Software-assisted Hardware Reliability: Enabling Aggressive Timing Speculation Using Run-Time Feedback From Hardware and Software

A dissertation presented

by

Vijay Janapa Reddi

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

May 2010

Thesis advisors | Author

David M. Brooks, Gu-Yeon Wei and Michael D. Smith | **Vijay Janapa Reddi**

Software-assisted Hardware Reliability: Enabling Aggressive Timing Speculation Using Run-Time Feedback From Hardware and Software

Abstract

In the era of nanoscale technology scaling, we are facing the limits of physics, challenging robust and reliable microprocessor design and fabrication. As these trends continue, guaranteeing correctness of execution is becoming prohibitively expensive and impractical. In this thesis, we demonstrate the benefits of abstracting circuit-level challenges to the architecture and software layers. Reliability challenges are broadly classified into process, voltage, and thermal variations. As proof of concept, we target voltage variation, which is least understood, demonstrating its growing detrimental effects on future processors: Shrinking feature size and diminishing supply voltage are making circuits more sensitive to supply voltage fluctuations within the microprocessor. If left unattended, these voltage fluctuations can lead to timing violations or even transistor lifetime issues. This problem, more commonly known as the dI/dt problem, is forcing microprocessor designers to increasingly sacrifice processor performance, as well as power efficiency, in order to guarantee correctness and robustness of operation. Industry addresses this problem by un-optimizing the processor for the worst case voltage flux. Setting such extreme operating voltage margins for those large and

infrequent voltage swings is not a sustainable solution in the long term. Therefore, we depart from this traditional strategy and operate the processor under more typical case conditions. We demonstrate that a collaborative architecture between hardware and software enables aggressive operating voltage margins, and as a consequence improves processor performance and power efficiency. This co-designed architecture is built on the principles of *tolerance*, *avoidance* and *elimination*. Using a fail-safe hardware mechanism to tolerate voltage margin violations, we enable timing speculation, while a run-time hardware and software layer attempts to not only predict and avoid impending violations, but also reschedules instructions and co-schedules threads intelligently to eliminate voltage violations altogether. We believe tolerance, avoidance and elimination are generalizable constructs capable of acting as guidelines to address and successfully mitigate the other parameter-related reliability challenges as well.

Contents

List of Figures

List of Tables

Citations to Previously Published Work

Portions of this dissertation have appeared in the following conference proceedings:

Vijay Janapa Reddi, Simone Campanoni, Meeta S. Gupta, Michael D. Smith, Gu-Yeon Wei, David Brooks, "Eliminating Voltage Emergencies Using Software-Guided Code Transformations", Proceedings of the ACM Transactions on Architecture and Code Optimization (TACO).

Vijay Janapa Reddi, Meeta S. Gupta, Glenn Holloway, Michael D. Smith, Gu-Yeon Wei, David Brooks, "Mitigating Voltage Noise Using Emergency Signatures", Proceedings of the ACM Transactions on Architecture and Code Optimization (TACO).

Vijay Janapa Reddi, Meeta S. Gupta, Glenn Holloway, Michael D. Smith, Gu-Yeon Wei, David Brooks, "Predicting Voltage Droops Using Recurring Program and Microarchitectural Event Activity", IEEE Micro's Top Picks in Computer Architecture Conferences (TopPicks).

Vijay Janapa Reddi, Simone Campanoni, Meeta S. Gupta, Michael D. Smith, Gu-Yeon Wei, David Brooks, "Software-Assisted Hardware Reliability: Abstracting Circuit-level Challenges to the Software Stack", Proceedings of the Design Automation Conference (DAC).

Vijay Janapa Reddi, Meeta S. Gupta, Krishna K. Rangan, Simone Campanoni, Glenn Holloway, Michael D. Smith, Gu-Yeon Wei, David Brooks, "Voltage Noise: Why It's Bad, and What To Do About It" Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE).

Vijay Janapa Reddi, Meeta S. Gupta, Glenn Holloway, Michael D. Smith, Gu-Yeon Wei, David Brooks, "Voltage Emergency Prediction: Using Signatures To Reduce Operating Margins" Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA).

Meeta S. Gupta, Vijay Janapa Reddi, Glenn Holloway, Gu-Yeon Wei, David Brooks, "An Event-Guided Approach to Handling Inductive Noise in Processors" Proceedings of the Design Automation and Test in Europe (DATE).

Acknowledgments

This is the end. And to this end, many people have been a great source of strength to muster through the difficult times. They have been an inspiring source to aspire ever more ambitiously, and many a time they have been a tremendous source of calmness. To each and everyone of you, I am extremely grateful. A lifelong goal and dream have finally come true.

I thank my family for all the wonderful support they have provided me despite the thick and thin we have been through. The effort is priceless, and I share the success of this culmination with them. They deserve it as much as I do. I thank my father for the countless hours of systematic encouragement to pursue further along, my mother for the emotional support and all those delicious meals that have kept me healthy, and my brother for always rooting on my behalf.

My dearest friends Karthik Pinnamaneni, Sati Hillyer and Sunil Verma have always been a priceless influence since my undergraduate days at Santa Clara University. I cannot thank the trio enough for all their moral support, continuous encouragement and all those deep philosophical exchanges that have not only shaped my drive to succeed, but also defined and refined me as an individual. All of this has also influenced me as a researcher. These folks are beyond just friends. They are family to me.

I have enjoyed a very enriching graduate school experience because of my numerous wonderful colleagues. Going back to the days of University of Colorado at Boulder, my thoughts and patterns of thinking have emerged from wild discussions overall several late nights in the lab with Alex Shye, Tipp Moseley and Rahul Saxena. Amongst others are Heidi Pan and Jason Mars. They provided me with the utmost support

and friendship in times of great need. More recently, my experiences have grown further along over several meaningful discussions with folks at Harvard. This group includes some amazing and aspiring individuals like Wonyoung Kim, Meeta Gupta, Simone Campanoni and Svilen Kanev. Benjamin C. Lee, a prior graduate from our lab, has been not only a good friend and mentor, but also a true source of inspiration. Knowing him has taught me to aspire for the best and to not give up prematurely, no matter how bad the odds seem. And I cannot say enough good things about Glenn Holloway, who acted not only as a mentor when I needed one, but he has also been by my side as a friend, a critique, an editor, and acted the role of many other such things when I needed one. Thank you Glenn. I am also grateful for the camaraderie I share with everyone in MD307.

None of my achievements would have been possible with great minds influencing, guiding and believing in my academic success. Without Prof. Silvia Figueira, I may have never embarked on this journey towards getting a PhD. I thank her for streamlining me during the course of my undergraduate studies. And without Dr. Connors at University of Colorado at Boulder, I would have never come across Dr. Robert Cohn and the Pin Team. All of them have been instrumental in my work and success. Dan and Robert are exceptional mentors. They have always supported my aspirations, even at the expense of their own time, allowing me to aspire and achieve my personal goals. It is thanks to them that I have had the opportunity to work with Profs. Michael D. Smith, Gu-yeon Wei and David Brooks. These members of my committee have seen to it that I consistently perform my best, opening new doors and constantly providing me with new insight on how I could do better each and

every single time. I am truly fortunate to be in the company of such great minds. I aspire to be like them some day, embodying the best traits of each and every single one of them.

While this PhD marks the ending of one major journey in my life, I see it as only the beginning of another, one of many more to come. I can only hope that I will continue to meet such amazing and influential people as I grow further along, both as an individual and as a researcher in the scientific community.

To my mother, father and brother.

Chapter 1

Introduction

Contents

The landscape of processor design is changing. Performance is no longer the only criteria. Power constraints are driving designers to consider performance-per-watt efficiency, trading performance for lesser power consumption. However, techniques and technology advances necessary to deliver good performance-per-watt efficiency are leading to transient processor reliability errors. Guaranteeing correctness into the presence of these errors is inadvertently degrading performance, consequently leading to lower efficiency. As the industry employs power-constrained processor design techniques more aggressively in the future, especially due to increasing core count per chip and the *power wall*, ramifications of these reliability errors will only continue to expand. Therefore, it is imperative we understand their implications and learn to address them effectively.

Traditionally, designers have been tackling reliability errors at the circuit-level, masking the issues from the processor microarchitecture above and the software running on top of it. However, these traditional solutions are not scaling well with reducing feature size and more aggressive power-constrained designs. Future systems will require adaptive processor design techniques. The underlying microarchitecture must dynamically detect and recover from reliability errors in the field.

In order to build such a resilient processor architecture, we propose abstracting circuit-level reliability challenges to the higher levels, the microarchitecture and software layers. The lower layers propagate relevant information to the higher layers, as illustrated in Figure 1.1. Using this information, the higher layers influence or mitigate problems at the lower layers. We envision this happening at run-time. Such dynamic feedback and reaction would enable an architecture that is efficient even in

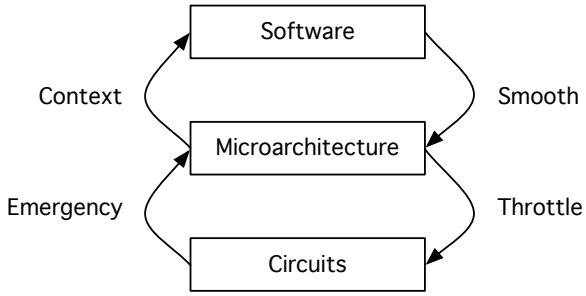


Figure 1.1: An abstract overview of exposing circuit-level reliability challenges to the higher levels of execution.

the presence of errors. In the short term, we propose an architecture solution. In the longer term, we capture the emerging role of software at the chipset-level, showing its potential for enabling long term resiliency in processors.

1.1 Challenges Facing Reliable Processor Design

Technology scaling has greatly improved transistor density over the past three decades. However, continued scaling has begun to introduce parameter variations, affecting both manufacturing yield and run-time efficiency. Parameter variations are broadly classified into three sub-categories: process, voltage and thermal variations.

Process variation refers to differences in transistor characteristics from one die to another, within-die or even wafer-to-wafer, resulting in differences between chips. It occurs during fabrication time because of imperfections in lithography techniques and unevenness in dopant injection. Post-fabrication, this variation does not change or affect chip behavior. It is static.

Voltage and thermal variations are more dynamic, affecting chip operation at run-time. These variations occur from execution-time interactions between the chip and

the workloads it is running. Temperature variation arises from aggressive utilization of certain circuit blocks, creating hotspots within the microprocessor. Voltage variation results from non-ideal power distribution. Both these variations are affected by the activity of a running workload.

In all cases, variations affect the worst-case delay within a circuit. Dynamic variations also reduce the lifetime of a processor through repeated stresses on individual components. These *emergencies* must be avoided at all costs to ensure robust processor operation.

Traditionally, designers have coped with parameter variations through careful design and testing, allocating sufficiently large margins or tolerance guardbands. This compromises the peak operational capacity of a circuit to ensure reliable and expected execution. For a processor relying on a single reference source signal (i.e., clock signal), the clock rate of the processor is set forth by the operational speed of the slowest logic path. As processor features have shrunk, parameter variations have amplified the difference between peak and worst-case operational delay in these slowest logic paths. Therefore, the effective clock rate is slowing down. Consequently, the maximum performance-per-watt efficiency we can extract from our processors suffers.

As we go into the future, designers must increasingly compromise peak efficiency for growing worst-case delays. Otherwise, designers must compromise on chip yield, throwing away processors that do not meet stringent delay specification, or they must invest in expensive manufacturing or packaging solutions to mitigate the effects of parameter variations. Considering that every part of a commodity processor affects chip price, these alternative solutions are impractical in the long term, leaving margins

as the most cost-effective solution.

Industry is yet to actively integrate and account for parameter variations during the chip development cycle. Today, industry designs, tests and optimizes only for power and performance targets during the chip development stage. Parameter variations are seen as a post-fabrication time issue. But margins necessary to guarantee correctness are progressively growing, and post-fabrication time solutions will become prohibitively impractical and ineffective in the future. Therefore, innovation is necessary in the presence of parameter variations. We must make parameter variations an active component of the processor development cycle.

1.2 Abstracting Circuit-level Challenges to Architecture

The problem with addressing emergencies at the circuit-level, like using worst-case margins, is that circuit techniques are inflexible. The solutions that designers put in place are not adaptable once the chips leave the fabrication plant. Therefore, designers make cautious and pessimistic assumptions about the conditions under which a chip may operate to ensure high reliability. But such conservative decisions lead to worst-case design that is not representative of typical case execution. Such worst-case design severely penalizes chip performance and power efficiency for very infrequent cases, rather than optimizing the chip for the average case behavior.

Solutions at the processor architecture level are better than circuit-level techniques because they are dynamic. Architectural level techniques are feedback-driven, observ-

ing run-time activity to determine the appropriate course of action to take. Such a higher level solution enables the processor to dynamically adapt to execution-time emergency activity, rather than being pessimistically penalized through conservative assumptions at the fabrication facility. Therefore, we would be able to operate the processor under more typical case conditions, and a consequence reap more efficiency from our chips.

In order to protect the chip from emergencies during operation in the field, designers can build recovery and rollback logic into the processor. The processor runs ahead assuming execution is always correct, but then rolls back execution upon detecting an error. This process is similar to branch speculation, where the processor predicts the branch outcome and continues executing speculatively, rolling back execution only if the prediction was incorrect.

1.3 Extending Processor Efficiency Using Software

Although architecture level solutions are dynamic and enable design for typical case operation, they lack the global perspective of software. Software can re-configure code running on a chip to eliminate emergencies. It can do this because software has global knowledge, such as what code is running on the processor, or which set of threads are co-scheduled together in a multicore chip. Therefore, in addition to innovating solutions at the architecture layer, we mitigate emergencies at the software layer as well.

Eliminating emergencies improves the overall efficiency of the processor. The architecture will need to recovery and rollback fewer times due to fewer emergencies.

Therefore, by intermittently using the architecture layer, and relying on the global view of software to eliminate recurring emergencies, we enable smoother run-time performance.

Another reason for relying on software is that the efficiency improvement we observe through architecture-level solutions is a function of emergency frequency. As technology scaling trends continue, processors will increasingly suffer from more number of emergencies (see Chapter 2). As a consequence, more aggressive utilization of the architecture's dynamic fail-safe mechanism will become necessary.

Since the fail-safe mechanism is a run-time feature, performance at execution time will suffer if the number of emergencies is significant, which will lead to larger number of rollbacks. Therefore, architecture-level solutions, although effective in the near term, are not an ideal solution. Software can sustain the existence of such hardware solutions by targeting and reducing recurring emergencies, keeping the overhead of dynamic rollback and recovery tolerable even in future, more emergency-prone, technology nodes.

Software is already playing a critical role in ensuring robustness. Large-scale companies like Google write error tolerant application code [15]. Google engineers assume that hardware failures are inevitable and write their code accordingly. System failures do not affect their application's correctness nor quality of service. Google search engine automatically detects failures via timeouts, and reissues requests to other available nodes, thus proving resilient to hardware failures.

However, compromising transparency between hardware and the application, as Google does, is not a generalizable solution for the mass market. Independent software

vendors (ISVs) will require that hardware is always robust. This is especially true for backward compatibility and legacy code reasons.

In the future, we envision that microprocessor companies will ship their processors with formally verified software that operates below the operating system. It will act as a transparent layer that guarantees resiliency in the presence of emergencies. Such software-assisted chipset-level reliability is simply an extension of present day application-level reliability like in the Google case.

1.4 Contributions

The embodiment of this thesis work is to demonstrate how to build a cost-effective and fault-tolerant platform in the presence of parameter variations. We make the following contributions:

1. Tolerance, avoidance and elimination. We believe these three principles should serve as guiding principles to solve process, voltage and thermal variation-related problems.

2. Resilient architecture design. We demonstrate that collaboration between hardware and software can effectively enable dynamic detection and recovery from emergencies in the field, even as the processor is running.

3. Proof-of-concept. We show that tolerance, avoidance and elimination are useful constructs for mitigating voltage variation, proving this via an instantiation of our resilient architecture design.

Overview. Since our solution relies on hardware and software co-design, there are two distinct methods for dealing with emergencies. First, the hardware-layer is responsible for guaranteeing reliable operation without the assistance of software using its fail-safe mechanism. The software layer seeks to eliminate emergency events from recurring in the future through code transformations, or even operating system-level thread scheduling. An advantage of our multi-layered approach is that it allows the hardware to focus on guaranteeing correct operation for the initial emergency, while the software focuses on eliminating or reducing the performance impact of all future emergencies in the steady state.

In order to facilitate software-assisted hardware reliability, a fault-tolerant system must gather and pass along pertinent information across all the different layers (circuits, microarchitecture and software). Figure 1.1 illustrates the layers and a high-level synopsis of the information flowing across the layers. Those interactions allow the processor to dynamically detect and resolve emergencies.

The bottom layer includes low-level circuit blocks that provide critical emergency information, like voltage and temperature readings from sensors, to the microarchitectural layer. This middle layer collects the context (sensor readings, executing code location etc.), possibly throttling (taking preventive measures) to avoid any immediate catastrophic failure, before propagating the context to the software layer. The software layer then smoothes the activity (or code) running on the processor to avoid recurring emergency events. Smoothing voids the microarchitecture from repeatedly activating its fail-safe mechanism whenever a similar context of activity recurs, thus improving performance.

Since the process we describe is feedback-driven, the hardware may profile the emergency initially before deciding to invoke the software layer. Profiling allows the hardware to first identify whether the emergency is recurring, since software can only fix recurring events. Moreover, profiling amortizes the overhead of invoking the software layer. An added benefit of relying on software is that the logic guaranteeing robustness at the hardware layer does not have to be very fine-grained or implemented costly.

Specific Contributions. To narrow the scope of work, and demonstrate proof of concept, in this thesis we focus only upon voltage variation. Efforts at reducing processor power have the unfortunate side-effect of causing large current variations within the processor. Due to parasitic inductance in the power-supply network, current oscillations may cause an undesirable swing in the microprocessors supply voltage [34]. This can result in supply voltages that violate the minimum or maximum voltage margins for the processor. Such transient reductions in supply voltage increase circuit delay [45]. Therefore, voltage variation can cause timing problems in a microprocessor that lead to incorrect execution. We refer to such violations as voltage emergencies.

Current designs are able to dangerous voltage emergencies through careful placement of decoupling capacitors and advanced packaging. However, such traditional means are being severely stretched due to recent technology trends. These trends will make voltage variation management a considerable challenge in the coming years.

In the context of voltage variation, we answer several research questions that demonstrate and validate our contributions. For each question, we provide a brief synopsis of our findings, encouraging the reader to read further details in subsequent

chapters. Overall, we show that it is possible to build a software-assisted hardware reliability platform:

- *What information should the circuit layer provide to the microarchitecture layer?* This involves sensing voltage at different points on the chip, and during each cycle monitoring for voltage emergencies. Upon detecting an emergency, these voltage sensors must signal the processor microarchitecture of an emergency.

- *How should we design the microarchitecture so that it can tolerate emergencies?* We can deal with voltage emergencies by leveraging existing hardware checkpoint-recovery logic in use for handling soft-errors. Whenever we detect an emergency, we simply rollback execution to some previously know safe state. Such recovery logic is already shipping in production systems, and is general purpose, serving additional tasks such as debugging, replaying execution.

- *What information should the microarchitecture propagate to the software layer?* We discovered that it is the interleaving of program and microarchitectural events that leads to voltage emergencies. Capturing and propagating this information allows the software to effectively smooth-away emergencies.

- *What should the software layer look like?* We demonstrate that a runtime compiler such as those used for application or process virtualization suffice. We also demonstrate that it is possible to handle emergencies at the operating system layer.

- *What smoothing techniques should the software utilize to eliminate emergencies?* From a compiler perspective, we can eliminate recurring emergencies by intel-

ligently constraining the instruction level parallelism of the processor via code transformation techniques. At an operating system level, scheduling threads cooperatively leads to fewer emergencies.

The thesis is structured in a bottom-up fashion. First, we provide the reader with background and motivation on voltage variation in Chapter 2. We discuss prior work. Additionally, we show that voltage emergencies will have a big impact in future systems as voltage supply scales (even if moderately) with increasing current draw, which is bound to happen as the chips grow denser with logic because of shrinking feature size.

Our scheme dictates that the hardware must be robust. So in Chapter 3 we elaborate the general-purpose checkpoint-recovery we assume is available in production systems. Since the recovery mechanism tolerates emergencies, we capture activity leading to voltage emergencies. This allows us to analyze the activity, explaining exactly what causes emergencies. Understanding this insight enables us to efficiently eliminate emergencies.

We state that software will frequently interject and eliminate emergencies. But software can be buggy. Moreover, software cannot eliminate all emergencies. Therefore, in Chapter 4 we introduce a low-cost emergency avoidance mechanism that works in conjunction with the general purpose recovery logic. The general purpose recovery logic is very expensive, ranging into hundreds to thousands of cycles per rollback. By comparison, the smaller avoidance mechanism penalizes performance by fewer than 10 cycles per activation. While this additional low-cost mechanism consumes chip area and additional power, the avoidance mechanism assures a robust hardware platform

that is more efficient than using only general purpose recovery logic.

We introduce our compiler and operating system thread scheduling software solutions in Chapter 5. We initially describe the information gathering resources necessary at the microarchitectural level. This middle architecture layer is the critical link in information exchange, as it monitors what is going on at the circuit-level and passes along information to the software for voltage smoothing. Thereafter, we discuss the specifics of our compiler and thread scheduling algorithms.

In summary, this thesis is a demonstration of software-assisted hardware reliability. We discuss details and findings that enable efficient hardware and software co-design. While the details we provide are specific to mitigating voltage emergencies, the concepts and structure we introduce and discuss in the following chapters are generalizable to other avenues of reliability research as well.

1.5 Impact

Efficiency at any cost is no longer an option. This is especially true in today's commodity market place. We focus primarily on building robust processors that deliver good performance and power efficiency within reasonable costs. As variation trends stretch the limits of existing strategies for mass markets, we believe our contribution can soften their impact in future technology nodes. To appreciate the impact of our contribution, it is important to understand evolving trends in computing and application domains.

The commoditization of hardware is forcing architects for applications involving high-volume web services (e.g., search engines), biological and physical analyses and

simulations (e.g., gene sequencing), and massively multi-player role-playing games and simulations are continuously striving for higher availability and better performance, but with decreasing costs as the need for compute power density increases. These applications contain enormous amounts of request/thread-level and application-level parallelism that encourages application architects to build computing clusters with large numbers of parallel processors. This is especially true as data migrates from clients to a cloud of distributed resources located in datacenters.

Clearly, for a fixed level of performance per processor, the application architects would prefer processors that cost less to purchase (e.g., cheaper packaging) and less to run (e.g., consume less power and produce less heat), because they can then purchase more computing power to solve their problems faster or make their infrastructures more available. Consequently, leading application domains and ISVs are focusing on energy efficiency and the ratio of price-to-performance much more than sole performance. This trend has been led by large companies that are increasingly relying on cheaper commodity desktop and workstation processors, rather than purchasing more costly high-end server processors because of the commoditization of hardware.

The Google Cluster Architecture is a perfect example, running their popular web search engine in a manner demonstrating that "price/performance beats peak performance" [14]. Barroso *et al.* describe an architecture where, for example, power reductions are extremely desirable if they can be obtained without a corresponding loss in performance or increase in price of the hardware. Furthermore, predictability in the speed of computation in each processor is important for the success of their load balancing algorithms.

Though massively parallel applications are where there exist obvious big savings, the work we propose in this thesis benefits desktop, as well as workstation computing domains. This stems from understanding that the massively parallel applications we mention above are increasingly run on architectures built out of commodity microprocessors. By focusing on this large segment of the microprocessor market, individuals purchasing single-processor systems will also experience savings, albeit on a smaller scale. Summing these individual savings on a national scale could however yield large savings.

Chapter 2

Voltage Noise: Why It's Bad and What to Do About It

Contents

Processors are designed to operate at fixed voltage levels called the nominal voltage. While it is possible to dynamically change this voltage setting to improve processor power and performance efficiency in a controlled manner, unexpected deviations from expected settings, called *voltage variation*, or *voltage noise*, or sometimes even referred to as the *dI/dt* problem, can lead to incorrect execution and affect processor lifetime. In this chapter, we introduce and explain voltage noise and its ramifications, specifically in the long-term. Briefly, voltage noise degrades peak processor efficiency. And while researchers have been proposing solutions to mitigate it, like the broad class of sensor-based schemes we will discuss, prior work cannot guarantee execution correctness at the very aggressive settings necessary to recover from the penalties associated with voltage noise.

Therefore, we introduce new hardware and software co-design principles to the voltage noise problem: *tolerance*, *avoidance* and *elimination*. By tolerating voltage noise we learn activity leading to it, which can then be used to build not only intelligent hardware that learns to anticipate and avoid dangerous voltage flux intelligently, but in fact design software that can eliminate the problem altogether. It is through these three constructs that we believe designers of future processors can tackle the voltage noise reliability problem.

2.1 Voltage Noise

Ideally, voltage within a processor is always steady at its fixed and expected nominal value. This nominal voltage determines how quickly designers can operate the processor's circuits because voltage has a direct relationship with circuit delay.

Consequently, voltage determines the clock frequency of a processor, which in turn translates to processor performance and power efficiency.

