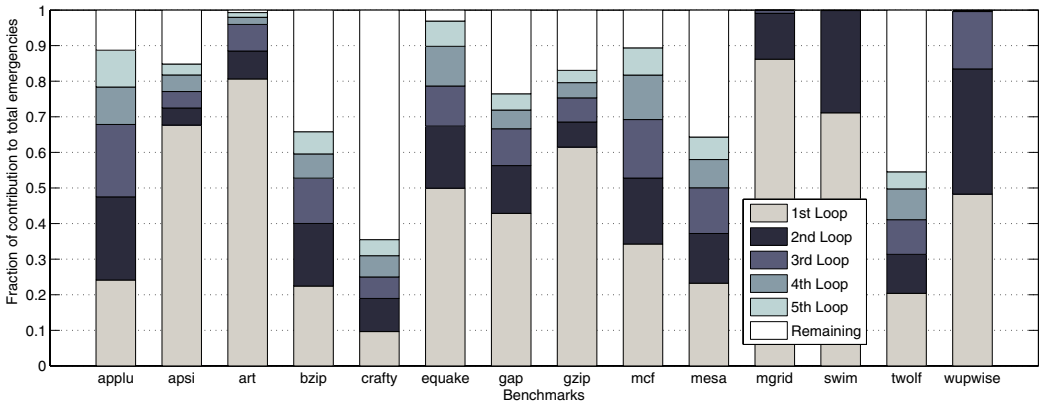

Figure 2: Contribution of top 5 loops to the emergencies for various SPEC benchmarks

region. Previous work by Hazelwood and Brooks [5] used the *last executed branch (LEB)* as a starting point for the compiler for identifying the candidate region for optimization. An earlier study by Joseph et al. [3] showed that periodic occurrences of high and low activity in the stressmark loop can lead to emergencies. Periodic behavior is generally associated with loops within applications. This observation was used as the starting point for our analysis.

Loops can be uniquely identified based on the sequence of instructions in the pipeline at any given instance. Loops are flagged with a specific type of emergency based on the sequence of instructions active at the time of the emergencies. These *emergency loops* represent the information which would be required by the compiler to optimize them, and hence, represent the number of times a compiler needs to be invoked. Our loop detection algorithm fails to perform procedural analysis and, hence, certain instructions are not classified with loops, which we call *procedural emergencies*.

Table 2 summarizes a few key statistics about the static loops present in the SPEC2000 benchmarks. Loops that are active at the time of emergencies are the loops that we are interested in. The third column of Table 2 tabulates the number of *emergency loops* identified, i.e. the number of static loops in which a voltage emergency occurs. Though the total number of loops ranges from 310 to 1806, the total number of *emergency loops* for each application is a small fraction of this total. This suggests that the runtime overhead of voltage-specific dynamic optimizations would be small. Most benchmarks have a small number of procedural emergencies and more than 90% of the emergencies can be associated with unique loops. The benchmarks *crafty* and *twolf* are the only two that do not follow this trend, with 33% and 25% of the voltage emergencies classified as procedural emergencies, respectively.

Since we are interested in the loops that incur the greatest numbers of voltage emergencies, we plot in Figure 2 the percentage of the total number of voltage emergencies occurring in the top five *emergency loops* for each SPEC benchmark. For each benchmark, around 2-5 loops account for more than 75% of the emergencies and, hence, we classify these loops as *hot loops*. Optimizing these hot loops can significantly reduce the overall number of emergencies. We further analyze the loops and code regions that cause the most voltage emergencies for each benchmark.

Identifying such problematic loops, understanding the characteristics of these loops, and investigating their interaction with other loops can provide insights into the sequences of events that lead to emergencies. This understanding is critical to finding perturbations to the code that will eliminate the emergencies.

