Eliminating Voltage Emergencies via Software-Guided Code Transformations

VIJAY JANAPA REDDI, SIMONE CAMPANONI, MEETA S. GUPTA,
MICHAEL D. SMITH, GU-YEON WEI, DAVID BROOKS
Harvard University
and
KIM HAZELWOOD
University of Virginia

In recent years, circuit reliability in modern high-performance processors has become increasingly important. Shrinking feature sizes and diminishing supply voltages have made circuits more sensitive to microprocessor supply voltage fluctuations. These fluctuations result from the natural variation of processor activity as workloads execute, but when left unattended, these voltage fluctuations can lead to timing violations or even transistor lifetime issues. In this paper, we present a hardware-software collaborative approach to mitigate voltage fluctuations. A checkpoint-recovery mechanism rectifies errors when voltage violates maximum tolerance settings, while a run-time software layer reschedules the program's instruction stream to prevent recurring violations at the same program location. The run-time layer, combined with the proposed code rescheduling algorithm, removes 60% of all violations with minimal overhead, thereby significantly improving overall performance. Our solution is a radical departure from the ongoing industry standard approach to circumvent the issue altogether by optimizing for the worst case voltage flux, which compromises power and performance efficiency severely, especially looking ahead to future technology generations. Existing conservative approaches will have severe implications on the ability to deliver efficient microprocessors. The proposed technique reassembles a traditional reliability problem as a runtime performance optimization problem, thus allowing us to design processors for typical case operation by building intelligent algorithms that can prevent recurring violations.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms: Performance, Reliability

Additional Key Words and Phrases: Voltage Noise, dI/dt, Inductive Noise, Voltage Emergencies

1. INTRODUCTION

Power supply noise directly affects the robustness and performance of microprocessors. With the use of ever lower supply voltages and aggressive power management techniques such as clock gating, resulting large current swings are becoming inevitable. 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, while significant overshoots in the voltage can cause long-term degradation of 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 [James et al. 2007] 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. For instance, Bowman et al. show that removing a 10% operating voltage margin leads to a 15% improvement in clock frequency [Bowman et al. 2008].

As an alternative to such conservative design, researchers have proposed designing for average-case operating conditions while providing a "fail-safe" hardware-based mechanism that guarantees correctness in the presence of voltage emergencies. Such a fail-safe mechanism enables more aggressive timing margins in order to maximize clock frequency, or even improve energy efficiency, but at the expense of some runtime penalty when violations occur. Architecture- and circuit-level techniques either proactively take measures to prevent a potentially impending voltage emergency [Ayers 2002; Joseph et al. 2003; Powell and Vijaykumar 2003; 2004], or operate reactively by recovering a correct processor state after an emergency corrupts machine execution [Gupta et al. 2008].

Traditional hardware techniques do not exploit the effect of program structure on emergencies. Figure 1 shows the number of unique static program locations or instructions that are responsible for emergencies[1] on our simulated platform (see Section 4.1), 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 program instructions 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, and cannot exploit the fact that there are just a few emergency code hotspots responsible for nearly all emergencies. 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 [Gupta et al. 2008]. When combined with implementation costs, potential changes to traditional architectural structures, and challenges like response-time delays [Gupta et al. 2008], design, validation and wide-scale retargetability all become increasingly difficult.

In this paper, we present a hardware-software collaborative approach for handling voltage emergencies. Hazelwood and Brooks [2004] suggest the potential for a collaborative scheme, but we demonstrate and evaluate a full-system implementation. The collaborative approach relies on a general-purpose fail-safe mechanism as infrequently as possible to handle emergencies, while 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.

Our software transformation to prevent emergencies is a form of performance optimization because preventing emergencies at aggressive margins leads to better performance, due to reduced fail-safe recoveries. The software layer relies on feedback from the hardware to identify and eliminate emergency-prone program addresses, which is similar to present day industrial-strength virtual machines that target runtime performance optimization using feedback from hardware performance counters [Schneider et al. 2007; Lau et al. 2006]. In the future, we envision run-time systems treating reliability transformations as a

[1] We use the event categorization algorithm described by Gupta *et al.* [Gupta et al. 2007] to identify the instruction that gives rise to an emergency.

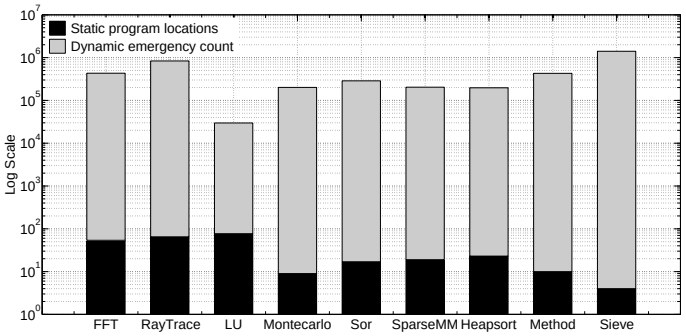


Fig. 1: A small set of static program locations or instructions (fewer than 100) are responsible for nearly all voltage emergencies. Any voltage crossing beyond the 4% operating margin is considered an emergency in our experimental setup, which is described in Section 4.1.

class of dynamic performance optimization.

Dynamic optimization systems [Bala et al. 2000] 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. In contrast to hardware techniques, a compiler-assisted scheme can exploit the fact that programs have so few static emergency-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 rescheduling existing code or injecting new code into the dynamic instruction stream of a program.

Unlike throttling-based hardware schemes, our solution does not require design-time package- and microarchitecture-specific solutions. A dynamic compiler is inherently fine-grained, code-aware, and machine-specific, and it can adapt to the run-time environment. Our collaborative design is a more holistic technique for handling voltage emergencies, as compared to prior hardware techniques. Therefore, our solution allows us to more easily harness the benefits of improved energy efficiency or performance improvement that aggressive margins enable.

The primary contributions of this paper 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 that general-purpose checkpoint-recovery hardware is useful to infrequently tolerate voltage emergencies at aggressive operating margins when combined with our hardware and software co-design approach.

The rest of the paper is organized as follows: Section 2 presents the structure of the proposed hardware-software collaborative approach along with design details for each of the individual hardware and software components. Section 3 presents a code transformation algorithm that we employ to smooth the voltage of the executing program, after the region has been identified. Section 4 discusses performance results, Section 5 discusses related work and Section 6 concludes the paper.

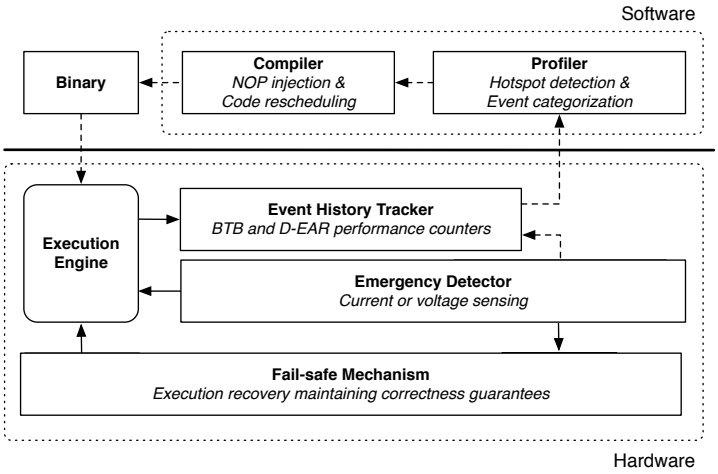


Fig. 2: Workflow diagram of the proposed software-assisted hardware-guaranteed architecture to deal with voltage emergencies.