However, voltage within a processor is always fluctuating from its nominal value. Consider Figure 2.1 that illustrates voltage activity within the processor over several cycles. Current swings over a small amount of time cause voltage to swing due to parasitics in the power-delivery subsystem. Quantitatively, the magnitude of the voltage swing is proportional to the rate of change of current over time times the magnitude of inductance. More abstractly, voltage noise depends on the interactions between the program, the microarchitecture of the processor, and the characteristics of the underlying power-delivery subsystem.

Voltage noise is becoming a major hurdle. As logic density is increasing with shrinking feature sizes, denser chips are consuming more current. To reduce or even cap the maximum power consumption as current increases, designers are using aggressive power saving techniques like clock gating. This gating technique turns off parts of the processor to reduce dynamic power consumption depending on processor utilization. Unfortunately, it causes unacceptable stress on the power delivery network. Suddenly turning off and turning on logic causes sudden changes in current draw. These large changes over short time-scales cause voltage to droop or overshoot rapidly because of impedance present throughout the power supply network.

Since voltage determines circuit delay, unexpected and intermittent voltage drops below nominal voltage can cause logic paths within the processor to slow down, leading to circuit timing violations that eventually manifest as incorrect execution. Voltage can also exceed the nominal setting value. Such voltage spikes or overshoots

Figure 2.1: Voltage within a processor fluctuates due to activity changes and interactions between a running program and the processor's underlying power-delivery subsystem, as well its run-time microarchitectural behavior.

can degrade the lifetime of a processor by affecting the reliability of the transistors within. To ensure reliable and correct operation of the processor at all times, designers allocate sufficiently large guardbands or operating voltage margins to tolerate voltage noise. Dangerous voltage swings beyond these operating margins, called *voltage emergencies*, must be avoided under all circumstances.

2.2 Why It's Bad

Processor designers strive to ensure robust processor behavior in the presence of voltage noise through cautious and conservative design and testing strategies at the circuit-level. While these approaches have been practical and suitable in the past, they are fast becoming outdated and inefficient as technology is scaling; the International Technology Roadmap for Semiconductors (ITRS) identifies voltage noise as one of the Grand Challenges to overcome in order to sustain high performance-per-watt

efficiency in future processors.

Industry standard practice today is to make pessimistic assumptions about voltage swings. Industry designs for the worst-case voltage swing to avoid voltage emergencies. In the following subsections, we explain the issues with the pessimistic present technique, explaining why it will not work in the future from a performance, area and cost perspective. We then conclude this section by demonstrating the limits of alternative solutions that researchers have been proposing. The points we make in this section build towards our case that existing, as well as previously proposed solutions, need improvements to sustain processor efficiency in the long-term.

2.2.1 Worst-case Design Penalties

The traditional way of dealing with voltage noise is to over-design the system to accommodate the worst-case voltage swing. In this way, designers prevent voltage emergencies. To determine the amount of over-design necessary, chip designers write a power virus in software that causes extremely rare and large voltage swings, as shown in Figure 2.2. According to recent research efforts, designers of the POWER6 processor [33] show the need for operating margins greater than 20% of the nominal voltage. These conservative processor designs with large margins ensure robustness.

However, conservative designs lower the peak operating frequency of the processor. Figure 2.3 plots peak frequency at different voltage margins across four PTM [55] technology nodes (45nm, 32nm, 22nm, and 16nm) based on detailed circuit-level simulations of an 11-stage ring oscillator consisting of fanout-of-4 inverters. Assuming a fixed power budget, we study the implications of margins. This is a valid assumption,

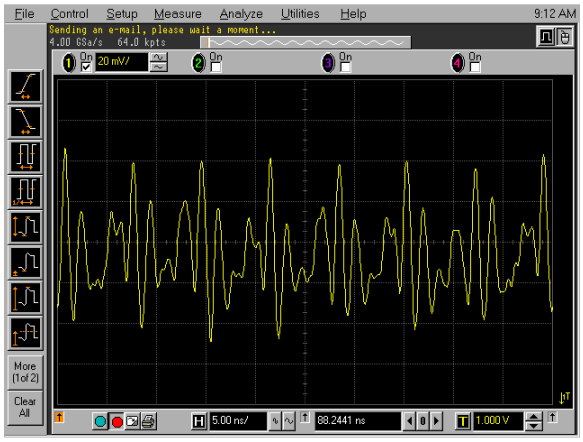


Figure 2.2: Designers use a power virus to determine the worst-case voltage swing.

since the industry has already reached maximum power consumption.

From the figure, we observe that the peak frequency for a given voltage margin is decreasing in newer technology nodes. The plot shows that at today's 32nm node, a 20% voltage margin translates to a 33% frequency degradation, and at future technology nodes the situation gets much worse. To understand this decreasing trend, we must understand power consumption. Power is equal to voltage times current. As we go into future technology nodes, the amount of current draw increases, since we have more logic packed into the chip. But because of the fixed power budget, we require scaling of the nominal supply voltage, which ITRS anticipates to happen gradually. Since threshold voltage scaling has all but stopped and nominal voltage is decreasing, circuit delay is increasing. And as a result, the peak operational frequency of the processor is reducing. Practical limitations on reducing power delivery impedance combined with large anticipated current draws therefore make margin-based solutions unsustainable.

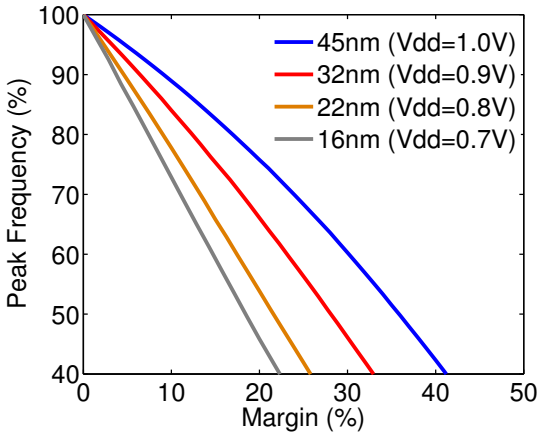


Figure 2.3: Worst-case margins limit peak operational frequency, and the problem is getting worse as technology trends are scaling.

2.2.2 Area and Cost Implications

To reduce voltage swings and to keep voltage margins within some reasonable bounds, processor designers rely on package and on-chip decoupling capacitance [49]. Figure 2.4 captures the belly-side view of a Intel Core^{TM}2 Duo processor package. The package has several capacitors, covering a spectrum of low and mid frequencies, that dampen the maximum voltage swing. On-chip decoupling capacitance targets high frequency noise. Traditionally, designers have been using oxide capacitors. But industry is making advances in integrating deep-trench decoupling capacitors into logic circuits, which would provide more capacitance per unit area than oxide capacitors.

However, the use of on-chip capacitors requires careful placement, estimation and allocation at design time. At present, the only quantifiable methodology that strongly establishes the precise amount of capacitance needed and its placement is to estimate for the worst-case voltage swing. Moreover, capacitors occupy precious area and increase the cost of a chip, both of which affect the cost of a chip. Commodity processors

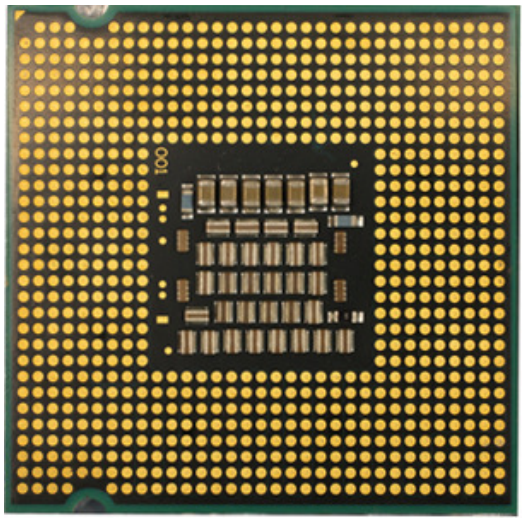


Figure 2.4: Processor designers rely on on-chip and package capacitance to keep the maximum amount of voltage swing within some reasonable bounds. This increases the cost of a chip, as well as requires valuable space. In the future, with increasing swing levels, both these resource requirements will have to increase.

must deliver good processor performance-per-watt efficiency within reasonable costs to stay competitive in the market.

Current designs are able to manage voltage swings through careful placement of decoupling capacitors and advanced packaging. But traditional means of reducing these swings are being severely stretched due to recent technology trends. While decoupling capacitors compensate for impedance due to the parasitic inductance of the power supply network, they do not suffice to compensate for the inductance of the wires between the die and the package.

2.2.3 Limitations of Prior Work

Traditional solutions are occurring at the expense of growing and intolerable operational efficiency. So researchers have been seeking alternative solutions. In this

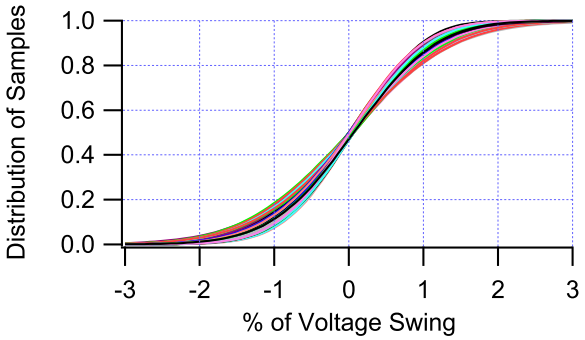


Figure 2.5: Cumulative distribution of voltage samples on a real production chip for several hundreds of benchmarks. All samples appear to fall within a 3% range, indicating that an aggressive voltage margin such as 4% would suffice under typical case operation conditions. However, the worst-case voltage swing is as large as 12%, indicating a fail-safe mechanism is necessary.

section, we explain this prior work, justifying why the general idea makes sense by corroborating it with our chip measurement data. However, we conclude why prior effort will not work in the long run.

Smaller voltage margins enable higher performance for the same nominal voltage. Therefore, recent efforts have been focusing on understanding the peak-to-peak voltage swings. Designers are considering to operate the processor under more typical case conditions, rather than allocating voltage margins sufficient for the worst-case voltage swing. We did analysis to determine the average voltage swing within a processor during the course of several benchmarking runs, finding that the worst-case voltage swing is overly conservative. Figure 2.5 shows that on average the peak-to-peak voltage swing is 3% of the processor's nominal supply voltage for a variety of workloads ranging from single-threaded SPEC CPU2006 [5] to the multi-threaded Parsec benchmark suite [16]. These are measured chip results, and we describe the measurement setup in Appendix A.

The Intel Core™2 Duo processor we use in our experiments can tolerate a worst-case voltage droop of 14%. During execution we find voltage swings as large as 12%. However, this happens very infrequently. From this we conclude that while the processor is robust against the worst-case voltage swing, it is severely over-designed for the more typical 3% voltage swing we see across most benchmarks.

It is a better design choice to tighten the worst-case voltage margin to 4%, while providing a fail-safe guarantee mechanism for those very infrequent large voltage swings. A 4% voltage margin translates to 15% improvement in clock frequency, assuming a 1.5x voltage to frequency scaling factor [17].

Therefore, instead of using worst-case design, researchers have been proposing sensor-based techniques that react to and mitigate on-chip voltage emergencies, as a means of designing the processor for typical case operation. A typical sensor-based proposal uses a tight feedback loop like that shown in Figure 2.6a. The loop includes a sensor that tries to detect impending emergencies and a throttling actuator that tries to avoid them. The sensor relies on a soft current or voltage threshold as a "canary". Crossing that threshold means that voltage is approaching its lower margin, so the actuator turns on throttling until the crisis is past. Proposed throttling schemes range from frequency throttling, to pipeline freezing/firing, to issue ramping, and altering the number of accessible memory ports [26, 34, 44, 43]. The behavior of the feedback loop is determined by two parameters, the setting of the soft threshold level and the delays around the feedback loop. Unfortunately, choosing those parameters to accommodate reduced operating margins is thwarted by correctness failures and/or performance penalties.

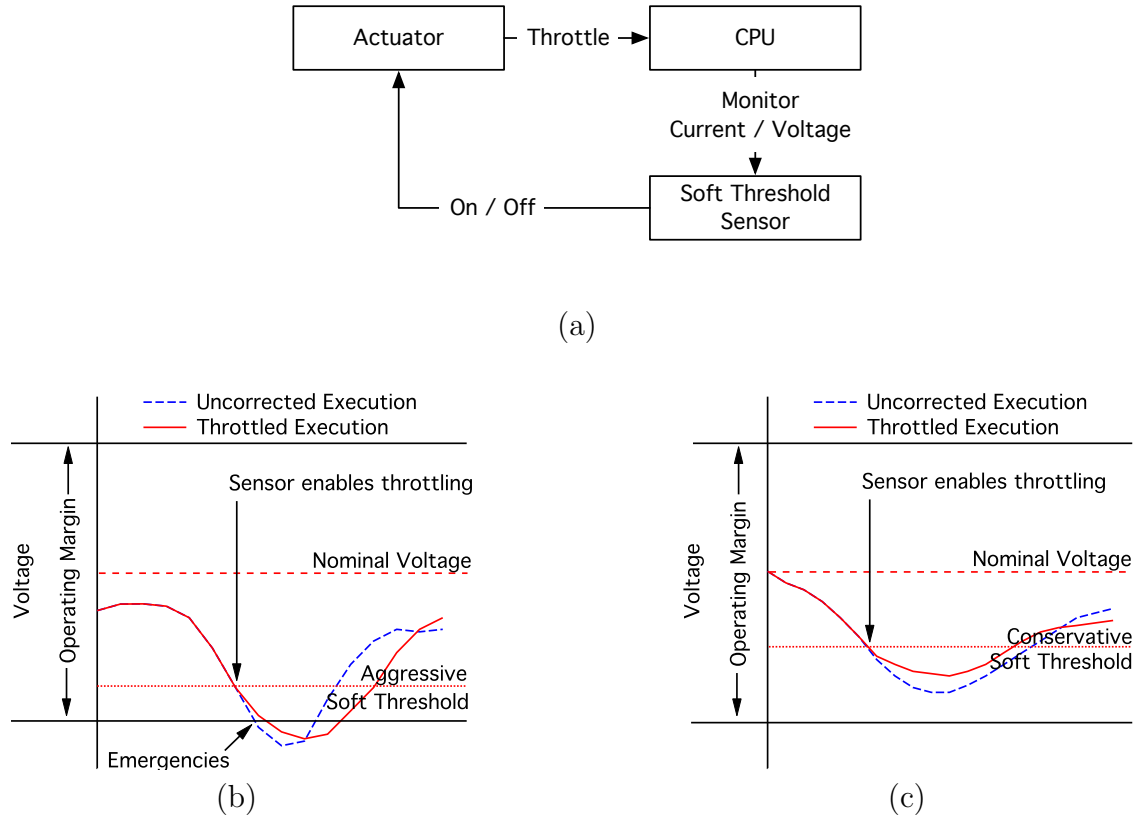


Figure 2.6: Sensor-based throttling. (a) A feedback loop is intended to detect and prevent emergencies. (b) Aggressive soft thresholds allow too little time to prevent emergencies. (c) Conservative soft thresholds trigger unnecessary throttling.

Correctness Failures. Figure 2.6a illustrates the use of a soft threshold to throttle execution and prevent an emergency. The graph shows voltage waveforms with and without sensor-based throttling (Throttled Execution and Uncorrected Execution, respectively). The solid horizontal line marked Aggressive Soft Threshold indicates the threshold at which a voltage sensor starts to take action to prevent an emergency. Setting the soft threshold aggressively (i.e., close to the lower operating margin) requires a very fast reaction by the sensor and actuation system. Failure to respond quickly enough results in a voltage emergency. In Figure 2.6b, the voltage starts to recover with throttling, but not in time to avoid crossing the lower operating margin.

Figure 2.7a shows the sensitivity of sensor-based mechanisms to feedback loop delays by plotting the number of emergencies that go unsuppressed in our benchmark suite as a function of sensor-loop delay times. This data is based on the experimental infrastructure we discuss in Appendix B.1. Here we assume the soft threshold to be 3% below the nominal voltage and the lower operating margin to be 4% below nominal. Feedback loop delays ranging between 0 and 5 cycles would require a nearly perfect sensor. Analog to digital conversion takes time and so does gathering data from all the sensor spread across the chip. Yet even a 2-cycle delay causes 50% of all soft threshold crossings to violate the simulated microprocessor's minimum operating margin specification. In other words, fail-safe execution is not possible at this margin using sensor-based schemes, as they cannot operate in a timely manner.

Performance Penalties. To accommodate slow sensor response times and ensure that throttling effectively prevents emergencies, sensor-based schemes can use conservative soft thresholds. Lifting the soft threshold away from the lower operating margin, as illustrated by the Conservative Soft Threshold in Figure 2.6c, gives the throttling system more time to prevent an emergency. But as the Uncorrected Execution waveform in Figure 2.6c shows, even in the absence of throttling, a soft threshold crossing may not be followed by an emergency. Throttling execution in such cases decreases performance without any compensating benefit. The more conservative the soft threshold setting, the greater the performance penalty. Figure 2.7b shows that this penalty can be quite large. Assuming an ideal sensor with no feedback loop delay (i.e., 0-cycle sensor delay), the percentage of benign soft threshold crossings is between 77% and 58% for soft thresholds ranging from 2% to 3%. So even if it

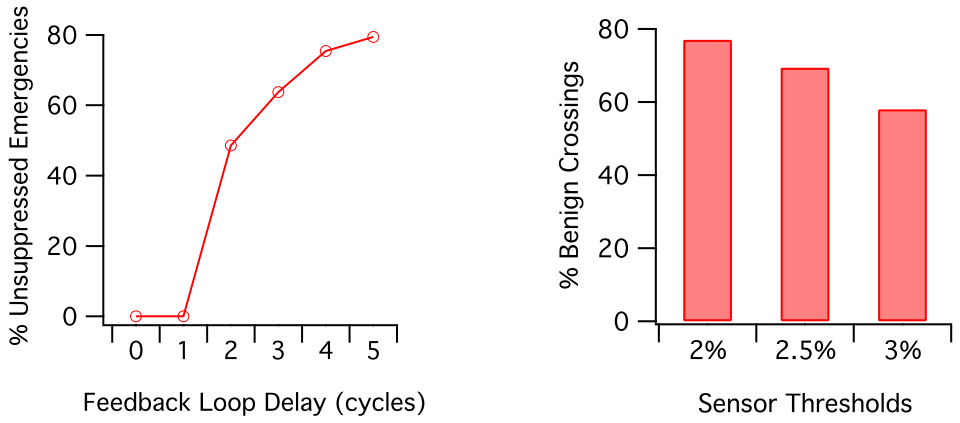


Figure 2.7: Implications of feedback loop delay and soft threshold settings on correctness and performance. (a) A large percentage of emergencies are not detected early enough to prevent them due to feedback loop delays. (b) Even assuming a 0-cycle feedback loop delay, the number of soft threshold crossings that are not followed by emergencies (i.e., benign crossings) is so large that performance suffers due to unnecessary throttling.

were possible to design a feedback loop with no delay, the large performance penalties would deter architects from reducing operating margins.

Resonant vs. Isolated Emergencies. A sensor-based scheme proposed by Powell and Vijaykumar [43] reduces sensitivity to feedback loop delay by focusing on voltage emergencies that are the result of resonating patterns. While resonance-induced emergencies are dominant for some packages, recent work by Gupta *et al.* [30] illustrates that non-resonant (pulse) events are also a major source of emergencies across a range of packages. James *et al.* [33] have observed isolated (non-resonant) pulses in a POWER6 chip implementation. And Kim *et al.* show that resonant emergencies are likely to become less important than isolated pulses in future chip multi-processors with on-chip voltage regulators, as package inductance effects are decoupled from the power grid via on-chip regulators [35]. Therefore, to realize the

benefits in improved energy efficiency or performance that reduced margins can enable, new solutions are needed that cope with both resonant and non-resonant voltage emergencies in future systems.

2.3 What To Do About It

In this section, we propose a new interdisciplinary solution for voltage noise, involving VLSI circuits, computer architecture, and software systems. In particular, we lean towards a hardware-guaranteed, software-assisted voltage noise management system.

As we have seen previously, conservative designs either lower the operating frequency. Practical limitations on reducing power delivery impedance combined with large current fluctuations make margin-based solutions unsustainable. And we have also seen that recent efforts proposing the tightening of noise margins by adding fail-safe mechanisms to the hardware cannot guarantee absolute correctness at very aggressive settings. In summary, the limitations of prior work are that they are not scalable solutions, specifically because they always attempt to proactively avoid emergencies altogether.

We take a radial route to the problem, advocating a system that allows emergencies to occur. We intend to tolerate emergencies infrequently, while eliminating frequently recurring emergencies using patterns in emergency behavior of a running code. We implement this vision in both hardware and software. Figure 2.8 illustrates an overview of the system. The system has an Emergency Detector (hardware) that triggers a Fail-safe Recovery Unit (hardware) to rollback execution whenever it de-

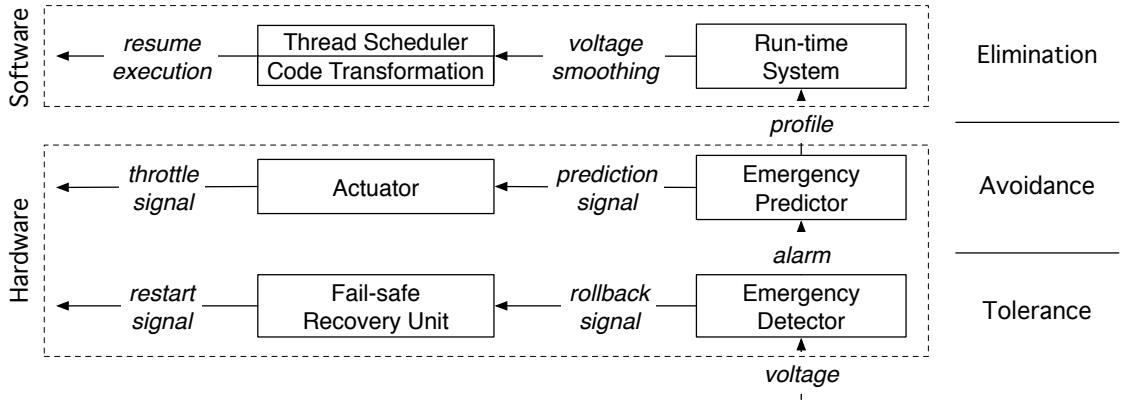


Figure 2.8: A co-design architecture to mitigate voltage noise that uses both hardware and software to tolerate, avoid and eliminate voltage emergencies.

tects an emergency. The detector then feeds an Emergency Predictor (hardware) with a signature that represents processor activity leading up to that emergency. The predictor quickly programs itself to suppress recurrences of the emergency by throttling processor activity. But if the profiler within the predictor identifies that the emergency is occurring very frequently, perhaps because it is in loop, then the hardware accumulates information to guide a dynamic Run-time System (software) that eliminates recurrences of that emergency. The run-time software layer eliminates the emergency either via Code Transformation (using compiler techniques) or by invoking the operating system's Thread Scheduler to co-schedule the suffering thread with an alternative program. The latter is useful in multi-core systems. However, when software is unsuccessful, the hardware emergency predictor takes over, re-arming itself with the signature pattern to instead predict and suppress the emergency.

The following subsections elaborate the three guiding principles that designers should use to build a noise-tolerant architecture. We explain the benefits of each of these three principles, and discuss our specific implementation schemes. We reserve

our discussions to a high level in this chapter. The rest of the thesis chapters provide in-depth and specific details on how to implement each of the schemes and evaluate them thoroughly.

2.3.1 Tolerance

Tolerating emergencies is useful both for tightening margins, and observing the emergency behavior of running code. By tolerating emergencies we can eliminate emergencies intelligently, as we empirically understand the activity leading to them. Our architecture relies on a hardware mechanism that allows voltage emergencies to occur, but when they do, the architecture has a built-in mechanism to recover processor state and resume execution.

We propose relying on a checkpoint-rollback mechanism to guarantee fail-safe execution. Checkpoint-rollback mechanisms have been proposed for handling soft errors [7, 10, 54]. They support execution rollback in the presence of an error. They are already available in existing production systems [9, 48], and more novel applications of this general-purpose hardware are continuously emerging [54, 51, 39, 36, 47, 41]. Our extension of its service to voltage noise is yet another addition.

2.3.2 Avoidance

Tolerating emergencies using coarse-grained checkpoint-recovery hardware is not always practical, since it is a prohibitively expensive rollback mechanism. Therefore, we built a voltage emergency predictor that identifies when emergencies are imminent and prevents their occurrence by predicting them. A voltage emergency predictor pre-

dicts voltage emergencies using voltage emergency signatures and throttles machine execution to prevent them. Throttling is the act of slowing down machine execution so that voltage recovers to its nominal level gracefully.

An emergency signature is an interleaved sequence of control-flow events and microarchitectural events leading up to an emergency. A voltage emergency signature is captured when an emergency first occurs (tolerated) by taking a snapshot of relevant event history and storing it in the predictor. Our emergency tolerating checkpoint-recovery mechanism then rolls the machine back to a known correct state and resumes execution. Subsequent occurrences of the same emergency signature cause the predictor to throttle execution and prevent the impending emergency. By doing so, the predictor enables aggressive timing margins in order to maximize performance, even in the presence of emergencies.

