Eliminating Voltage Emergencies via Microarchitectural Voltage Control Feedback and Dynamic Optimization

Kim Hazelwood and David Brooks
Division of Engineering and Applied Sciences
Harvard University
{hazelwood,dbrooks}@eecs.harvard.edu

ABSTRACT

Microprocessor designers use techniques such as clock gating to reduce power dissipation. An unfortunate side-effect of these techniques is the processor current fluctuations that stress the power-delivery network. Recent research has focused on hardware-only mechanisms to detect and eliminate these fluctuations. While the solutions have been effective at avoiding operating-range violations, they have done so at a performance penalty to the executing program.

Compilers are well equipped to rearrange instructions such that current fluctuations are less dramatic, with minimal performance implications. Furthermore, a dynamic optimizer can eliminate the problem at run time, avoiding the difficult task of statically predicting voltage emergencies.

This paper proposes complementing existing hardware solutions with additional run-time software to address problematic code sequences that cause recurring voltage swings. Our proposal extends existing hardware techniques to additionally provide feedback to a dynamic optimizer, which can provide a long-term solution, often without impacting the performance of the executing application.

We found that recurring voltage fluctuations do exist in the SPEC2000 benchmarks, and that given very little information from the hardware, a dynamic optimizer can locate and correct many of the recurring voltage emergencies.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance; C.0 [**Computer Systems Organization**]: General—*hardware/software interfaces*

General Terms

Performance, Design, Experimentation

Keywords

Power-aware computing, Hardware-software co-design, dI/dt, Voltage emergencies.

ISLPED'04, August 9–11, 2004, Newport Beach, California, USA.

1. INTRODUCTION

Recent efforts at reducing processor power have the unfortunate side-effect of causing large current variations within the processor, referred to as the *dI/dt problem.* Due to parasitic inductance in the power-supply network, current oscillations may cause an undesirable swing in the processor's supply voltage [10]. The dI/dt problem (also referred to as voltage emergencies) can result in supply voltages that violate the minimum or maximum voltage thresholds for the processor. This can potentially cause timing problems in a microprocessor and result in incorrect calculations [5].

Researchers have focused on hardware mechanisms to characterize, detect, and eliminate voltage emergencies [6, 7, 8, 9]. While these solutions have been effective at reducing dI/dt to the operating range of the processor, the executing program incurs performance penalties as a result. Joseph et al. [7] showed an example of an instruction loop that resulted in a large voltage swing. The swing was reduced by allowing the hardware to turn on or off functional units to control the necessary current. Their instruction sequence 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, but a loop incurring repeated voltage deviations is best handled by a compiler. A compiler typically has several options when choosing the order of instructions, and many of the options result in equally performing software. Therefore, in the case voltage-emergency loops, the compiler may be able to rearrange the instructions to avoid the voltage emergency without impacting performance.

Currently, static compilers do not account for voltage fluctuations when scheduling instruction sequences. While techniques exist for producing power-efficient code by the static compiler, extending these static optimizations to solve the dI/dt problem is difficult because there is a general lack of understanding about instruction sequences that result in voltage fluctuations. Furthermore, even if algorithms were developed for locating potentially dangerous instruction sequences, the decision of whether or not to intervene would depend on characteristics of the power-supply network and operating voltage range of the target processor, which typically are not known at compile time. Finally, static techniques may not avoid all voltage emergencies; many emergencies occur due to dynamic instruction sequencing, which is difficult to predict prior to program execution.

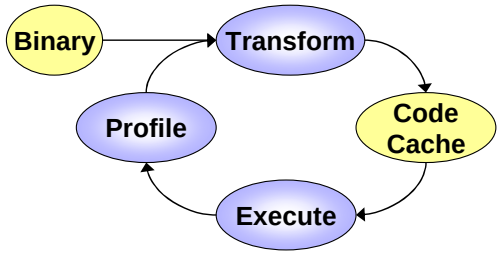

Figure 1: Control flow of a software-based dynamic optimization system.

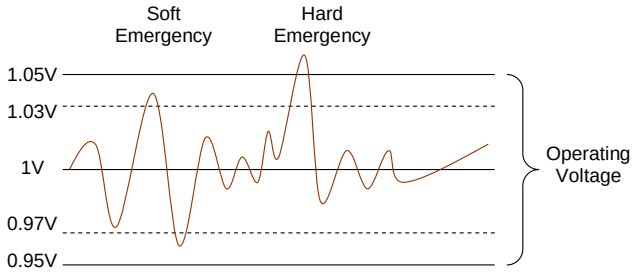

Figure 2: Soft and hard voltage emergencies.

