Software-Assisted Hardware Reliability: Abstracting Circuit-level Challenges to the Software Stack

Vijay Janapa Reddi, Meeta S. Gupta,
Michael D. Smith, Gu-yeon Wei, David Brooks
Harvard University
Cambridge, USA
{vj, meeta, smith, guyeon, dbrooks}@eecs.harvard.edu

Simone Campanoni
Politecnico di Milano
Milano, Italy
simone.campanoni@mail.polimi.it

ABSTRACT

Power constrained designs are becoming increasingly sensitive to supply voltage noise. We propose a hardware-software collaborative approach to enable aggressive operating margins: a checkpoint-recovery mechanism corrects margin violations, while a run-time software layer reschedules the program's instruction stream to prevent recurring margin crossings at the same program location. The run-time layer removes 60% of these events with minimal overhead, thereby significantly improving overall performance.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/Software interfaces and System architectures.*

General Terms

Performance, Reliability.

Keywords

Runtime Optimization, Hardware Software Co-Design.

1. INTRODUCTION

Power supply noise impacts the robustness and performance of microprocessors. In order to meet power requirements, processor designs are relying on ever lower supply voltages and aggressive power management techniques such as clock gating that can cause large current swings. These current swings when coupled with the parasitic inductances in the power-delivery subsystem can cause voltage fluctuations that violate the processor's operating margins—a significant drop in the voltage can lead to timing-margin violations, due to slow logic paths. On the other hand, significant overshoots in the voltage can also cause long-term degradation in transistor characteristics. For reliable and correct operation of the processor, large voltage swings, also called *voltage emergencies*, should be avoided.

The traditional way of dealing with voltage emergencies has been to over-design the system to accommodate the worst-case voltage swing. A recent paper analyzing supply noise in a Power6 processor [15] shows the need for operating margins greater than 20% of the nominal voltage (200mV for a nominal voltage of 1.1V). Conservative processor designs with large timing margins ensure robustness. However, conservative designs either lower the operating frequency or sacrifice power efficiency.

Alternatively, researchers have proposed designing for the average case operating conditions while providing a "fail-safe" hardware mechanism that guarantees correctness in the presence of voltage emergencies. Such a fail-safe mechanism enables more aggressive timing margins in order to maximize performance, but at the expense of some performance penalty. Architecture- and circuit-level techniques either proactively take measures to prevent a potentially impending voltage emergency [11, 16, 22, 23], or operate reactively by recovering a correct processor state after an emergency corrupts machine execution [13].

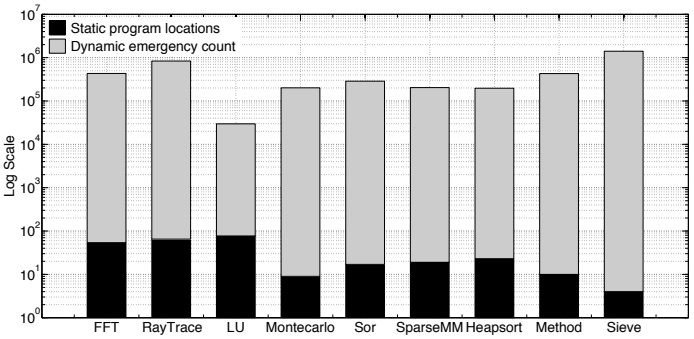


Figure 1: Fewer than 100 static program addresses are responsible for the large number of voltage emergencies. We assume a 4% operating margin, but this trend is the same across margins.

Traditional hardware techniques do not exploit the effect of program structure on emergencies. Figure 1 shows the number of unique static program addresses responsible for emergencies,[1] and the total number of emergencies they contribute over the lifetime of a program. The stacked log-scale distribution plot indicates that on average fewer than 100 static program addresses are responsible for several hundreds of thousands of emergencies. Even an ideal oracle-based hardware technique will need to activate its fail-safe mechanism once per emergency. Additionally, hardware-based schemes must ensure that performance gains from operating at a reduced margin outweigh the fail-safe penalties. They therefore rely on tuning the fail-safe mechanism to the underlying processor and power delivery system specifics [13]. When combined with implementation costs, potential changes to traditional architectural structures, and challenges like response-time delays [13], design, testing, validation and wide-scale retargetability all become increasingly difficult.

In this paper, we present a hardware-software collaborative approach for dealing with voltage emergencies. Hazelwood and Brooks *et al.* [14] suggest the potential for a collaborative scheme, but we demonstrate and evaluate a full-system implementation design. The collaborative approach relies on a general-purpose fail-safe mechanism as infrequently as possible to handle emergencies, by having a software layer dynamically smooth bursty machine activity via code transformation to prevent frequently occurring emergencies. Ideally, the fail-safe mechanism activates only once per static emergency location, and therefore only a few times in all, as shown in Figure 1.