The cost of signature-based throttling is fewer than 10 cycles, which is much cheaper than the thousands of cycles that it costs to rely on the general-purpose checkpoint-recovery mechanism. This middle layer plays an important role in our hardware-software co-design solution to voltage noise, as it serves two purposes: First, it serves as a low-cost profiling hardware mechanism, determining when to invoke software. Therefore, it allows us to amortize the cost of invoking software. Second, it acts as a cheaper fail-safe alternative when software cannot eliminate the voltage emergency.

2.3.3 Elimination

While avoidance is cheaper than tolerating emergencies, software can eliminate emergencies altogether. Software has a much more global view of execution than the hardware does. For instance, it knows what threads are running on a chip, and it can also know the instructions that a program is executing. By relating emergencies to such high-level information, software can relieve the hardware of repeatedly taking action to ensure correctness, be it via either tolerance or avoidance.

Software can handle recurring emergency activity better than hardware. Consider a frequently executing loop that experiences recurring emergencies every iteration of the loop simply because the program is taking the same error-prone code path every iteration. Such a scenario can be handled by software, rather than hardware. Hardware would repeatedly tolerate, or throttle execution to ensure correctness to avoid or tolerate that emergency. But an intelligent software piece, like a compiler, is capable of performing fine-grained instruction-level tweaks to eliminate the emergency. A compiler typically has several options when choosing the order of instructions, and many of the options result in equally performing software. Therefore, in the case of this voltage emergency-prone loop, the compiler can rearrange instructions along the problematic code path to avoid recurring emergency activity without impacting performance.

In multi-core chips, an operating system thread scheduler can smooth out voltage noise from threads that are interfering with one another. Thread scheduling is an important topic of study in symmetric or chip multiprocessors. Prior work demonstrates that threads can hurt each other's performance by destructively interfering

with one another [50, 25, 38, 37, 56, 23, 22]. For instance, scheduling two cache intensive programs together is bad, since the cache resource becomes a bottleneck and both programs suffer. It is better to schedule one of those cache intensive programs with another program that is more CPU-bound (i.e., less intensive on the cache), resulting in less interference and better overall system performance.

We find that similar thread interference also exists in the context of voltage noise. The number of emergencies varies depending on which threads are co-scheduled together. Therefore, a noise-aware operating system thread scheduler can schedule threads intelligently to minimize the number of emergencies. By reducing emergencies, the overall throughput of a system increases due to fewer rollbacks; in our system we assume a global checkpoint-recovery mechanism across all cores sharing a power supply source.

However, we must be judicious in our use of software. Invoking software is expensive, costing several thousands of cycles. Thus, it is important to carefully determine when it is economical to pay the penalty of invoking software. Moreover, software might not always be able to eliminate every emergency. In both of these cases, our low-cost avoidance mechanism becomes invaluable. It allows us to cheaply profile and prevent emergencies, while relying initially on the general purpose checkpoint-recovery hardware to identify recurring signature patterns.

Voltage noise management will be a considerable challenge in the future. Increasing processor currents, decreasing supply voltages, and a significant increase in current variability due to power saving techniques all contribute to this issue. Our holistic hardware and software design enables aggressive voltage margins, mitigating

the inefficiencies of worst-case voltage margins and overcoming the limitations of prior work, based on tolerance, avoidance and elimination.

Chapter 3

Tolerating Voltage Noise to Learn Activity Leading to Emergencies

Contents

Rather than trying to avoid voltage swings beyond certain operating margins that guarantee correctness and reliability, in this body of work we rely on a hardware mechanism that allows such voltage emergencies to occur. When they do, the architecture has a built-in mechanism to recover processor state. In this way we (1) always guarantee execution correctness, and (2) have a means of identify leading indicators of dangerous voltage fluctuations, or that which we refer to as voltage emergencies.

We start off by first understanding voltage fluctuations within a production processor using our measurement setup (see Appendix A). In this chapter, we limit our discussion to studying voltage noise in a single core, since it simpler and provides a more comprehensible starting point. Subsequent chapters tackle multicore systems.

Processor stalls cause voltage to fluctuate. We study events that cause stalls like branch mispredictions and cache misses. By studying them in isolation using microbenchmarks, we are able to classify their effect on voltage. However, real programs experience an intermingling of stalls that are more convoluted and harder to digest. Nevertheless, we successfully construct a strong relationship between processor stalls and the amount of voltage swing a program experiences during its execution. We use performance counters to determine stalling activity. But this analysis is at a very coarse granularity, giving us insight only at the granularity of several tens of seconds. Finer timescales lead to measurement error (see Appendix A).

To gain further clarity into the interactions between those microarchitectural events that cause voltage noise and program code that influences such activity, we switch to our simulation framework (see Appendix B) to provide more depth. By observing activity at a cycle-by-cycle granularity, we conclude that it is the interleaving

of program control and microarchitectural events that are leading indicators of voltage emergencies. These voltage emergency signatures allow us to predict emergencies, which is useful for both avoidance (Chapter 4) and elimination (Chapter 5). But we must first capture them. We demonstrate our approach to tracking these signatures using specialized, yet simple logic called the event history register (EHR), showing that certain flavors of voltage emergency signatures yield better prediction accuracy than others. Accuracy is our quantitative metric of evaluating how representative a certain signature is of the emergency it represents.

3.1 Characterizing Voltage Droops and Overshoots

In this section, we demonstrate measurements of voltage noise in a production chip. We show oscilloscope snapshots in response to certain current stimuli that we create either explicitly by toggling power circuitry, or by inducing execution stalls within the processor. We then stitch these microbenchmarking effects to noise profiles of full programs, showing that there is a strong correlation between stalling behavior and voltage noise. Our effort is first in demonstrating that such a working relationship exists in a production processor. In the next section, we investigate this relationship deeper, examining the interactions between program control flow, microarchitectural events and voltage noise.

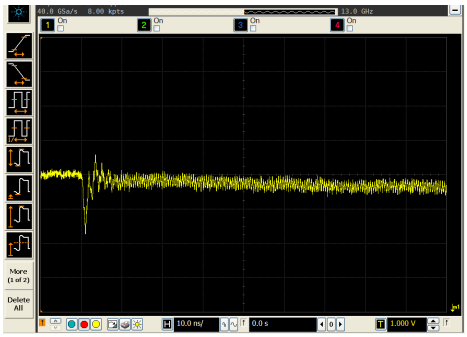
3.1.1 Changes in Current

To demonstrate that even processors in production experience large voltage swings, we reset the processor as it is idling, running the `idle` loop of the operating system.

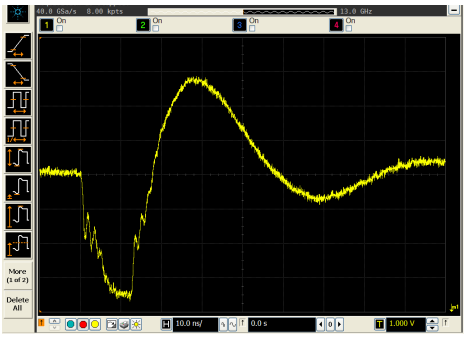
The reset signal turns off the processor and turns it back on immediately. During this toggle period, the processor invalidates all its internal caches without writing anything back to memory and reinitializes itself. Consequently, the processor draws a significant amount of current within a very short period of time. Impedance in the power delivery network temporarily causes a voltage "droop" (a transient expression for drop).

We observe the effect of a reset signal on the core's supply voltage in Figure 3.1a. There is a sharp 150mV voltage droop for about five nanoseconds towards the left side of the image. If core supply voltage droops for a more extended period of time, the processor may experience circuit delays that cause timing closure problems. In our case the processor does not experience a timing glitch because production processors are made robust against such reset signals using de-coupling capacitance (see Chapter 2.2.1). But the processor cannot boot up if we were to remove the de-coupling capacitance. Extremely large voltage droops over an extended period of time, such as the 350mV drop we see over 20 nanoseconds in Figure 3.1b, lead to timing violations that prevent the processor from even booting up. The large overshoot that follows the droop can damage transistors, shortening the lifetime of the processor.

While reset signals cause very large voltage swings, it is not the only reason voltage can droop or overshoot. We find similar behavior in the presence of microarchitectural events. Events can cause stalls during execution, forcing sharp changes in processor current draw. Such sudden changes may translate to large voltage swings, depending on a variety of factors which we explore next.



(a) Intel Core™2 Duo with all package capacitors.



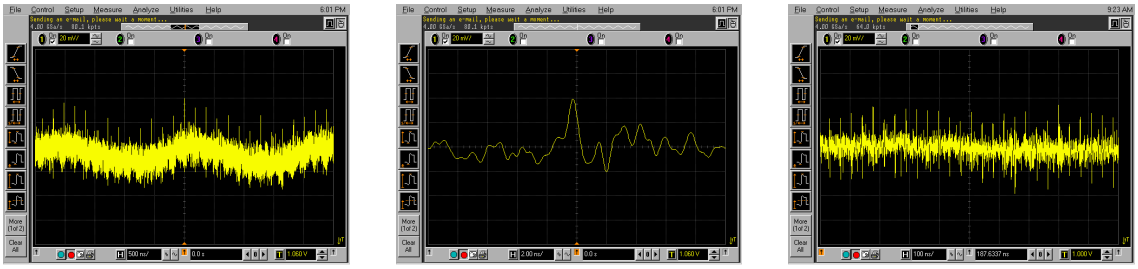
(b) Intel Core™2 Duo with no package capacitors.

Figure 3.1: (a) Voltage droop in a Intel Core™2 Duo processor when the processor is reset. A reset signal causes a large voltage droop, since current rapidly jumps from some nominal value to zero and back up. This leads to a voltage droop because of the impedance in the power delivery network. (b) A similar reset test with no package capacitors to protect against voltage drops causes a much larger voltage droop that prevents the processor from startup because of timing violations.

3.1.2 Effect of Stalls

Microarchitectural execution behavior causes current fluctuations within a processor that can lead to voltage swings. In our study, we considered several microarchitectural parameters, such as the reorder buffer, the instruction fetch queue, and the load/store queue, along with microarchitectural events like cache misses and pipeline flushes (caused by branch mispredictions). For clarity and to provide deep insight, in this section we constrain our discussion to the impact of microarchitectural events. In the following section, we broaden our discussion to encompass all events.

We hand-crafted microbenchmarks that cause cache misses, translation lookaside buffer (TLB) misses and branch mispredictions to study their effect on processor current and voltage. Table 3.1 shows how we construct these microbenchmarks. Please refer to the table for details. We construct our microbenchmarks such that



(a) Voltage snapshot showing recurring TLB misses, including the regulator's switching frequency.

(b) Closeup of one of the TLB overshoots from (a).

(c) Overshoot followed by a successive droop due to consecutive L1 cache misses.

Figure 3.2: Snapshots of voltage within the processor, as we execute microbenchmarks that stall processor activity every few cycles, leading to voltage swings.

the processor is experiencing nothing but the expected stalling effects over several seconds in time. Each microbenchmark kernel is put in a loop, so that the activity is recurring long enough to study it.

To prove that our microbenchmarks exhibit steady and repeatable behavior for measurements, Figure 3.2a presents a snapshot of voltage within the processor as it is experiencing recurring TLB misses. The sawtooth-like triangular waveform that is about two periods longs is the voltage regulator's switching frequency, or regulator noise. We are not interested in this, and consider it as background activity. But embedded within that same waveform are recurring high frequency voltage spikes or overshoots that correspond to our TLB Invalidation microbenchmark.

A TLB miss causes voltage within the processor to swing because a TLB miss stalls processor execution briefly, causing a momentous drop in current. As a consequence, voltage overshoots because of impedance. We see this overshoot in Figure 3.2b.

We observe similar effects with other microarchitectural events, but with the ex-

Microbenchmarks	Description

L1 Cache Miss	Generates L1 cache misses using back-to-back-loads and pointer-chasing. Pre-initializes an array larger than the cache to form a circular linked list of memory addresses, such that accessing one memory location yields the next memory location to load from. The offset between subsequent accesses determines the frequency of cache misses, and thus ideally matches the cache's line size. Such back-to-back loading of memory locations serializes execution and renders out-of-order execution and memory disambiguation ineffective, thus allowing clean noise measurement of individual cache miss events:
	`register char **ptr = &array[0];`
	`while (ptr != NULL)`
	`  ptr = (char **)*ptr;`

L2 Cache Miss	Uses the same back-to-back loading technique that the L1 cache miss microbenchmark uses to force second level cache misses. The array size and memory access offset are adjusted to the second level cache's size and its corresponding line size, respectively. Alternatively, certain x86 processors support the `CLFLUSH` instruction. However, this does not guarantee serialization.

TLB Invalidation	Re-loads the `cr3` register to purge the virtual to physical address mappings from the TLB:
	`asm ("movl %%cr3, %0;`
	`  movl %0, %%cr3;`
	`  : "=r" (tmpreg) ::  "memory");`
	An alternative approach is to use the `INVLD` instruction to purge the entry corresponding to the page containing the instruction. Re-execution of the instruction forces a stall, as a result of the TLB miss. Either way, this microbenchmark requires `Ring 0`, or root, privilege for execution.

Branch Misprediction	An if-then-else statement whose condition variable depends upon the output of a randomization function (e.g., `rand()` from standard `C` library) to thrawt the branch predictor from successfully knowing the outcome of a branch, thus forcing pipeline flushes. To avoid function call perturbations during measurement, we pre-initialize an array that is larger than the predictor's history register with values from `rand()`. As the static recurring pattern does not fit in the history register, performance counter data indicates 50.59% branch misprediction on the if-then edge and the remainder on the else path.
	`if (array[i] & 0x1)`
	`  ...`
	`else`
	`  ...`

Table 3.1: Descriptions for microbenchmarks that cause observable voltage swings. All microbenchmarks run in a loop, providing us sufficient time to make measurements.

ception that some other events can induce a correspondingly strong voltage droop following an initial overshoot. Consider the L1 Cache Miss microbenchmark from Table 3.1. This microbenchmark serializes processor execution by repeatedly accessing

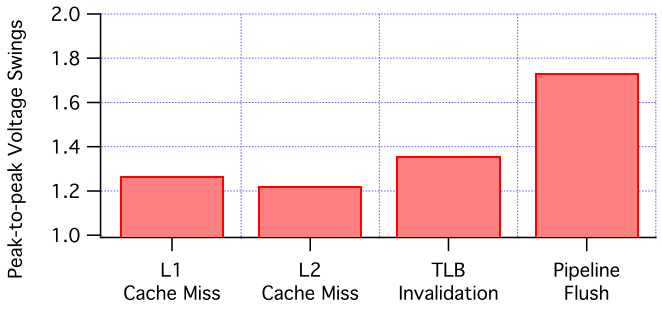


Figure 3.3: Peak-to-peak voltage swing analysis because of microarchitectural events that cause sudden stalls within the processor. These results are relative to an idling system.

cache lines that always miss in the L1 cache, but hit in the L2 cache. Pipeline activity ramps down during the time it takes to service the L1 miss. Current draw drops sharply, leading to an overshoot (see Figure 3.2c). However, after the L1 miss data is available, functional units become busy and there is a sudden increase in current activity. This steep increase in current causes the voltage to droop, which is also observable in Figure 3.2c.

From these microbenchmarks, we learn that the effects on voltage vary depending on the event. But in addition to the subtle differences between voltage overshoots and droops, the magnitude of the voltage swing also varies depending on the event. We summarize the magnitude of voltage swing relative to an idling system. In addition to the events discussed thus far, we studied the effect an L2 Cache Miss event and Pipeline Flush event have on voltage. Figure 3.3 shows that branch mispredictions cause the largest amount of voltage swing compared to other events (over 1.7x the effective nominal voltage band that includes the regulator noise).

Our evaluation using microbenchmarks allowed us to distinctly understand how specific events cause voltage to swing. But real programs experience a confluence of

events. Therefore, it is imperative to also gain insight into how the intermingling of microarchitectural event activity affects voltage swings.

3.1.3 Workload Differences

Every program during execution experiences a unique mix of microarchitectural event activity. Therefore, the amount of voltage swing per program will vary substantially based on the program's activity. In this section, we will validate this claim, in addition to showing a relationship between program stall behavior and the amount of voltage droop. While in the previous section we discuss processor stalls in the context of a few select microarchitectural events, here we take a more holistic approach. For example, we look at front-end stalls as a whole, rather than looking at individual microarchitectural events like L1 or L2 cache misses or pipeline flushes.

In order to understand how voltage droops vary across programs, we define the term "Droops per 1K cycles." A droop here refers to a voltage dip below idling conditions when only the `idle` loop of the operating system is running. No work is being done during this process. Therefore, it represents steady microarchitectural state for the purposes of our evaluation. We analyze droops in this aggregate droops per 1000 clock cycles form, since it is a form of metric commonly used to characterize the access behavior of workloads (like cache misses).

We quantify voltage droops across the CPU2006 benchmark suite in Figure 3.4. The data indicates that droops vary largely across programs. The trend is that programs fall into one of three categories: around 40, 80 or 110 droops per 1K cycles. But interestingly, we find that the number of droops has a relationship with the raw

Hardware Performance Counter	Description
RESOURCE_STALLS.BR_MISS_CLEAR	Stalls due to branch misprediction.
RESOURCE_STALLS.FPCW	Stalls due to floating point unit (FPU) control word write.
RESOURCE_STALLS.LD_ST	Pipeline exceeded load or store limits.
RESOURCE_STALLS.ROB_FULL	Reorder buffer (ROB) full stalls.
RESOURCE_STALLS.RS_FULL	Reservation station (RS) full stalls.

Table 3.2: Breakdown of counters that define stall ratio (see Figure 3.4).

performance of the machine.

Based on our understanding of how microarchitectural event stalling leads to voltage swings, one can conclude that fewer stalls will result in a smaller number of droops. In other words, if the machine utilization is good (high IPC), then the processor is not stalling, and therefore the number of droops will be smaller. We should therefore be able to construct a metric that captures machine performance, as a leading indicator of voltage noise.

We introduce a metric called Stall Ratio to assist us in understanding the relationship between processor resource utilization and voltage swings. It is a representation of processor stalling activity. Stall ratio is a comprehensive metric including several counters such as reorder buffer occupancy, reservation station usage, branch prediction rate etc. Table 3.2 summarizes all the counters, and the behavior they represent/capture. We gather this metric for each workload using hardware performance counters with the help of VTune [1].

We gathered the stall ratio corresponding to each program, and overlay it onto the "Droops per 1K cycles" plot in Figure 3.4. The lineplot corresponds to stall ratio, and its y-axis is on the right. From this overlay, we visually see a relationship between voltage swing and stall ratio. Quantitatively, the correlation between droops and stall

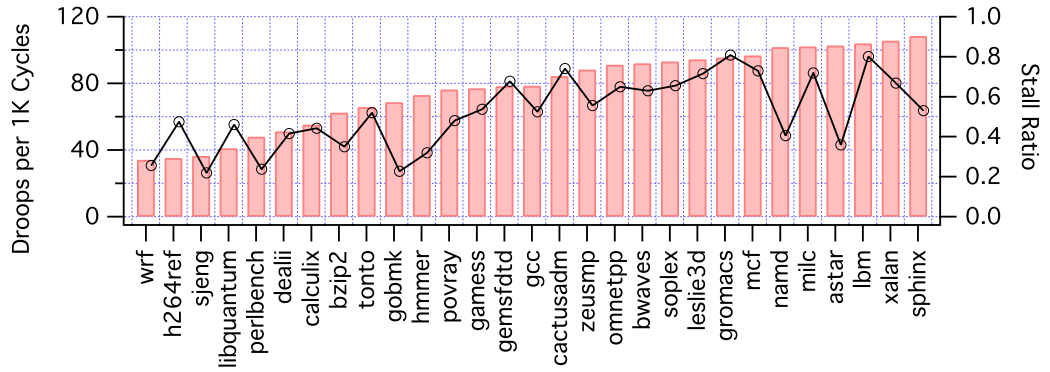


Figure 3.4: Voltage droop characteristics of different programs. The figure illustrates that the noise characteristics of programs vary based on their behavior, which we capture with various hardware counters (see Table 3.2).

ratio is ∼97%; we normalize the correlation result to 1.0 at the maximum value where both stalls and droops are most similar. This result confirms that machine utilization, or more generally stalling activity of a processor is indicative of the amount of droop.

Stall ratio is a good leading indicator of the amount of droop a program experiences. However, it falls short of helping us understand how program control flow influences voltage swings. We explore the relationship between microarchitectural events and program control flow next.

3.2 Exploiting Recurring Activity as Voltage Emergency Signatures

Programs are highly repetitive. Repeating code patterns give rise to repeating patterns of memory access and data flow through the processor. As we have shown using measurements, and Gupta *et al.* [28] have shown using simulation, repeating

sequences of processor microarchitectural event activity have the potential to cause dangerous voltage swings. What we are yet to understand is *when* microarchitectural events are benign versus harmful. In other words, there is no guarantee that a pipeline flush or any recurring event will always cause large intolerable voltage swings. From here on, we refer to such dangerous voltage fluctuations as voltage emergencies. In this section, we show it is possible to predict the likelihood of an emergency more accurately by taking into account the context leading up to the emergency.

We explore the working principles underlying voltage swings occurrence using a specific, but real-life, scenario from benchmark *403.gcc*. We switch over to our simulation framework for this part of the work (see Appendix B), since it allows us to monitor both microarchitectural stalls and program control flow precisely in order to better understand voltage swings.

A microarchitectural event acting in complete isolation only sometimes causes an emergency by itself. To help illustrate when an event causes an emergency, Figure 3.5 shows pipeline activity over 880 cycles for benchmark *403.gcc* while it is executing the nested loop illustrated in Figure 3.6. Figure 3.5 illustrates pipeline flushing due to branch mispredictions using a vertical bar in the Flush subgraph. The number next to each vertical bar in the Flush graph corresponds to the basic block number in Figure 3.6 containing the mispredicted branch. Other relevant pipeline activities across different parts of our simulated microprocessor ranging from cache access, to functional unit usage, to the rate at which instructions are being dispatched, issued and committed are also shown for the same time frame. The resulting current draw and voltage activity are also shown. Lastly, Figure 3.5 shows three distinct phases A,

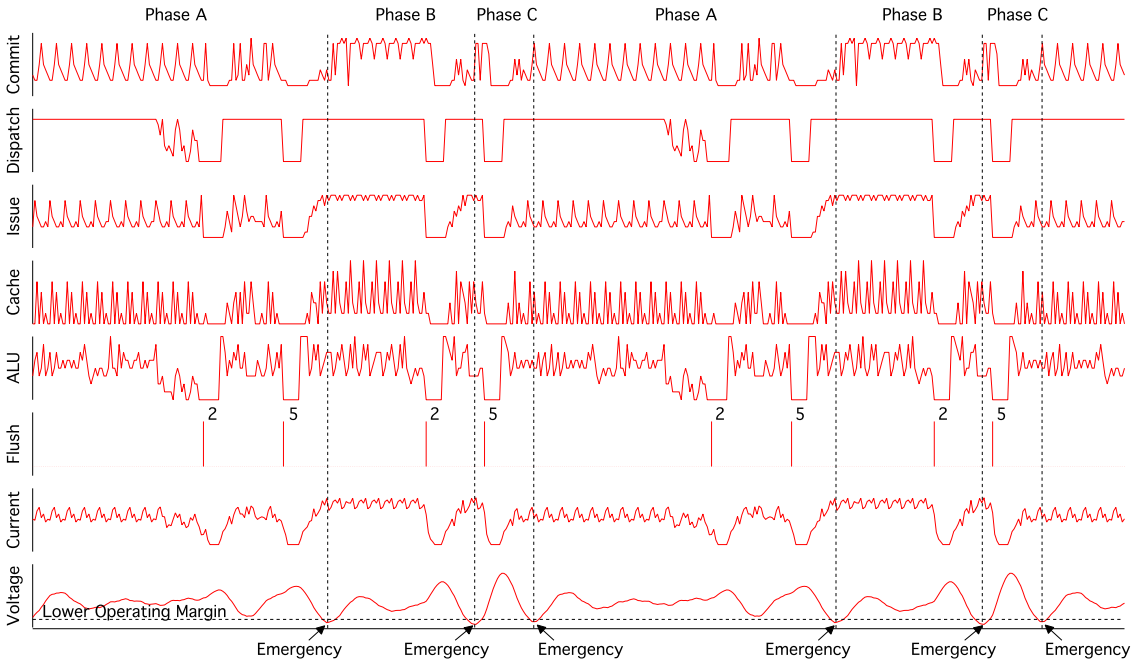


Figure 3.5: Voltage emergencies are associated with recurring activity (phases A, B and C) over 880 cycles. The numbers next to the vertical bars in the Flush graph correspond to the basic block number in Figure 3.6 containing the mispredicted branch.

B and C (see top of figure) and each phase terminates at an emergency (see bottom of figure).