Therefore, we propose extending the hardware mechanisms to additionally provide feedback to a software-based dynamic optimization system [3]. Figure 1 shows the control flow of a typical dynamic optimizer. These systems observe execution and perform code transformations to a cached copy of the frequently-executed instructions. The cached, transformed code is then executed in lieu of the original instructions. Finally, run-time feedback and profile information is used to guide other transformations, and the process continues.

A dynamic optimizer has the ability to effectively balance the performance/power trade-off, 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 code at the granularity of *code traces*—dynamic instruction sequences that span procedure call and branch boundaries. Therefore, this system can correct voltage emergencies that arise due to dynamic instruction sequencing that spans module boundaries.

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

- A description of a collaborative hardware/software approach to the dI/dt problem.
- A characterization of voltage fluctuations within the SPEC2000 benchmark suite assuming modern technology, as well as an extrapolation to future processors over the next 10 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 some existing solutions. 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 we can expect to occur within the SPEC2000 benchmarks over the next 10 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.

Voltage emergencies aren't a critical problem today because hardware designers have taken a conservative approach when designing power supply systems and CPUs. But, implementing conservative designs is wasteful, and future trends may make it much more difficult to do so. The International Technology Roadmap for Semiconductors (ITRS) lists noise management (dI/dt, ground bounce, etc.) as one of their *grand challenges* for the 2010+ time frame [1].

There are three major contributors to the increasing seriousness of the dI/dt problem. The first contributor is the low-power design trend to reduce average power dissipation, including aggressively clock gating more and more portions of the microprocessor, i.e. 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.

The second contributor is the decreasing voltage trends in high-performance microprocessors. ITRS estimates suggest that the operating voltage will drop to 0.7V over the next 15 years, which means that the tolerance for processor voltage variations will drop to +/- 0.035V.

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 [11] 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. [6] 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 online calculations, as part of the voltage control mechanism. Their work focused on voltage ranges rather than specific voltage values and defined two thresholds. A *control threshold* was derived from

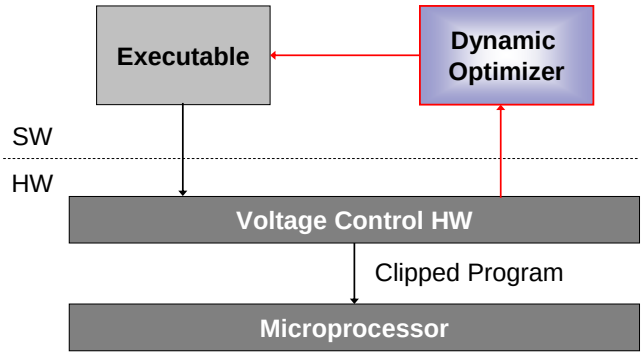

Figure 3: Our collaborative architecture.

a control-theoretic model that included architectural, voltage sensor, and power supply parameters. The 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

The hardware-based voltage control mechanisms described in Section 2 respond whenever the source voltage moves outside of a predefined control range. The range is defined to minimize false alarms, while guaranteeing that sufficient time will be available to stabilize the voltage through the actuation response. As the voltage moves outside of the control threshold range, the actuation mechanism 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 energy. While these hardware solutions have been shown to be effective at reducing voltage swings, the executing program incurs performance penalties as a result.

Therefore, we extend the hardware mechanisms to additionally provide feedback to a software-based dynamic optimization system. This system can 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. Hardware-based voltage control mechanisms remain intact, while the extensions are shown at the software level. 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 processor state during the emergency, including instructions that are currently in-flight or recently completed.

Execution Core	
Clock Rate	3.0 GHz
Instr Window	256-RUU, 128-LSQ
Func Units	8 iAlu, 4 fpAlu, 2 iMul/iDiv, 2 fpMul/fpDiv
Front End	
Fetch Width	8 instructions
Decode Width	8 instructions
Branch Penalty	10 cycles
Branch Predictor	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 using the SimpleScalar toolset with Wattch power extensions.