Dynamic optimization systems [5] are well suited for scenarios where "90% of the execution time is spent in 10% of the code". Figure 1 shows similar behavior with respect to emergencies. A compiler-assisted scheme, in contrast to hardware techniques, can exploit the fact that programs have so few static emergency-

[1] We use the event categorization algorithm described by Gupta *et al.* [12] to identify the instruction that gives rise to an emergency.

DAC 2009, July 26 - 31, 2009, San Francisco, California, USA.

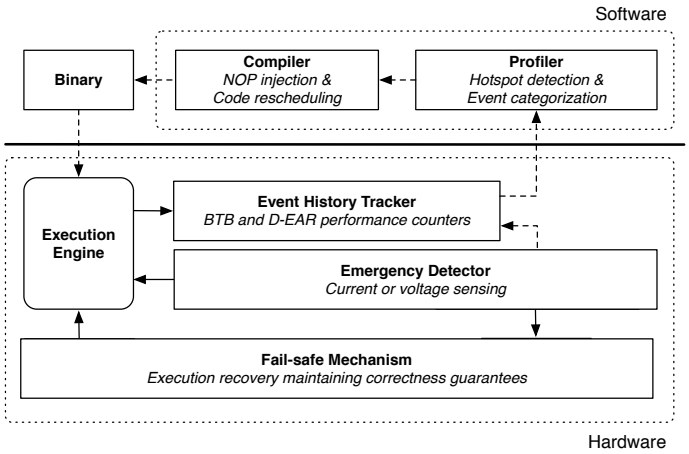


Figure 2: Workflow diagram of the proposed software-assisted hardware-guaranteed approach to deal with emergencies.

prone hot spots. In our scheme, a dynamic compiler eliminates a large fraction of the Dynamic emergency count. We demonstrate a compiler-based issue rate staggering technique that reduces emergencies by applying transformations such as code rescheduling or injecting new code into the dynamic instruction stream of a program. Our primary contributions are as follows:

1. Design and implementation of a dynamic compiler-based system for suppressing recurring voltage emergencies.
2. An instruction rescheduling algorithm that prevents voltage emergencies by staggering the issue rate.
3. Demonstration and evaluation of general purpose checkpoint-recovery hardware to handle voltage emergencies even at aggressive operating margins with our software co-design.

The rest of the paper is organized as follows: Section 2 presents an overview of the proposed hardware-software collaborative approach along with design details for the hardware and software components. Section 3 discusses performance results, and Section 4 concludes the paper.

2. A COLLABORATIVE SYSTEM

The benefits of a collaborative hardware-software approach are twofold: First, recurring emergencies are avoidable via software code transformation. Second, a collaborative scheme allows hardware designers to relax worst-case timing margin requirements because of the reduced number of emergencies. The net effect is better energy efficiency or improved performance. In this section, we first present an overview of how our collaborative architecture works and highlight the critical components. Following that, we present details about each of the hardware and software components.

2.1 Overview

Figure 2 illustrates the operational flow of our system. An Emergency Detector continuously monitors execution. When it detects an emergency, it activates the hardware's Fail-safe Mechanism. We assume that a general-purpose checkpoint-recovery mechanism restores execution to a previously known valid processor state whenever an emergency is detected. After recovery, the detector notifies the software layer of the voltage emergency.

The software operates in *lazy* mode; it waits for emergency notifications from the hardware. Whenever a notification arrives, the software's Profiler extracts information about recent processor activity from the Event History Tracker, which maintains information about cache misses, pipeline flushes, and so on. The Profiler uses this information to identify the code region corresponding to an emergency. Subsequently, the Profiler calls a run-time Compiler to alter the code responsible for causing the emergency in an attempt to prevent future emergencies at the same location.

2.2 Hardware Design

Emergency detector. To detect operating margin violations, we rely on a voltage sensor. The detector invokes the fail-safe mechanism when it detects an emergency. After recovery, the detector invokes the software layer for profiling and code transformation to eliminate subsequent emergencies.

Fail-safe mechanism. Our scheme allows voltage emergencies to occur in order to identify emergency-prone code regions for software transformation. We therefore require a mechanism for recovering from a corrupt processor state. We use a recovery mechanism similar to that found in reactive techniques for processor error detection and correction that have been proposed for handling soft errors [1, 26]. These are primarily based on checkpoint and rollback. Checkpoints can be made either explicitly [2,17,18] or they can be saved implicitly [13]. We only evaluate the explicit scheme since it is already shipping in production systems [3, 10] and does not require modifications to traditional microarchitectural structures. The interval between checkpoints is just tens of cycles.

Several researchers have proposed a variety of diverse applications using checkpoint-recovery hardware [17, 18, 20, 24, 26]. Our use of checkpoint-recovery for handling inductive noise in collaboration with software is another novel application of this general-purpose hardware. However, explicit-checkpointing by itself cannot be used to handle voltage emergencies because the performance penalties are too large (as discussed in Section 3.2).