Benchmark	Total Loops	Total Emergencies	Total Emergency Loops	Procedural Emergencies
applu	479	389897	13	0
apsi	718	21056	25	537
art	293	120885	13	0
bzip	383	403798	35	6181
crafty	1406	623977	302	190181
equake	423	174293	9	6942
gap	1806	243259	48	2115
gzip	310	67324	40	17
mcf	338	88828	23	620
mesa	536	279904	41	41
mgrid	411	1111152	32	75
swim	425	2332	4	0
twolf	1271	514970	81	138241
wupwise	425	42598	14	81

Table 2: Loops associated with emergencies

4. MICROARCHITECTURAL EVENT-BASED CATEGORIZATION OF EMERGENCIES

To employ a dynamic optimizer to mitigate future voltage emergencies in an emergency loop, we must categorize the types of emergencies encountered at runtime and use this categorization to determine the optimization to apply. We developed one such categorization algorithm and used it to analyze the emergencies occurring in the SPEC benchmarks. The microarchitectural events we consider are L1/L2/TLB misses, branch mispredicts and long latency operations. Our algorithm is based on a moving window containing the history of all events occurring within a certain time period before each emergency. The window size is dependent on the longest latency event: an L2 miss with a simulated latency of 300 cycles. Whenever an emergency occurs, a search of the event window is performed for the various events in a fixed priority order. The highest priority is given to L2 misses that stall the pipeline, as they are most likely to cause the longest stalls in the pipeline. We believe that a priority-based categorization is simple to implement and can directly point to the relevant optimization strategy (Section 5).

