Eliminating Voltage Emergencies via Microarchitectural Voltage Control Feedback and Dynamic Program Modification

Kim Hazelwood

TR-18-03

2003

Computer Science Group
Harvard University
Cambridge, Massachusetts

Eliminating Voltage Emergencies via Microarchitectural Voltage Control Feedback and Dynamic Program Modification

Kim Hazelwood
Division of Engineering and Applied Sciences
Harvard University

Abstract

As processor clock gating becomes more and more prevalent, the resulting processor current fluctuations increase the chance of the power supply violating its operating voltage range. Today, low-power research has focused on hardware mechanisms to reduce the chances of these voltage emergencies. While these hardware solutions are very effective at reducing di/dt to an acceptable range, they do so at a performance penalty to the executing program. On the other hand, a compiler is well-equipped to rearrange instructions such that current fluctuations are less dramatic, while minimizing the performance implications. Furthermore, a software-based dynamic optimizer can eliminate the problem at the source-code level during program execution. This paper proposes complementing the hardware techniques with additional compiler-based techniques for eliminating power virus loops, and other recurring power problems. We propose that hardware solutions remain intact, but we extend them to additionally provide feedback to the dynamic optimization system, which can provide a permanent solution to the problem, often without affecting the performance of the executing program. We found that recurring voltage fluctuations do exist in the SPEC-cpu2000 benchmarks, and that given very little information from the hardware, a dynamic optimizer can locate and correct many of the recurring voltage emergencies.

1 Introduction

Low-power architecture has been an active area of research in recent years. Yet, many of the efforts at reducing processor power have the unfortunate side-effect of causing current oscillations within the processor. These oscillations are typically referred to as the *di/dt problem*, because this change in current may cause a swing in the source voltage. The di/dt problem can result in source voltages that violate the minimum or maximum voltage threshold for which the processor was rated. These threshold violations can cause timing problems, incorrect calculations, or in extreme cases, can cause damage to the microprocessor.

Recent research has focused on hardware mechanisms to reduce the chances of voltage emergencies [4, 5]. While these hardware solutions have been shown to be effective at reducing di/dt to an acceptable range, the executing program incurs performance penalties as a result. Joseph et al. [5] showed an example of a software code loop that approximated a large voltage swing. The swing was reduced by allowing the hardware to turn on or off functional units to control the necessary current. Yet, this software code loop provides the motivation for our work because if such loops exist in real applications, then it seems logical to apply a permanent solution at the application level, and therefore limit the performance penalty of activating control hardware.

We claim that hardware-based solutions work well for intermittent voltage emergencies that may arise, but a repeated voltage emergency loop is best handled by a compiler. A compiler typically has several options when choosing the order of instructions, and often, many of the options result in equally performing software. Therefore, in the case of the voltage-emergency loop, it may be the case that the compiler could've rearranged the instructions as to avoid the voltage emergency, without impacting performance.

Currently, compilers do not account for voltage fluctuations when scheduling instruction sequences. While techniques can and have been developed to produce power-efficient code by the static compiler, it would be difficult to extend these static optimiza-

tions to solve the di/dt problem. Currently, there is a general lack of understanding about software sequences that result in voltage fluctuations. Yet, even if algorithms were developed for locating potentially dangerous instruction sequences, the decision whether or not to intervene would depend on the operating voltage range of the target processor, which typically is not known at compile time. Finally, static techniques would not avoid all voltage emergencies, because many of the emergencies occur due to dynamic instruction sequencing, which is difficult to predict prior to program execution.

Therefore, we propose the use of a dynamic optimizer to solve the problem of repeating voltage emergencies. A dynamic optimizer has the ability to effectively balance the performance/power tradeoff, and has the added benefit of being able to know in real time when a voltage emergency occurs. Additionally, most dynamic optimizers already optimize and cache the frequently-executed portions of a program at the granularity of *code traces*—dynamic instruction sequences that span procedure call and branch boundaries. Therefore, this system can correct problems that span software module boundaries. Reoptimization is furthermore simplified, requiring only a replacement of the offending trace in the software-based code cache.