Event history tracker. The software layer requires pertinent information to locate the instruction sequence responsible for an emergency in order to do code transformation. For this purpose, we require the processor to maintain two circular structures similar to those already found in existing architectures. The first is a *branch trace buffer (BTB)*, which maintains information about the most recent branch instructions, their predictions, and their resolved targets. The second is a *data event address register (D-EAR)*, which tracks recent memory instruction addresses and their corresponding effective addresses for all cache and TLB misses. The software extracts this information whenever it receives a notification about an emergency.

2.3 Software Design

Profiler. The profiler is notified whenever an emergency occurs. The profiler identifies emergency-prone program locations for the compiler to optimize. It records the time and frequency of emergency occurrences in addition to recent microarchitectural event activity extracted from the performance counters. Using this information the profiler locates the instruction responsible for an emergency using an *event categorization* algorithm [12]. We refer to this problematic instruction as the *root-cause* instruction. Event categorization identifies root-cause instructions based on the understanding that microarchitectural events along with long-latency operations can give rise to pipeline stalls. A burst of activity following the stall can cause the voltage to drop below the minimum operating margin due to a sudden increase in current draw. Such a violation of the minimum voltage margin is by definition a voltage emergency.

Figure 3(a) illustrates a scenario where a data dependence on a long-latency operation stalls all processor activity. When the operation completes, issue rate increases rapidly as several dependent instructions are successively allocated to different execution units. This gives rise to a voltage emergency because of the sudden increase in current draw. The categorization algorithm associates the long-latency operation as the root cause since it caused the burst of activity that gave rise to an emergency. The algorithm also takes into account other processor activity such as cache and TLB misses and branch mispredictions, as they can also cause emergencies.

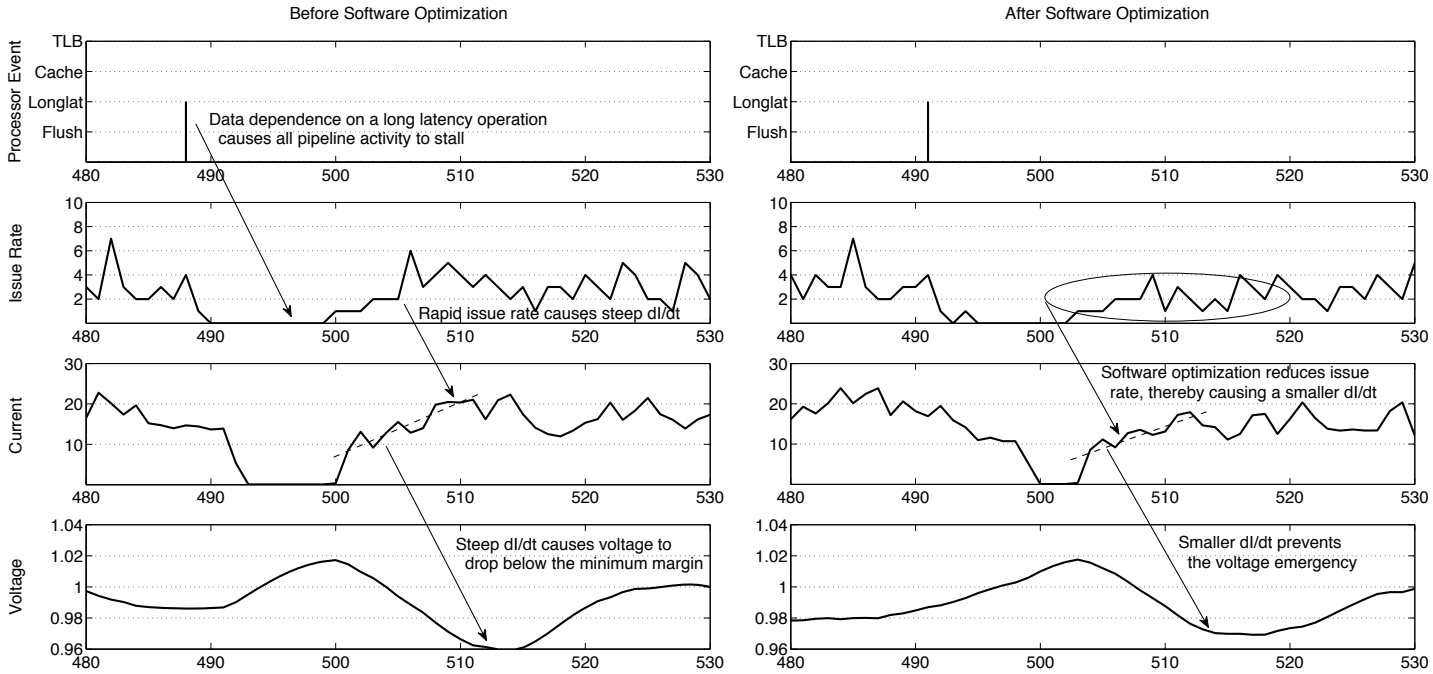


Figure 3: Snapshot of benchmark *Sieve* showing the impact of a pipeline stall due to data dependency. An operating margin of 4% is assumed (i.e., max. of 1.04V and min. of 0.96V). (a) Before Software Optimization shows how a stall triggers an emergency as the issue rate ramps up quickly once the long-latency operation completes. (b) After Software Optimization demonstrates how compiler-assisted code rescheduling slows the issue rate to prevent the emergency illustrated in (a).