3.2.1 Contextual Information

Microarchitectural events perturb machine activity significantly, but by themselves are not responsible for voltage emergencies. Pipeline flush Event 2 in Figure 3.5 is an ideal candidate for illustrating this point. Event 2 in Phase A causes a voltage droop a few cycles before Event 5 (also in Phase A), but it does not cause an emergency. The same event, however, always causes an emergency in Phase B (at the end of B). Understanding the processor activity leading up to these events explains this

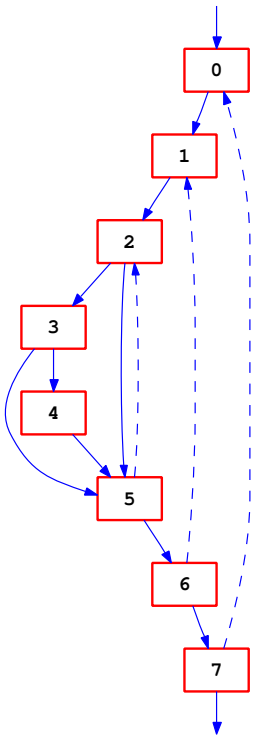


Figure 3.6: An emergency prone nested-loop in function `init_regs` of benchmark *403.gcc*. `init_regs`'s activity snapshot is shown in Figure 3.5.

inconsistent behavior. The Issue, as well as other rates prior to Event 2 are different between Phase A and Phase B, so the perturbation effects of Event 2 are different between the phases. By comparison, pipeline flush Event 5 always occurs just prior to an emergency in both Phase A and Phase C. Nevertheless, our argument that activity prior to an event matters holds true. The voltage just prior to Event 5 in Phase A is rising versus falling in Phase C. The latter occurs because the voltage is already in flux due to the perturbation brought about by Event 2 in Phase B. For this reason, any scheme attempting to characterize and exploit recurring patterns must take into account the execution context preceding an emergency.

3.2.2 Microarchitectural Events and Program Control Flow Interleaving

Voltage emergencies are uniquely identifiable by tracking control flow instructions and microarchitectural events in order of occurrence. Rapid fluctuations in a program's control and data flow and in its level of utilization of processor resources lead to changes in current flow that induce large voltage swings. For instance, the distinct current and voltage activity between phases A, B and C are the result of different control flow paths exercised by the program combined with the voltage droops induced by pipeline flush Events 2 and 5. During the early part of Phase A, the program is executing basic blocks $2 \rightarrow 3 \rightarrow 5$ (from Figure 3.6) in a steady-state manner. The stable and repetitive Issue rate pattern during the early part of Phase A in Figure 3.5 confirms this. Slightly past the midpoint of Phase A, the program switches control flow from basic blocks $2 \rightarrow 3 \rightarrow 5$ to basic blocks $2 \rightarrow 5$. This switch triggers a pipeline flush to recover from speculatively executing incorrect code along Edge $2 \rightarrow 3$ to executing correct code along Edge $2 \rightarrow 5$. The activity on the recovery path following the pipeline flush causes the voltage to droop slightly but not enough to violate the operating margin (shown using Lower Operating Margin). After a few cycles, a misprediction on basic block 5's control instruction eventually leads to a voltage emergency. So the emergency in Phase A is because of the activity including, as well as following, basic blocks $2 \rightarrow 3 \rightarrow 5$ combined with pipeline flush Events 2 and 5. In contrast, the emergency in Phase B arises from executing basic blocks $2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ followed by the single flush Event 2. Consequently, tracking control flow sequence along with pipeline flush events in order of occurrence yields two unique activity patterns representing

Phase A and Phase B.

3.2.3 Repeatability and Stability

Voltage emergencies, like program phases, are repetitive over a program's lifetime, which make them predictable. Consider the three phases illustrated in Figure 3.5. The phases are recurring because execution sequence flows through phases $\mathsf{A} \rightarrow \mathsf{B} \rightarrow \mathsf{C}$ and back to Phase A. A subsequent occurrence of the same phase leads to yet another emergency. For instance, Event 2 always causes an emergency as execution flows through phases $\mathsf{B} \rightarrow \mathsf{C}$, but not through phases $\mathsf{A} \rightarrow \mathsf{B}$. Thus, a pattern of voltage emergency occurrence emerges. Identifying and exploiting such recurring activity is the basis for predicting voltage emergencies in terms of program behavior, as well as microarchitectural behavior.

3.3 Capturing Voltage Emergency Signatures

We refer to activity leading up to a voltage emergency as a voltage emergency signature. These signatures are the enabling mechanism behind effectively suppressing emergencies, either at the hardware or software layer, as we will discuss in subsequent chapters. But first, we shall understand how to capture a voltage emergency signature. This section describes the hardware necessary to capture program control flow and microarchitectural event interleaving. Since voltage emergencies contribute to timing faults, all logic capturing signatures must be designed carefully with sufficiently conservative timing margins.

3.3.1 Emergency Detection

Capturing a voltage emergency signature, with our scheme, requires an emergency to occur at least once. This requires a mechanism to monitor operating margin violations. We rely on a voltage sensor. The sensor is used to signal that an emergency has occurred and the system ought to take appropriate actions.

3.3.2 Fail-safe Recovery Mechanism

Processor state is potentially corrupted as emergencies occur, since voltage emergencies induce timing faults. So we rely on a fail-safe checkpoint-recovery mechanism to recover from emergencies. The fail-safe mechanism initiates a recovery whenever the sensor detects an emergency, and in that process also captures a voltage emergency signature. Checkpoints can be taken at varying intervals (e.g., 10-1000 cycles). We assume a 100-cycle rollback penalty.

Gupta *et al.* [30] have proposed a low-overhead implicit checkpointing scheme to handle voltage emergencies by buffering commits until it is confirmed that no voltage emergencies have occurred while the buffered sequence was in flight. While shown to be effective, implicit checkpointing is specialized to a specific style of processor design (i.e., out-of-order superscalar execution).

Instead, we propose relying on coarse-grained checkpoint-recovery that is already shipping in today's production systems [48, 9]. Researchers are proposing a broad range of novel applications that use traditional checkpoint-recovery [54, 51, 39, 36, 47, 41]. With ever-increasing applications of this fail-safe mechanism, we believe checkpoint-recovery will become part of future mainstream processors. However,

checkpoint-recovery alone as a solution for handling voltage emergencies is unacceptable since the overhead of tolerating repeated emergencies degrades performance.

We assume explicit-checkpointing is robust against sensor delays involving detection to notification time lag. Any checkpoint falling after an emergency, but before its subsequent detection due to sensor delays, can be corrupt. Therefore, providing correct recovery semantics requires maintaining two checkpoints.

3.3.3 Activity History Tracking

Event history tracking is a well-studied topic in the area of branch prediction. Our contribution is unique in that we can identify the information flow that precisely captures activity prone to voltage emergencies. We rely on a shift register to capture the interleaved sequence of control flow instructions and architectural events that give rise to an emergency. Figure 3.7 portrays a high-level schematic of an example 4-entry signature capturing mechanism (a thorough evaluation of signature details will follow in Section 3.4). As a control flow instruction is executed or a microarchitectural event occurs, the corresponding instruction address or event type (e.g., pipeline flush or L2miss) encoding is shifted into the event history register; the oldest entry is simply discarded. Whenever an emergency is detected, a snapshot of the event history register is captured instantly (as illustrated using the dotted lines). The captured snapshot is a voltage emergency signature.

The interleaving of events in the event history register is important for capturing the dynamic current and voltage activity resulting from program interactions with the underlying microarchitecture (as described in Section 3.2). The purpose of tracking

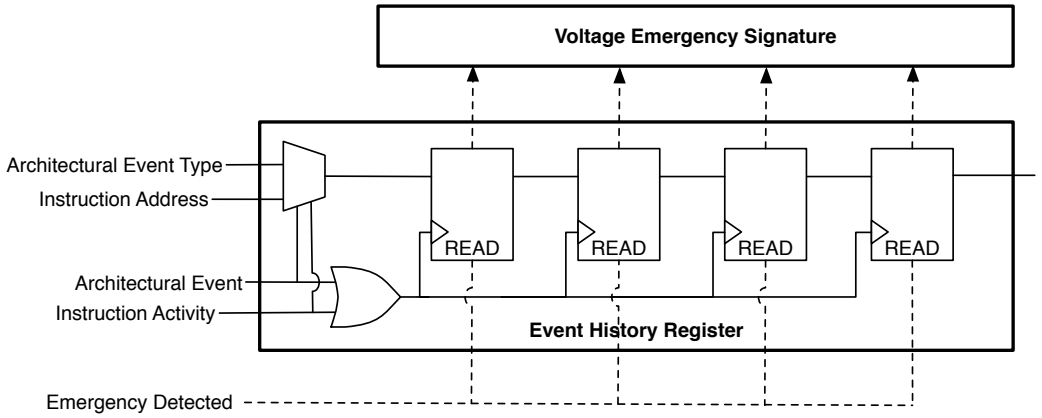


Figure 3.7: Event history register for tracking the interleaving sequence of program control flow and processor events. A voltage emergency signature is a snapshot of the register when an emergency occurs.

the instruction stream is to capture the dynamic path of a program. Consequently, control flow instructions are ideal candidates for tracking a program's dynamic execution path.

Figure 3.8 illustrates example snapshots of the emergencies shown in Figure 5.6 across phases A, B and C. The updates into a 4-entry wide event history register are shown over time. At the point of the emergency in Phase B, the history register contains the following (from oldest to most recent): two control flow instruction addresses (illustrated as BR) and an event encoding for the pipeline flush (illustrated as 2), followed by another branch. It is important to never clear the event history register after capturing a snapshot to maintain a rolling window of contextual information. For example, the oldest BR in Signature C overlaps with the most recent entry in Signature B.

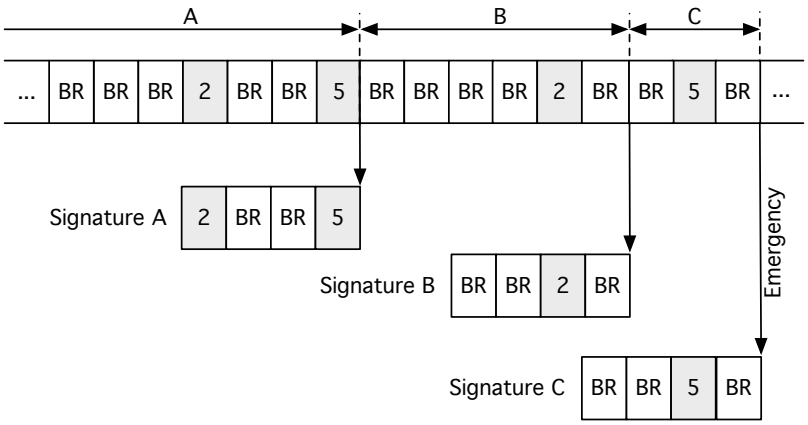


Figure 3.8: Overview of voltage emergency signatures. Taking snapshots of a 4-entry event history register for emergencies illustrated in Figure 5.6 across phases A, B and C.

3.4 Semantics of Voltage Emergency Signatures

We discuss factors that influence the quality of a voltage emergency signature, such as the type and amount of information recorded. The function of a voltage emergency signature is to precisely indicate whether a pattern of control flow and microarchitectural event activity will give rise to an emergency. To evaluate the effectiveness of different flavors of signatures, we define accuracy as the fraction of predicted emergencies that become actual emergencies.

3.4.1 Contents

Information tracking in the event history register must correspond to parts of the execution engine that experience large current draws, as well as dramatic spikes in current activity. The event history register can collect the control flow trace at different points in a superscalar processor: in-order fetch and decode, out-of-order issue, and in-order commit. Each of these points contribute different amounts of

information pertaining to an emergency. For instance, tracking execution in program order fails to capture any information regarding the impact of speculation on voltage emergencies. Tracking information at the in-order fetch and decode sequence captures the speculative path, but it does not capture the out-of-order superscalar issuing of instructions.

The accuracies of different signature types are illustrated in Figure 3.9a, assuming a signature size of 32 entries. Later in Section 3.4.2 we investigate the effect of tracking more or less number of entries. Tracking committed control flow sequences in the event history register gives an accuracy of only 40%. If the history register tracks information at the decode stage, an accuracy of 72% is possible because the decode stage captures the speculative control flow path. Accuracy improves further by 12%, from 72% to 84%, if the history register tracks control flow at the issue stage, since we can now capture interactions more precisely at the level of hardware instruction scheduling and code executed along a speculative path.

Interleaving microarchitectural events with program control improves accuracy even further, as processor events provide additional information about swings in the supply voltage. For instance, pipeline flushes cause a sharp change in current draw as the machine comes to a near halt before recovering on the correct execution path. The last two bars of Figure 3.9a show accuracy improvements from adding microarchitectural event activity to the event history register. The second to last bar represents the effect of capturing events that have the potential to induce large voltage swings—pipeline flushes and secondary (L2) cache misses. An improvement of five percentage points is achieved by taking flushes and L2 misses into account (i.e., total accuracy

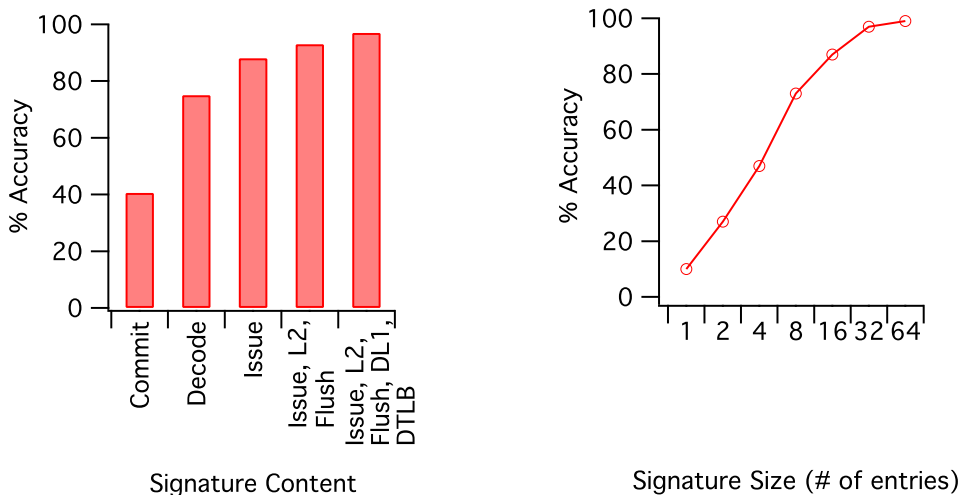


Figure 3.9: Prediction accuracy improves as (a) signature contents represent machine activity more closely and as (b) the number of entries per signature increases.

of 89%). Capturing the more frequently occurring events like DTLB and DL1 misses contributes additional improvements of ∼4%. Microarchitecture perturbations resulting from instruction cache activity (i.e., IL1 and ITLB) are negligible and do not lead to an improvement in accuracy.

From here on, we assume the event history register resides at the issue stage of the pipeline and captures microarchitectural-event activity. More formally, the event history register is updated whenever a control flow instruction is executed, along with Level 1 and Level 2 cache and TLB misses. Lastly, pipeline flushes are also events recorded in the event history register.

3.4.2 Size

Accuracy depends not only on recording the right interleaving of events, but also on balancing the amount of information the event history register keeps. Accuracy

improves as the length of history register increases. However, it can be detrimental to increase the number of register entries beyond a certain count. Large numbers of entries in a signature can cause unnecessary differentiation between similar signatures—signatures whose most recent entries are identical and whose older entries are different, but not significantly so.

Figure 3.9b shows prediction accuracy improves as signature size increases. Accuracy is only 13% on average for a signature containing only 1 entry, which supports our discussion earlier on that voltage emergencies do not solely depend upon the last executed branch or a single microarchitectural event. It is the history of activity that determines the likelihood of a recurring emergency. Prediction accuracy begins to saturate once signature size reaches 16, and peaks at 99% for a signature size of 64 entries.

3.4.3 Coverage

For signatures to be effective and to amortize the cost of discovering signatures at execution time, emergency-prone activity that signatures represent must be recurring. Figure 3.10 demonstrates that signatures are recurring, since the number of new signatures we discover over time decreases. We capture this in the form of Compulsory Misses. A compulsory miss occurs when we record a signature pattern for an emergency that was previously unknown.

Figure 3.10 illustrates the percentage of new signatures over time (in terms of the number of committed instructions). The dropping percentage of compulsory misses over time demonstrates that the coverage of emergency-prone activity is increasing.

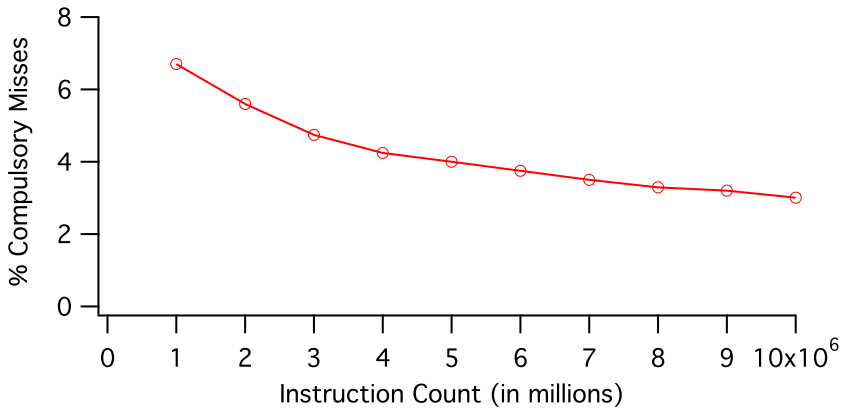


Figure 3.10: The number of new emergency signatures we discover over time (Compulsory Misses) is decreasing, which indicates that signatures are recurring.

However, the number of misses does not asymptotically approach zero because we continuously discover new signatures as the program goes through different activity or phase changes.

3.5 Accuracy of Voltage Emergency Signatures

In this section, we demonstrate the robustness of signatures assuming a signature size of 32 entries. We show that it is robust across different machine configurations and power delivery subsystems with no need for fine-tuning. We also demonstrate they are capable of anticipating emergencies some 16 cycles ahead of time with 90% accuracy.

3.5.1 Robustness

Applications exhibit different characteristics that drive the machine into different levels of activity and, therefore, varying rates of current draw. Figure 3.11 plots

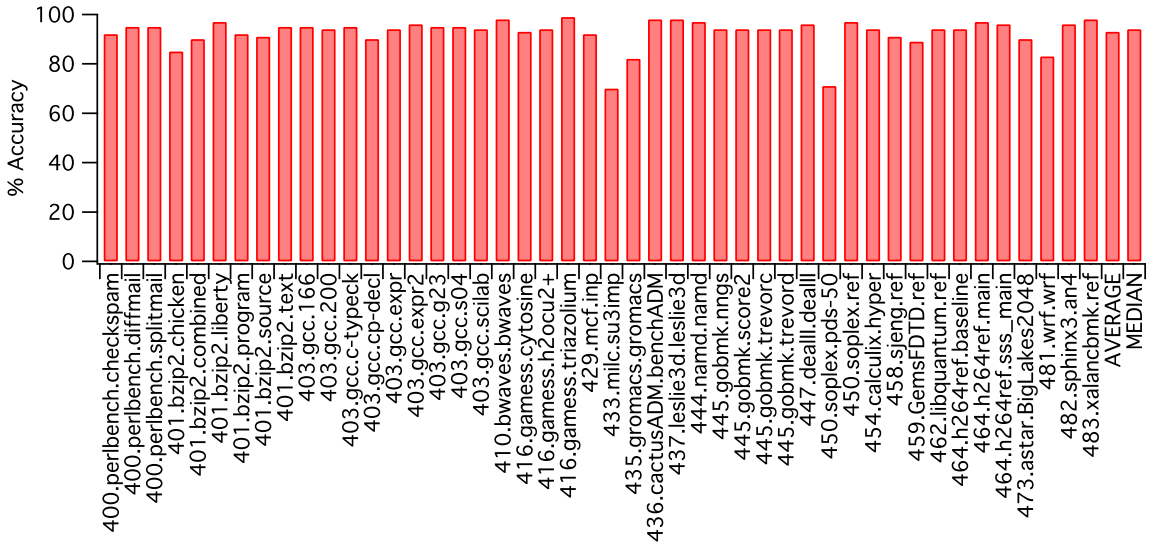


Figure 3.11: In order to evaluate the robustness of voltage emergency signatures we ran several benchmarks, including all the different input data sets provided by Spec for the CPU2006 benchmark suite. We find that the prediction accuracy of voltage emergency signatures consistently remains high across very different program types.

prediction accuracy across the spectrum of benchmarks from CPU2006. For benchmarks with multiple inputs, we present the average prediction accuracy across different inputs. The signatures enable high prediction accuracy with an average of 93% and a median of 94%. Voltage emergency signatures are able to handle a range of benchmarks from control-flow-intensive benchmarks like *403.gcc* and *400.perlbench* to memory-intensive benchmarks like *429.mcf*, and to *462.libquantum* that exhibit a large number of microarchitectural events such as cache misses. Overall, high prediction accuracy is observed across both the integer and floating-point benchmarks.

3.5.2 Retargetability

Figure 3.12 shows prediction accuracy when we pair different power delivery packages Pkg 1, Pkg 2, and Pkg 3 with our baseline microprocessor design Arch 1 (see Ta-

ble B.1 in Appendix B for details), average prediction accuracy remains high (93%, 96%, and 95%, respectively) despite decreasing package quality. Signatures consistently enable emergency prediction with over 90% accuracy without specialization. By comparison, sensor-based schemes require careful configuration of soft thresholds [30]. When we pair package Pkg 1 with a simpler out-of-order processor Arch 2 (one with the same structure as that in Table B.1, but with half-sized fetch and decode widths and half-sized buffers, queues, and caches), the accuracy of our predictor still remains high at 97%.

3.5.3 Lead time

Up to this point, voltage emergency signatures represent activity up until the moment of an emergency. This is an optimistic assumption, allowing us to verify the effectiveness of signatures as good leading indicators, or *predictors* of voltage emergencies. However, real systems require non-zero lead times to account for circuit delays in order to make effective use of signatures.

To experiment with these delay times, we erase trailing segments of emergency signatures. Figure 3.13 shows accuracy slightly degrades from 93% as lead time increases. However, even with 16 cycles of lead time, prediction accuracy remains high at 90%. This indicates that signatures can anticipate emergencies 16 cycles before an emergency is imminent.

As we demonstrate in the next chapter, it is possible to build intelligent hardware prediction logic using voltage emergency signatures. These predictors can anticipate impending emergencies effectively several cycles ahead of time using signatures (as we

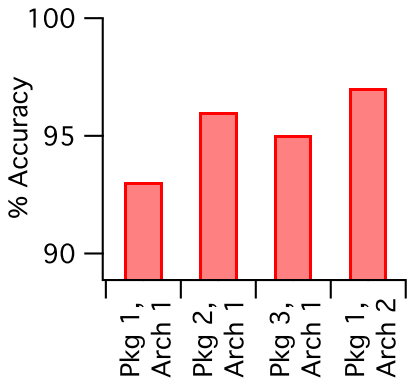


Figure 3.12: The predictor sustains high prediction accuracy across different of power delivery packages and microarchitecture combinations.

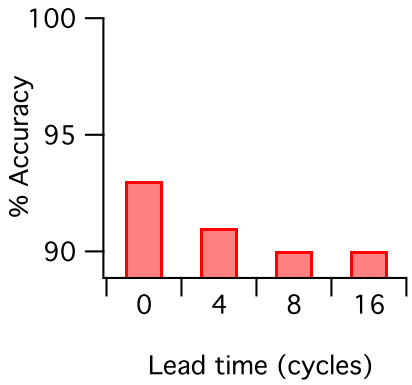


Figure 3.13: The predictor predicts emergencies with sufficient time to actuate a throttling mechanism to avoid an impending emergency.

see here), and therefore avoid them by slowing processor activity down appropriately and just briefly. Since there is sufficient lead time, voltage recovers back to its nominal level gradually. Therefore, the processor can continue executing smoothly without an abrupt glitch. This is only possible because the prediction accuracy of signatures is high.

Chapter 4

Avoiding Emergencies Using Voltage Emergency Signatures

Contents

To reduce the gap between nominal and worst-case operating voltages, this chapter introduces a *voltage emergency predictor* that identifies when emergencies are imminent and avoids their occurrence. The emergency predictor anticipates dangerous voltage swings using *voltage emergency signatures*. Using these signatures, the predictor throttles (or slows) machine execution to prevent emergencies. An emergency signature is an interleaved sequence of control-flow events and microarchitectural events leading up to an emergency. The predictor captures these voltage emergency signature when an emergency first occurs by taking a snapshot of relevant event history and storing it within its prediction tables. A built-in checkpoint-recovery mechanism then rolls the machine back to a known correct state and resumes execution. Subsequent occurrences of the same emergency signature cause the predictor to throttle execution and avoid the impending emergency. By doing so, the predictor enables aggressive timing margins in order to maximize performance.

An effective emergency avoidance predictor mechanism must meet two criteria: First, it must anticipate an emergency accurately to prevent performance degradation due to unnecessary throttling. Second, it must initiate the emergency avoidance mechanism with enough lead time to throttle and successfully prevent the emergency from occurring. Voltage emergency signatures inherently demonstrate both these traits, as shown in the previous chapter. This chapter starts off with Section 4.1 showing that it is possible to use the predictable behavior of voltage emergencies to smooth away recurring emergency activity. The subsequent section, Section 4.2, compares our new solution against prior work, demonstrating the robustness of our predictor scheme. Finally, Section 4.3 concludes with implementation details.