2. A COLLABORATIVE FRAMEWORK TO MITIGATE VOLTAGE NOISE

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 eliminate future emergencies at the same program location.

2.2 Hardware Design

The hardware support mechanism consists of a voltage emergency detector that identifies when an emergency has occurred, a fail-safe mechanism that engages after every emergency to provide a rollback mechanism, and an event history tracker that is used to

communicate to the software component.

2.2.1 *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.

2.2.2 *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 [Wang and Patel 2006; Agarwal et al. 2004]. These are primarily based on checkpoint and rollback. We use explicit checkpointing, which is a scheme already shipping in production systems [Ando et al. 2003; Slegel et al. 1999].

Explicit-checkpoint mechanisms rely on explicitly saving the architectural state of the processor, i.e., the architectural registers and updated memory state. But there is substantial overhead associated with restoring the register state, and there are additional cache misses at the time of recovery (a buffered memory update is assumed, with updated lines between checkpoints marked as volatile). Moreover, a robust explicit-checkpoint mechanism for noise margin violations must be independent of sensor delays. Any checkpoint falling after a violation but before its subsequent detection due to sensor delays must be considered corrupt. Therefore, providing correct recovery semantics requires maintaining two checkpoints. The interval between checkpoints is just tens of cycles.

While we choose explicit checkpointing for evaluation in this paper, the overall approach is independent of the specific checkpointing implementation. So we refer readers to Section 5 for alternative checkpointing schemes that could be used in place of the explicit checkpointing mechanism.

2.2.3 *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 like the IPF and PowerPC systems. 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 translation lookaside buffer (TLB) misses. The software extracts this information whenever it receives a notification about an emergency.

2.3 Software Design

The software component consists of a profiler that converts the information gathered by the hardware event history tracker into a particular location in the code, and a compiler that analyzes and modifies the program to prevent future recurrences.

2.3.1 *Profiler.* The profiler is notified whenever a hardware 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

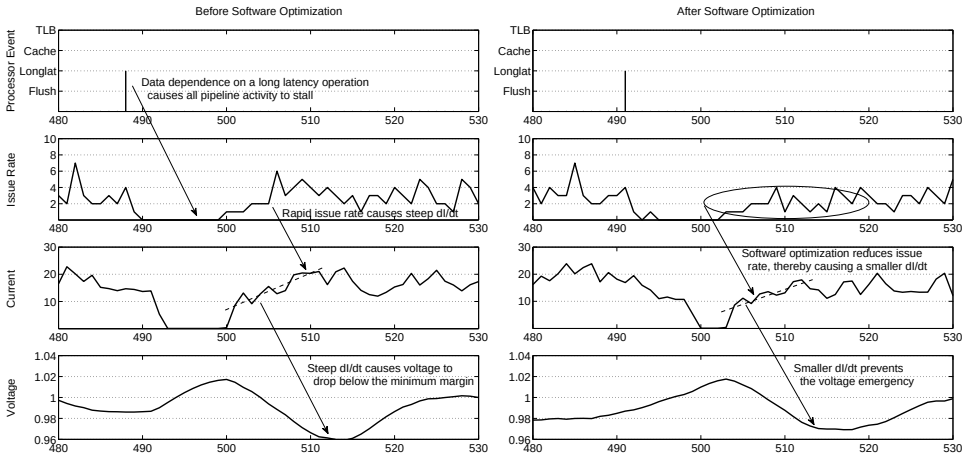


Fig. 3: A 50-cycle execution snapshot of benchmark *Sieve*. It shows the impact of a pipeline stall due to a long latency operation on processor current and voltage. An operating margin of 4% is assumed (i.e., a maximum of 1.04V and minimum of 0.96V). (a) Before Software Optimization shows how a stall 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 after the long latency operation to eliminate the emergency illustrated in (a).

profiler locates the instruction responsible for an emergency using the *event categorization* algorithm [Gupta et al. 2007]. The algorithm works on an out-of-order superscalar machine and it is important to note that the compiler is sensitive to the algorithm's effectiveness, as the algorithm is responsible for directing the compiler to the appropriate code location to target. We refer to this problematic instruction as the *root-cause* instruction and we rely on the robustness of the algorithm provided by prior work to identify the root-cause correctly.

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 such a scenario using the experimental setup we describe in Section 4.1. A data dependence on a long-latency operation stalls all processor activity. When the operation completes, the 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.

Generally, there are several other causes of voltage emergencies, ranging from cache misses to branch mispredictions and TLB misses. We characterize these for the benchmarks we evaluate later in Section 4.1.3. The profiler is equipped to detect the root-cause for all types of emergencies. In this work, we do not focus on eliminating the events that lead to an emergency, rather we focus on smoothing activity following the event to prevent an emergency, since in reality it is impossible to eliminate every microarchitectural event from a real system.

2.3.2 *Compiler*. Figure 3(a) illustrates that voltage emergencies depend on the issue rate of the machine. Therefore, slowing the issue rate of the machine at the appropriate point can prevent voltage emergencies. We can achieve the same goal in software by altering the program code that gives rise to emergencies at execution time, and can do so without large performance penalties. 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 a 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.

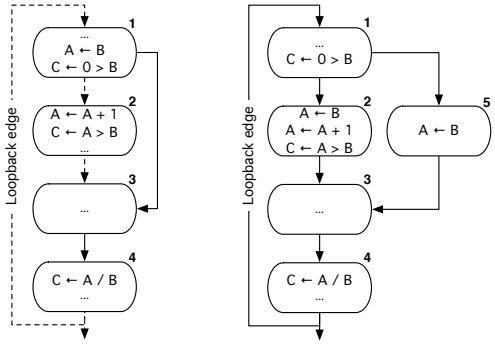
In the following sections, we discuss two approaches we explored for injecting pipeline delays at the software level. We outline one simple approach consisting of inserting nops, and a more sophisticated approach that exploits existing RAW dependencies. Later, in Section 4, we evaluate each approach in turn.

NOP Injection. A simple way for the compiler to slow the pipeline is to insert 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. Since these *pseudo-NOP* instructions perform no useful work, this approach often degrades performance.

The compiler attempts to construct the pseudo-NOP instruction sequence utilizing only dead registers. However, this is not always feasible. In such cases, the compiler spills the contents of live general purpose registers needed for pseudo-NOP code generation. Following the creation and insertion of the pseudo-NOP code in the appropriate location, the compiler fills back live register state and returns control back to the original program code instruction sequence. Therefore, in addition to wasted cycles due to pseudo-NOP code execution, the system may experience additional performance loss due to register spills and fills.