Compiler. Figure 3(a) illustrates that voltage emergencies are dependent on the issue rate of the machine. Therefore, slowing the issue rate at the appropriate point can prevent voltage emergencies. Hardware-based solutions have been proposed that prevent emergencies by altering machine behavior via execution throttling [11, 16, 22, 23] or staggering the issue rate [21, 22]. But as high issue rate alone is insufficient to cause emergencies, throttling in all cases can penalize performance unnecessarily.

Alternatively, Toburen [25] and Yun and Kim [27] demonstrate static compiler techniques that target voltage emergencies. However, emergencies are the result of complex interactions between the application, the execution engine, and the power delivery subsystem. Therefore, these static optimizations are not easily retargetable across different combinations of platform and application. In other words, the emergency-prone static program locations discussed in Figure 1 differ depending on platform specifics.

In contrast, our software approach prevents emergencies by altering the program code that gives rise to emergencies at execution time and does so without slowing down the machine. The compiler tries to exploit pipeline delays by rescheduling instructions to decrease the issue rate close to the root-cause instruction. Pipeline delays exist because of NOP instructions or read-after-write (RAW), write-after-read (WAR), or write-after-write (WAW) dependencies between instructions. Hardware optimization techniques like register renaming in superscalar machines can optimize away WAR and WAW dependencies, so a RAW dependence is the only kind that forces the hardware to execute in sequential order. The compiler tries to exploit RAW dependencies that already exist in the program to slow the issue rate by placing the dependent instructions close to one another.

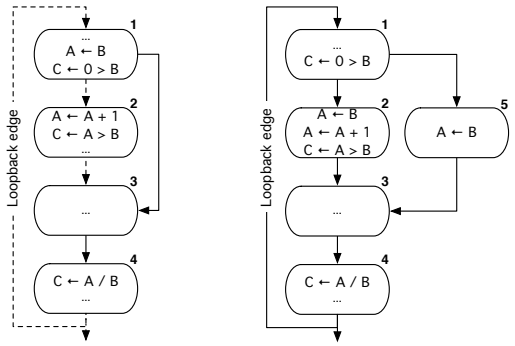
NOP injection. The compiler can slow down pipeline activity by inserting NOP instructions specified in the instruction set architecture into the dynamic instruction stream of a program. However, modern processors discard NOP instructions at the decode stage. Therefore, the instruction does not affect the issue rate of the machine. Instead of real NOPs, the compiler can generate a sequence of instructions containing RAW dependencies that have no effect. But since these *pseudo-NOP* instructions perform no real work, this approach degrades performance.

Code rescheduling. A better way reduce processor activity is to exploit RAW dependencies already existing in the original control flow graph (CFG) of the program. The compiler attempts to relocate RAW dependencies to a point following the root cause of an emergency, thereby constraining the burst of activity after the stall and consequently preventing the emergency.

Whether the compiler can successfully move instructions to create a sequence of RAW dependencies depends on if moving the code violates either control dependencies or data dependencies. The compiler's instruction scheduler does not break data dependencies, but it works around control dependencies by cloning the required instructions and moving them around the control flow graph carefully such that the original program semantics are still maintained. Aggressive cloning can potentially impact the performance of a program by increasing the total number of instructions executed at run time. For this reason, our scheduler does not migrate instructions if the estimated instruction count of the program is likely to increase from cloning.

For illustrative purposes, we present in Figure 4(a) a simplified sketch of the code corresponding to the activity shown in Figure 3(a). The long-latency operation illustrated in Figure 3 corresponds to the *divide* instruction shown in basic block 4 of Figure 4. An emergency repeatedly occurs in basic block 3 along the dotted loop backedge path $4 \rightarrow 1 \rightarrow 2 \rightarrow 3$. The categorization algorithm identifies the *divide* instruction corresponding to $C \leftarrow A \ / \ B$ in basic block 4 as the root-cause instruction. The compiler identifies the control flow path using the branch history information extracted by the profiler from the BTB counters, and recognizes that moving instruction $A \leftarrow B$ from basic block 1 to 2 will constrain the issue rate of the machine because of a tighter sequence of RAW dependencies. But the compiler also recognizes that the result of $A \leftarrow B$ is live along edge $1 \rightarrow 3$, so it clones the instruction into a new basic block (basic block 5) along that edge to ensure correctness.