Finally, by operating in a *lazy optimization* mode, the dynamic optimizer can wait until it is informed by the hardware of a voltage problem (after the hardware activates control mechanisms to eliminate the emergency), and it can then re-optimize and cache a version of the loop that exhibits more voltage stability. Therefore, in the ideal case, only one iteration of a power-virus loop would require hardware intervention, and the remaining iterations would be executed from the software-based dynamically-optimized code cache.

In this paper, we propose a combined compiler-microarchitectural approach for handling the di/dt problem. The contributions are as follows.

- A description of a collaborate hardware/software approach to the di/dt problem.
- A characterization of the voltage fluctuations within the SPECcpu2000 benchmark suite assuming modern technology, as well as an extrapolation to future processors over the next 15 years.
- Techniques for mapping voltage emergencies back to original source code.
- Compiler-based techniques for solving voltage emergencies at the application level.

The remainder of the paper is organized as follows. Section 2 provides a detailed explanation of the di/dt problem, and discusses the prior work in the the area. Section 3 introduces a collaborative design for handling both intermittent and repeated voltage emergencies. Section 4 characterizes the number and type of voltage emergencies that occur within the SPECcpu2000 benchmarks, both present day and over the next 15 years. Section 5 explores the problem of mapping voltage emergencies back to source code, and proposes techniques for reducing hardware-to-software communication during an emergency. Section 6 discusses three compiler algorithms for altering a problematic area of source code to alleviate a voltage-emergency region. Finally, Section 7 concludes and Section 8 discusses ideas for future work.

2 Background and Prior Work

Dramatic changes in processor current over a relatively short time frame are referred to as *the di/dt problem*. These changes in current are problematic because they result in corresponding voltage changes in the power delivery system. Processors require a relatively stable power source, and typically cannot tolerate supply voltage variations exceeding 5%. Therefore, if the changing demands in processor current result in supply voltage variations greater than 5%, this can result in a malfunction within the CPU.

Interestingly, a major contributor to the di/dt problem is the recent focus on low-power techniques. Many of the proposed methods for reducing dynamic power work by turning off idle resources. However, the simple act of turning on and off microprocessor elements results in dramatic variations in the amount of current required by the processor as a whole each cycle.

Currently, voltage emergencies aren't a critical problem because hardware designers have taken a conservative approach when designing power supply systems and CPUs. However, future trends may make it much more difficult to design conservative systems. In fact, the International Technology Roadmap for Semiconductors (ITRS) lists noise management (di/dt, ground bounce, etc) as one of their *grand challenges* for the 2008+ time frame [1].

There are three major contributors to the increasing seriousness of the di/dt problem. The first con-

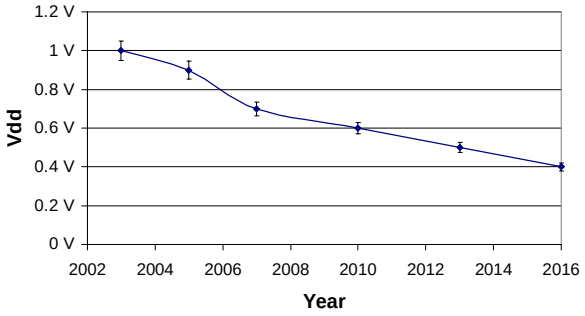

Figure 1: Future processor voltage trends.

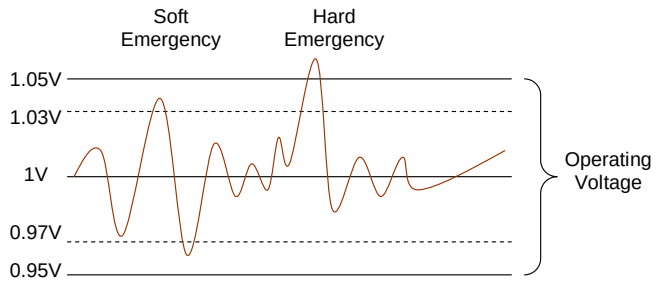

Figure 2: Soft and hard voltage emergencies.

tributor is the low-power design trend of aggressively clock gating more and more portions of the microprocessor. The continuous change in the state of various internal processor elements results in dramatic changes in processor current. As the number of independently gated processor elements increases, the problem will become much more serious.