Code Rescheduling. A better way to smooth processor activity is to exploit RAW dependencies already existing in the original control flow graph (CFG) of the program. This constrains the burst of activity when the machine resumes execution after the stall, which prevents the emergency. Whether the compiler can successfully move instructions to create a sequence of RAW dependencies depends on whether moving the code violates either control dependencies or data dependencies. From a high level, 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 such that the original program semantics are still maintained.

To illustrate our code rescheduling approach, in Figure 4(a) we present 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



(a) Before. (b) After.

Fig. 4: Effect of code rescheduling on an emergency-prone loop from benchmark *Sieve*. (a) An emergency consistently occurs 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.

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 after rescheduling is illustrated in Figure 3(b). Activity in this figure is slightly offset to the right by about 5 clock cycles from Figure 3(a) due to subtle changes to the loop structure from code rescheduling. Nevertheless, the stall event still occurs at the same program location. 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 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. Using this anchor point, the software code 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 increases the chances that the machine's scheduling window will fill with dependent code, reducing the issue rate. Otherwise, the compiler must generate multiple sets of smaller RAW dependence chains.

Instruction	Description	Event Type
Root-cause	Instruction identified by the hardware as the cause of an emergency.	All but BR.
Last-writeback	Most recent instruction in the write back stage of the pipeline.	All but BR.
Wrong-path	First instruction along the speculative path prior to detecting a misprediction.	All (including BR).

Table I: Types of instructions that the code rescheduling algorithm targets depending on the event responsible for an emergency.

In the following section, we present a detailed description of our algorithm, which is a specific instantiation of the general concept we propose to prevent emergencies—staggering the issue rate using RAW dependence chains.

3. SOFTWARE-BASED CODE RESCHEDULING ALGORITHM

Given a root-cause instruction, our scheduler constrains the instruction issue rate at different points within the CFG. The scheduler transforms the code differently depending on whether or not the emergency was caused by a branch misprediction. In the simple case, such as an emergency caused by a sudden burst of activity following a cache miss event or a long latency stall (as illustrated in Figure 3), the scheduler targets the root-cause instruction and the last writeback instruction to successfully remove emergencies. Table I describes these instruction types and indicates under which event conditions the code rescheduler targets them. We consider these two particular locations to prevent the out-of-order issue logic from intelligently bypassing the RAW dependence chain put in place to prevent the emergency. The hardware may discover some other instruction sequences also ready for execution. These other sequences could lead to a burst of activity that can cause an emergency, thus rendering our transformations ineffective. Therefore, we conservatively target two locations to constrain the issue rate.

When an emergency is caused by a branch misprediction (BR), the scheduler must take into account the speculative set of instructions executed by the machine. We experimentally discovered that constraining the issue rate before a pipeline flush event along the wrong path significantly increases the chances of preventing an emergency. Therefore, to prevent branch misprediction-related emergencies, the scheduler targets the root-cause instruction, the last writeback instruction, as well as the first instruction along the speculative path that is executed just prior to detecting the branch misprediction.

Algorithm 1 illustrates the highest-level pseudocode that the compiler invokes to transform the code at the point of an emergency (i.e., root-cause instruction r). It takes as input the three input instructions described above that the Profiler mechanism (illustrated in Figure 2) identified. The algorithm then invokes the Scheduler function to transform the code in order to constrain the issue rate just before a specific instruction: the algorithm constrains the issue rate on the last write back instruction regardless of the emergency type and before every successor of the root-cause instruction. However, depending on the emergency type, we decide the successor paths on which to constrain the issue rate. In the case of a branch misprediction-related emergency, we constrain the issue rate on the fallthrough, as well the taken path, thereby smoothing voltage along the speculative path as well.

Determining Candidates for Code Motion. The Scheduler function discovers and schedules a RAW chain before its input parameter instruction a. To locate the closest and longest RAW chain, the Scheduler invokes the GlobalCandidate function. The GlobalCandidate function defines the scope or range of basic blocks from within which the LocalCandidate

Algorithm 1: Highest-level routine for performing instruction scheduling to prevent voltage emergencies

```
Input: Emergence type t
Input: Root-cause instruction r
Input: Last write-back instruction l
Input: Wrong instruction w
Scheduler(l) ;
switch t do
    case Branch misprediction-related emergency
        a ∈ Succ(r)|a ≠ w ;
        Scheduler(a) ;
        Scheduler(w) ;
    end
    otherwise
        a ∈ Succ(r) ;
        Scheduler(a) ;
    end
end
```

Function `Scheduler(`a`)`

```
Input: Instruction a
l = GlobalCandidate(a) ;
if length(i) > 0 then
    MarkScheduled(i) ;
    GCSMove(i, a) ;
end
```

function attempts to construct the longest RAW dependence chain. When LocalCandidate fails (for instance, when no dependent instructions can be found), GlobalCandidate enlarges the range of basic blocks to consider and the process repeats.

The return value of GlobalCandidate is a linked list of instructions l that can be successfully scheduled. If this list is not null, the Scheduler function notes these instructions as already visited using the MarkScheduled function. Visited or previously scheduled instructions cannot be subsequently rescheduled, as that would perturb or invalidate a previously scheduled RAW chain, or could lead to schedule thrashing.

Performing Code Motion. Upon identifying a useful RAW chain from GlobalCandidate, the Scheduler function calls GCSMove to migrate the necessary set of instructions from one location to another. GCSMove is based on the standard *Global Code Scheduling* (GCS) algorithm [Aho et al. 2006]. Briefly, the GCS algorithm clones instructions as necessary to move instructions. It discovers the necessary set of clones by means of the pre and post dominance relations computed using the CFG. An instruction a predominates instruction b if, and only if, instruction a always executes before instruction b. Instruction b postdominates instruction a if, and only if, instruction b is always executed after executing instruction a. If the instruction to schedule, say b, postdominates target instruction a, and a predominates b, then no instruction cloning is necessary. However, if this condition does

Function `GlobalCandidate(`a`)`

Input: Instruction a
Output: Linked list of instructions
S = $BasicBlock(a)$;
i = $\{\}$;
while $i == \{\} \wedge S \neq CFG$ **do**
  i = $LocalCandidate(S, a)$;
  S_1 = S ;
  forall $s \in S$ **do**
    S_1 = $S_1 \cup Succ(s) \cup Prev(s)$;
  end
  S = S_1 ;
  forall $s \in S$ **do**
    $S_1 = S_1 \cup BasicBlock(s)$;
  end
  S = S_1 ;
end
return i ;

not hold, instructions must be cloned and inserted in positions found by the *anticipated expressions* computed using data-flow analysis [Aho et al. 2006].