The resulting effect (on the more complex actual code sequence) after rescheduling is illustrated in Figure 3(b). The slight change in current activity between cycles 490 and 500 is a result of code rescheduling. After dependent instructions are packed close to one another in basic block 2, the issue rate in Figure 3(b) does not spike as high as it does in Figure 3(a) once



(a) Before. (b) After.

Figure 4: Effect of code rescheduling on an emergency-prone loop from benchmark *Sieve*. (a) Emergencies consistently occur in basic block 3 along the dotted loop backedge path $4 \rightarrow 1 \rightarrow 2 \rightarrow 3$. (b) Moving instruction $A \leftarrow B$ from block 1 to block 2 puts dependent instructions closer together, thereby constraining the issue rate. This prevents all subsequent emergencies in basic block 3.

pipeline activity resumes after the stall.

Code rescheduling alters the current and voltage profile. Therefore, the scheduler must be careful not to simply displace emergencies from one location to another by arbitrarily moving code from far away regions. To retain the original activity, the code rescheduling algorithm searches for RAW dependencies starting with the basic block containing the root-cause instruction. As the program is running, the profiler tracks the instructions that are giving rise to voltage emergencies. Using this information, the compiler computes a set of program instructions, P, for rescheduling. For each instruction in P, the scheduler searches for RAW dependencies starting from the root-cause instruction. The scheduler enlarges its search window iteratively over the CFG until it finds a RAW dependence to exploit or it reaches the scope of a function body, at which point it gives up.

Out-of-order execution complicates instruction rescheduling, as the machine can bypass the RAW dependence chain generated by the compiler if there is enough other code available for execution in the hardware's scheduling window. The scheduler handles this by choosing a RAW candidate from a set C_1 of candidates by computing the subset $C_2 \subseteq C_1$ such that each element of C_2 has the longest RAW dependence chain after moving the instructions to the required location. By targeting long RAW dependence chains, the compiler causes the machine's scheduling window to fill up with dependent code that reduces the issue rate. Otherwise, the compiler must generate multiple sets of smaller RAW dependence chains. But the more code the compiler moves around, the higher the chances that it will render optimization ineffective because of statically unpredictable interactions among the dependence chains.

3. RESULTS

Evaluation of our system demonstrates the effectiveness of the compiler at reducing voltage emergencies and shows the impact of its code changes on performance. After showing in Section 3.1 that the compiler can reduce over 60% of emergencies, we present a performance study in Section 3.2 showing that our software-assisted scheme overcomes the challenges of existing hardware techniques effectively.

Hardware simulator. We use SimpleScalar/x86 to simulate a Pentium 4 with the characteristics shown in Table 1. The modified 8-way superscalar $x86$ SimpleScalar gathers detailed cycle-accurate current profiles using Wattch [7]. Voltage variations are calculated by convolving the simulated current profiles with an impulse response of the power delivery subsystem [16,23]. In this work, we focus on a power delivery subsystem model based on the characteristics of the Pentium 4 package [4], which exhibits a mid-frequency resonance at 100MHz with a peak impedance of 5mΩ. Finally, we assume peak current swings of 16-50A.

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

Table 1: Architecture parameters for SimpleScalar.

Software infrastructure. We use the ILDJIT [9] CIL compiler as our framework for optimizing emergencies at run time. The compiler dynamically generates native $x86$ code from CIL byte code, which it then executes directly on the simulator. We extended the ILDJIT compiler to include the code injection and scheduling algorithms described in Section 2.3. The compiler has access to metadata such as the complete control flow graph and data flow graph, all of which is utilized at run time for optimization. The C# benchmarks evaluated in this paper come from the Java Grande benchmark suite [8,19].

Due to space constraints, we omit discussion about the negligible overheads of run-time code transformation. Figure 1 shows that the number of static emergency-prone program locations (root-cause instructions) is fewer than a hundred. Therefore, our compiler is rarely invoked during execution to transform the code. When combined with the fact that the ILDJIT compiler does its profiling and code transformation in a separate thread on a separate core, performance overhead on the original program is negligible. Also, in our experiments we observe that the fraction of emergencies encountered during ILDJIT's own execution is around 1% across all benchmarks. Since ILDJIT cannot recompile itself, we incur rollback penalties during the compiler's execution. But because the fraction of emergencies is so small, the rollback overhead is insignificant.

3.1 Compiler Efficiency

The goal of our software-based voltage emergency elimination is to: (1) reduce the number of voltage emergencies, and (2) ensure that performance does not suffer as a result of our code transformations. In the next section, we factor in all costs to evaluate full-system performance. In this section, we evaluate the effectiveness of NOP injection and code rescheduling. The main observations are that (1) the choice of transformation affects performance, and that (2) the transformation itself can introduce new emergencies if the scheduler is not careful.