The second contributor is the decreasing voltage trends in high-performance microprocessors. Figure 1, extracted from the ITRS, shows the decreasing voltages for microprocessors over the next 15 years. It is important to also note the decreasing operating voltage range, which is calculated as +/- 5% of the source voltage. Therefore, this figure shows that while operating voltages will drop from 1.0V to 0.4V, the tolerance for processor voltage variations will decrease from +/- 0.05V to +/- 0.02V.

Finally, as more and more features become incorporated into future high-performance microprocessors, the overall device current is expected to increase [1]. This increase in current results in the potential for larger per-cycle current variation, or increased di/dt.

Several solutions have been proposed for reducing processor current and voltage fluctuations. In 1999, Toburen et al. [6] proposed heuristics for reducing the number of bit-flips between successive instructions in the execution core of high-performance microprocessors. In 2002, Grochowski et al. [4] proposed disabling and enabling functional units to reduce voltage variation based on a complex calculation of the voltage, and in 2003, Joseph et al. extended this idea to use on-chip voltage sensors, rather than calculation, as part of the voltage control mechanism. Furthermore, they were able to simplify the problem by focusing on voltage ranges rather than specific voltage values. To do so, they defined two thresholds. A *control threshold* (defined as +/- 3% of the source voltage) triggered corrective action by the control system. Ideally, this action prevented the power supply from exceeding the *operating voltage threshold* (+/- 5% of the source voltage), which could result in processor malfunction. Figure 2 distinguishes these thresholds, and also aids in distinguishing *soft emergencies* from *hard emergencies*. Soft emergencies occur when the supply voltage violates the control threshold and triggers the hardware voltage control mechanisms, while hard emergencies occur when the microprocessor's operating voltage threshold is violated.

3 Collaborative Architecture

As mentioned in Section 2, prior research has proposed hardware-based control mechanisms for handling voltage emergencies. These control mechanisms respond whenever the source voltage moves outside of a predefined control range. The range is defined so as to minimize false alarms, while maximizing the time available to stabilize the voltage. As the source voltage moves outside of the control threshold range, the processor reacts by performing one of two actions. If the emergency is resulting from an abnormally low current draw, the processor responds by producing *phantom firings* of one or more functional units. Furthermore, if the current is abnormally high, the processor disables one or more functional units. While these techniques effectively correct impending voltage emergencies, the latter case does so at the expense of program performance, while the former case wastes power.

Therefore, we extend the hardware mechanisms to additionally provide feedback to a software-based dynamic optimization system. This runtime system can then determine whether a similar voltage emergency has occurred in the past, making this region of code a candidate for re-optimization.

Figure 3 provides a high-level view of the proposed architecture. The previously-proposed

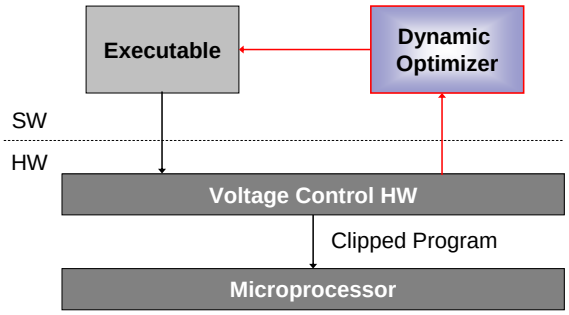

Figure 3: Collaborative architecture.

hardware-based voltage control mechanisms are shown in gray, while the extensions are outlined in red. As shown in the figure, the voltage control hardware monitors execution of the application. Upon detection of an imminent voltage emergency, the control mechanism intercepts execution and performs various actions to correct the emergency. Simultaneously, the control mechanism provides feedback to the dynamic optimizer relaying pertinent information about the state of the processor during the emergency, and the instructions that are currently in-flight or recently completed.

Using the feedback from the voltage control mechanism, the dynamic optimizer performs the following actions:

1. Determining the location in the original source code that is the apparent cause of the voltage fluctuation.
2. Deciding whether this region of code has caused a voltage fluctuation in the past, and is therefore a candidate for region modification.
3. Determining the best plan of action for altering the code region.
4. Performing the optimization.
5. Inserting the new code into the code cache, thereby making it the default version for the remainder of execution.