The LocalCandidate function attempts to construct the longest dependence chain using the MoveableBefore function. This intermediate MoveableBefore function checks to see if the first instruction s given as its input can be moved just prior to its target a by means of GCS. We impose constraints within MovableBefore to prevent perturbing the original voltage profile so much so that our constructive code transformations become ineffective. Specifically, we impose instruction cloning rules:

(1) The head of the RAW chain, instruction s, can be scheduled before target a assuming no limit on the number of clones necessary to migrate s anywhere within the scope defined by the GlobalCandidate function.
(2) All other instructions belonging to the RAW chain can be cloned at most once.
(3) Allowed cloning cannot increase the dynamic instruction of the program, since aggressive cloning can potentially impact performance.

If these conditions are not satisfied, the LocalCandidate function returns a null list of instructions, forcing GlobalCandidate to enlarge the scope and retry. Readers are welcome to relax these constraints in an attempt to improve the chances of finding a suitable RAW dependence chain. However, there is a risk of increasing the execution time, and even potentially perturbing neighboring code so much so that the transformed code leads to new emergencies.

A Demonstration of the Code Rescheduling Algorithm. To facilitate better understanding, here we illustrate the functionality of the code rescheduling algorithm with a simplified example extracted from a real scenario in benchmark *RayTrace*. Consider the original program CFG and its related Data-Dependence Graph (DDG) shown in Figure 5a and Figure 5c, respectively. Instruction `4` is the root-cause related to a branch misprediction. Instruction `8` corresponds to the wrong path instruction, or the first instruction

Function `LocalCandidate(`S, a`)`

Input: Instruction set S
Input: Instruction a
Output: Linked list of instructions
$C = \emptyset$;
forall $s \in S$ **do**
  if *MovableBefore(*s*,* a*)* $\wedge\neg$ *Marked(*s*)* **then**
    $C = C \cup \{s\}$;
  end
end
$j \in C$;
forall $c \in C$ **do**
  if *DataDependencesLength(*c*,* a*)* $>$ *DataDependencesLength(*j*,* a*)* **then**
    $j = a$;
  end
end
return $LongestRAWDependenceChain(j)$;

executed along the incorrectly speculated path. In order to smooth the voltage emergency at the root-cause, the scheduler attempts to add a RAW dependence chain of instructions between instructions 4 and 5, instructions 4 and 8 and just before the last writeback instruction. For simplicity, we only elaborate the steps taken to construct the chain between instructions 4 and 5.

The algorithm starts by looking for the best RAW chain by calling the GlobalCandidate function, giving instruction 5 as its input. The GlobalCandidate function calls LocalCandidate to find the longest RAW chain inside the present scope of interest, which is the basic block containing instruction 5. The LocalCandidate function returns null upon first invocation. Consequently, GlobalCandidate enlarges the scope and re-invokes the LocalCandidate function. Figure 5a illustrates this scope enlargement process using the initially small dotted inner circle, and subsequently enlarging the scope to include more basic blocks.

During the subsequent call to LocalCandidate, several additional blocks are chosen for creating the RAW chain. These basic blocks are chosen because they are within one edge distance away from all basic blocks previously considered. At this point, the algorithm finds six candidate instructions (1, 2, 9, 10, 11 and 14) as heads of RAW chains. Hence, we have $C = \{1, 2, 9, 10, 11, 14\}$. From this set of six potential chains, LocalCandidate chooses the longest RAW chain it can create without violating our cloning rules. It finds instruction 1 as the best candidate. Moving instruction 1 along with its data-dependent sequence (instructions 1, 2 and 3) between instructions 4 and 5 leads to an optimum solution with a chain length of three. Note that while instruction 9 can lead to a RAW chain length of 4, LocalCandidate cannot choose this alternative because we specified that cloning cannot increase the dynamic instruction of the program. Alternative implementations of our algorithm that relax this constraint are possible for improving emergency coverage, albeit at the risk of potentially slower runtime performance. The transformed CFG is shown in Figure 5b, where we see that instructions 1, 3, and 5 have been replicated and migrated down the CFG.

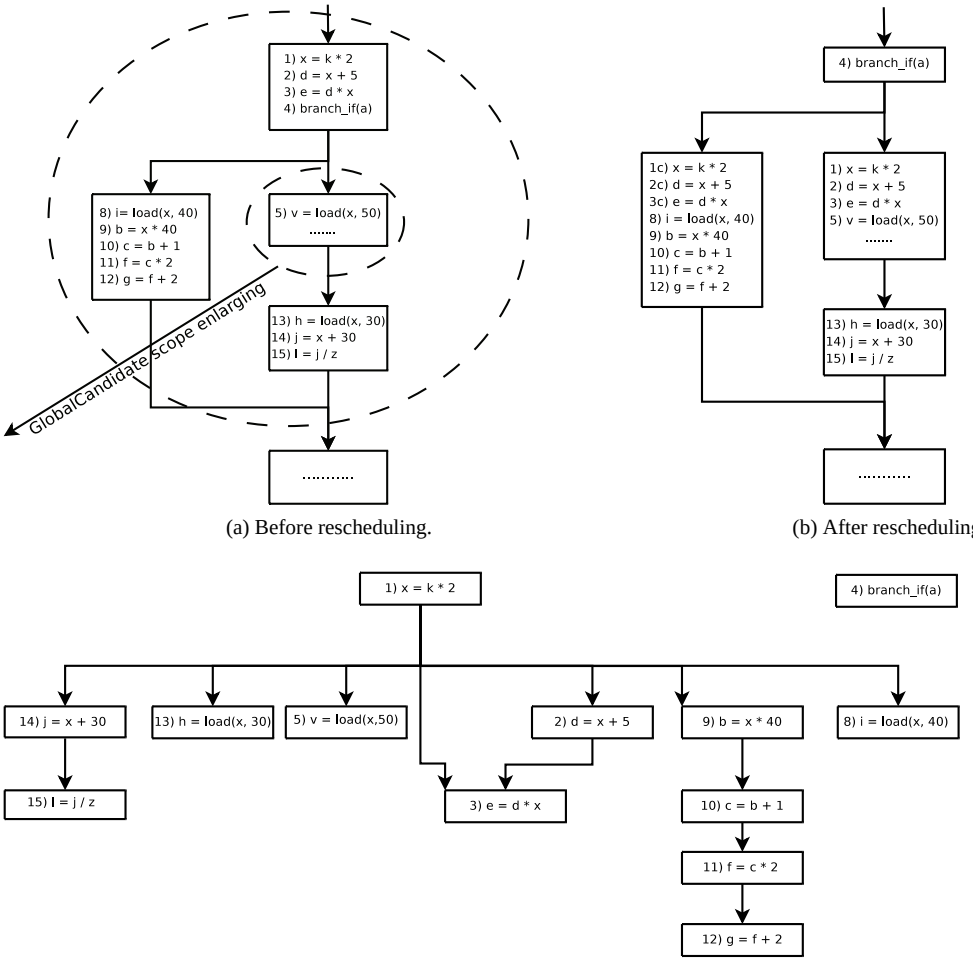


Fig. 5: (a) Control flow graph of an emergency-prone piece of code from benchmark *RayTrace*. (b) Rescheduled code after the compiler moves instructions to remove the emergency caused by the frequently mispredicted branch at location 4. (c) Data dependence graph corresponding to the original code that the rescheduling algorithm uses to extract the safest RAW dependence chain.