NOP injection. In this transformation, the compiler modifies the original program to contain new instructions that simulate a NOP instruction immediately following the root-cause instruction. The effectiveness of the transformation is shown by the left bar in Figure 5(a). The bar shows the fraction of emergencies remaining after the compiler has attempted to prevent emergencies by injecting pseudo-NOP code. The number of emergencies is reduced by ∼50% in benchmarks *FFT*, *RayTrace*, *Method*, *Sieve*, and *Heapsort*, which shows that the transformation can be effective. However, the transformation is ineffective across the remaining benchmarks *LU*, *Montecarlo*, *Sor* and *SparseMM*.

Analysis reveals that pseudo-NOP injection does reduce the original program's emergencies, but the transformation itself gives rise to new emergencies. The compiler might occasionally have to spill and fill registers to generate pseudo-NOP code. This has the adverse effect of not only increasing the number of instructions needed to simulate the NOP, but also potentially causing architectural events like cache misses (from the spill and fill code) that dramatically alter the current and voltage profile. These side effects depend on the number of registers available for use, the properties of the original instruction schedule, and other conditions. It is hard to predict what current and voltage activity will result from injecting new code, so it may give rise to new emergencies. That is what we observe with the poorly performing

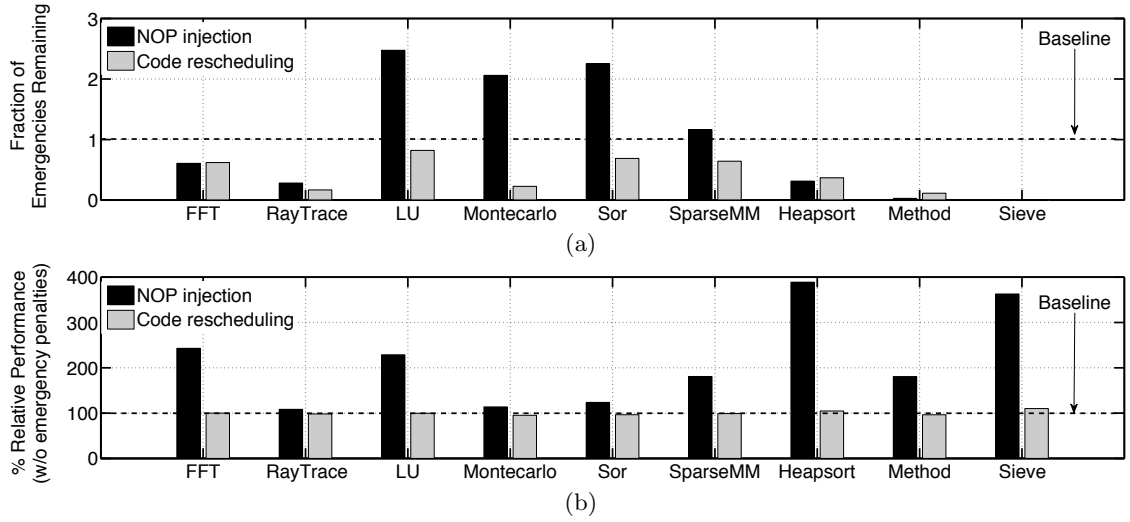


Figure 5: (a) Fraction of emergencies remaining after code transformation. (b) Code performance after transformation. The cost for handling emergencies is not shown in this plot to isolate the effect of code transformation on the run-time performance. Section 3.2 evaluates overall performance after factoring in code performance costs, along with penalties for handling emergencies.

benchmarks *LU*, *Montecarlo*, *Sor*, and *SparseMM*.

Additionally, the run-time performance of the original program suffers with the injection of pseudo-NOP code, as the new code does not serve the original program's purpose. The left bar in Figure 5(b) shows execution performance of the program with the injected code. The data indicates that the effect of simply adding new code to prevent emergencies can be severely detrimental to performance. In the case of benchmarks *Heapsort* and *Sieve* performance degrades by as much as 300%. Large execution overheads indicate that while a transformation can be very effective at reducing voltage emergencies (e.g., benchmark *Sieve* has fewer than 10 emergencies remaining), the compiler must be sensitive to its run-time performance implications.

Code rescheduling. A compiler approach that relocates RAW dependencies following the root-cause instruction does not suffer from the severely unpredictable behavior of injecting code to prevent emergencies. Code rescheduling is superior to simple NOP injection for the following reasons. First, it successfully reduces more emergencies across all the benchmarks (illustrated by the bars on the right in Figure 5(a)). Second, it does so without dramatically increasing the execution time of a program (as shown in Figure 5(b)). Our analysis also shows that it does not introduce new emergencies, as the compiler does not inject new code that significantly alters the current and voltage profile.