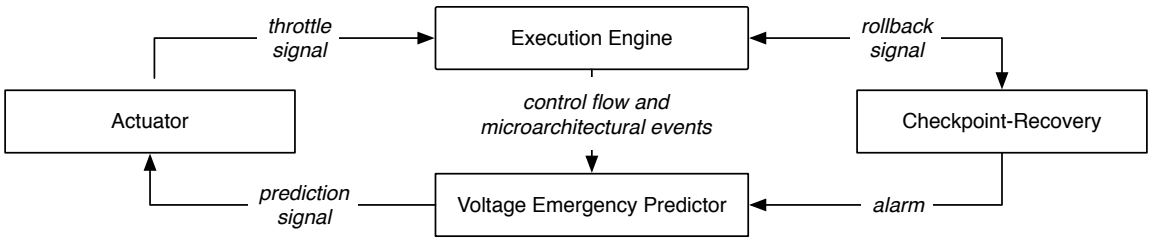


Figure 4.1: Overview of our voltage emergency predictor. The predictor relies on code and microarchitectural event activity (i.e., voltage emergency signatures) instead of current and voltage activity as prior schemes do to decide when to throttle. It is trained using a fail-safe checkpoint-recovery mechanism.

4.1 Signature-based Throttling to Prevent Emergencies

A voltage emergency predictor is a structure that learns recurring voltage emergency activity during runtime and prevents subsequent occurrences of said emergencies via execution throttling. By doing so, the predictor enables aggressive timing margins in order to maximize performance. This section explains the components of the predictor and the overall scheme that it fits within.

4.1.1 Voltage Emergency Predictor

Figure 4.1 is a block diagram of the overall scheme. The Voltage Emergency Predictor monitors control flow and microarchitectural events and keeps track of these voltage emergency signatures that lead to emergencies. The predictor captures these signatures just before the Checkpoint-recovery block initiates a rollback. Voltage sensors scattered across the chip trigger the checkpoint-recovery rollback signal whenever voltage droops below the minimum operating voltage margin. Checkpoint-recovery

mechanism rolls the machine back to a known correct state after an emergency occurrence and resumes execution.[1] Subsequent occurrences of the same emergency signature cause the predictor to throttle execution and prevent the impending emergency. The predictor does this by invoking the Actuator, telling it to slow machine execution, which allows voltage to recover back to its nominal level gradually. Unlike prior work, the prediction-based approach allows the microprocessor to operate with margins much tighter than otherwise possible.

A voltage emergency predictor outperforms previously proposed architecture-centric techniques [26, 34, 43, 44] that rely on voltage sensors to detect and react to emergencies via throttling. Prior schemes detect emergencies by solely relying on voltage sensors. These voltage sensors monitor the supply voltage for specific soft threshold crossings. Whenever the supply voltage falls below the soft threshold setting of a sensor, the machine throttles execution in pursuit of emergency prevention. But delay in detection and soft threshold settings can severely impact or limit how effective these sensor-based schemes are.

Prior schemes cannot always guarantee correctness without incurring large performance penalties. Aggressively setting the soft threshold close to the operating margin limits time available to throttle and successfully prevent an emergency. Alternatively, setting the threshold too conservatively leads to unnecessary throttling that degrades performance. Not every conservative soft threshold crossing eventually crosses the lower operating voltage margin.

Figure 4.2 illustrates why the predictor outperforms sensor-based throttling. As

[1]Please refer back to the previous chapter for specific details on capturing voltage emergency signatures.

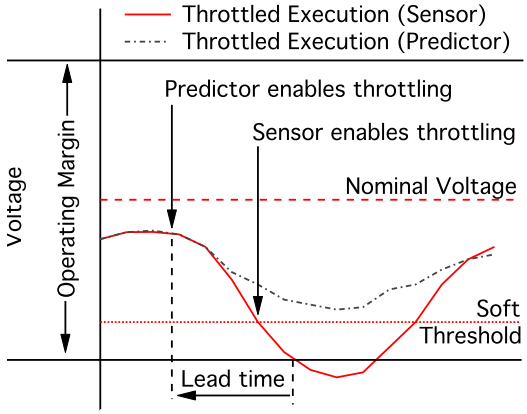


Figure 4.2: The voltage waveforms help illustrate how the predictor throttles execution with sufficient lead time to prevent emergencies instead of relying on soft thresholds. The predictor is therefore able to prevent emergencies better.

soon as the predictor identifies a voltage emergency signature, the predictor starts to throttle execution with sufficient lead time to prevent an emergency from occurring. The predictor recognizes and tracks patterns of emergency-prone activity to proactively throttle execution well before an emergency can occur. In contrast, sensor-based throttling, corresponding to waveform Throttled Execution (Sensor) from Figure 2.6b, fails to avoid the emergency with aggressive soft threshold settings. Relaxing this soft threshold allows more detection and throttling time, but the system incurs large performance penalties due to false warnings.

An additional benefit is that our voltage emergency predictor does not require fine tuning based on specifics of the microarchitecture nor the power delivery subsystem, as is the case with sensor-based schemes. The current and voltage activity of a microprocessor are products of machine utilization that are specific to the workload's dynamic demands. Relying on voltage emergency signatures allows the predictor to dynamically adapt to emergency-prone behavior patterns resulting from

the processor's interactions with the power delivery subsystem without having to be preconfigured to reflect the characteristics of either.

4.1.2 Feedback Mechanism

Voltage emergency signatures are dynamic and, as such, the overall scheme requires that the predictor discovers emergency signatures at runtime. Initially, the predictor does not know any emergency signatures. The predictor detects emergencies as margin violations occur at run-time. Since an emergency can potentially corrupt machine state, the predictor relies on checkpoint-recovery to recover and resume execution.

Detecting an emergency using checkpoint-recovery is robust since the predictor may occasionally mispredict, thereby allowing an emergency to corrupt execution. The predictor cannot prevent all emergencies. In such cases, checkpoint-recovery acts as a fail-safe mechanism. It recovers processor state and the machine incurs a rollback penalty.

However, our experimental data indicates that the number of checkpoint-recoveries necessary is small. In tests, only $\sim$1% of emergencies result in rollback penalties and all other emergencies are avoided successfully. In other words, the predictor is very good at preventing emergencies. Over time, the predictor collects a history of emergency-prone activity and uses this history to successfully prevent future emergencies via throttling.

4.1.3 Throttling Actuator

Predictor designers can choose the flavor of throttling they wish to implement. The choice may vary depending upon the margin setting and the aggressiveness of the underlying microarchitecture. Power hungry cores require aggressive throttling. Previously proposed throttling solutions range from frequency throttling, to pipeline freezing/firing, to issue ramping, and altering the number of accessible memory ports [26, 34, 44, 43].

The benefit of using the voltage emergency predictor to trigger the actuator is that it does not require fine tuning based on specifics of the microarchitecture nor the power delivery subsystem. But such is the case with reactive sensor-based schemes. According to prior work, the throttling mechanism must respond fast enough to prevent the emergency, especially considering that sensor delays are a large component of the throttling actuation loop [30]. Therefore, the choice of throttling for sensor-based schemes will depend upon how quickly the sensors can detect and identify an impending emergency. The choices are limited, as compared to our scheme which can anticipate emergencies with 90% accuracy even some 16 cycles ahead of time (see Section 3.5.3).

4.2 Efficiency Comparison to Prior Work

An aggressive reduction in operating voltage margins can translate to higher performance or higher energy efficiency. Since performance and power are inextricably tied, this section focuses on clock frequency performance improvements for a proces-

sor configuration that is representative of a typical out-of-order superscalar processor like the Pentium 4 microarchitecture design. Assessing performance also enables straightforward accounting of penalties resulting from throttling and rollbacks.

This section evaluates the maximum attainable performance within the context of all runtime costs for our scheme and compare to a variety of idealized and practical approaches. More specifically, this section compares the signature-based predictor to a variety of other schemes that also use throttling and/or checkpoint-recovery. Initial analysis makes optimistic assumptions about hardware implementation of the voltage emergency predictor, but subsequently we explore design tradeoffs, showing that a resource-constrained predictor achieves performance improvements. All schemes assume a half-rate throttling mechanism that gates every other clock cycle. For sensor-based schemes, we assume sensors are ideal with zero delay, and can instantly react to either resonant or single-event-based voltage emergencies. We also test them with respect to tolerable delays. For our predictor, we assume an unbounded prediction table with a voltage emergency signature predictor with 16 cycle lead time.

Worst-case Operating Voltage Margin. Designers typically build in conservative margins (guardbands) to safeguard against potentially large voltage dips that can lead to timing violations. Such margins translate to clock frequency reductions and performance loss. Recent papers on industrial processor designs have shown that 15% to 20% operating voltage margins would be required to protect against voltage emergencies [33, 17]. Similarly, our setup experiences a worst-case droop of 13.5%.

Voltage Margin to Frequency Scaling. The roughly linear relationship between operating voltage and clock frequency facilitates translation of voltage margin

reductions into performance gains. Based on detailed circuit-level simulations of an 11-stage ring oscillator consisting of fanout-of-4 inverters, we observe a 1.5x relationship between voltage and frequency at the PTM 32nm node [55]. This relationship is consistent with results reported by Bowman *et al.* [17], which show that a 10% reduction in voltage margins leads to a 15% improvement in clock frequency.

While the evaluation in this section relies on a 1.5x voltage-to-frequency scaling factor, we also see a disconcerting trend across technologies. Simulation results reveal voltage-to-frequency scaling factors of 1.2x, 1.5x, 2.3x, and 2.8x for PTM nodes at 45nm, 32nm, 22nm, and 16nm, respectively. Given a slowdown in traditional constant-field scaling trends, sensitivity of frequency to voltage is growing, which increases the need for techniques that can efficiently reduce voltage noise in future processors.

Based on the 1.5x scaling factor, a 4% operating voltage margin corresponds to a 6% loss in frequency. Similarly, a conservative voltage margin of 13.5%, enough to cover the worst-case dips observed, leads to 20% lower frequency. If we take this conservative margin as the baseline for comparisons and the 13.5% margin can reduce to 4% while avoiding voltage emergencies, the corresponding clock frequency improvement suggests system performance gains of 17.5%. This sets the upper bound on maximum performance gains achievable. We make the simplifying assumption that frequency improvements directly translate to higher overall system performance.

Calculation of performance gains shown for each scheme begins with the maximum 17.5% gains possible, which then scales down by accounting for all performance overheads. Again, a conservative voltage margin of 13.5% allows for emergency-free,

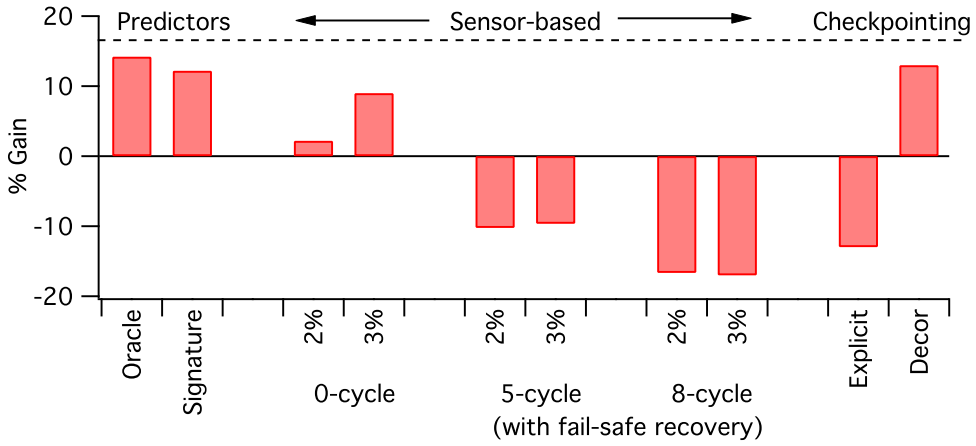


Figure 4.3: Performance gains because of reducing the voltage margin from a conservative 13.5% (assuming the worst-case voltage swing) to an aggressive 4% with different fail-safe mechanisms to handle voltage emergencies. The dotted line indicates the ideal gain from reducing the margin.

lower-frequency operation and is the common baseline for all comparisons. Figure 4.3 shows the performance gains for the different schemes while Figure 4.4 breaks down the associated penalties into throttling and rollback costs.

4.2.1 Predictors

This section begins with an ideal oracle predictor evaluation. It is an important comparison point. An oracle predictor sets the upper bound on the potential benefits of all other prediction-based schemes.

Oracle Predictor. An oracle predictor throttles exactly when an emergency is about to occur, and it always prevents the emergency. It does not waste throttles nor does it incur rollback penalties. By removing all voltage emergencies, it gives the best performance gain achievable by a predictor (14% in Figure 4.3), while incurring only 2.9% throttling overhead (see Figure 4.4).

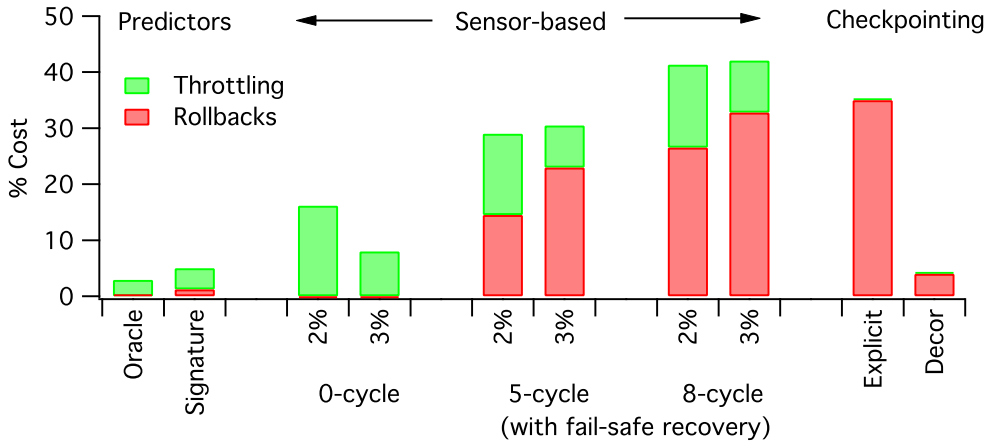


Figure 4.4: Breakdown of the throttling and rollback costs associated with achieving the gains shown in Figure 4.3 across the different schemes.

Voltage Emergency Signature-based Predictor. The signature-based prediction scheme incurs total performance overhead of 5% on average across all benchmarks. This overhead includes the rollback cost for detecting emergencies, as well as subsequently throttling to avoid them. The rollback penalty for discovering signatures is ∼1.2% and the throttling penalty is ∼3.8%, as the breakdown in Figure 4.4 shows. This slight overhead translates to performance gain of 12.3% relative to our baseline, which is just 2.2 percentage points less than the oracle predictor despite rollback costs for discovering signatures at execution time. The overhead is low since the predictor is very good at preventing emergencies once it learns the recurring emergency signature patterns.

4.2.2 Sensor-based Schemes

Ideal Sensor. Still using a 4% operating margin as the hard lower operating voltage margin, we evaluate sensor-based schemes for two soft voltage threshold set-

tings, a conservative threshold of 2% and an aggressive one of 3%. We optimistically assume that the sensor has no delay and that all emergencies that would occur after voltage crosses the soft threshold are prevented (i.e., there is no rollback cost). Note that an actual sensor would have a delay of several cycles and so would give worse performance results (as we discuss below).

Despite optimistic assumptions about sensor delay, performance gains for the 2% and 3% soft thresholds are only 2.2% and 9.0%, respectively. Gains for the sensor-based schemes are low because of the high fraction of benign soft threshold crossings that lead to unnecessary throttling penalties, as shown earlier in Figure 2.7b.

Sensor-based Throttling with Fail-safe Recovery. We extended the sensor-based scheme with checkpoint-recovery to test whether we can leverage the simpler sensor-based mechanism by combining it with a fail-safe guarantee to protect against those emergencies that go undetected due to delay. We evaluated delay times of 5-cycle and 8-cycles.

Our results in Figure 4.3 indicate that we cannot achieve performance gains by extending sensor-based technology with checkpoint-recovery. Both the 2% and the 3% soft threshold schemes suffer from negative gains. The gain with a 5-cycle delay is -9%, and the gain drops further to -18% as delay increases to 8 cycles. Delay causes a large fraction of emergencies to be missed. Therefore, Figure 4.4 shows that the system experiences high rollback cost because the processor must frequently recover previous safe state.

4.2.3 Checkpoint-recovery

Explicit Checkpoint and Recovery. Gupta *et al.* propose the use of checkpointing specifically for the purpose of handling voltage emergencies [30]. They demonstrate that explicit checkpoint-recovery schemes cannot be directly applied to handling voltage emergencies due to their high rollback costs. The system experiences a -13% performance gain when using an explicit checkpoint-recovery mechanism that has a 100-cycle rollback penalty. These results confirm prior work.

Delayed Commit and Rollback. To overcome limitations of explicit checkpoint-recovery, Gupta *et al.* propose an implicit checkpointing scheme called DeCoR that speculatively buffers register file and memory updates until it has been verified that no emergency has occurred during a period long enough to detect one [30]. The commit proceeds as usual unless an emergency is detected, in which case the machine rolls back and resumes execution at a throttled pace. This system assumes a 5-cycle sensor delay for DeCoR to detect emergencies, representing the best case as demonstrated by its designers.

DeCoR's performance gain is 13.0%. The signature-based predictor outperforms DeCoR, but only slightly. However, the benefits of using a signature-based predictor outweigh using DeCoR for a general-purpose processor design. DeCoR's implicit checkpointing requires changes to traditional microarchitectural structures. In comparison, coarse-grained checkpoint-recovery is already shipping in production systems [48, 9] and can serve multiple purposes ranging from boosting processor performance [51, 39, 36] to fault detection [47] and debugging [41]. A signature-based predictor leverages the coarse-grained checkpoint-recovery hardware, thereby retain-

ing all the benefits of coarse-grained checkpoint-recovery while also reducing voltage emergencies.

4.3 Implementing a Voltage Emergency Predictor

Up to this point, prior evaluation assumes unbounded or infinite resources for matching voltage emergency signatures. This section discusses implementing the predictor under resource constraints. First, we discuss a Content Addressable Memory (CAM) approach, but because this requires an impractically large 64KB CAM to achieve good performance, we subsequently evaluate the more space-efficient Bloom filter structure.

A prediction table is a hardware structure for recognizing voltage emergency signatures. Lookups in the prediction table happen whenever the processor updates the contents of the event history register. The processor combines the event sequence from the history register with the address of the last issued branch instruction to form a signature, and then tries to match that signature in the prediction table. If the match succeeds, the processor throttles execution to prevent a potential emergency.

Prediction table management takes place by a software component in firmware. The use of firmware to manage the prediction table is consistent with systems in which firmware manages energy and deals with processor design errors [24, 42, 52, 46]. When an emergency occurs, the emergency predictor firmware is responsible for managing the signature.

To avoid large space overheads, a realistic predictor implementation uses a compact 3-bit encoding per signature entry regardless of the implementation. This encod-

ing captures processor events and it records the outcome of each conditional branch (fall-through or taken). But encoding causes aliasing between signatures. Therefore, an encoded signature also contains the program counter for the most recently issued branch—the *anchor PC*. Combining an anchor PC with branch outcomes gives complete path information for a signature. The 3-bit encoding compactly captures all of the relevant information consisting of different processor events, and takes into account the edge taken by each branch (i.e., fall-through paths are encoded as 000 versus 001 for taken edges). This compact representation results in a total signature length of 16 bytes (4 bytes for the anchor PC and 12 bytes for a signature size of 32 entries with 3 bits per entry).

4.3.1 Content Addressable Memory (CAM)

A CAM is a natural structure for implementing a prediction table. As per our investigation of different CAM sizes (see Figure 4.5), we find a large CAM of 64KB is necessary to achieve a gain that is comparable to the unbounded predictor. However, since a CAM-based structure consumes large amounts of power and area, it is only practical in small sizes. Unfortunately, at small sizes capacity misses prevent emergencies from being detected, which leads to severe rollback cost. Performance gain is negative for a 4KB CAM.

4.3.2 Bloom filter

A Bloom filter is a compact lookup structure that saves space, but may sometimes return a false match. It is a probabilistic hash table that maps keys to boolean values,

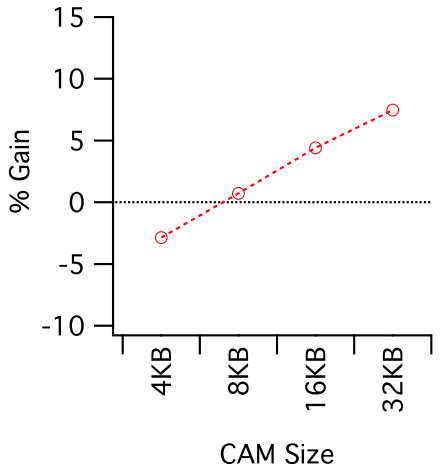


Figure 4.5: Performance gains using a CAM-based signature predictor. CAM must be sufficiently large to tolerate capacity misses, but large CAMs are impractical and inefficient.

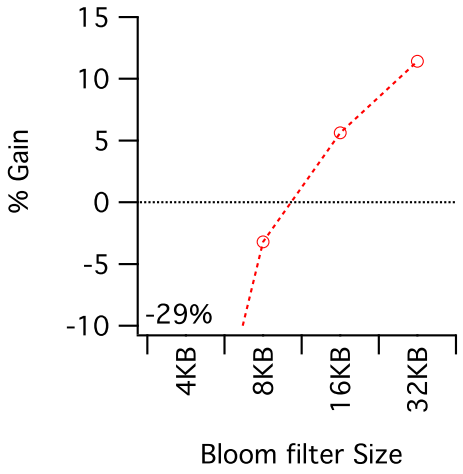


Figure 4.6: A Bloom filter-based signature predictor does not suffer rollback penalties unlike a CAM. However, sizing the structure appropriately is important to tolerate its false positives.

implemented using a bit vector and k hash functions. The procedure to add a key to the Bloom filter hashes the key k ways and sets the bits in the bit vector corresponding to the k indices returned by the hash functions. A key matches in the Bloom filter if and only if the bits for all k indices hashed from that key are set. With some probability, all of the indices for a key that has never been entered may nevertheless be set, in which case matching that key produces a false positive result. False positives only affect performance, not correctness. Therefore, the predictor can tolerate false positive activity.

Figure 4.6 plots the performance gains when using a Bloom filter implementation. A Bloom filter with three hash functions achieves better performance than a CAM at sizes past 8KB. Gains are 5.62% versus 4.42% using a 16KB Bloom filter instead of a 16KB CAM. A Bloom filter is also comparatively more energy-efficient.

However, at smaller sizes Bloom filters have higher false positive rates, which causes unnecessary throttling that degrades performance. Figure 4.6 shows that at 4KB, performance gain is -29%. Therefore, a Bloom filter needs to be sufficiently large to give acceptable performance. It takes a size of 32KB to achieve gains that are comparable to an unbounded signatures-based predictor. But at more practical sizes, like 8KB and 4KB, performance gains are still negative. Consequently, intelligent structures/optimizations are necessary to achieve performance gains within reasonable structure sizes.

4.3.3 CAM Bloom filter

This section proposes three optimizations that allow us to improve the performance of a naive Bloom filter. Due to these optimizations, the system achieves better performance by reducing the number of throttles due to false positive lookup hits. The optimizations consist of thresholds, CAM-based filtering of Bloom filter lookups, and signature compaction. We explain why these optimizations enable better performance than a CAM or a Bloom filter by itself. In summarizing, we demonstrate their effectiveness by showing the percentage of throttles they reduce by constraining the number of false positives.

Thresholds. An effective way of reducing false positives is to keep the occupancy of the Bloom filter low. That is done by excluding the less frequently occurring emergency signatures. The trade-off is that with higher thresholds, the predictor misses more emergencies and will therefore incur more rollback cost. The firmware that manages the prediction table could at the same time profile signature occurrences

Benchmark	Number of Signatures	Number of Emergencies	Benchmark	Number of Signatures	Number of Emergencies
400.perlbench.checkspam	12317	255923	400.perlbench.diffmail	7906	136066
400.perlbench.splitmail	62	146425	401.bzip2.chicken	2430	126905
401.bzip2.combined	88617	986728	401.bzip2.liberty	80	202660
401.bzip2.program	70819	897442	401.bzip2.source	90923	1068100
401.bzip2.text	2641	99015	403.gcc.166	31245	1280457
403.gcc.200	47233	1272642	403.gcc.c-typeck	57026	1310646
403.gcc.cp-decl	13856	626146	403.gcc.expr	35311	1348139
403.gcc.expr2	87174	978192	403.gcc.g23	5255	299808
403.gcc.s04	2525	341422	403.gcc.scilab	49065	1303976
410.bwaves.bwaves	3739	849930	416.gamess.cytosine	34481	596727
416.gamess.h2ocu2+	31018	450079	416.gamess.triazolium	41202	1169664
429.mcf.inp	7571	1006028	433.milc.su3imp	792	522592
435.gromacs.gromacs	12625	352528	436.cactusADM.benchADM	17714	333852
437.leslie3d.leslie3d	26013	822309	444.namd.namd	64	2097
445.gobmk.nngs	68060	632549	445.gobmk.score2	67003	617961
445.gobmk.trevorc	68398	654497	445.gobmk.trevord	67189	624310
447.dealII.dealII	5266	170814	450.soplex.pds-50	3906	309634
450.soplex.ref	1347	180947	454.calculix.hyper	366	254818
458.sjeng.ref	96779	1025541	459.GemsFDTD.ref	2982	1051679
462.libquantum.ref	39	159201	464.h264ref.baseline	19065	400705
464.h264ref.main	46127	539755	464.h264ref.sss_main	58928	688194
473.astar.BigLakes2048	9704	135274	481.wrf.wrf	4101	222365
482.sphinx3.an4	4701	348221	483.xalancbmk.ref	9968	1064952

Table 4.1: Number of voltage emergency signatures and the number of emergencies they represent across the different benchmarks and their inputs.

and exclude those signatures whose occurrence counts fall below a chosen threshold.