In summary, the proposed architecture is a collaborative approach between two components – one hardware-based and one software-based component. The hardware component includes the voltage control mechanisms proposed by Joseph [5], with extensions to provide feedback to the other software-based component—the dynamic optimizer—which may apply a permanent solution to the executing program.

Execution Core	
Clock Rate	3.0 GHz
Instruction Window	256-RUU, 128-LSQ
Functional Units	8 iALU, 2 iMul/iDiv
	4 fpALU, 2 fpMul/fpDiv
	4 memory ports
Front End	
Fetch Width	8 instructions
Decode Width	8 instructions
Branch Penalty	10 cycles
Branch Predictor	Combined - 64Kb chooser
	64Kb bimodal/64Kb gshare
BTB	1K entry
RAS	64 entry
Memory Hierarchy	
L1 D-Cache	64KB 2-way
L1 I-Cache	64KB 2-way
L2 I/D-Cache	2KB 4-way, 16 cycle latency
Main Memory	300 cycle latency

Table 1: Simulated processor parameters.

4 Characterizing Voltage Emergencies

Before moving forward with the proposed architecture, it is first important to gain an understanding of the processor current fluctuations and voltage profile resulting from the execution of modern benchmark programs. This will allow us to solidify the motivation for voltage control mechanisms, and more specifically, determine whether or not applications experience recurring voltage variation trends throughout execution (within loops, for example) that would be best handled by a runtime system such as a dynamic optimizer.

4.1 Experimental Framework

To characterize voltage trends in the SPECcpu2000 benchmarks, a framework similar to [5] was used. This included a modified version of the Wattch 1.03 architectural-level power simulator [3] configured for a 1.0 V supply voltage. Wattch is based on the SimpleScalar toolset [2] and we simulated an aggressive 8-way superscalar out-of-order processor configured with the parameters shown in Table 1.

Wattch was modified to calculate the voltage variation each cycle. This was done by performing a convolution of the current estimates (produced by Wattch) and an impulse response generated by an

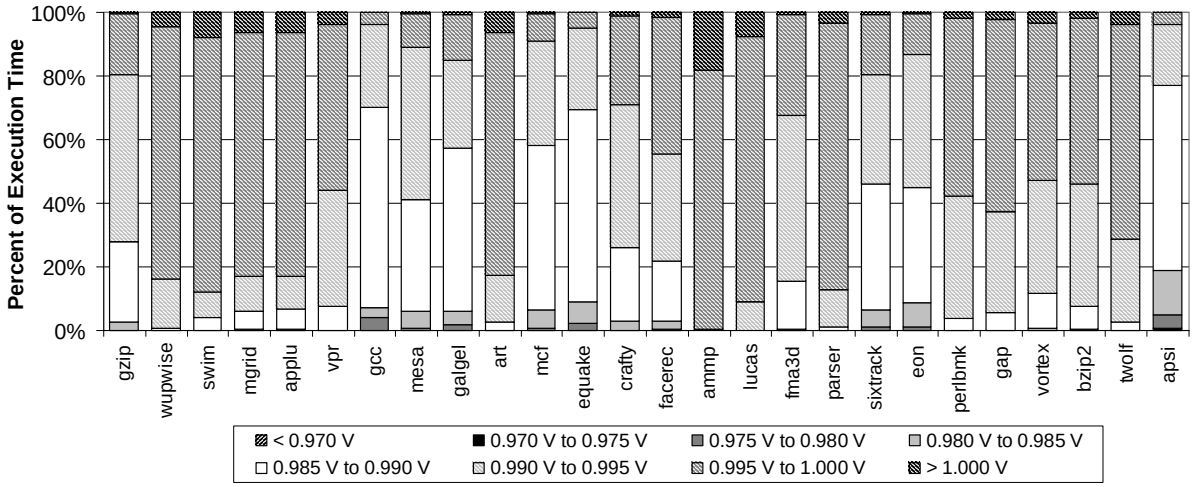

Figure 4: Voltage classification of SPECcpu2000.