For instance, consider benchmark *FFT*. The NOP injection transformation and the code rescheduling transformation eliminate approximately the same number of emergencies. However, the effect on performance between the two transformations is substantially different. The NOP injection transformation causes the original program to take twice as long to execute, whereas code rescheduling has a negligible effect on the original program's performance. That is because the NOP code wastes processor cycles, while the rescheduled instructions are real program code that is simply restructured to prevent emergencies.

Overall, changes in the run-time performance of the rescheduled code are negligible across all the benchmarks, and the reduction in emergencies averages ∼61%. Reductions are smaller over benchmarks *LU*, *Sor* and *SparseMM* (around 30%) because the compiler could not find enough RAW dependencies that it could relocate to slow the issue rate at the frequently occurring root-cause locations.

3.2 Full-System Performance Evaluation

Reducing operating voltage margins allows for frequency improvements or improved energy efficiency. However, there are fail-safe mechanism penalties associated with handling voltage emergencies at tighter margins. In this section, we demonstrate that by using our dynamic compilation strategy, it is possible to leverage general-purpose checkpoint-recovery for voltage emergencies at very aggressive margins. Performance gains for our collaborative approach are within 4 percentage points of an oracle-based throttling scheme. Results are presented in Table 2.

Bowman et al. show that removing a 10% operating voltage margin leads to a 15% improvement in clock frequency [6]. This indicates a scaling factor of 1.5 from operating voltage margin to clock frequency. We assume an aggressive operating margin of 4% in our experiments as compared to a 18% worst-case margin[2]. Based on the 1.5x scaling factor, the 4% operating voltage margin assumed in this paper corresponds to a 6% loss in frequency. Similarly, a conservative voltage margin of 18%, sufficient to cover the worst-case drops observed, leads to 27% lower frequency. If we take this conservative margin as the baseline for comparisons and the 18% margin can reduce to 4% while avoiding voltage emergencies, the resulting clock frequency increase could be ∼29%. This sets the upper bound on frequency gains achievable. We make the simplifying assumption that frequency improvements translate to higher overall system performance.

Fail-safe mechanism. An explicit-checkpointing scheme recovers from an emergency by rolling back execution. The explicit-checkpoint scheme suffers from the penalty of rolling back useful work done whenever a voltage emergency occurs. The restart penalty is a direct function of the sensor delay in the system, i.e., the time required to detect a margin violation. An explicit-checkpoint scheme incurs additional overhead associated with restoring the registers (assumed to be 8 cycles, for 32 registers with 4 write ports) and memory state (when volatile lines are flushed, additional misses can occur at the time of rollback).

Assuming a 50-cycle rollback penalty per recovery, an explicit-checkpoint scheme incurs an average increase of 25% in CPI over the set of benchmarks evaluated in Figure 5. Performance gains from scaling the operating margin down to 4% are negligible at only 3%. This minimal improvement in performance implies that explicit-checkpointing by itself cannot handle voltage emergencies successfully at aggressive margins.

Fail-safe mechanism with code rescheduling. While the performance gains using only explicit-checkpointing are minimal, the gains are larger when the fail-safe mechanism is combined with a software counterpart (as proposed in Section 2 and illustrated in Figure 2). Of the two compiler transformations discussed in Section 2.3 we evaluate the code rescheduling transfor-

[2]The worst voltage drop we observe for our power delivery package is 18%.

Scheme	CPI Overhead	Performance Gain
Fail-safe mechanism	25.0%	3.0%
Fail-safe mechanism with code rescheduling	7.6%	19.8%
Oracle-based throttling	4.0%	23.8%

Table 2: Increase in CPI to handle voltage emergencies, and net performance improvement after scaling the operating margin and factoring in the overheads. The upper bound on performance improvement is 29% assuming the margin is scaled from 18% to 4%. These results are the average measured across all benchmarks.

mation only, since its changes effectively reduce the number of emergencies (as discussed in Section 3.1), but are not detrimental to program performance.

The profiler identifies root-cause instructions as the fail-safe checkpoint scheme initiates rollbacks. So there is some amount of rollback penalty associated with initially discovering root-cause instructions for transformation. Thereafter, however, the compiler optimizes the root-cause instructions to prevent subsequent occurrences of emergencies at the same program location. If the rescheduling algorithm is ineffective at fixing certain emergency points, rollback penalties may still arise at those points (as shown in Figure 5(a) and discussed in Section 3.1). Combining explicit checkpointing with compiler assistance reduces checkpointing overhead substantially, from 25% to 7.6%. This translates to a net performance gain of ∼20%.

Comparison to other schemes. Several researchers have proposed mechanisms that spread out a sudden increase in current via execution throttling. Several kinds of throttling have been proposed [11,16,22,23]. For evaluation purposes, we compare the performance of our scheme against a frequency throttling mechanism that quickly reduces current load. The frequency of the system is halved whenever throttling is turned on, which results in performance loss.