4. EVALUATION

Our system evaluation demonstrates the effectiveness of the compiler at reducing voltage emergencies and shows the impact of its code changes on performance. After showing that the compiler can reduce over 60% of emergencies (Section 4.2) with minimal overheads (Section 4.3), we present a performance study (Section 4.4) showing that our software-assisted scheme overcomes the challenges of existing hardware techniques effectively.

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 II: Baseline architecture parameters for SimpleScalar.

4.1 Experimental Setup

Given that modern hardware does not support fine-grained access to voltage sensors, we explored our design using a hardware simulator together with an existing software compilation infrastructure.

4.1.1 *Hardware simulator.* We used SimpleScalar/x86 to simulate a Pentium 4 with the characteristics shown in Table II. The modified 8-way superscalar x86 SimpleScalar gathers detailed cycle-accurate current profiles using Wattch [Brooks et al. 2000]. This tool is an architectural simulator that estimates CPU power consumption based on a set of parameterizable power models for different hardware structures using per-cycle resource accounting. To model voltage variations, the simulator convolves the simulated current profiles with an impulse response of the power delivery subsystem [Powell and Vijaykumar 2004; Joseph et al. 2003] each cycle. In this work, we focus on a power delivery subsystem model based on the characteristics of the Pentium 4 package [Aygun et al. 2005], which exhibits a mid-frequency resonance at 100MHz with a peak impedance of 5mΩ. Finally, we assume peak current swings of 16-50A.

4.1.2 *Compiler infrastructure.* We use the ILDJIT [Campanoni et al. 2008] 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.

4.1.3 *Benchmarks.* We use the C# benchmarks that come from the Java Grande benchmark suite [Bull et al. 2000]. Table III presents a summary description of each of the benchmarks. While the programs run for an extended period of time, on the order of billions of instructions, we shorten their execution time to approximately 150 million instructions because of the hardware simulation overhead exhibited by SimpleScalar.

To briefly characterize the voltage emergencies in our benchmarks, Figure 6 shows the distribution of root-causes across the benchmarks. The majority of the emergencies in the Java Grande benchmark suite arise because of stalls due to Long Latency operations, Cache Miss and Branch Misprediction events. The Others category corresponds to those events we were unable to successfully attribute to any specific microarchitectural event. This likely resulted from code-based bursts of activity such as the “power virus” demonstrated by other researchers [Joseph et al. 2003]. Finally, TLB Miss events did not tend to result in

Benchmark	Description
FFT	Performs a one-dimensional forward transform of N different complex numbers
RayTrace	Measures the performance of a 3D ray tracer on a scene containing 64 spheres
LU	Linear system solver that is based on Linpack
Montecarlo	Financial simulation using MonteCarlo techniques
Sor	Performs successive over-relaxation over a grid
SparseMM	Matrix-vector multiplication using an unstructured sparse matrix
Heapsort	Sorts an array of integers using a heap sort algorithm
Method	Determines virtual machine method call overheads
Sieve	Algorithm for finding the prime numbers in a given interval

Table III: Benchmark descriptions.

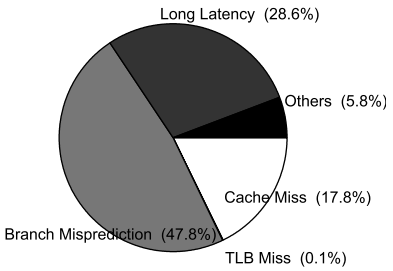


Fig. 6: Aggregate distribution of root-causes across benchmarks in the Java Grande benchmark suite.

emergencies in our evaluated benchmark suite. The absolute number of emergencies per benchmark is shown in Table V.

The emergency distribution we present allows readers to compare the traits of our benchmarks to more traditional benchmarks such as CPU2006. CIL byte code is unavailable for SPEC workloads, so we were unable to evaluate them directly. However, since the distribution and number of emergencies for the Java Grande programs is representative of prior hardware-based work using SPEC workloads [Gupta et al. 2009], we expect our results to generalize, and we feel that the results and contributions of this paper outweigh this limitation of the experimental infrastructure.

4.2 Effectiveness of the Compiler-Based Transformations

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. We first evaluate the effectiveness of NOP injection and code rescheduling, where we find that (1) the choice of transformation affects performance, and that (2) the transformation itself can introduce new emergencies if the scheduler is not careful. Following this analysis, in the next section, we will factor in all costs to evaluate full-system performance.

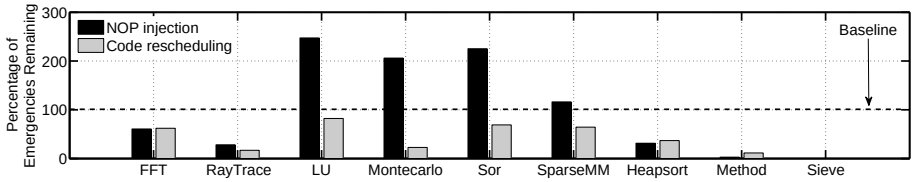


Fig. 7: Percentage of emergencies remaining after code transformation. Lower than the baseline 100% is good, implying fewer emergencies than the original code. Otherwise, it means the transformation lead to more emergencies than the original code.

4.2.1 *NOP injection.* As described earlier, the NOP injection algorithm inserts new instructions into the program path that slows down the machine issue rate as needed to prevent an emergency. In our specific implementation, the sequence is made up of three instructions that form a RAW chain at the intermediate representation-level. But after code generation we find that the sequence typically grows to between six and eight instructions due to register allocation.

The effectiveness of the scheme is shown by the left bar in Figure 7. The bar shows the percentage of emergencies remaining after the compiler has attempted to prevent emergencies by injecting pseudo-NOP code. The number of emergencies is reduced by $\sim$50% or more in benchmarks *FFT*, *RayTrace*, *Method*, *Sieve*, and *Heapsort*, which shows that the transformation can be effective. However, the transformation is not as effective across the remaining benchmarks *LU*, *Montecarlo*, *Sor* and *SparseMM*. In fact, the number of emergencies increases by over twofold for benchmark *LU*. The loops and the specific code paths within that the compiler targets in these benchmarks are under extreme register pressure. Consequently, adding new code leads to frequent spills and fills during each loop iteration. These memory loads and stores cause additional cache and TLB misses. Some become new root-causes that lead to more emergencies than the original code experiences.

Analysis reveals that pseudo-NOP injection does reduce the original program's emergencies, but the transformation itself also gives rise to new emergencies. The compiler generates spill and fill code to create the pseudo-NOP code sequence. 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 and the properties of the original instruction schedule, among other conditions. It is difficult to predict the current and voltage response activity that will result from injecting new code, so the new emergencies are not easy to avoid, as we see in the case of *LU*, *Montecarlo*, *Sor*, and *SparseMM*.