By operating in a *lazy optimization* mode, the dynamic optimizer can wait until it is informed by the hardware of a voltage emergency (after the hardware activates control mechanisms to eliminate the emergency), and it can then re-optimize and cache a version of the code that exhibits more voltage stability. 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.

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

1. Determines the location in the original source code that is the apparent cause of the voltage fluctuation.
2. Decides whether this region of code has caused a voltage fluctuation in the past, and is therefore a candidate for region modification.
3. Determines the best plan for altering the code region and performs the optimization.
4. Inserts the new code into a code cache, thereby making it the default version for subsequent execution.

In summary, the proposed architecture is a collaborative hardware/software approach. The hardware component includes voltage control mechanisms and feedback to the software component—the dynamic optimizer—which applies a long-term solution to the executing program.

4. CHARACTERIZING EMERGENCIES

To explore voltage trends in the SPEC2000 benchmarks, a framework similar to [7] was used. This included a modified version of the Wattch 1.03 architectural-level power simulator [4] configured for a 1.0 V supply voltage. Wattch is based on the SimpleScalar toolset [2] and we simulated an 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 by performing a convolution of Wattch's current estimates and an impulse

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 SPEC2000 benchmarks experiencing voltage emergencies.

response to the power-supply network for parameters derived from a model of the Alpha 21364 package [12]. The convolution is calculated using the following equation:

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

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 for the first 100 million instructions (after skipping 1 billion instructions) were output to a voltage trace file. Finally, the voltage traces were analyzed to produce the results presented in the following sections.

4.1 Voltage Emergency Results

Using our modified Wattch simulator to output a trace of supply voltages each cycle during the execution of each benchmark, we determined the number of voltage emergencies expected in future microprocessors. We used methodology similar to [7], where we assume that modern technology allows the power supply to match the target impedance necessary to avoid voltage emergencies, then look at the case where the impedance is 200%, 300%, and 400% 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 microprocessors can be shipped with less expensive, less conservative power supplies.

Table 2 shows the number of emergencies that will occur at these impedance values across the SPEC benchmarks. While it's not until 400% target impedance that voltage emergencies occur, almost 20% of the benchmarks will trigger the voltage control system (due to a soft emergency) at only 200% impedance. Since it doesn't matter whether a hard emergency would have occurred for the hardware-based control system to engage, performance implications would be experienced after the increase to 200% impedance. Our software extensions could permanently alleviate any recurring false alarms, and therefore reduce the performance effects of a conservative hardware-only control system.

5. MAPPING EMERGENCIES TO SOURCE

Since our design relies on hardware-initiated feedback to a dynamic optimizer, an important question becomes: *After an emergency, what information should the hardware provide to the dynamic optimizer?* To answer this question, we explored several options.

All Instructions To form a baseline for comparison, we began by investigating the option of marking all instructions

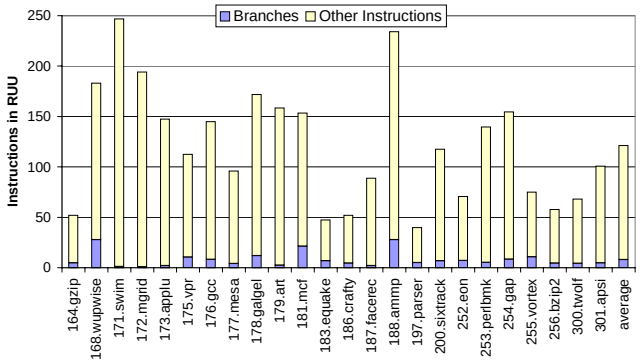

Figure 4: Average number of instructions in the RUU during a voltage emergency.

in SimpleScalar's Register Update Unit (RUU) as potential causes of the voltage emergency, and communicating all instructions back to the dynamic optimizer. Figure 4 shows the average number of instructions in the RUU during a voltage emergency, for each SPEC2000 benchmark. Overall, an average of 113 instructions would require communication to the software system using this heuristic. It is inefficient and unrealistic to design a feedback system that carries such a high communication overhead. Therefore, this motivates the need for *address pruning*.