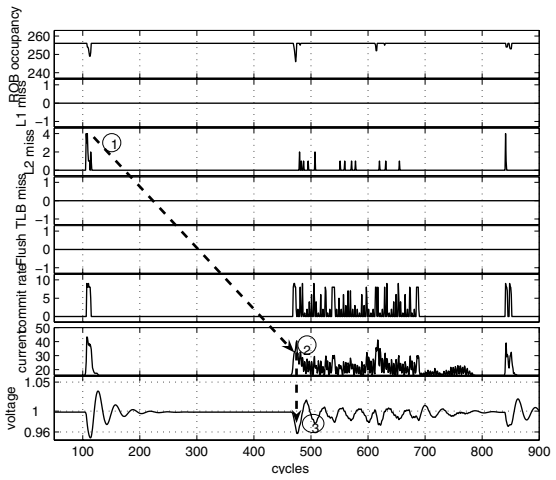

Figure 3: A snapshot of a loop in procedure: *smvp* of *equake*

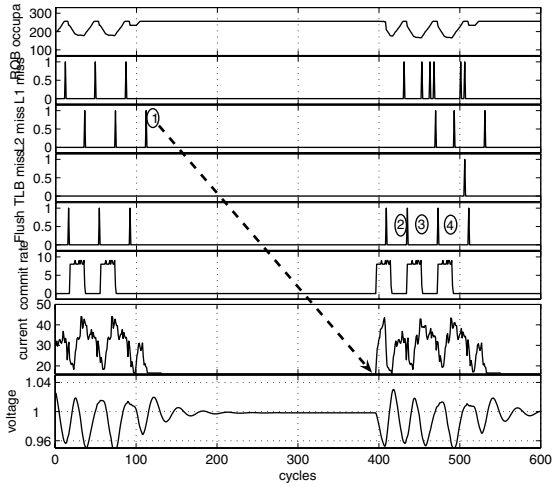

Figure 4: A snapshot of a loop in procedure: *match* of *art*

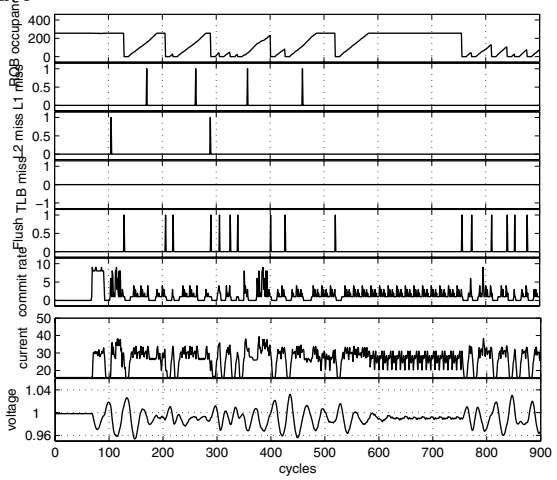

Figure 5: A snapshot of a loop in procedure: *longest_match* of *gzip*

Figure 3 shows a snapshot of pipeline activity for the topmost loop in *equake*. Pipeline statistics such as occupancy of reorder buffer and commit rate are depicted along with microarchitectural events: L1/L2/TLB misses and pipeline flushes. We considered several other microarchitectural parameters, such as number of entries in the Instruction Fetch Queue and Load/Store Queue, but none of these helped us to better characterize the analyzed emergencies. The current and voltage profile of the benchmark is also depicted by the lower two plots. In the figure, a presence of a long stall due to an L2 cache miss can be observed (shown by reference point 1). During the time it takes to service the L2 miss, pipeline activity ramps down as seen in the current profile. However, after the L2 miss data is available, functional units become busy and there is a sudden increase in activity (shown by reference point 2). This steep increase in current leads to a significant voltage drop (shown by reference point 3). Other events like TLB misses, L1 misses or flushes are not present in the pipeline during this window, which clearly suggests that the L2 miss in this code region caused the voltage fluctuation in *equake*. We now categorize such events into four distinct categories:

Memory Miss Event: Stalls can occur in the pipeline due to loads missing the L1 cache, and larger stalls occur when the loads miss the L2 cache. The large miss penalty associated with L2 or higher cache misses can drain the active instructions in the pipeline and can result in long periods of inactivity. It is important to note that an L2 miss that results in a pipeline stall may lead to an emergency, while an L2 miss that does not stall the pipeline will likely not. This period of inactivity following a miss is characterized by low current draw (as shown in Figure 3). A sudden increase in activity happens when the L2 miss returns, leading to execution of all dependent instructions. These bursts of activity following long period of inactivity must be avoided. Pre-fetching the loads associated with these emergencies can help remove the long stalls and hence avoid the voltage emergencies.

Pipeline Flush Event: Misprediction of branches leads to flushing the entire pipeline when the branch is resolved. This leads to a sudden decrease in activity following a flush event; however, a few cycles later, activity ramps up because of non-control instructions at the branch target. If the period of low and high current draw coincides with the periodicity of the package characteristic, resonance build up of voltage occurs. Figure 4 shows one such example in a snapshot of *art*. The L2 miss (shown by reference point 1) is responsible for the initial inactivity and subsequent increase in activity when it returns leading to a drop in voltage but the subsequent pipeline flushes (shown by reference points 2, 3, 4) occur periodically leading to further voltage drops. This snapshot also shows the presence of L1 misses but our detailed analysis showed that these were not responsible for these emergencies. For emergency loops with branch mis-predicts, improved static or hardware prediction could smooth out the impact of this noise.

Long Latency Event: A long chain of dependent floating-point operations, such as divides, can lead to long delays for dependent instructions, and hence pipeline stalls. Such events were detected in benchmarks like *applu*. One possible way to optimize these long latency events is to reschedule the code to schedule non-dependent instructions to avoid the long stalls. This can be achieved by loop unrolling.

Mixed Events: Certain applications display a complex interplay of events, and it is difficult to identify a single cause for the emergencies. We observed that periods of high

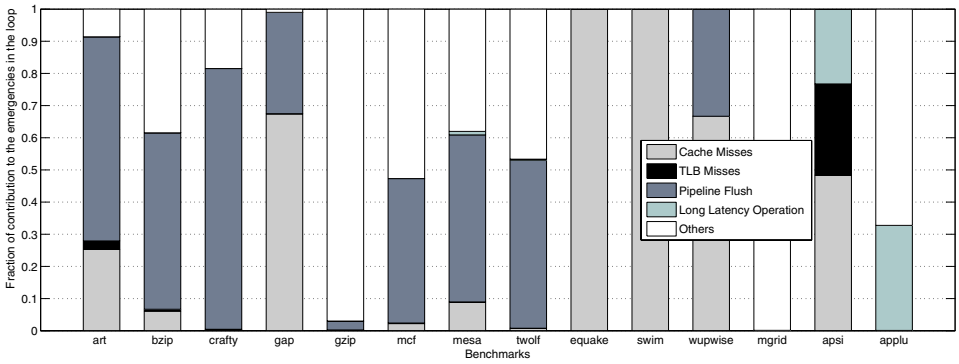

Figure 6: Distribution of various microarchitectural events to voltage emergencies for the top most loop

activity in loops leads to high-frequency noise in the voltage. Figure 5 shows one such example with high frequency noise occurring around the nominal voltage. In such cases, it is difficult to identify a single dominant cause for the emergency and a specific optimization to eliminate emergencies in the loop.