Target Impedance	Soft Emergencies	Hard Emergencies
100%	0 / 26	0 / 26
200%	5 / 26	0 / 26
300%	20 / 26	0 / 26
400%	24 / 26	3 / 26

Table 2: Number of SPECcpu2000 benchmarks experiencing voltage emergencies.

external perl script. The convolution is calculated using the following equation

$$v(t) = \sum_{i=0}^{t} h(i) * i(t-1) \qquad (1)$$

where i(t) is the instantaneous current and h(i) is the impulse response. This calculation was performed each cycle, based on Wattch's per-cycle power calculation and the simple $I = P/Vdd$ transformation. After calculation, per-cycle voltages were output to a voltage trace file. Finally, the voltage traces were sent through customized analysis tools to produce the results presented in the following sections.

4.2 Voltage Emergency Results

We began evaluating the importance of the voltage emergency problem by first establishing the number of voltage emergencies that occur in the SPECcpu2000 benchmarks using modern technology, then extrapolating to future technology. Using our modified Wattch simulator to output a trace of the supply voltage values each cycle during the execution of each benchmark, we categorized the individual voltage trace values during benchmark execution. Figure 4 shows the resulting breakdown of supply voltage values on the modern-day microprocessor specified in Table 1. From this figure, the main observation is that while none of the SPECcpu2000 benchmarks result in voltage emergencies on modern-day processors, individual benchmarks vary greatly in terms of their voltage stability profile. For example, while `ammp` spends 82% of its execution in one centralized voltage category (0.995V–1.000V), benchmarks like `facerec` and `sixtrack` spend no more than 30% of their execution in one category. Furthermore, `apsi` stands out as the benchmark that comes closest to a lower-bound voltage violation[1].

Next, we look at extrapolating the voltage analysis to future microprocessors. We do so using methodology similar to [5], where we assume that modern technology allows the power supply to match the target impedance necessary to avoid voltage emergencies. We then look at the case where the necessary impedance is 200%, 300%, and 400%

[1]Another more subtle observation to make from Figure 4, is that we have come very close to replicating the results of prior work [5], upon which our ideas are built. Therefore, we can feel more comfortable that we are making a fair comparison to prior work.

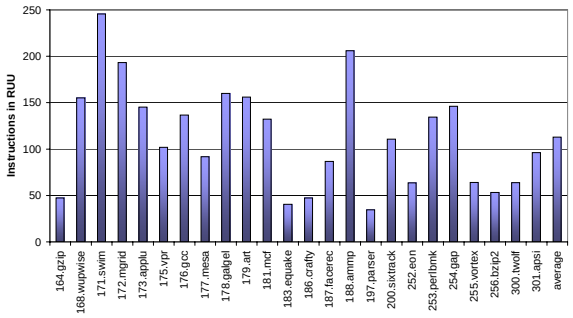

Figure 5: Average number of instructions in the RUU during a voltage emergency.

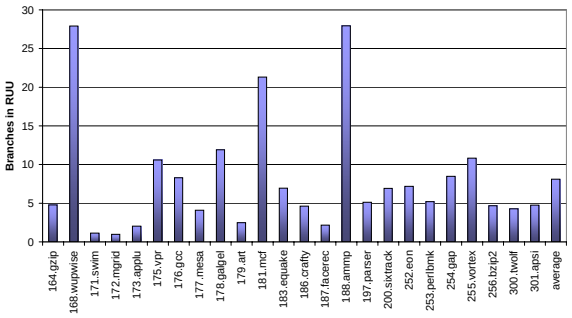

Figure 6: Average number of branches in the RUU during a voltage emergency.

of what the power supply can attain. This situation would happen if either (a) power supply designers can no longer match the required impedance due to technological limitations, or (b) voltage control mechanisms mature to the point where microprocessor developers feel comfortable shipping their products with less expensive, less conservative power supplies.

Table 2 shows the number of soft and hard emergencies that will occur at these impedance values. We see that while it's not until 400% target impedance that we actually experience voltage emergencies, at only 200% impedance, almost 20% of the benchmarks will trigger the voltage control system. Since it doesn't matter whether a real voltage emergency would've occured for the hardware-based control system to engage, performance implications would undoubtedly be experienced after the increase to 200% impedance. Fortunately, the dynamic optimization system would also permanently alleviate the recurring false alarms, reducing the performance affects of previously-proposed hardware-only techniques.