We compare against an oracle-based throttling scheme, which throttles once per emergency and always successfully prevents the emergency. As a result, an oracle scheme does not suffer from rollback costs, nor does it suffer from performance loss due to throttles that cannot prevent emergencies. Oracle-based throttling enables ∼24% improvement in performance for tightened margins, which is just 4 percentage points better than our scheme. Of course, our scheme represents a practical design.

While an oracle-based scheme always successfully prevents emergencies, it is important to remember that realistic sensor-based implementations suffer from a tight feedback loop that involves detecting an imminent emergency and then activating the throttling mechanism in a timely manner to avoid the emergency. The detectors are either current sensors or voltage sensors that trigger when a certain threshold is crossed, indicating that a violation is likely to occur. Unfortunately, the delay required to achieve acceptable sensor accuracy inherently limits the effectiveness of these feedback-loop schemes, and operating margins must remain large enough to allow time for the loop to respond [13].

In contrast, our collaborative approach does not suffer from the limitations of sensor-based schemes. It leverages general-purpose checkpointing hardware that is already shipping in production systems [3, 10] to reduce voltage emergencies at very aggressive margins that enable significant performance gains.

4. CONCLUSION

The primary contribution of this work is a full system implementation design for a hardware-software collaborative approach to handle voltage emergencies. The collaborative approach reduces hardware penalties associated with handling voltage emergencies by having the software (a dynamic compiler) permanently fix the code region responsible for emergencies. The hardware provides fail-safe guarantees via a checkpoint-recovery mechanism, while the software layer identifies the emergency-prone code regions and reschedules that code to prevent further emergencies. The compiler eliminates over 60% of the emergencies on average, and therefore dramatically reduces the recurring overhead of the fail-safe mechanism. We show that by scaling the operating margin down from a conservative 18% to an aggressive 4% setting, we can achieve ∼20% higher performance, which is within 4 percentage points of an oracle-based throttling scheme.

Acknowledgments

We are grateful to Glenn Holloway and the anonymous reviewers for their comments and suggestions. This work is supported by gifts from Intel Corporation, National Science Foundation grants CCF-0429782 and CSR-0720566 and in part by ST Microelectronics and the European Commission under Framework Programme 7 (the OpenMedia Platform project).

5. REFERENCES

[1] S. Agarwal et al. Adaptive incremental checkpointing for massively parallel systems. In *ICS*, 2004.
[2] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO-36*, 2003.
[3] H. Ando and et al. A 1.3 GHz Fifth-Generation SPARC64 Microprocessor. In *Design Automation Conference*, 2003.
[4] K. Aygun et al. Power delivery for high-performance microprocessors. *Intel Technology Journal*, 9, 2005.
[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI*, 2000.
[6] K. A. Bowman et al. Energy-efficient and metastability-immune timing-error detection and instruction replay-based recovery circuits for dynamic variation tolerance. In *ISSCC*, 2008.
[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA-27*, 2000.
[8] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking java grande applications. In *The Practical Applications of Java*, 2000.
[9] S. Campanoni, G. Agosta, and S. C. Reghizzi. A parallel dynamic compiler for cil bytecode. *SIGPLAN Not.*, 2008.
[10] S. et al. Ibm's s/390 g5 microprocessor design. *Micro, IEEE*, 1999.
[11] E. Grochowski et al. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *HPCA-8*, 2002.
[12] M. S. Gupta et al. Towards a software approach to mitigate voltage emergencies. In *ISLPED*, 2007.
[13] M. S. Gupta et al. DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors. In *HPCA-14*, 2008.
[14] K. Hazelwood and D. Brooks. Eliminating Voltage Emergencies via Microarchitectural Voltage Control Feedback and Dynamic Optimization. In *ISPLED*, 2004.
[15] N. James et al. Comparison of split-versus connected-core supplies in the POWER6 microprocessor. In *ISSCC*, 2007.
[16] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *HPCA-9*, 2003.
[17] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed early load retirement. In *HPCA-11*, 2005.
[18] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO-35*, 2002.
[19] J. A. Mathew, P. D. Coddington, and K. A. Hawick. Analysis and development of java grande benchmarks. In *Java Grande Conference*, 1999.
[20] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*, 2005.
[21] M. D. Pant et al. An architectural solution for the inductive noise problem due to clock-gating. In *ISLPED*, 1999.
[22] M. D. Powell and T. N. Vijaykumar. Pipeline muffling and a priori current ramping: architectural techniques to reduce high-frequency inductive noise. In *ISLPED*, 2003.
[23] M. D. Powell and T. N. Vijaykumar. Exploiting resonant behavior to reduce inductive noise. In *ISCA-28*, 2004.
[24] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *ASPLOS*, 2006.
[25] 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.
[26] N. J. Wang and S. J. Patel. ReStore: Symptom-based soft error detection in microprocessors. *TDSC.*, 2006.
[27] H.-S. Yun and J. Kim. Power-aware Modulo Scheduling for High-Performance VLIW Processors. In *ISLPED*, 2001.