Additionally, the run-time performance of the original program suffers with the injection of pseudo-NOP code, as the injected code does not serve the original program's purpose. The left bar in Figure 8 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.

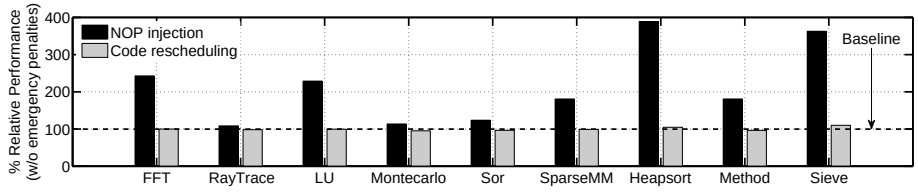


Fig. 8: Code performance after transformation. The cost for handling emergencies is not shown in this plot to isolate the effect of code transformation on the run-time performance. Section 4.4 evaluates overall performance after factoring in code performance costs, along with penalties for handling emergencies.

4.2.2 *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 7). Second, it does so without dramatically increasing the execution time of a program (as shown in Figure 8). 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.

By restricting the compiler's scheduling algorithm to the strict cloning rules described in Section 2.3.2, we were able to effectively limit performance loss from injecting new instructions. Table IV shows that the number of instructions added due to cloning is in the order of tens of instructions. Thus when the dynamic instruction count of the program does increase (resulting from the register allocator generating spill/fill code) it does so by a tiny amount. These instruction increases are especially insignificant when considering that the benchmarks execute hundreds of millions of instructions. In some benchmarks such as *Method* and *Heapsort*, the dynamic instruction count decreases by a small percentage because code transformations change register allocation, leading to fewer register spills and fills along the specialized paths.

Changes in the run-time performance of the rescheduled code are generally in the noise for all benchmarks, and the reduction in emergencies averages $\sim$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. Therefore, some emergencies continue to persist. Making code transformations can lead to new emergencies root-causes as well. Figure 9 illustrates this breakdown. As we are careful to not aggressively modify the code surrounding a root-cause, we see that the percentage of new emergencies introduced is a very small fraction of all emergencies.

Ideally, the scheduling algorithm should attempt to create a RAW dependence chain long

Benchmark	# of Instructions		% Change in
	Cloned	Moved	Dynamic Instructions
FFT	7	30	0.0
RayTrace	20	40	-0.24
LU	28	64	0.1
Montecarlo	23	53	5.2
Sor	39	77	2.3
SparseMM	33	67	3.3
Heapsort	37	61	-1.0
Method	2	8	-3.7
Sieve	7	11	0.0

Table IV: Only a small percentage of the static code (in the order of tens of instructions) need modification to eliminate emergencies. Additionally, the changes the compiler makes has minimal impact on the dynamic instruction count.

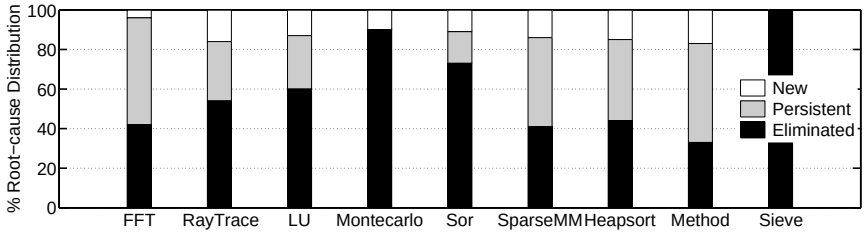


Fig. 9: Not all emergencies can be eliminated. Some root-causes cannot be fixed because the compiler cannot find sufficient code to construct RAW dependence chains. Also, new emergencies can be introduced as a result of making transformations to existing code.

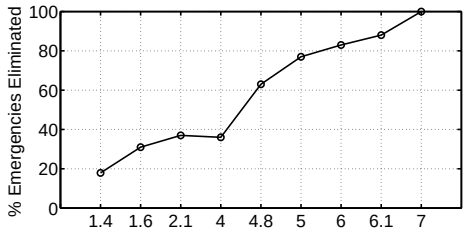


Fig. 10: There is a correlation between the number of emergencies the compiler can eliminate and the average length of the dependence chains it creates. The compiler can eliminate more emergencies as it creates chain lengths that approach the machine's issue width. Our machine is 8-wide.

enough to block the issue width of the machine. We find that there is a strong correlation between the length of the RAW dependence chain and how successfully the compiler can eliminate emergencies. Figure 10 plots the average RAW chain length on the x-axis. The percentage of emergencies eliminated across the different benchmarks is presented on the y-axis. The simulated machine has an issue width of 8 instructions, and we find that the number of emergencies eliminated steadily grows towards 100% as the length of the RAW chain approaches the issue width of the machine.

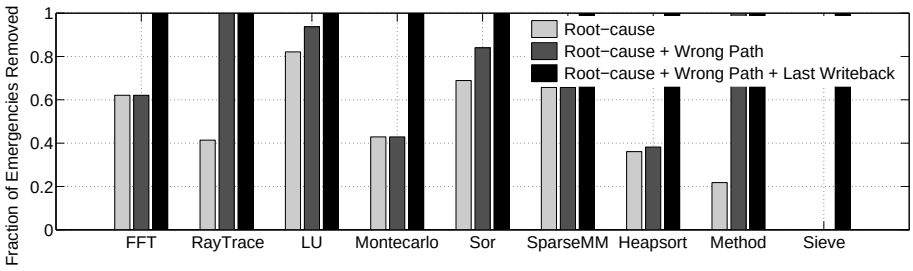


Fig. 11: This figure justifies the use of three program points for resolving voltage emergencies. The combination of the root-cause instruction, the wrong path instruction, and the last writeback instruction, results in the ability to identify and resolve nearly all of the voltage emergencies encountered.

In Section 2.3.2, we mentioned that the compiler's instruction scheduler targeted three specific points of interest in the CFG for an emergency: the root-cause instruction, the last write-back instruction, and in the case of a branch misprediction-related emergency, the first instruction along the speculative path. We made a qualitative argument that these three points provided good coverage to eliminate emergencies successfully, but here we quantitatively justify that claim.

Assuming all three points are covered as the baseline, Figure 11 shows how effective the compiler is at removing emergencies as we increase the number of points the scheduler targets. We examine the three points here cumulatively, starting with the root-cause, but we do not claim they are disjoint. The graph is normalized to 1, which indicates the utmost number of emergencies we are able to eliminate using the code rescheduling algorithm. This number corresponds to the Code rescheduling bar shown in Figure 7.

The left-most bar in Figure 11 shows the effect of targeting only the Root-cause instruction. Since higher values mean fewer emergencies, we observe that the root-cause instruction alone is insufficient, and the effectiveness of the scheduler increases as we consider the Last writeback and Wrong path points. This is especially the case for programs that are control intensive such as benchmarks *RayTrace* and *Method*. Most of the emergencies in these benchmarks arise because of branch mispredictions, therefore ignoring the issue rate on the incorrectly speculated path can have a significant impact. However, by covering the speculative execution path as well, efficiency improves on *RayTrace* by 60% and *Method* by nearly 80%.