Figure 6 provides the distribution of microarchitectural reasons for emergencies in the top loop in each of the benchmarks. Other loops in the applications showed similar emergency breakdowns and we focus on the top loop for brevity. As seen from the figure, the two primary reasons for emergencies are the L2 misses and the pipeline flushes caused by branch mispredicts. *apsi, equake, gap, swim* and *wupwise* are examples of benchmarks with memory misses, L2 or TLB misses. Benchmarks *apsi* and *applu* show the presence of some emergencies attributed to long latency operations. We also note that some emergencies are attributed to multiple events. Finally, some emergencies are not easily characterized with these microarchitectural events and are placed in the *other* category: *mgrid* and *gzip* fall into this category.

5. OPTIMIZATION

The previous section categorizes emergency-causing events and links them to possible optimization techniques. Before developing a dynamic optimizer to apply these optimizations, we must understand whether the elimination of the identified microarchitectural events will actually reduce the number of voltage emergencies incurred during execution. As such, the results in this section are based on our categorization algorithm forcing changes in our machine simulator to remove the identified microarchitectural event in the targeted emergency loop. For example, if categorization indicates that an L2 miss is the cause of an emergency in a code region, then removing that L2 miss should result in that emergency disappearing. Our results support this basic premise, but the effect of the "optimization" is not that simple and localized, as we will describe.

Based on the analysis in Section 3, we applied appropriate optimization techniques and studied their effect on the emergencies within each emergency loop and across the whole application. We chose the same top loops in each benchmark, which we categorized in Section 3. For loops with L2/TLB misses as the cause for emergencies, we prefetched the loads/stores which are causing misses. The top loop in *apsi* had 32% of its emergencies attributed to TLB misses, and we prefetched the address translations causing those misses. For loops with branch mispredicts leading to emergencies, we provided perfect prediction that replaced the mispredicts.

The top loop in *twolf* is a very small loop nested inside an outer, larger loop. The emergencies are linked to the mispredicts in the outer loop and we applied the branch optimization to this loop. For loops with long latency operations, we reduced the latency of the long instructions. This method, done as a replacement to compiler driven code reordering, is sufficient to validate that these instructions in the code region cause the emergencies. For loops with several interesting events, we chose the most recurring event and performed the optimization related to that event. We did this in the interest of keeping our scheme simple, so that a runtime solution could be employed.

Table 3 shows the effects of different optimization techniques based on the distribution shown in Figure 6. The third column shows the effectiveness of the optimization in reducing the emergencies in that loop for each benchmark and the fourth column represents the overall effect of the optimization in the loop on the total emergencies across the entire benchmark. This table shows that any optimization can either have an isolated effect on the loop (*self-contained optimization*) or have secondary effects on loops close to the optimizing loop region (*spill-over optimization*). The secondary effects can be further divided into *positive spill-over*, in which the optimization for that particular loop altered the current signature of the application so as to remove emergencies from other loops, or *negative spill-over* in which the optimization removed emergencies from that loop but caused more emergencies elsewhere.

As can be seen in Table 3, the optimizations were successful in removing the emergencies in the code region. The extent of success with an optimizing event appears proportional to the extent that event was responsible for causing emergencies in the loop. For example, in *equake*, L2 misses are the primary cause of emergency. Hence, prefetching loads causing L2 misses removed most of the emergencies in the loop. On the other hand, in *apsi*, a variety of microarchitectural events contributed to emergencies in the top loop. Hence, prefetching the TLB misses helped, but resulted in less than optimal reduction in emergencies. Removing the long latency operation in *applu* was a self-contained optimization that did not have any effect outside of the loop. In *bzip*, change in the current signature of the application as a result of removing mispredicts had a significant *positive spill-over*(20%) effect and resulted in other emergencies disappearing as well. In contrast, *gzip* shows a *negative spill-*