5 Mapping Emergencies to Source

As the design is predicated on hardware-initiated feedback to a dynamic optimizer, the next crucial question becomes: *After an emergency, what information should the hardware provide to the dynamic optimizer?* There are many instructions in the processor at any given time, therefore, the feedback mechanism must minimize the amount of information that it sends back to the dynamic optimizer, while still providing enough information to correct the problem.

We explored several options when determining the granularity of data to be sent to the runtime system. In this section, we investigate various means for pruning the instructions in the processor in order to determine an effective working set to communicate to the runtime system. Each option is evaluated in terms of the amount of data communicated, and the possible flaws of each pruning algorithm.

Baseline To form a baseline for comparison, we began by investigating the option of marking all instructions in SimpleScalar's Register Update Unit (RUU) as potential causes of the voltage emergency, and communicating all instructions back to the dynamic optimizer. Figure 5 shows the average number of instructions in the RUU during a voltage emergency, for each of the SPECcpu2000 benchmarks. Overall, an average of 113 instructions must be communicated to the runtime system using this heuristic. Clearly, the major weakness of this heuristic is its communication overhead. It is inefficient and unrealistic to design a feedback system that typically communicates 113 instruction addresses to a software system. Therefore, *address pruning* will clearly be necessary.

Instructions vs. Branches The next heuristic for pruning the information sent to the runtime system we explored was to focus on the branch instructions in the RUU. The logic behind this heuristic is that we can identify loops using only branch instructions. The runtime system can then focus effort on optimizing that particular loop. Figure 6 shows the average number of branch instructions in the RUU during a voltage emergency, for each of the SPECcpu2000 benchmarks. In this case, an average of 8 instructions must be communicated to the software system, while individual benchmarks vary from 1 branch (as in the case of `swim` and `mgrid`) to 27

Benchmark	Failures	Total	Percent
gzip	59	57376	0.10%
wupwise	0	54	0%
swim	0	218193	0%
mgrid	43809	176668	24.80%
applu	6515	119133	5.47%
vpr	99	45789	0.22%
gcc	11	6346	0.17%
mesa	4579	98509	4.83%
galgel	8	58914	0.01%
art	0	59133	0%
mcf	0	3525	0%
equake	0	119753	0%
crafty	3938	201847	1.95%
facerec	1	99140	0.00%
ammp	0	94	0%
parser	201	28049	0.72%
sixtrack	38	55234	0.07%
eon	1882	306698	0.61%
perlbmk	0	35	0%
gap	1	6528	0.02%
vortex	469	139072	0.34%
bzip2	0	1284	0%
twolf	1323	98947	1.34%
apsi	14	241056	0.01%

Table 3: Number of pruning failures occurring during voltage emergencies in SPECcpu2000. *Failures* shows the number of voltage emergencies where all of the data was pruned (i.e. no branches existed in the RUU.) *Total* presents the number of emergencies that occurred per benchmark. *Percentage* calculates $failures/total$ to depict the percentages of emergencies resulting in branch pruning failures.

branches (in `wupwise` and `ammp`.) While communicating eight instruction addresses is certainly more reasonable than 113, the worst case of 27 addresses still seems excessive. Furthermore, the best case of 1 address begs the question: *Is it ever the case that there are no branches in the RUU to be communicated to the dynamic optimizer?* A pruning model that prunes all data is clearly problematic! Table 3 shows how often such over-pruning occurs with this model. As shown in the table, the major outlier is `mgrid`, where 25% of emergencies occur while no branches exist in the RUU. It is likely the case that the problematic area(s) of `mgrid` are located inside a large loop. Therefore, the back-edge branch has already retired from the processor by the time the voltage emergency occurs.

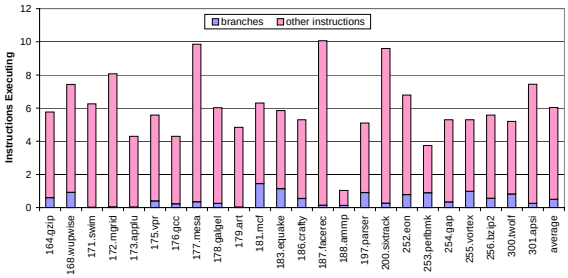

Figure 7: Instructions in-flight during a voltage emergency.