Branch Only One heuristic we explored for pruning the information sent to the dynamic optimizer was to focus on the in-flight branch instructions. The logic behind this heuristic is that we can identify loops using only branch instructions. The dynamic optimizer can then focus effort on optimizing that particular loop. Figure 4 also shows the average number of in-flight branch instructions during a voltage emergency, for each SPEC2000 benchmark. Using this heuristic, an average of eight instructions must be communicated to the dynamic optimizer, while individual benchmarks vary from one branch (as in the case of `swim` and `mgrid`) to 27 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 one address begs the question: *Is it ever the case that there are no in-flight branches to be reported to the dynamic optimizer?* Table 3 shows how often such over-pruning occurs. A major outlier is `mgrid`, where 25% of emergencies occur while no branches are in-flight. This is likely the result of a problematic code region that is located inside a large loop, and the back-edge branch has already retired from the processor by the time the voltage emergency occurs.

Last-Executed Branch In the ideal case, the hardware would only pass a single address to the dynamic optimizer to trigger an update of a problematic code region. This would allow the hardware to communicate via a single performance counter register. What single instruction is likely to provide the most information about a problematic area? One solution is the last-executed branch instruction. In hardware, this heuristic could be implemented by extending the branching hardware to write the PC of every executed branch to a Last Executed Branch register (LEB). Then, when a voltage emergency occurs, the LEB register contents can be reported to the dynamic optimizer.

SpecInt	Failures	Percent	SpecFP	Failures	Percent
gzip	59	0.10%	wupwise	0	0%
vpr	99	0.22%	swim	0	0%
gcc	11	0.17%	art	0	0%
mcf	0	0%	mgrid	43809	24.80%
crafty	3938	1.95%	applu	6515	5.47%
perlbmk	0	0%	mesa	4579	4.83%
eon	1882	0.61%	galgel	8	0.01%
parser	201	0.72%	equake	0	0%
gap	1	0.02%	facerec	1	0.00%
vortex	469	0.34%	sixtrack	38	0.07%
bzip2	0	0%	ammp	0	0%
twolf	1323	1.34%	apsi	14	0.01%

Table 3: Pruning failures during voltage emergencies. *Failures* is the number of voltage emergencies where there were no in-flight branches. *Percentage* is the dynamic fraction of branch pruning failures.

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

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

Using this heuristic, we 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* is actually the number of times that the dynamic optimizer would need to intervene to correct voltage emergencies, while the *Total* column is the number of times that hardware must intervene if the dynamic optimizer was not part of the solution. 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, hardware-only solutions would need to intervene between 35 and 306,698 times, for `perlbmk` and `eon`, respectively. The fact that emergencies are clearly associated with particular static branch instructions strongly suggests the existence of source-level voltage emergency loops that are well-suited for a dynamic optimizer. The case becomes even more compelling as we scale the number of instructions executed from 100 million to 1 and 10 billion. The number of distinct emergencies stays constant, while the total emergencies increases linearly with instructions executed signaling, that for realistic, long-running applications, the execution cycles required to perform code optimization can easily be amortized.

6. COMPILER-BASED SOLUTIONS

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

Prior work [7] 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 processor's resonant frequency 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 (depicted in Figure 5) that provided the initial motivation for our work, as the source code consists of a single loop body, but causes thousands of voltage emergencies during execution. We will now discuss a set of existing compiler optimizations that can alleviate these periods of high and low activity dynamically.

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. Typically, the result of software pipelining is that n-iterations of a loop will be combined to form one larger loop iteration. The nature of the software pipelining algorithm has two interesting side-effects. First, the technique allows high-activity periods in one loop iteration to be combined with low-activity periods of the next loop iteration potentially leading to a more stable sequence of instructions that will often complete faster than the original sequences. Second, by changing the amount of work done in a loop iteration, periods of high and low activity that fall on the resonant frequency will be disrupted.

Figure 6 depicts the result of applying software pipelining to the loop body in Figure 5. By unrolling the loop body once, and therefore lengthening the period of low activity originally resulting from three subsequent divide operations, we were able to move the stressmark off of the resonant frequency. This reduced the resulting voltage fluctuations and potentially eliminated numerous invocations of the hardware throttling mechanism.

dI/dt Stressmark

```
        BEFORE                          AFTER
ldt     $f1, ($4)           ldt     $f1, ($4)
ldt     $f2, ($6)           ldt     $f2, ($6)
divt    $f1, $f2, $f3       divt    $f1, $f2, $f3
divt    $f3, $f2, $f3       divt    $f3, $f2, $f3
divt    $f1, $f2, $f3       divt    $f1, $f2, $f3
stt     $f3, 8($4)          divt    $f3, $f2, $f4
ldq     $7,  8($4)          divt    $f4, $f2, $f4
cmovne  $31, $7, $3         divt    $f3, $f2, $f4
stq     $3, ($4)            stt     $f4, 8($4)
stq     $3, ($4)            ldq     $7,  8($4)
stq     $3, ($4)            cmove   $31, $7, $3
stq     $3, ($4)            stq     $3, ($4)
...                         ...
stq     $3, ($4)            stq     $3, ($4)
```

Figure 5: Alpha instructions in a dI/dt stressmark loop before and after loop unrolling with software pipelining.