Benchmark	Contribution of the loop to application's emergencies(%)	Reduction of Emergencies in the given loop (%)	Overall Reduction (%)	Spillover (%)
Optimization: Branch Prediction				
twolf	14.9	99.9	20.9	6.0
mesa	20.9	98.8	24.3	3.6
bzip	22.1	94.9	41.4	20.4
art	80.6	79.1	68.9	5.1
mcf	31.8	77.1	68.1	43.6
gap	10.8	52.7	6.1	0.41
crafty	9.7	22.1	1.9	-0.24
gzip	61.5	19	5.4	-6.3
Optimization: Prefetching Loads				
equake	47.9	97.75	52.8	5.9
swim	71.1	80.2	57.0	-0.02
wupwise	48.2	66.6	16.0	-16
applu	23.8	32.3	7.7	0.01
gap	10.7	7.52	0.7	-0.11
Optimization: Long Latency Operations				
applu	23.8	23.4	5.6	0.03
Optimization: Prefetching TLB misses				
apsi	65.9	26.4	17.7	0.31

Table 3: Effectiveness of various optimization schemes

over, in which emergencies in the loop reduced but more emergencies appeared elsewhere in the code region.

In summary, our results show that directed optimization could be successful overall, which suggests that our categorization of voltage emergencies is correct. However, the presence of *negative spill-over* also suggests that the process of optimization has to be continuously carried out, perhaps throughout the runtime of the application. The simplicity of the optimization mechanisms and the narrowness of code regions that need to be altered (for example, inserting prefetch instructions before particular load instructions) make it suitable for a dynamic compiler to perform them in real time without much overhead.

6. CONCLUSIONS AND FUTURE WORK

This paper presents a categorization of the various loops detected at the time of emergencies for SPEC benchmarks. This is a step towards understanding the required solution at the microarchitectural or software level to handle noise-margin violations. This study highlights that only a few loops (typically 2-5) are responsible for more than 75% of the emergencies across most of the applications.

Further characterization of code regions that experience voltage emergencies shows that certain microarchitectural events like cache misses and branch mispredicts lead to increase/decrease in activity of the pipeline leading to voltage fluctuations. Occurrence of such events at the resonant frequency of the system can worsen the voltage drop (as shown in *art*). We developed a simple but effective categorization scheme to highlight the prominent cause for emergencies in the code regions contributing most of the emergencies. To validate our characterization we applied simple solutions like prefetching loads, branch prediction or TLB prefetching, which were generally effective in decreasing the occurrence of voltage violations across our applications.

7. REFERENCES

[1] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors," 2005.

[2] E. Grochowski, D. Ayers, and V. Tiwari, "Microarchitectural Simulation and Control of di/dt-induced Power Supply Voltage Variation," in *Int'l Symposium on High-Performance Computer Architecture*, 2002.

[3] R. Joseph, D. Brooks, and M. Martonosi, "Control Techniques to Eliminate Voltage Emergencies in High Performance Processors," in *Int'l Symposium on High-Performance Computer Architecture*, 2003.

[4] M. D. Powell and T. N. Vijaykumar, "Pipeline muffling and a priori current ramping: architectural techniques to reduce high-frequency inductive noise," in *Int'l Symposium on Low Power Electronics and Design*, 2003.

[5] K. Hazelwood and D. Brooks, "Eliminating Voltage Emergencies via Microarchitectural Voltage Control Feedback and Dynamic Optimization," in *International Symposium on Low-Power Electronics and Design*, August 2004.

[6] M. Powell and T. N. Vijaykumar, "Exploiting Resonant Behavior to Reduce Inductive Noise," in *Int'l Symp. on Computer Architecture*, Jun 2004.

[7] M. Toburen, "Power Analysis and Instruction Scheduling for Reduced di/dt in the Execution Core of High-Performance Microprocessors," Master's thesis, NC State University, USA, 1999.

[8] H.-S. Yun and J. Kim, "Power-aware Modulo Scheduling for High-Performance VLIW Processors," in *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001, pp. 40–45.

[9] W. El-Essawy and D. Albonesi, " Mitigating Inductive Noise in SMT Processors," in *International Symposium on Low Power Electronics and Design*, August 2004.

[10] K. Aygun, M. J. Hill, K. Eilert, R. Radhakrishnan, and A. Levin, "Power Delivery for High-Performance Microprocessors," *Intel Technology Journal*, vol. 9, no. 4, Nov. 2005.

[11] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a Framework for Architectural-level Power Analysis and Optimizations," in *27th Annual International Symposium on Computer Architecture*, 2000.