Before proceeding to understand the effect of thresholds on Bloom filter population, we must initially understand the number of voltage emergency signatures and the dynamic number of voltage emergencies they represent (assuming we do not throttle to avoid emergencies). Table 4.1 shows that the number of signatures varies significantly. For example, benchmark *403.gcc* has over 87000 signatures that repeatedly give rise to emergencies under input data set *expr2*. At the other end of the spectrum are benchmarks like *444.namd* and *462.libquantum* which have only 64 and 39 signatures, respectively. The number of emergencies is in the hundreds of thousands.

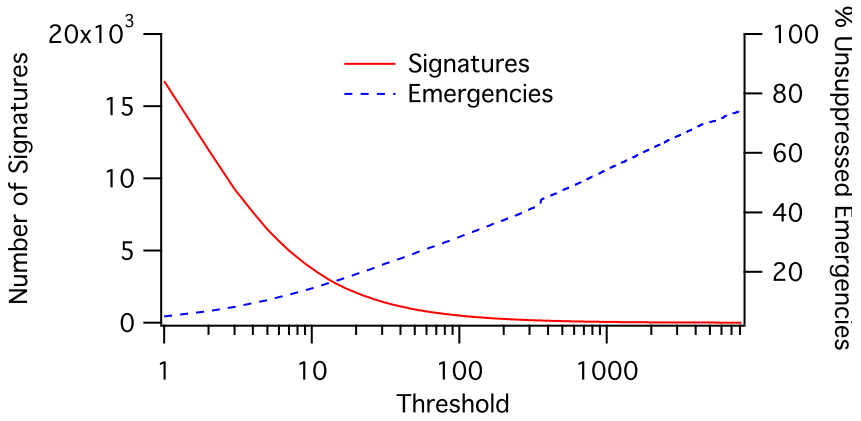


Figure 4.7: When resources are limited, thresholds help to identify *hot* signatures that are resource-worthy. But thresholds cause some emergencies to go unsuppressed since their signatures are omitted from the predictor's lookup table.

Figure 4.7 plots the effect of applying thresholds on the number of signatures we insert into the Bloom filter on the log scale. It also shows the fraction of emergencies that go unsuppressed as a result of applying thresholds. Consider a threshold of one at which only 4.9% of all emergencies go unsuppressed (i.e., they cause rollbacks). Just waiting for an emergency signature to recur drastically reduces the number of signatures per benchmark. On average, at a threshold of one only 16000 signatures remain across the entire benchmark suite. The knee in the curve for thresholds is slightly past 10 emergencies, at which point the number of signatures drops down even further to ∼2000. The number of signatures continues to fall further to only a few hundred signatures at an aggressive threshold of 100, which indicates that few signatures contribute to most emergencies. Therefore, we can reduce the number of false positives by only storing the *hot* (or frequently occurring) signatures in the Bloom filter.

Bloom filter Plus CAM. By screening the anchor PC components of signatures using a CAM, we can reduce the number of lookups in the Bloom filter. This

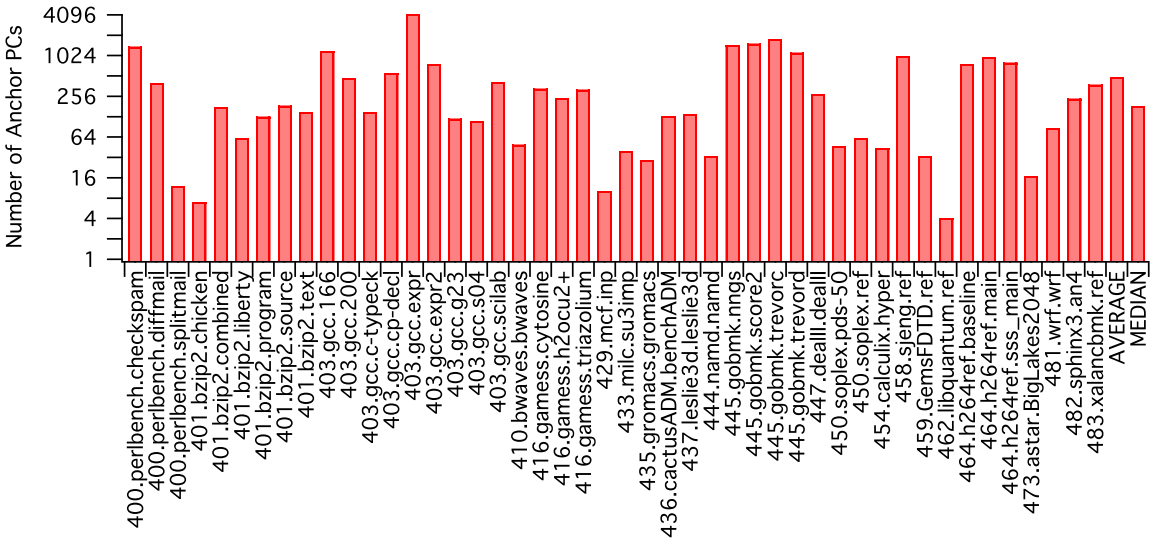


Figure 4.8: The number of static program locations where emergencies occur (i.e., anchor PCs) is only a few hundred across a large spectrum of benchmarks. Therefore, a small CAM can be used to enable the lookup logic only when execution reaches these locations.

effectively reduces the number of times false positives cause throttling. Figure 4.8 shows the number of anchor PCs across the different benchmarks on a binary log scale. The maximum number of such PCs we discover, even considering a large code footprint application like *403.gcc*, is around 4000. Caching such a large number of 32-bit addresses will require an impractical 16KB CAM.

However, we find that by caching only the working set of anchor PCs, a small CAM is sufficient assuming it relies on a Least Recently Used (LRU) replacement algorithm. Carefully sizing the CAM is important because capacity misses allow emergencies to happen, which leads to rollbacks. Figure 4.9 shows that rollback cost is relatively small when no thresholds are active. It is between 1.5% and 0.5% for CAM sizes of 128 bytes (32 entries) and 256 bytes (64 entries), respectively. Performance loss due to capacity misses is even more negligible at 512 bytes (128 entries). However, rollback

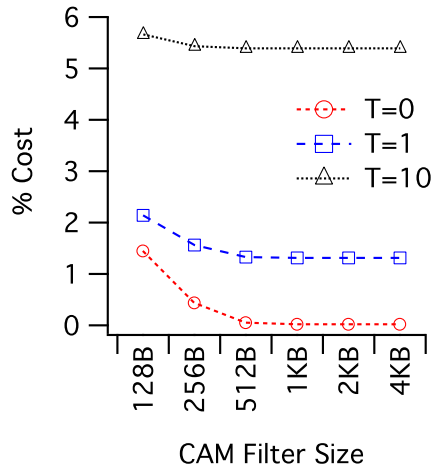


Figure 4.9: Rollback cost due to capacity misses in the CAM, which we use to control lookup access into the Bloom filter as a means of reducing the number of false positive throttles.

penalties due to thresholds dominate at a threshold of one (T=1) and a threshold of 10 (T=10), so much so that even a large CAM cannot mitigate the cost.

Signature Compaction. Signatures corresponding to a specific anchor PC sometimes exhibit similarity. Therefore, we can congregate similar signatures together. The Bloom filter experiences fewer false positives by folding multiple signatures corresponding to a specific anchor PC into a single representative signature. By using a weighted similarity metric based on Manhattan distance, we determine how much compaction is possible for a set of signatures corresponding to a particular benchmark. Let x and y be k-element signatures associated with the same instruction address. We define the similarity of x and y to be:

$$s = \frac{2}{k(k+1)} \sum_{i=1}^{k} \begin{cases} i & \text{if } x_i = y_i \\ 0 & \text{otherwise} \end{cases}$$

If the signatures are identical, s is one. If no two corresponding elements are the same, it is zero. The later elements in x and y correspond to later events in time.

They are more heavily weighted in s, because they are more significant for emergency prediction. Other measures of similarity might yield better compaction, but they would be more expensive to compute in hardware. For a given instruction address, we partition the signatures into maximal sets in which each signature x is related to one or more other signatures y with similarity of 0.9 or greater. The resulting partition is then used instead of the original signature set.

Signature compaction reduces the total number of signatures by over 61% on average. Figure 4.10 illustrates the percentage of compaction we achieve across the benchmarks. The stack plot shows the upper bound across the different thresholds. The biggest winners are usually benchmarks that experience a large number of signatures. For instance, applying compaction reduces the number of signatures in *403.gcc.expr2* by ∼60% when we do not apply thresholds (T=0). The effectiveness of compaction drops (slightly) with thresholds (T=1 and T=10) since thresholds discard infrequent or noisy signatures. Therefore, the quality of signatures is higher and there is less similarity among signatures. Other benchmarks reflect this trend as well.

Figure 4.11 shows the percentage of false positives we reduce by applying the different optimizations progressively. A naive Bloom filter with no thresholds (T=0) suffers from a large percentage of false positives, especially at small sizes. The naive Bloom filter throttles unnecessarily over 90% of the time at a 4KB size, which explains the -29% gain we show in Figure 4.6. Thresholds help reduce false positive throttles.

While thresholds reduce a naive Bloom filter's false positives by identifying only the hot signatures to store, the associated rollback cost of thresholding can be high. Alternatively, we can reduce the percentage of false positives by 60% by using a CAM

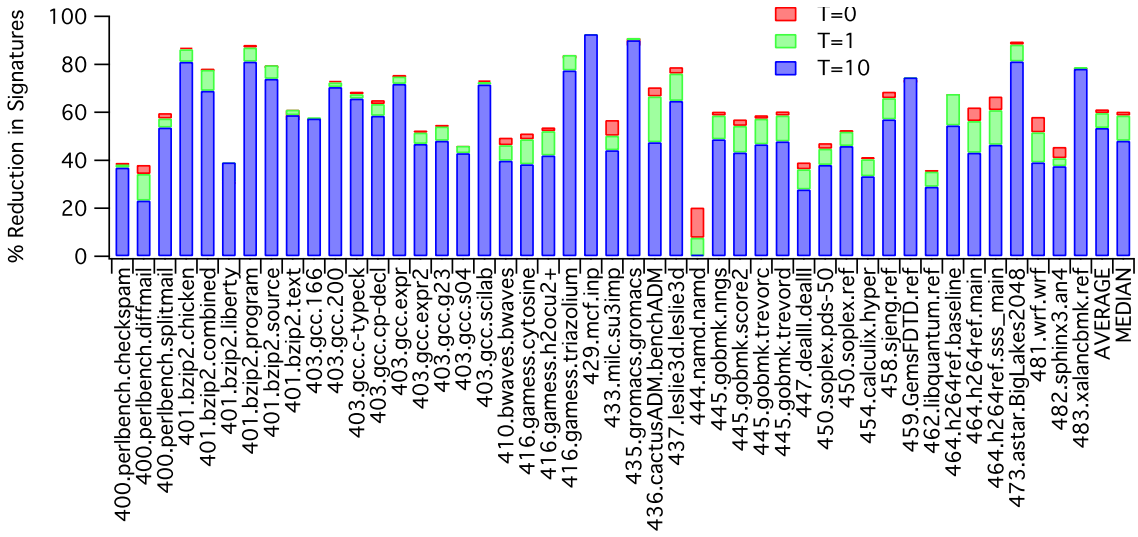


Figure 4.10: The signature compaction optimization folds similar signatures into one more representative signature. On average, the number of signatures drops by over 61%.

to avoid unnecessary lookups (compare T=0 in Bloom filter to T=0 in Bloom filter + CAM at the 4KB size). Applying thresholds lowers false positive throttling even further. Lastly, combining these schemes with signature compaction negates nearly all false positive throttles, as the number drops to just 2.3%.

Performance Evaluation with Optimizations. We now analyze the net effects of the optimizations discussed previously. We use a prediction table combining a 128-entry CAM (512 bytes) with the naive Bloom filter, and we refer to this implementation as the Bloom filter + CAM. Briefly, we find that using a 8KB table with the proposed optimizations enables 11.1% gain in performance, as compared to the 12.3% gain for the unbounded predictor we covered in the previous section. We do not evaluate the naive Bloom filter with thresholds because the structure has a high number of false positives regardless of the threshold value even at reasonable sizes like 4KB and 8KB. The left-most cluster of bars in Figure 4.11 confirm this statement.

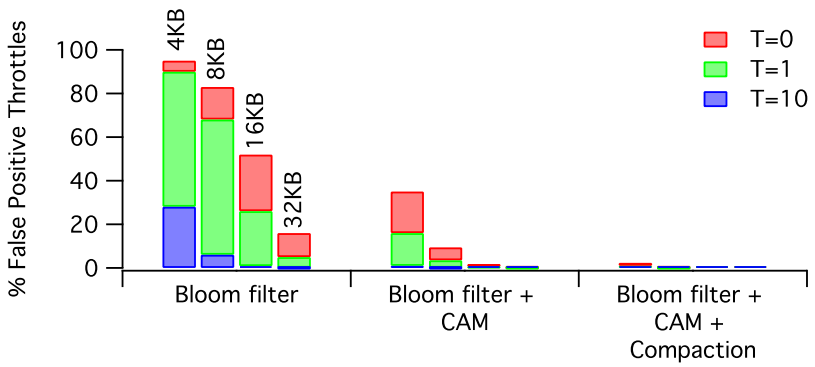


Figure 4.11: Event with thresholds, a plain Bloom filter of reasonable size gives many false positives, but this number decreases significantly as we restrict Bloom filter lookups using a CAM (Bloom filter + CAM). False positive throttles drop even further when we combine this latter structure with signature compaction (Bloom filter + CAM + Compaction).

Figure 4.12 shows the performance of the Bloom filter + CAM without thresholds (T=0) and with them (T=1 and T=10). Even when we do not apply thresholds, the Bloom filter + CAM consistently performs better than the naive Bloom filter. This shows the value of the CAM-based screening structure—lookups cause large false positives penalties when the Bloom filter set is heavily populated. For instance, at a 4KB size, we observe an improvement of 19 percentage points at T=0.

Thresholds are important at small table sizes. A higher threshold yields better performance at small table sizes because it reduces the false positive rate. With a 4KB prediction table size, performance loss is ∼10% without a threshold (T=0). But a threshold of T=10 reduces false positive throttling so much that performance gain increases to 7.3% despite rollback penalties from applying a threshold. Thresholds become less important as the table size grows to 16KB and 32KB because the Bloom filter's population count goes down, and so does its false positive rate.

We are able to improve performance gains of a Bloom filter + CAM even further by

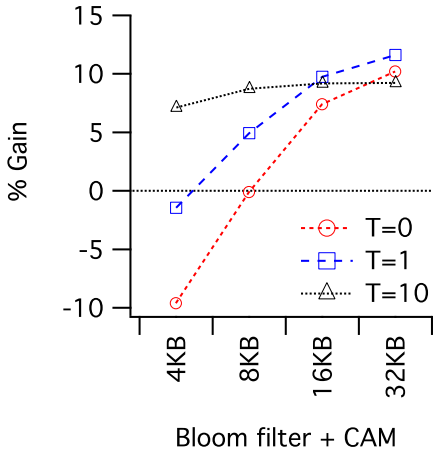


Figure 4.12: Performance gains using a Bloom filter whose lookups are initially screened using a CAM. T is the threshold we apply.

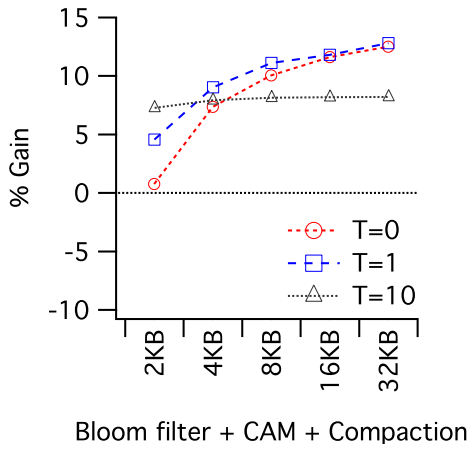


Figure 4.13: Performance gains using only compacted signatures in the Bloom filter + CAM structure.

relying on compacting similar signatures together (see Figure 4.13). Large improvements for T=0 are seen in Figure 4.13 as compared to Figure 4.12, since compaction reduces the number of signatures by over 61%. The best reductions are at smaller Bloom filter sizes, since the false positive rate is much higher for these smaller structures. In fact, a 2KB Bloom filter + CAM + Compaction outperforms a 4KB Bloom filter + CAM.

Thresholds are less effective at reducing overall cost when signatures are compacted. Nevertheless, the benefits of thresholding can be significant when a Bloom filter + CAM is smaller than 2KB in size. They are also effective when there are a large number of infrequently recurring signatures that pollute the Bloom filter.

As predictor table size grows, the false positive rate drops so that lower thresholds are more attractive. For an 8KB prediction table, performance gain for a threshold of T=10 is 3 percentage points less than that for a threshold of T=1, because false

positives are reduced so much that rollback penalties dominate. With T=1 (which simply excludes all non-recurring emergency signatures), the performance gain for an 8KB table is 11.1%, as compared to the 13.5% gain for the unbounded prediction table.

To this end, we have demonstrated it is possible to use recurring voltage emergency signatures to prevent emergencies with minimal performance loss. We achieve performance improvements despite tolerating emergencies using coarse-grained checkpoint-recovery to identify recurring signatures. Checkpoint-recovery when used by itself results in negative performance improvement. Moreover, our performance is very comparable to an oracle-based predictor.

Chapter 5

Eliminating Emergencies via Hardware and Software Co-design

Contents

Hardware solutions deal with voltage emergencies ineffectively. Hardware repeatedly, and consequently inefficiently, throttles and rolls back execution to mitigate emergencies. Even an ideal voltage emergency predictor that never mispredicts must pay recurring penalties, throttling once per impending voltage emergency.

However, voltage emergency signatures demonstrate that voltage fluctuations within a processor are not random, as previously thought. Rather, it is recurring program and corresponding microarchitectural activity combined with the interactions of the underlying power delivery subsystem that repeatedly give rise to emergencies. We can leverage this information to build better solutions that mitigate voltage emergencies much more effectively at the software layer. As technology innovation continues and software layers like virtualization become an integral part of the hardware platform, voltage emergency signatures can enable software-layer solutions that go beyond just repeatedly avoiding emergencies, software can *eliminate* emergencies altogether.

This chapter presents a collaborative architecture between hardware and software to mitigate emergencies. The resulting architecture is much more efficient than pure hardware-based solutions in the presence of emergencies. By infrequently tolerating (and/or even avoiding) emergencies and by ultimately eliminating emergencies through algorithmic code changes at a higher level, software enables more fluid execution at the hardware layer, due to fewer rollbacks and throttles.

In this chapter, we present an instantiation of hardware and software co-design to reduce voltage emergencies. Our collaborative approach relies on the general-purpose fail-safe mechanism as infrequently as possible to handle emergencies, while a software layer dynamically smoothes out bursty machine activity to eliminate emergencies. We

discuss this co-design architecture bottom-up, starting with a compiler scheme that targets single cores systems. Subsequently, we expand our discussion to multiple cores per chip, dealing with voltage noise at the operating system level. This thread scheduling solution builds on top of the architecture necessary to support uncore-level compiler code transformations. Therefore, we cover these topics in that order.

We cover the design and implementation of a dynamic compiler-based system for suppressing recurring emergencies. 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.

For multi-core systems, we discuss co-scheduling of threads to mitigate voltage emergencies. Typically, scheduling efforts focus on improving performance. We show that scheduling for performance, independently of voltage emergencies in a noise-tolerant architecture such as ours, can in fact lead to more emergencies. System performance will actually degrade due to underlying rollback penalties.

In an effort to present our software solutions elegantly, we evaluate the compiler and thread scheduler solutions as separate isolated techniques. We do not assess the combined benefits, although such a scheme is feasible. Similarly, we limit our fail-safe hardware mechanism to checkpoint-recovery, not assessing checkpoint-recovery in combination with the voltage emergency predictor. We stay true to our goals, identifying new co-design solutions and demonstrating the improvements they enable even in the presence of voltage emergencies at very aggressive voltage margin settings.

5.1 From Emergencies to Error-prone Code

Hardware techniques do not exploit the effect of program structure on voltage emergencies. They work best for intermittent voltage emergencies. For instance, consider a loop incurring repeated voltage emergencies. Although an efficient hardware mechanism, such as our voltage emergency predictor, can avoid a recurring emergency, it must repeatedly throttle. Therefore, hardware mechanisms waste precious processor cycles even when they operate near perfection.

Leaving recurring emergency activity to software is better. Software can instead eliminate that recurring penalty altogether by restructuring the loop through permanent code transformations. In this section, we demonstrate both illustratively and quantitatively the role that software can play in mitigating emergencies. We use specific examples that motivate our fine-grained instruction scheduling solution, as well as the more coarser-grained operating system level scheduling of threads to reduce voltage noise.

5.1.1 Problematic Loops

Joseph *et al.* [34] show that program code can include successive periods of high and low current profiles that lead to emergencies. While their synthetic hand-crafted microbenchmark contains only a single loop body, it consistently causes voltage emergencies during execution. Real programs have several loops. Programs spend majority of their execution time running through code within these loops. Therefore, if emergency-prone loops exist in real applications, then it is logical to apply a permanent solution at the code level. By doing so, we limit the recurring performance

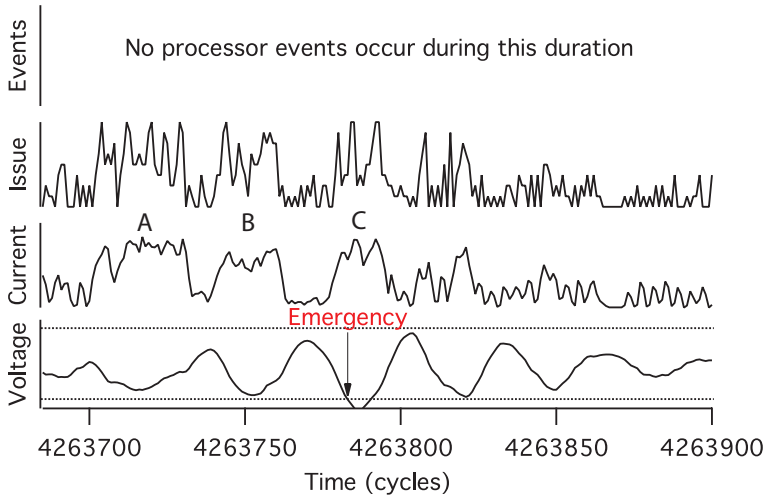


Figure 5.1: Voltage swing grows progressively larger because of pulsing current activity (see markers A, B and C). As that activity subsides, the voltage swing reduces.

penalty of activating control hardware to either tolerate or avoid emergencies.

We find that real programs have emergency-prone hotspots embedded within frequently executing loops. One such program is benchmark *436.cactusADM* from SPEC CPU2006. Figure 5.1 illustrates a voltage emergency that recurs every several thousands of processor clock cycles. Prior to the emergency, we find that there are no emergency-causing microarchitectural events in the Events subgraph embedded within Figure 5.1. This confirms that only the executing sequence of instructions is responsible for the emergency.

Additionally, we find that the emergency-prone activity in Figure 5.1 is recurring. In-depth inspection reveals that this part of the execution takes place within a loop. This finding further relates Joseph *et al.'s* microbenchmark to real world examples, demonstrating that emergency-prone code can reside in program loops.

Figure 5.2 is a control flow graph (CFG) representation of the loop responsible for

this emergency. Our Pin Tool [3] that generates this CFG reveals loopback edge paths in Figure 5.2 using dotted lines. There are two dotted edges between bbl:42 and bbl:47, revealing loop nesting. The loop body consists of basic block bbl:42. Power-hungry floating point instructions dominate the basic block instruction sequence. Execution dependencies in between the power-hungry instruction sequences cause the processor to intermittently stall. Current activity drops as the processor waits for operand values to become available during stalls. Once these values become available, execute resumes and current surges for few tens of cycles. This fluctuating activity causes current oscillations around the resonant frequency of the power delivery subsystem between cycle times 4263700 and 4263850. Such current pulse trains around the resonant frequency cause voltage to swing in progressively increasing magnitudes. Markers A, B and C in Figure 5.1 illustrate this behavior. As current fluctuation subsides, voltage swings also subside. In our investigation of such activity across other benchmarks in the CPU2006 harness, we find similar instances, indicating that this is not unique to the benchmark under investigation in Figure 5.1.

A compiler that reschedules such problematic instruction sequences can eliminate this recurring behavior altogether. The loop could be restructured to prevent all future occurrences of the emergency. Using hardware assistance, the compiler could identify the code corresponding to this emergency-prone region of execution, and target its transformation accordingly.