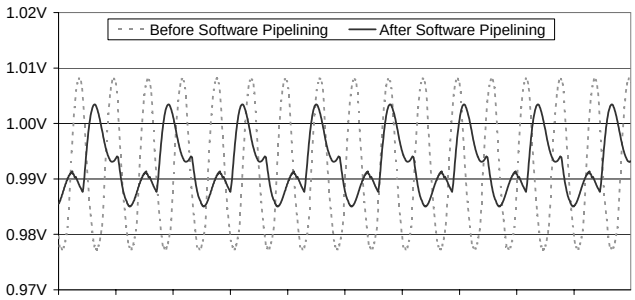

Figure 6: The effect of software pipelining on the voltage stressmark loop.

Code Motion Since we have no guarantee that the last-executed branch is a loop back-edge, algorithms that target acyclic code regions are also necessary.

When a static compiler schedules instructions, it often has several options for scheduling an instruction that result in equal run-time performance of the application. Thus, the compiler may inadvertently create regions of high and low processor activity simply 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. This technique can result in the removal of a voltage emergency without degrading application performance.

Instruction Padding A final optimization is one that can be applied to acyclic regions when performing code motion is not possible. Instruction padding involves inserting unnecessary calculation into a low-utilization code region. This transformation 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 as it has no performance benefits. While the processor will ideally schedule the unnecessary instructions off the critical path on idle functional units, this approach may degrade performance of an instruction sequence, and therefore should be considered as a last resort.

7. CONCLUSIONS

Voltage emergencies are becoming a 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, which avoids the performance penalties of a hardware-only solution and the inaccuracies of a software-only solution. We use hardware to correct intermittent dI/dt problems, but provide feedback to a software-based dynamic optimizer for correcting repeatable problems at the source-code level. We found that hardware-to-software communication can be effectively minimized using a Last-Executed Branch (LEB) register. Using a combination of software pipelining, code motion and instruction padding, we can move problematic code regions off of the resonant frequency of the microprocessor package, resulting in a reduction in soft emergencies.

8. FUTURE WORK

Now that the potential for improved performance during dI/dt emergencies has been identified, our next logical step is to modify a dynamic optimizer to enable it to receive hardware feedback during an emergency, and to implement the optimizations described in Section 6 in response to that feedback. 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. Low-power versions of commonly executed portions of code can be generated, cached, and executed all at run time.

Acknowledgments

We would like to thank Russ Joseph for his guidance during the process of replicating his infrastructure and stressmark. This work was sponsored by a Harvard DEAS Fellowship.

9. REFERENCES

[1] International technology roadmap for semiconductors. *Semiconductor Industry Association*, 2003.

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

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, pages 1–12, 2000.

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

[5] A. P. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. Wiley-IEEE Press, 2000.

[6] E. Grochowski, D. Ayers, and V. Tiwari. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *HPCA-8*, pages 7–16, 2002.

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

[8] R. Joseph, Z. Hu, and M. Martonosi. Wavelet analysis for microprocessor design: Experiences with wavelet based di/dt characterization. In *HPCA-10*, 2004.

[9] M. D. Powell and T. N. Vijaykumar. Pipeline damping: A microarchitectural technique to reduce inductive noise in supply voltage. In *ISCA-30*, pages 72–83, 2003.

[10] L. Smith, R. Anderson, D. Forehand, T. Pelc, and T. Roy. Power distribution system design methodology and capacitor selection for modern CMOS technology. *IEEE Transactions on Advanced Packaging*, 22(3):284–291, August 1999.

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

[12] M. Tsuk, R. Dame, D. Dvorscak, C. Houghton, and J. St.Laurent. Modeling and measurement of the Alpha 21364 package. In *2001 Electrical Performance of Electronic Packaging (EPEP)*, October 2001.