Executing vs. Recently Completed The next interesting question we must answer is whether the currently-executing instructions were likely the cause of the voltage emergency, or whether the recently-completed instructions were more likely to blame. More importantly, we must determine whether the characteristics of program loops make it inconsequential which instruction state we focus on, as the dynamic optimizer would likely optimize the same region of code regardless.

Figure 7 shows the average number of in-flight instructions during a voltage emergency. As we can see, roughly 6 instructions on average would need to be communicated to the dynamic optimizer were we to select this heuristic. Interestingly, this heuristic provides a much more stable information exchange, as all of the benchmarks average less than 5 instruction variance from the mean of 6 instructions. Furthermore, in terms of implementation, it is much easier for the processor to record the in-flight (currently executing) instructions than the recently completed instructions.

Hybrid–Last-Executed Branch In the ideal case, the hardware would only need to communicate a single program counter to the dynamic optimizer to trigger an update of the problematic code region. This would allow the hardware to communicate via a single performance counter register. In this case, the question then becomes, *which single instruction is likely to provide the most information about the problematic area?* One solution would be the first branch instruction encountered as we scan the executing and completed instruction stream in the microprocessor.

This heuristic can be considered a hybrid of the previous pruning algorithms, because we first eliminate non-branch instructions from the candidate list, then we eliminate the least-recently completed branches, until we're left with the most recently ex-

Benchmark	Distinct	Total
gzip	47	57376
wupwise	4	54
swim	5	218193
mgrid	24	176668
applu	18	119133
vpr	86	45789
gcc	64	6346
mesa	102	98509
galgel	7	58914
art	11	59133
mcf	37	3525
equake	7	119753
crafty	329	201847
facerec	14	99140
ammp	1	94
parser	278	28049
sixtrack	88	55234
eon	40	306698
perlbmk	4	35
gap	45	6528
vortex	197	139072
bzip2	12	1284
twolf	57	98947
apsi	37	241056

Table 4: Number of distinct voltage emergencies, as indicated by the LEB register.

ecuted branch. In hardware, this heuristic could be easily implemented by extending the branching hardware to write the source PC of the branch to a Last Executed Branch register (LEB). Then, when a voltage emergency occurs, the contents of the LEB register would be communicated to the dynamic optimizer.

Using this heuristic, we then characterized the number of *distinct voltage emergencies* that occurred during program execution. Table 4 shows the number of different branch instructions found in the LEB register during a voltage emergency. Interestingly, the column labeled *Distinct* can actually be equated with the number of times that the dynamic optimizer would need to intervene to correct voltage emergences, while the *Total* column is the number of times that hardware must intervene if the dynamic optimizer was not part of the solution. From this table, we see that the software would intervene between 1 and 329 times during benchmark program execution, for the cases of `ammp` and `crafty`, respectively. On the other hand, previously-proposed hardware-only solutions would need to intervene between 35 and 306,698 times, for `perlbmk` and `eon`, respectively. This clearly indicates the existence of source-level voltage emergency loops that are well-suited for a dynamic optimizer.

6 Compiler-Based Solutions

Now that we have established the existence of repeatable power problems, the next problem becomes: *What can a compiler do to correct a di/dt problem at the source-code level?* To answer this question, we begin by defining the salient features of an instruction sequence that tends to result in large current swings.

Prior work [5] pointed out that the most problematic processor current profiles include successive periods of high and low processor activity. It is when these high and low durations approach the resonant frequency of the processor that the problem becomes more serious. To prove their point, Joseph et al. developed an artificial application that was hand-tuned to simulate periods of high and low activity that matched the processor's resonant frequency. It was this synthetic benchmark that provided the initial motivation for this work, as the source code consisted of a single loop body, but caused thousands of voltage emergencies during execution.

Unfortunately, this synthetic benchmark is unavailable, and all attempts to recreate a similarly performing benchmark have thus far been unsuccessful. Nevertheless, we will now discuss a set of new and existing compiler optimizations that should alleviate these periods of high and low activity.