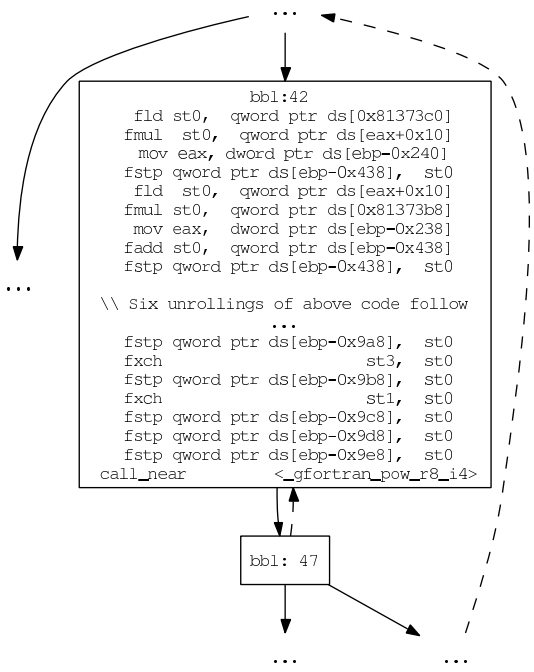


Figure 5.2: This snippet of high power floating point instruction execution mix experiences frequent execution stalls on operand values at around the resonant frequency. These stalls are responsible for current swings that lead to voltage noise in Figure 5.1.

5.1.2 Emergency Hotspots

Only a few code locations within loops are responsible for nearly all emergencies. By identifying these hotspot code locations and the paths along which emergencies occur, a compiler can target custom and specific code transformations only along those executions paths. By doing so, it can preserve original program behavior and performance as much as possible.

Using the event categorization algorithm described by Gupta *et al.* [28] we identify the instruction responsible for an emergency, demonstrating in Figure 5.3 that only a small number of unique static program addresses are responsible for all emergencies. The stacked plot shows the number of unique static program addresses responsible for

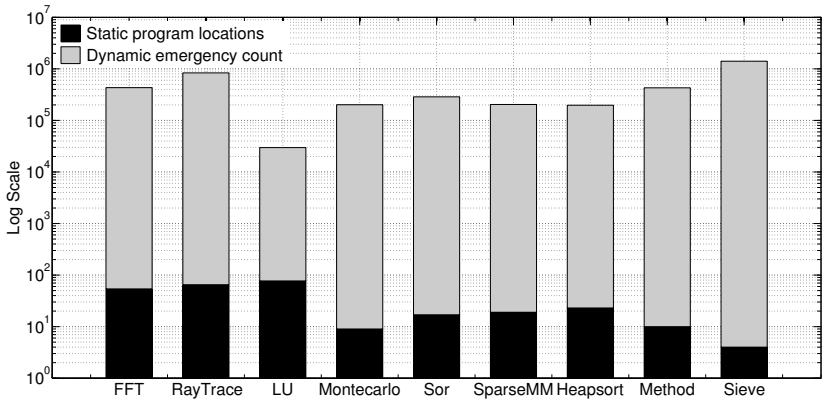


Figure 5.3: A small set of static program addresses (fewer than 100) are responsible for the large number of voltage emergencies. We assume a 4% operating margin, but this trend remains across different margins.

emergencies and the total number of emergencies they contribute over the lifetime of a program. The log-scale distribution indicates that on average fewer than 100 static program addresses are responsible for several hundreds of thousands of emergencies.

Hardware, because of its limited view of program execution and structure, cannot exploit the fact that there are so few program locations responsible for all emergencies. Even an ideal oracle-based hardware technique will activate its fail-safe mechanism once per emergency, paying recurring penalties. Additionally, hardware must ensure that performance gains from operating at a reduced margin outweigh the fail-safe penalties. When combined with implementation costs, potential changes to traditional architectural structures, and challenges like response-time delays [30], design, testing, validation and wide-scale retargetability all become increasingly difficult.

Unlike hardware schemes, a software-based solution does not require design-time package- and microarchitecture-specific solutions. Dynamic software systems can adapt their solution to their run-time environment. Moreover, since there are just

a few static hotspots, dynamic software assistance is necessary only intermittently during execution, assuming software is able to eliminate the emergency. Therefore, the overheads of software invocation are amortizable.

5.1.3 Inter-thread Interference

In multi-core systems, we find that microarchitectural behavior across two independent cores can interfere with one another, leading to voltage emergencies. Therefore, carefully scheduling noise-compatible threads together can reduce voltage emergencies

Voltage noise interference in multi-core systems is not yet understood in literature, especially from a detailed microarchitectural perspective. Therefore, in Figure 5.4 we present our initial findings from measurement. Our Intel Core^{TM}2 Duo chip consists of two cores in one chip. We stimulate each core with a specific microarchitectural event. We then capture the magnitude of voltage swing across the entire chip, since both cores are affected by the same power supply source. The figure's legend shows the magnitude of the swing relative to an idling machine; the colors correspond to the intensity of the swing. The y-axis corresponds to Core 0 and the x-axis corresponds to Core 1.

We introduced microbenchmarking details for the events in Figure 5.4 before in Table 3.1. But briefly, L1 and L2 events correspond to Level 1 and Level 2 cache miss events, respectively. TLB corresponds to translation lookaside buffer flushes. BR indicates a pipeline flush due to branch misprediction.

From observing the intensity of the colors in the heatmap, we conclude that the

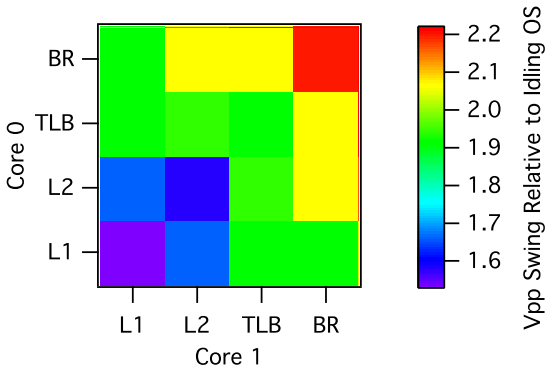


Figure 5.4: Microarchitectural event activity across two separate cores connected to the same power plane leads to voltage swings. The magnitude of voltage swing varies depending on which two events are happening together.

maximum voltage swing is when both Core 0 and Core 1 experience pipeline flushes due to branch mispredictions. Consequently, scheduling a branch heavy program onto one core, while avoiding a similar noise-characteristic schedule on the adjacent core, reduces the magnitude of the voltage swing. For instance, co-scheduling the BR program with a L2 heavy or a TLB intensive program reduces peak swing across the entire chip.

5.2 A Collaborative Architecture

Software elimination of voltage emergencies requires collaboration between hardware and software. There are two benefits of such a collaborative noise-tolerant architecture: First, software code transformation avoids recurring emergencies. Second, a collaborative architecture allows hardware designers to relax worst-case timing margin requirements because of the reduced number of emergencies. The net effect is better performance. We present an overview of how our collaborative architecture works and highlight the critical components. We present details about each of the

hardware and software components.

Our software component includes two pieces: a compiler and an operating system-level thread scheduler. For clarity, we spend more time discussing the needs of our more fine-grained compiler approach, as it requires specific attention to details. But we expand our discussion to incorporate the thread scheduler towards the end of Section 5.2.3. The thread scheduler builds on top of the same underlying architecture needed for the compiler solution.

Figure 5.5 illustrates the operational flow of our system in the context of a single core. 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 Run-time System software layer of the voltage emergency, passing along relevant information.

Whenever software receives a notification, the run-time system extracts the information about recent processor activity from the Event History Register. This register maintains the voltage emergency signature pertaining to the current emergency. The run-time system then uses this information to identify the code region corresponding to an emergency. Subsequently, the run-time system calls a dynamic Compiler to alter the code responsible for the emergency in an attempt to eliminate future emergencies at the same program location.

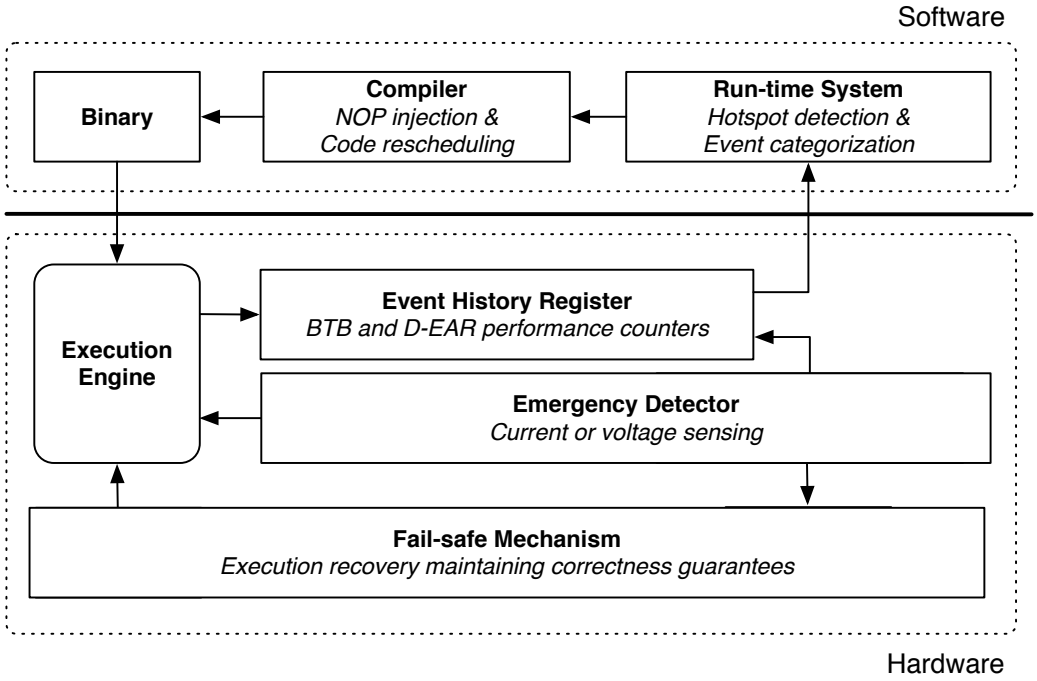


Figure 5.5: Workflow diagram of the proposed software-assisted hardware-guaranteed architecture to deal with voltage emergencies. For clarity of thought, we limit our discussion to the compiler scheme only in this illustration.

5.2.1 Emergency Tolerance

Tolerating or allowing voltage emergencies to occur is useful. Allowing emergencies enables us to identify emergency-prone code regions for software transformation. Therefore, we need to detect operating margin violations, and for that we rely on a voltage sensor.

However, since we allow emergencies to occur, we require a mechanism to recover from corrupt processor state. The detector invokes the fail-safe mechanism upon detecting an emergency. We rely on the checkpoint-recovery mechanism discussed in Chapter 3 to tolerate emergencies. It is similar to existing implementations found in reactive techniques for processor error detection and correction, previously proposed

for soft error recovery [54, 7]. These are primarily based on checkpoint and rollback. The form of checkpoint-recovery we rely on is explicit checkpointing, which is already shipping in production today's systems [9, 48]. While we choose explicit checkpointing for evaluation in this work, our overall approach is independent of the specific checkpointing implementation.

Explicit-checkpoint mechanisms rely on explicitly saving the architectural state of the processor, i.e., the architectural registers and updated memory state. We want to rely on this mechanism infrequently, since 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).

We assume one recovery unit per power plane (or power supply source). So if there are multiple cores per chip that a power plane is supplying, then a single global recovery unit supports all cores. Regardless of which core is responsible for a voltage emergency, all cores tied to the common source are affected by a voltage droop. They all run the risk of incorrect execution due to circuit delays. Therefore, a common recovery unit must restore state across all cores tied to the same power plane in the event of a voltage emergency.

5.2.2 Hardware Feedback to Software

Enabling a software solution requires that the underlying hardware provide pertinent information to software. Otherwise, code transformation efforts cannot be made intelligently. Information pertaining to an emergency is available in its corresponding

voltage emergency signature(s). The software extracts this information whenever it receives a notification about an emergency.

The event history register captures emergency signatures and it is very similar to structures already found in existing architectures. The history register tracks an interleaving of microarchitectural event activity (like cache misses, branch mispredictions etc.) along with program control flow path information. It uses issued branch instructions as control flow path indicators. Production systems today provide similar logic as the event history register. For instance, a branch trace buffer (BTB) maintains information about the most recent branch instructions, their predictions, and their resolved targets. A data event address register (D-EAR) tracks recent memory events like cache activity and translation lookaside buffer (TLB) misses.

Additionally, we require a firmware component in our hardware. The firmware, in addition to managing signatures and keeping track of their occurrences (see Chapter 4), effectively acts as a hardware performance counter, tracking the number of emergencies that are occurring. This helps the firmware decide when to invoke software. Tolerating emergencies within some limit allows the system to amortize the overhead of invoking the run-time software layer.

5.2.3 Software Layer

The software component consists of a run-time system, a compiler and an operating system thread scheduler. For clarity, here we limit our discussion to the compiler. Towards the end of this section we expand our discussion to include the thread scheduler, its resources requirements are a subset of the compiler's requirements.

The voltage emergency detector only communicates with the run-time system, which converts the information gathered by the hardware event history register into a particular location in the code. It then invokes the compiler to analyze that information and modify the corresponding program or thread to prevent future recurrences.

The run-time system component records the time and frequency of emergency occurrences in addition to recent microarchitectural event activity extracted from the performance counters. Using this information the run-time system locates the instruction responsible for an emergency using an *event categorization* algorithm [28]. 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 5.6(a) illustrates such a scenario. 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.

There are several other causes of voltage emergencies, ranging from cache misses to branch mispredictions and TLB misses. The run-time system is equipped to detect the root-cause for all types of emergencies. Figure 5.7 shows the distribution of root-

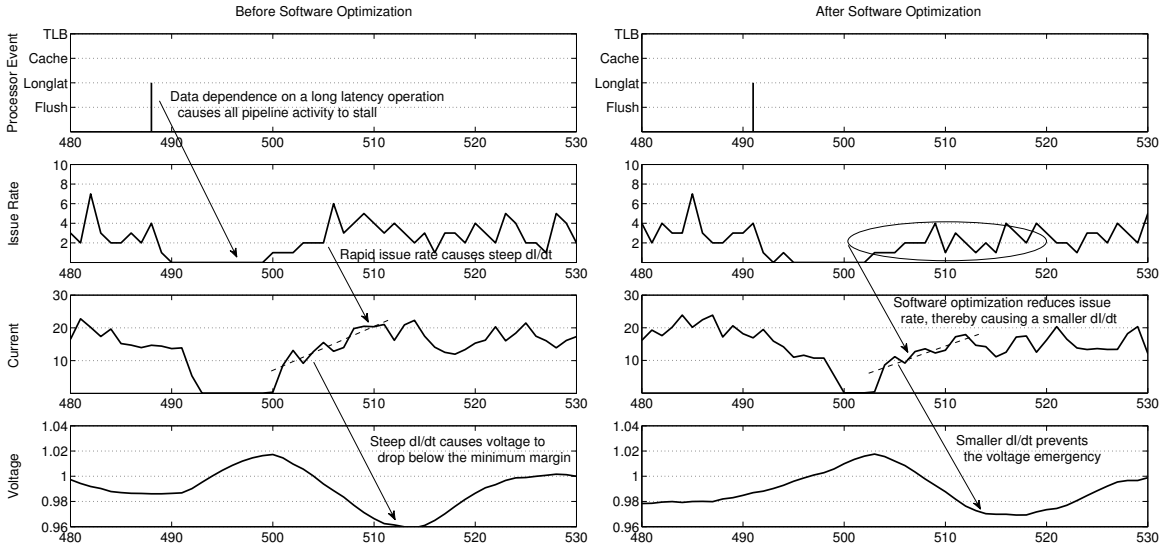


Figure 5.6: A 50-cycle execution snapshot of benchmark *Sieve* showing the impact of a pipeline stall due to data dependency. An operating margin of 4% is assumed (i.e., a maximum of 1.04V and minimum of 0.96V). (a) Before Software Optimization shows how a stall triggers an emergency as the issue rate ramps up quickly once the long-latency operation completes. (b) After Software Optimization demonstrates how compiler-assisted code rescheduling slows the issue rate to eliminate the emergency illustrated in (a).

causes across the benchmarks in order to characterize the activity leading to voltage emergencies in our benchmarks.

A 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 observable microarchitectural event. This likely resulted from code-based bursts of activity such as the "power virus" demonstrated by other researchers [34]. Finally, TLB Miss events did not tend to result in emergencies in our evaluated benchmark suite.

Thus far, we have discussed our software component within the context of our com-

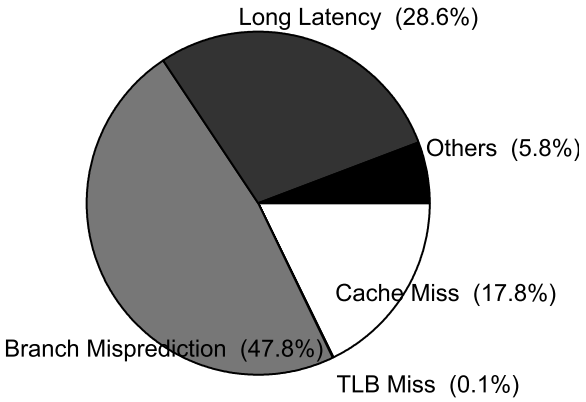


Figure 5.7: Aggregate distribution of root-causes across benchmarks in the Java Grande benchmark suite.

piler solution. However, differences exist between this technique and the operating system thread scheduler solution. Our thread scheduler does not require fine-grained root-cause identification, since it is not fixing instruction sequences. Rather, it relies on more coarse-grained mixing of instruction sequences across processor cores to smooth out voltage emergencies.

However, the thread scheduler relies on dynamic feedback from hardware. It requires the hardware to tell it how often emergencies are occurring. Using this information, the scheduler decides whether the currently scheduled set of running threads are interacting smoothly. Ideally, they are causing few or no voltage emergencies. Otherwise, it changes the set of scheduled threads at the next scheduling interval to dynamically obtain better performance.

5.3 Compiler Code Transformations

Figure 5.6(a) illustrates that voltage emergencies can 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.

Dynamic optimization systems [13] are well suited for scenarios where "90% of the execution time is spent in 10% of the code". In our scheme, a dynamic compiler eliminates a large fraction of recurring emergencies by applying transformations such as rescheduling existing code or injecting new code into the dynamic instruction stream of a program. Unlike 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.

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 5.3.3, we evaluate each approach in turn.

5.3.1 No Operation 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, as we later demonstrate.

5.3.2 Code Rescheduling

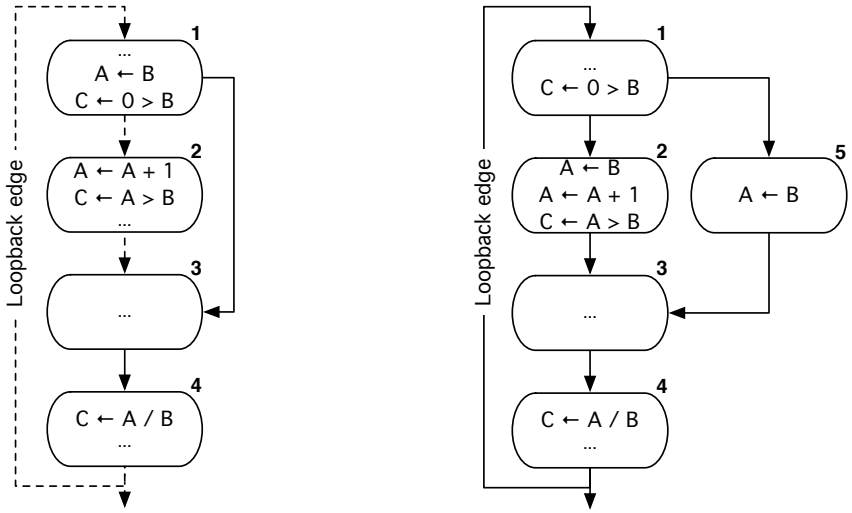
A better way to smooth processor activity is to exploit RAW dependencies already existing in the original control flow graph (CFG) of the program. This constrains the burst of activity when the machine resumes execution after the stall, which prevents the emergency. Whether the compiler can successfully move instructions to create a sequence of RAW dependencies depends on whether moving the code is possible given the program's control and data dependencies. In general, our approach to code rescheduling maintains data dependencies and works around control dependencies by cloning instructions and then moving them around the control flow graph such that

the original program semantics are all still maintained.

To illustrate our code rescheduling approach, in Figure 5.8(a) we present a simplified sketch of the code corresponding to the activity shown in Figure 5.6(a). The long-latency operation illustrated in Figure 5.6 corresponds to the *divide* instruction shown in basic block 4 of Figure 5.8. 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 result after rescheduling is illustrated in Figure 5.6(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 5.6(b) does not spike as high as it does in Figure 5.6(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



(a) Before. (b) After.

Figure 5.8: 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.

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 schedul-

ing window will fill with dependent code, reducing the issue rate. Otherwise, the compiler must generate multiple sets of smaller RAW dependence chains to fill up the scheduling window. This requires the compiler to move around more code, but the more code the compiler transforms, the higher the chances that it will render an optimization ineffective, since we cannot statically predict the interactions amongst the newly created set of dependence chains.

In the following paragraphs, 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.

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. For instance, when an emergency is caused by a branch misprediction, the scheduler takes 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 following locations:

1. **The root-cause instruction**: the instruction identified by the hardware as the cause of the voltage emergency.

2. **The last write-back instruction**: the most recent instruction in the write back stage of the pipeline.

3. **The wrong-path instruction**: the first instruction along the speculative path

that is executed prior to detecting a branch misprediction.

In the simpler case, such as an emergency caused by a sudden burst of activity following a cache miss or a long latency stall (as illustrated in Figure 5.6), the schedule only targets the root-cause instruction and the last writeback instruction to successfully remove emergencies. 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 by discovering other instruction sequences available for execution. These other instruction 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.

Algorithm 1 illustrates the entry point/function to transform the code corresponding to an emergency hotspot (i.e., root-cause instruction r). It takes as input the three input instructions described above that the run-time system mechanism (illustrated in Figure 5.5) 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

Algorithm 1: Entry routine that reschedules instructions to eliminate a voltage emergency hotspot

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*
$a \in Succ(r) | a \neq w$;

Scheduler(a) ;

Scheduler(w) ;

end

otherwise
$a \in Succ(r)$;

Scheduler(a) ;

end

end

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 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.

Function `Scheduler(`a`)`

Input: Instruction a

l = GlobalCandidate(a) ;

if $length(i) > 0$ **then**
    MarkScheduled(i) ;

    GCSMove(i, a) ;

end

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 [8]. 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 not hold, instructions must be cloned and

inserted in positions found by the *anticipated expressions* computed using data-flow analysis [8].

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 must not increase the dynamic instruction count of the program, since aggressive cloning can potentially degrade performance.

If these conditions are not satisfied, the LocalCandidate function returns a null list of instructions, forcing GlobalCandidate to enlarge the scope and retry. When these constraints are relaxed in an attempt to improve the chances of finding a suitable RAW dependence chain, 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

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 ;

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 5.9a and Figure 5.9c, respectively. Instruction 4 is the root-cause related to a branch misprediction. Instruction 8 corresponds to the wrong path instruction, or the first instruction executed along the incorrectly speculated path. In order to smooth the voltage emergency at the root-cause, the scheduler

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**
$j = a$;

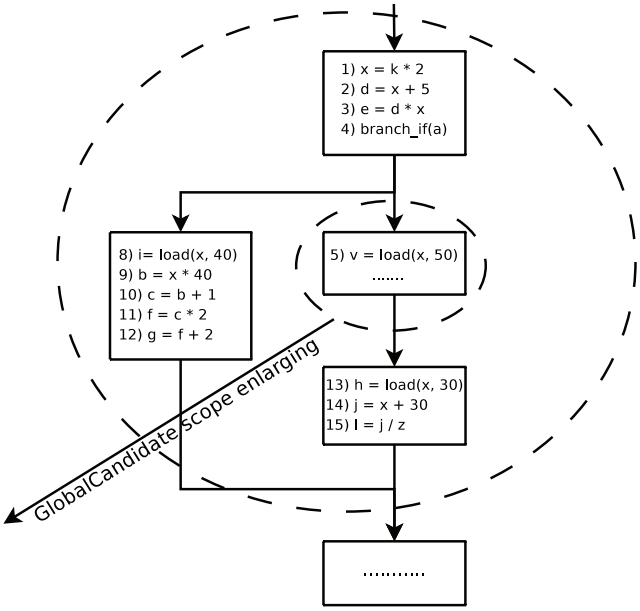
end

end

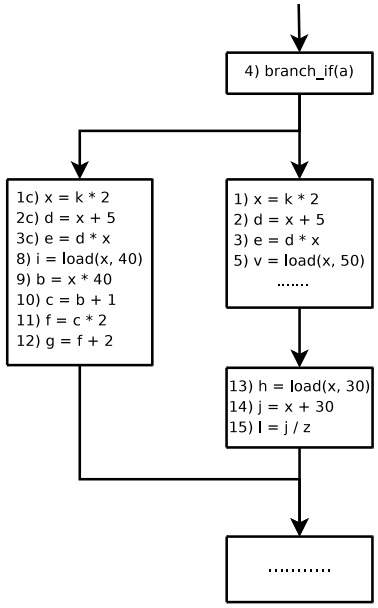
return $LongestRAWDependenceChain(j)$;

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.