Finally, all benchmarks, with the exception of *RayTrace* and *Method*, cover 100% of emergencies when we take into account the Last writeback point. Our general consensus is that if the program is highly data intensive with few control flow changes, then throttling the issue rate at the last writeback instruction has a positive effect. The benchmark that benefits the most from the Last writeback transformation is *Sieve*, where all emergencies eliminated were the result of focusing on the Last writeback instruction.

4.3 Compiler-Based Transformation Overhead

Our compiler cannot recompile itself, therefore we incur rollback penalties whenever the compiler is itself executing. This includes the scenario when the compiler is generating new dynamic code, as well as when the compiler is transforming existing code to prevent emergencies. Table V shows the distribution of emergencies between the compiler and

Benchmark	Number of Emergencies	
	Runtime Compiler	Application Code
FFT	639	431368
RayTrace	16	834753
LU	2	29639
Montecarlo	0	201355
Sor	16	286487
SparseMM	203	203759
Heapsort	299	196915
Method	763	428671
Sieve	0	1407500

Table V: Number of emergencies that arise as the compiler generated application code is running versus when the compiler is itself running (either for generating newly requested dynamic code or while transforming existing application code to prevent emergencies).

Benchmark	% of Execution Time	
	Runtime Compiler	Application Code
FFT	0.087	99.913
RayTrace	0.151	99.849
LU	0.082	99.918
Montecarlo	0.010	99.990
Sor	0.020	99.980
SparseMM	0.024	99.976
Heapsort	0.021	99.979
Method	0.010	99.990
Sieve	0.001	99.999

Table VI: Distribution of execution time spent handling emergencies in the compiler versus running application code.

generated application code. The data strongly indicates that the fraction of emergencies encountered during compiler execution is less than 1% on average across all benchmarks. Since the fraction of emergencies is so small, compiler-associated rollback overhead is insignificant.

Based on these results, the overhead of run-time code transformation to fix and eliminate emergencies appears to be insignificant. Figure 1 showed 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. Table VI substantiates this claim by demonstrating that the percentage of execution time spent running generated application code is substantially larger than the time spent in the compiler executing the rescheduling algorithm.

4.4 Full-System Performance Evaluation

Reducing operating voltage margins allows for frequency improvements and/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 our dynamic compilation strategy complements general-purpose checkpoint-recovery for

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 VII: 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.

voltage emergencies, enabling very aggressive operating margins in the processor. Performance gains for our collaborative approach are within four percentage points of an oracle-based throttling scheme. Results are presented in Table VII.

Bowman et al. show that removing a 10% operating voltage margin leads to a 15% improvement in clock frequency [Bowman et al. 2008]. This indicates a 1.5x scaling factor 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, leads to 27% lower frequency. If we take this conservative margin as the baseline and reduce the 18% margin to 4% while avoiding voltage emergencies, the resulting ideal clock frequency improvement could be $\sim$29%. This sets the upper bound on frequency gains achievable. We make the simplifying assumption that frequency improvements directly translate to higher overall system performance.

4.4.1 *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 for the benchmarks we evaluated. Performance gains from scaling the operating margin down to 4% are minor at only 3%. This minimal improvement in performance implies that explicit-checkpointing by itself cannot handle voltage emergencies successfully at aggressive margins.

4.4.2 *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 our proposed software counterpart. Of the two compiler transformations discussed in Section 2.3 we evaluate the code rescheduling transformation only, since it appeared to be the most promising technique for effectively reducing the number of emergencies without a detrimental performance impact.

The profiler identifies root-cause instructions as the fail-safe checkpoint scheme initiates

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

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 permanently 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 7 and discussed in Section 4.2). Combining explicit checkpointing with compiler assistance reduces checkpointing overhead substantially, from 25% to 7.6%. This translates to a net performance gain of $\sim$20%.

4.4.3 *Performance 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 [Ayers 2002; Joseph et al. 2003; Powell and Vijaykumar 2003; 2004]. 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 $\sim$24% improvement in performance for tightened margins, which is just four 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 [Gupta et al. 2008].

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 [Ando et al. 2003; Slegel et al. 1999] to reduce voltage emergencies at very aggressive margins that enable significant performance gains.

5. RELATED WORK

Fail-safe mechanism alternatives. There are unique tradeoffs one should consider in choosing a fail-safe mechanism. One that is relevant to this work involves balancing the complexity of the checkpoint-recovery hardware with the recovery scheme's impact on runtime performance while the machine is executing smoothly.

Gupta et al. [2008] propose an implicit-checkpoint-restart scheme based on delayed commit and rollback that speculatively buffers processor updates to the machine state until it is verified that no noise margin violations have occurred within the time it takes to detect an emergency. To guarantee system correctness, the implicit-checkpoint mechanism distinguishes between a *noise-verified* state and a *noise-speculative* state. In the noise-verified state, the machine is known to be free of corruption caused by inductive noise. Completed results are buffered in the reorder buffer (ROB) or store queue (STQ) until they are verified

to reflect no ill effects from noise violations.

The buffer time for the implicit scheme is determined by the emergency detector's sensor delays; it takes time for voltage sensors across the chip to detect a droop event and subsequently broadcast the error signal across the processor to initiate recovery. However, this delay is only in the order of a few clock cycles, therefore its impact on performance while the machine is executing smoothly is small, if not even negligible in some workloads. Moreover, the cost of restoring state under this scheme is effectively as low as flushing the pipeline due to a branch misprediction. Overall, this design greatly simplifies the complexity of the checkpoint-recovery hardware since it leverages existing traditional microarchitectural structures. But therein lies the problem. The scheme is intrusive and requires changes to traditional microarchitectural structures that increase design cost and validation time. Moreover, it is a highly custom solution to deal with voltage emergencies with no general purpose applicability.

By comparison, the explicit checkpointing we use as our fail-safe mechanism is less intrusive addition to existing processor designs, and it is likely to be useful for other purposes than suppressing voltage emergencies. Several researchers have proposed a variety of diverse applications using checkpoint-recovery hardware [Wang and Patel 2006; Sorin et al. 2000; Martínez et al. 2002; Kirman et al. 2005; Shyam et al. 2006; Narayanasamy et al. 2005]. Our use of checkpoint-recovery for handling inductive noise in collaboration with software is another novel application of this general-purpose hardware. Explicit-checkpointing by itself, however, cannot be used to handle voltage emergencies because the performance penalties are too large (as discussed in Section 4.4.1).

Hardware-based solutions. Prior work suggests preventing emergencies by altering machine behavior via execution throttling [Ayers 2002; Joseph et al. 2003; Powell and Vijaykumar 2003; 2004] or staggering the issue rate [Powell and Vijaykumar 2003; Pant et al. 1999]. Hardware mechanisms face increasing sensor delay problems as the margins are reduced aggressively. The feedback loop delay between detecting and engaging the preventive mechanism becomes a limiting factor to how aggressively we can reduce the operating voltage margins. By comparison, the mechanism we propose allows emergencies to occur and then recovers and eliminates them, thus avoiding the sensor delay issue altogether.