Software Pipelining A widely used compiler algorithm for increasing the instruction-level parallelism of cyclic code is software pipelining. By unrolling loops and overlapping the execution of instruction sequences from several loop iterations, the instructions can be scheduled more tightly. It is the removal of the data dependence on the loop iteration variable that allows the most amount of code reordering. Typically, the result of software pipelining is that n-iterations of a loop will be combined to form one larger loop iteration. The value of n is determined heuristically.

The nature of the software pipelining algorithm has the interesting side-effect of allowing high-activity periods in one loop iteration to be combined with low-activity periods of the next loop iteration. This will lead to a more stable, high-activity sequence of instructions that will often complete faster than non-pipelined instruction sequences.

While the hardware feedback mechanism does specify the last-executed branch, there is no guarantee that this branch is a loop back-edge. It may in fact be a forward branch such as an if-then-else statement or a function call. Therefore, the next two algorithms are designed to be applied to both cyclic and acyclic code regions.

Code Motion Often, when the static compiler schedules instructions, it has several options for scheduling an instruction without affecting application performance. Therefore, the compiler may inadvertently create regions of high and low processor activity due to its predefined settings for scheduling instructions in the event of a performance tie. By recognizing these schedule slips, a dynamic optimizer can later apply *code motion* to move instructions from high to low processor utilization regions, thereby alleviating a voltage emergency without affecting the runtime performance of the application.

Instruction Padding The final optimization can also be applied to acyclic regions when code motion is not possible. Instruction padding occurs when we insert unnecessary calculation into a low-utilization code region. This transformation essentially masks the low-utilization region in a manner similar to the hardware technique of *phantom firings* of the functional units. Instruction padding is not used in traditional compiler optimization phases due to the fact that it has no performance benefits. While the processor will ideally schedule the unnecessary instructions off the critical path on idle functional units, it may in some cases degrade performance of an instruction sequence, and therefore should be considered a last resort.

7 Conclusions

Voltage emergencies are becoming a larger and larger problem in the design of microprocessors. Aggressive clock gating, decreasing voltage, and increasing current trends are aggravating the problem. In this paper, we propose a hybrid hardware/software approach to solving the di/dt problem. We use hardware to correct intermittent di/dt problems, but provide feedback to a software-based runtime system for correcting repeatable problems at the source-code level. We found that we can effectively minimize hardware-to-software communication using our novel Last-Executed Branch (LEB) algorithm, and we can balance voltage variations using a combination of software pipelining, code motion and instruction padding.

8 Future Work

Applying the compiler optimizations to the power virus example that was used in prior research to evaluate the performance and voltage impact is the next step[2]. After evaluating the algorithms, the next logical step in this research is to modify a dynamic optimizer to enable it to receive hardware feedback. It can then respond to this feedback by applying the optimizations described in Section 6. Interestingly, this infrastructure can be used for much more than reducing di/dt. As temperature-related research closely mirrors di/dt research, we can apply this architecture to the temperature domain, allowing the optimizer to reschedule *hot* sections of code. Furthermore, low-power versions of commonly executed portions of code can be generated, cached, and executed all at runtime.

Acknowledgments

I wish to acknowledge the continuous guidance and assistance of Professor David Brooks throughout the duration of this research. I also wish to thank Russ Joseph for his guidance during the process of replicating his infrastructure.

References

[1] International technology roadmap for semiconductors. *Semiconductor Industry Association*, 2002.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, February 2002.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture (ISCA)*, 2000.

[4] E. Grochowski, D. Ayers, and V. Tiwari. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *8th International Conference on High Performance Computer Architecture*, pages 7–16, 2002.

[5] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *9th International Conference on High Performance Computer Architecture*, 2003.

[6] M. C. Toburen. Power analysis and instruction scheduling for reduced di/dt in the execution core of high-performance microprocessors. Master's thesis, North Carolina State University, 1999.

[2] Interestingly, the difficulty we encountered in attempting to recreate the exact effects of the published instruction sequence is reassuring, as it shows that even slight variations of the original instruction sequences can alleviate a voltage emergency!