Software-based prior effort. Toburen [1999] and Yun and Kim [2001] demonstrate static compiler techniques that can target voltage emergencies. However, voltage 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. Our mechanism dynamically discovers the emergency hotspots, and can adapt to them effectively. So our scheme is more robust for wide-scale deployment in the coming era where designing reliable processors is becoming increasingly challenging.

6. CONCLUSION

The primary contribution of this work is a full system design and implementation 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 coarse-grained 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 $\sim$20% higher performance, which is within 4 percentage points of an oracle-based throttling scheme.

Our rescheduling algorithm and general framework are a first step towards exposing voltage noise to the higher-level software stack. But with even tighter coupling between hardware and software, we can reduce the complexity of noise-reduction algorithms, making it more readily feasible for software to play an integral role in assisting with hardware issues. For instance, rather than trying to dynamically construct dependence chains that throttle the issue rate of the machine, a hardware hook that allows the compiler to more directly request temporary reductions in issue width would greatly simplify the algorithm. The compiler would then only need to identify and customize the path along which the issue throttling request is triggered, and not have to worry about finding dependence chains. With such integrated effort designers can recoup increasing operating voltage margin inefficiencies using software assistance and compiler-guided code transformations.

REFERENCES

AGARWAL, S., GARG, R., GUPTA, M. S., AND MOREIRA, J. E. 2004. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*. ACM, New York, NY, USA, 277–286.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 2006. *Compilers: Principles, Techniques and Tools*. Prentice Hall.

ANDO, H., YOSHIDA, Y., INOUE, A., SUGIYAMA, I., ASAKAWA, T., MORITA, K., MUTA, T., MOTOKURUMADA, T., OKADA, S., YAMASHITA, H., SATSUKAWA, Y., KONMOTO, A., YAMASHITA, R., AND SUGIYAMA, H. 2003. A 1.3ghz fifth generation sparc64 microprocessor. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*. ACM, New York, NY, USA, 702–705.

AYERS, D. 2002. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 7.

AYGUN, K., HILL, M. J., EILERT, K., RADHAKRISHNAN, K., AND LEVIN, A. 2005. Power delivery for high-performance microprocessors. *Intel Technology Journal 9*.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: a transparent dynamic optimization system. In *PLDI*.

BOWMAN, K. A. ET AL. 2008. Energg-efficient and metastability-immune timing-error detection and instruction replay-based recovery circuits for dynamic variation tolerance. In *ISSCC*.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA-27*.

BULL, M., SMITH, L., WESTHEAD, M., HENTY, D., AND DAVEY, R. 2000. Benchmarking java grande applications. In *The Practical Applications of Java*.

CAMPANONI, S., AGOSTA, G., AND REGHIZZI, S. C. 2008. A parallel dynamic compiler for cil bytecode. *SIGPLAN Not.*.

GUPTA, M. S., RANGAN, K. K., SMITH, M. D., WEI, G.-Y., AND BROOKS, D. 2007. Towards a software approach to mitigate voltage emergencies. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*. ACM, New York, NY, USA, 123–128.

GUPTA, M. S., RANGAN, K. K., SMITH, M. D., WEI, G.-Y., AND BROOKS, D. 2008. DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors. In *HPCA-14*.

GUPTA, M. S., REDDI, V. J., SMITH, M. D., WEI, G.-Y., AND BROOKS, D. M. 2009. An event-guided approach to handling inductive noise in processors. In *DATE*.

HAZELWOOD, K. AND BROOKS, D. 2004. Eliminating Voltage Emergencies via Microarchitectural Voltage Control Feedback and Dynamic Optimization. In *ISPLED*.

JAMES, N., RESTLE, P., FRIEDRICH, J., HUOTT, B., AND MCCREDIE, B. 2007. Comparison of split-versus connected-core supplies in the POWER6 microprocessor. In *ISSCC*.

JOSEPH, R., BROOKS, D., AND MARTONOSI, M. 2003. Control techniques to eliminate voltage emergencies in high performance processors. In *HPCA-9*.

KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTINEZ, J. 2005. Checkpointed early load retirement. In *HPCA-11*.

LAU, J., ARNOLD, M., HIND, M., AND CALDER, B. 2006. Online performance auditing: using hot optimizations without getting burned. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*.

MARTÍNEZ, J. F., RENAU, J., HUANG, M. C., PRVULOVIC, M., AND TORRELLAS, J. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO-35*.

NARAYANASAMY, S., POKAM, G., AND CALDER, B. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA*.

PANT, M. D., PANT, P., WILLS, D. S., AND TIWARI, V. 1999. An architectural solution for the inductive noise problem due to clock-gating. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*. ACM, New York, NY, USA, 255–257.

POWELL, M. D. AND VIJAYKUMAR, T. N. 2003. Pipeline muffling and a priori current ramping: architectural techniques to reduce high-frequency inductive noise. In *ISLPED*.

POWELL, M. D. AND VIJAYKUMAR, T. N. 2004. Exploiting resonant behavior to reduce inductive noise. In *ISCA-28*.

SCHNEIDER, F. T., PAYER, M., AND GROSS, T. R. 2007. Online optimizations driven by hardware performance monitoring. In *PLDI*.

SHYAM, S., CONSTANTINIDES, K., PHADKE, S., BERTACCO, V., AND AUSTIN, T. 2006. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. *ASPLOS-XII*.

SLEGEL, T. J., AVERILL III, R. M., CHECK, M. A., GIAMEI, B. C., KRUMM, B. W., KRYGOWSKI, C. A., LI, W. H., LIPTAY, J. S., MACDOUGALL, J. D., MCPHERSON, T. J., NAVARRO, J. A., SCHWARZ, E. M., SHUM, K., AND WEBB, C. F. 1999. Ibm's s/390 g5 microprocessor design. *IEEE Micro 19*, 2, 12–23.

SORIN, D. J., MARTIN, M. M. K., HILL, M. D., AND WOOD, D. A. 2000. Fast Checkpoint/Recovery to Support Kilo-instruction Speculation and Hardware Fault Tolerance. Computing science technical report, University of Wisconsin-Madison.

TOBUREN, M. 1999. Power Analysis and Instruction Scheduling for Reduced di/dt in the Execution Core of High-Performance Microprocessors. M.S. thesis, NC State University, USA.

WANG, N. J. AND PATEL, S. J. 2006. ReStore: Symptom-based soft error detection in microprocessors. *TDSC.*.

WILLIAMS, D., SANYAL, A., UPTON, D., MARS, J., GHOSH, S., AND HAZELWOOD, K. 2009. A cross-layer approach to heterogeneity and reliability. In *Proceedings of the 7th ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. Cambridge, MA, USA, 88–97.

YUN, H.-S. AND KIM, J. 2001. Power-aware Modulo Scheduling for High-Performance VLIW Processors. In *ISLPED*.