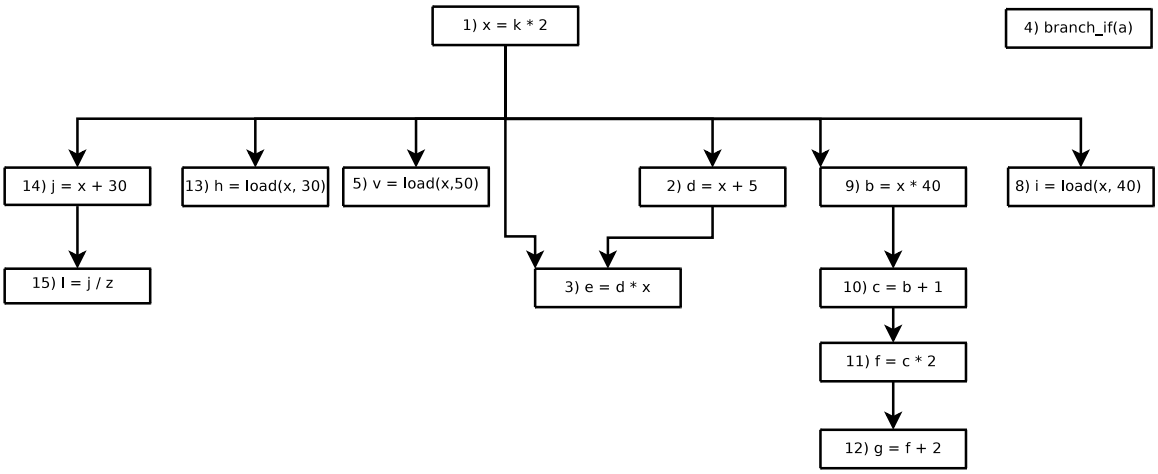
The algorithm starts by looking for the best RAW chain by calling the GlobalCandidate function, giving instruction 5 as its input. The GlobalCandidate function calls LocalCandidate to find the longest RAW chain inside the present scope of inter-



(a) Before rescheduling.



(b) After rescheduling.



(c)

Figure 5.9: (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.

est, 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 5.9a 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 5.9b, where we see that instructions 1, 3, and 5 have been replicated and migrated down the CFG.

5.3.3 Efficiency Comparison to Hardware-based Schemes

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 with minimal over-

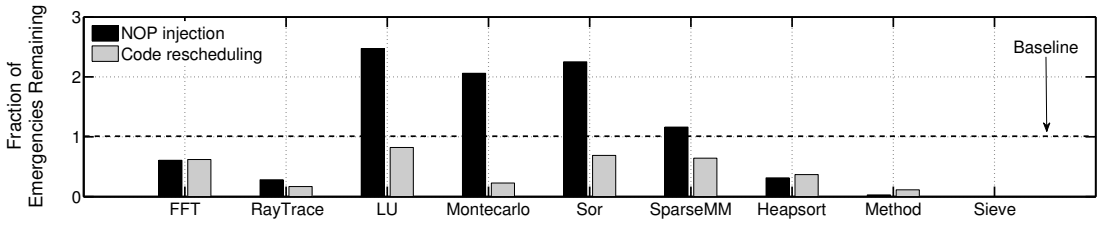
heads, we present a performance study showing that our software-assisted scheme overcomes the challenges of existing hardware techniques effectively.

Given that modern hardware does not support fine-grained access to voltage sensors, we explored our design using a hardware simulator together with our software compilation infrastructure. We used SimpleScalar/x86 to simulate a Pentium 4. We use the configuration setup shown in Table B.1. Please refer to that appendix for further details.

We use the C# benchmarks from the Java Grande benchmark suite [19]. Table B.3 gives a summary of the benchmarks. We invite the readers compare the traits of our benchmark setup to more traditional benchmarks like SPEC CPU2006 using the emergency distribution previously discussed in Section 5.2.3.

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 [29], we expect our results to generalize, and we feel that the results and contributions of our work outweigh this limitation of the experimental infrastructure.

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



(a)

Figure 5.10: Fraction of emergencies remaining after code transformation.

to evaluate full-system performance.

NOP injection. As described earlier, the NOP injection algorithm inserts new instructions in the program that simulate NOP instructions immediately following the root-cause instruction. The effectiveness of the transformation is shown by the left bar in Figure 5.10. 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*. In fact, the number of emergencies increases by over twofold for benchmark *LU*.

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 may 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 and the properties of the

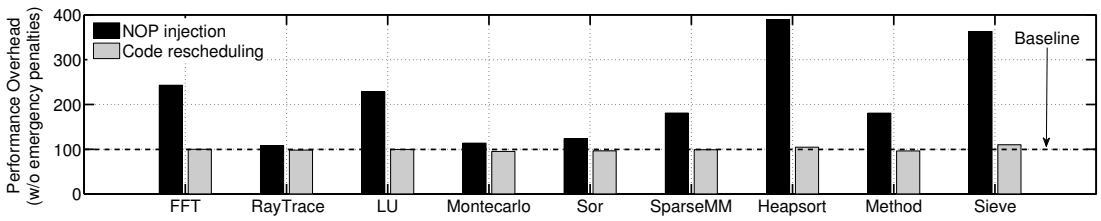


Figure 5.11: Code performance after transformation. The cost for handling emergencies is not shown in this plot to isolate the effect of code transformation on the run-time performance. We evaluate overall performance after factoring in code performance costs later on, along with penalties for handling emergencies.

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 for each benchmark in Figure 5.11 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 5.10). Second, it does so without dramatically increasing the execution time of a program (as shown in Figure 5.11). 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 5.3.2, we were able to effectively limit performance loss from injecting new instructions. However, there are side-effects. One of the rules states that cloning cannot increase the dynamic instruction count of a program. But in Table 5.1 we see that cloning leads to new instructions. These additional instructions come from the register allocation pass that takes place after the cloning pass in our compiler. The register allocator may generate necessary spill and fill compensation code to accommodate changes to the original code. Therefore, although cloning itself obeys the rules, there are side effects that lead to an increase in the instruction count.

Benchmark	# of Instructions		% Change in Dynamic Instructions
	Cloned	Moved	
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 5.1: Only a small fraction 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.

But even under such circumstances, the increase is small, in the order of tens of instructions. 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 the code transformation changes 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 ∼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 inadvertently lead to new emergencies root-causes as well. Figure 5.12 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.

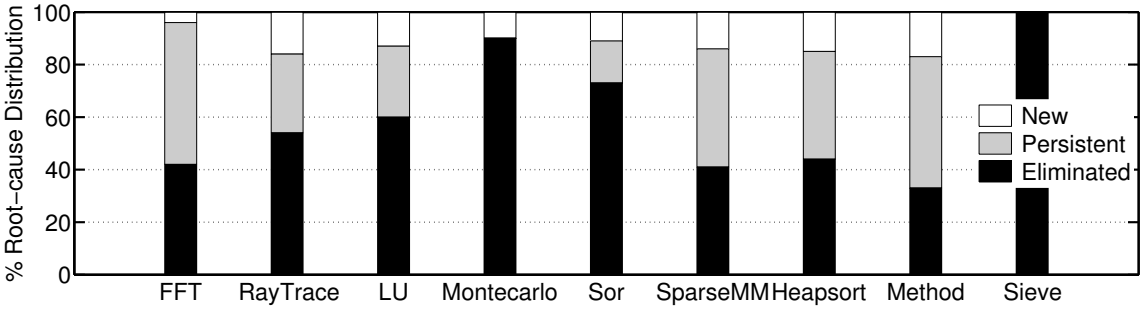


Figure 5.12: Not all emergencies can be eliminated. Some root-causes cannot be fixed because the compiler cannot find sufficient code to construct RAW dependence chains. Also, new emergencies can be introduced as a result of making transformations to existing code.

Ideally, the scheduling algorithm should attempt to create a RAW dependence chain long 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 5.13 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.

In Section 5.3.2, 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.

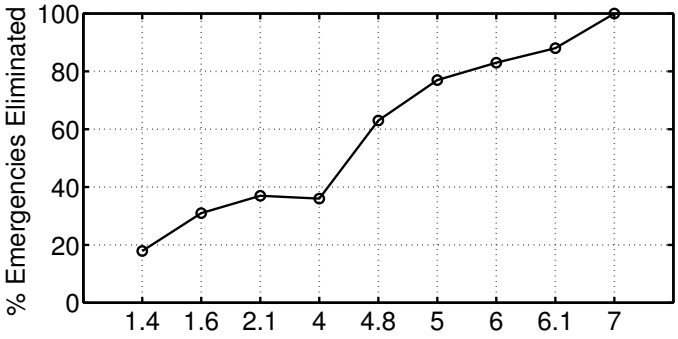


Figure 5.13: There is a correlation between the number of emergencies the compiler can eliminate and the average length of the dependence chains it creates. The compiler can eliminate more emergencies as it creates chain lengths that approach the machine's issue width. Our machine is 8-wide.

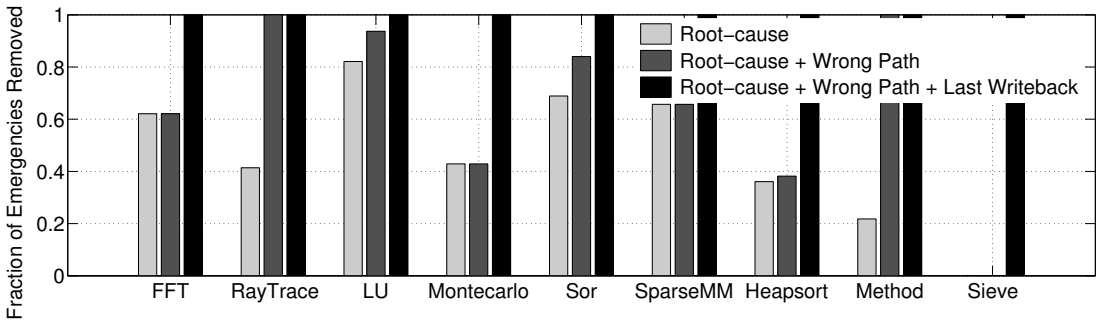


Figure 5.14: This figure justifies the use of three program points for resolving voltage emergencies. The combination of the root-cause instruction, the wrong path instruction, and the last writeback instruction, results in the ability to identify and resolve nearly all of the voltage emergencies encountered.

Assuming all three points are covered as the baseline, Figure 5.14 shows how effective the compiler is at removing emergencies as we reduce the number of points the scheduler targets. The graph is normalized to 1, indicating the utmost number of emergencies we are able eliminate using the code rescheduling algorithm implemented in the scheduler. This number corresponds to the Code rescheduling bar shown in Figure 5.10.

The left-most bar in Figure 5.14 shows the effect of targeting only the Root-cause

instruction. Since higher values mean fewer emergencies, 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 taken 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 of emergencies eliminated were the result of focusing on the Last writeback instruction.

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 5.2 shows the distribution of emergencies between the compiler and 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 in-

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 5.2: 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).

significant.

Based on these results, the overhead of run-time code transformation to fix and eliminate emergencies appears to be insignificant. Figure 5.3 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 5.3 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.

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 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

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

scheme. Results are presented in Table 5.4.

Bowman *et al.* show that removing a 10% operating voltage margin leads to a 15% improvement in clock frequency [17]. 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[1]. Based on the 1.5x scaling factor, the 4% operating voltage margin we assume 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 ∼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.

Fail-safe mechanism. An explicit-checkpointing scheme recovers from an emergency by rolling back execution. The explicit-checkpoint scheme suffers from the

[1]The worst voltage drop we observe for our power delivery package is 18% as we ran through the Java Grande benchmark suite.

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 5.4: 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.

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.

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 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 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. Combining explicit checkpointing with compiler assistance reduces checkpointing overhead substantially, from 25% to 7.6%. This translates to a net performance gain of ∼20%.

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 [26, 34, 44, 43]. 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 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 [31].

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 [9, 48] to reduce voltage emergencies. By doing so, we enable the processor to operate at very aggressive margins that translate to significant performance improvement.

Although our compiler solution is fine-grained, it is highly thread specific. In future systems, we require techniques that span multiple threads. Additionally, we also want our software to scale to multiple cores, dampening voltage swings that arise due to interactions across cores.

5.4 Operating System Thread Scheduling

As the proliferation of core count per die or chip continues, increasingly one core will either constructively or destructively interfere with other cores leading to more or less voltage noise, respectively. In such cases, a software solution bigger than a compiler becomes necessary. Virtual machine monitors or operating systems become appealing, since these systems see and control all threads executing on hardware. They can therefore decide (based on runtime feedback from hardware) if the running

set of threads are collaborative from a voltage noise perspective or not.

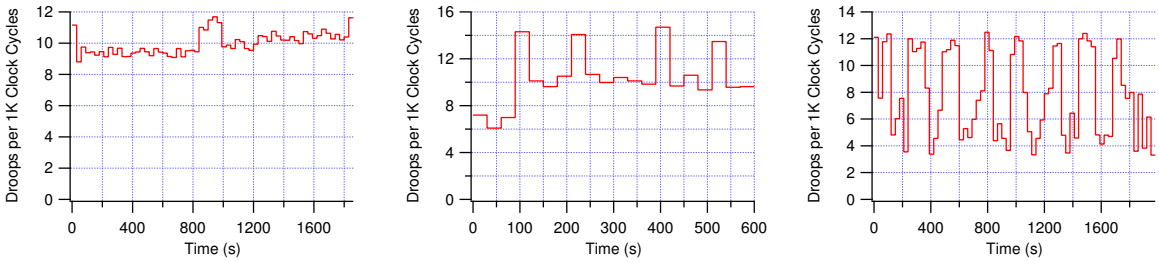
Ideally, scheduling threads for lower number of voltage emergencies leads to better performance due to fewer rollbacks. In order to schedule intelligently, we introduce the notion of voltage noise phases. Using these phases behavior changes, a thread scheduler can co-schedule threads to reduce the number of emergencies.

5.4.1 Voltage Noise Phases

Programs experience differing emergency activity over the course of their execution. While prior work exists showing that programs go through execution phase changes, no such behavior has thus far been identified in the context of voltage noise. Our experimentation reveals that programs also go through voltage noise phase changes. Emergency phases are periods of execution during which the average amount of voltage noise is significantly larger or lower than at other times.

To study voltage noise behavior, we sample, collect and plot core voltage measurements every 60 seconds in Figure 5.15. Our experimental setup limits us to this time granularity, and we can measure no finer. Therefore, it is possible that there are more finer-grained phases that we cannot observe using our setup. Nevertheless, this timescale still allows us to demonstrate that voltage noise phases exist.

Figure 5.15 is a plot that shows Droops per 1K Clock Cycles assuming a 4% voltage margin across three different SPEC CPU2006 programs. This metric is similar to the metric that designers typically use to study cache performance with respect to application behavior or its execution time—"Misses per 1000 instructions". We explicitly use the term droops instead of emergencies to draw distinction between



(a) *482.sphinx* (b) *416.gamess* (c) *465.tonto*

Figure 5.15: While some programs show no phases in voltage noise like benchmark *482.sphinx*, others like *416.gamess* and *465.tonto* experience simple and more complex phases, respectively.

measurements and interpretation. Our Intel Core™2 Duo processor is built robustly using a 14% voltage margin, so no real emergencies occur during execution. However, assuming the system is noise-tolerant, we would expect emergencies at a 4% voltage margin during execution. We infer the expected emergency count per 1K clock cycles using this droop metric.

The number of voltage noise phases and the number of voltage emergencies we observe varies from one program to another. See Figure 5.15. Benchmark *482.sphinx* experiences nearly no phase changes. The average number of "emergencies" is stable around 10 droops per 1K clock cycles. By contrast, benchmark *416.gamess* experiences four phase changes where voltage emergencies vary between 10 and 14 emergencies per 1K clock cycles transiently. Benchmark *465.tonto* goes through more complicated phase changes in Figure 5.15c, oscillating strongly and more frequently between 4 and 12 emergencies every 1K cycles.

In Section 3.1 we demonstrated that there is a relationship between voltage noise and processor stalling activity. We state that processor stalls are indicative of the number of emergencies a program experiences, showing a strong correlation of 97%.

Extrapolating from that same analysis, we believe it is underlying changes in microarchitectural activity, caused by program behavioral changes, that are causing these voltage noise phases.

Another interesting finding that Figure 5.15b and Figure 5.15c reveal is that not only do programs experience voltage noise phases, but that these phases are recurring. We previously claimed this through our simulation analysis in Section 3.2, but measurement results here validate our claim. Additionally, these phases are over the course of full program execution. Such recurring behavior is useful for amortizing the cost of making decisions at the software layer.

5.4.2 Phase Scheduling

Scheduling programs with different voltage noise phases or characteristics together on a multi-core system can lead to either an increase or decrease in emergencies. It is important to minimize emergencies in a multi-core system, since one power plane is shared across multiple cores and a droop anywhere on a common power plane forces recovery across all connected cores. Therefore, one misbehaving thread can penalize other running threads, severely degrading overall system performance.

We setup a sliding window experiment to evaluate the impact of co-scheduling different voltage noise phases together. Figure 5.16 illustrates this setup. It resembles convolving two execution windows together. But more precisely, in this experimental setup one program is tied to Core 0. This program called Prog X on Core 0, is run once until it completes execution. During the course of its execution, we spawn a second program onto Core 1 called Prog Y. However, we do not let this program run

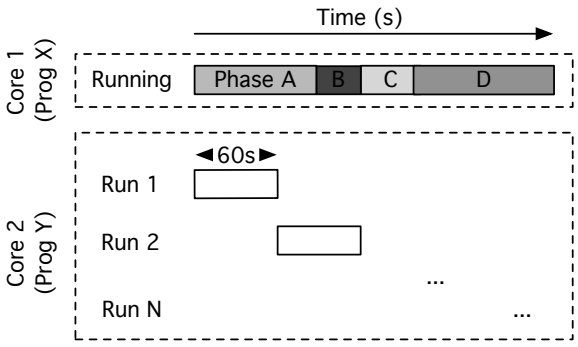
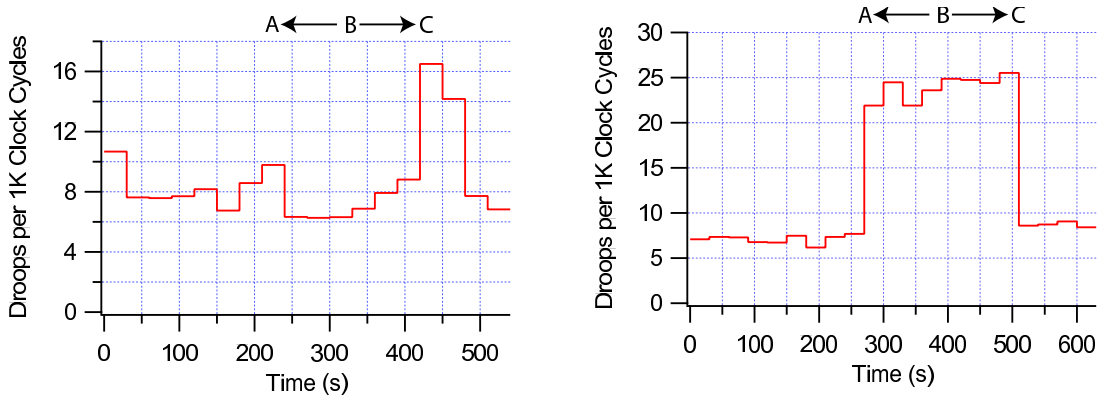


Figure 5.16: Experimental setup showing how we evaluate the impact of co-scheduling different phases together. We tether one program to Core 0. It runs to completion during the experiment. Then every 60 seconds we launch another program onto the second core. But we terminate this program after 60 seconds and repeat this with another instantiation. At the end of every run we collect our voltage measurements.

to completion. Instead, we prematurely terminate it after 60 seconds, launching it multiple times (Run 1, Run 2, ..., Run N in Figure 5.16). In this way, we capture the interaction of the first 60 seconds of program Prog Y as program Prog X is going through different voltage noise phases (A, B, C and D in Figure 5.16). To quantify the effects on voltage noise, we take measurements at the end of each Prog Y instance. We only have two cores on our system, so Prog X and Prog Y together maximize the running thread count, keeping all cores busy.

We demonstrate that co-scheduling impacts voltage emergency count in Figure 5.17 using benchmark *473.astar*. Running by itself benchmark *473.astar* does not exhibit much variance in emergencies other than towards the end of execution. The benchmark gets briefly noisier around marker C in Figure 5.17a. During this process the second core is idling.

However, we observe destructive interference leading to more emergencies as we convolve a 60-second execution window of one *473.astar* instance running on Core



(a) Single-core noise profile of benchmark *473.astar*. The second core is idling, allowing us to study the noise characteristics of the program in isolation.

(b) Noise profile of co-scheduled threads. Two instances of *473.astar* are running together, as per the experimental setup illustrated in Figure 5.16.

Figure 5.17: Voltage noise profiles with and without co-scheduling of benchmark *473.astar*.

0 with another *473.astar* instance running on Core 1. See Figure 5.17b. When *473.astar* is running by itself and it enters the execution region around marker B (in Figure 5.17a), the benchmark goes through microarchitectural activity changes. When this region is convolved with the first 60 seconds of another instance of the same program, emergencies quadruple. Emergency counts goes up from around 8 droops per 1K cycles to 24 droops per 1K cycles during co-scheduling. See Figure 5.17b.

While we noted destructive interference, it is also important to note the presence of constructive smoothing of voltage noise. Between the start of execution and marker A, the number of emergencies is the same across both graphs. Compare the y-axis values around region B across Figure 5.17a and Figure 5.17b, which represent single core and multi-core emergency activity respectively. Despite both cores running actively, the data indicates that the threads are not interfering and causing more emergencies.

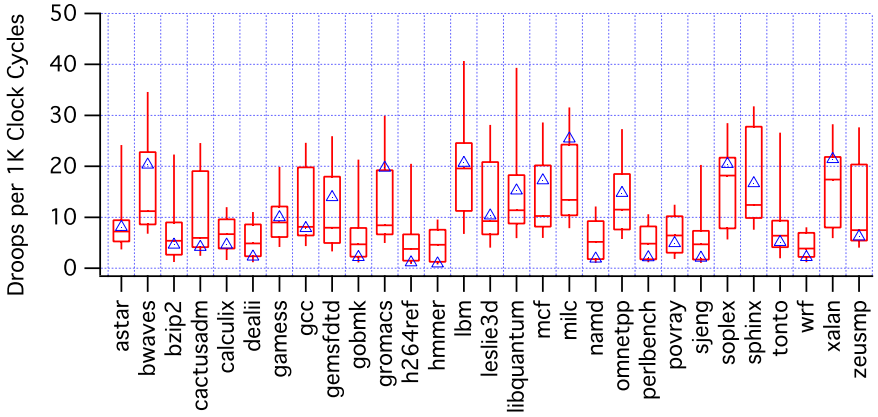


Figure 5.18: Boxplot showing the variance in emergencies (or droops) as each program on x-axis is co-scheduled with every other program shown on the same axis.

We expanded our analysis to the entire SPEC CPU2006 benchmark suite, finding that the same constructive and destructive interference behavior exists over other schedules as well. We present this analysis using a box plot in Figure 5.18. The figure illustrates the range of emergencies we observe as each program is co-scheduled with every other program on the x-axis. The blue triangles in the figure correspond to emergency counts as the benchmark is running with itself. These blue triangle are a useful baseline for comparing schedules.

In some cases co-scheduling a program with itself results in the lowest number of emergencies. We find that such is the case with benchmarks like *cactusADM*, *h264ref*, *hmmer*, *namd*, *wrf* and *zeusmp*. However, scheduling these programs with any other program causes emergencies to increase. In the extreme cases benchmarks *cactusADM* and *zeusmp* experience a 2.5x increase in emergencies.

5.4.3 Scheduling for Noise versus Performance

Determining which set of programs to co-schedule, or run together, on a multi-core system for improving system performance is a well studied topic [50, 25, 38, 37, 56, 23, 22]. Prior scheduling for performance work shows that it is possible to improve the aggregate throughput and performance of all running workloads by reducing stalls (like those due to cache misses). This is done by exploiting different program and microarchitectural execution characteristics.

A similar extension is possible allowing the operating system thread scheduler to pair threads together to reduce the number of emergencies. Voltage swings occur primarily because of fluctuations in activity due to stalls (see Section 3.1). Although scheduling for performance includes eliminating stalls, that same metric does not necessarily guarantee fewer emergencies. The operating system must explicitly schedule for voltage noise.

Co-scheduling threads to reduce voltage emergencies differs from scheduling for performance. In order to prove this point we evaluate different operating system scheduling policies, measuring emergencies over the course of a batch job schedule consisting of 50 jobs. The job pool consists of randomly chosen SPEC CPU2006 benchmarks. Some programs may be repeatedly selected to construct the job pool, since there are only 29 CPU2006 programs. For this selected set of programs, we evaluate a range of scheduling policies. We investigate random selection (Random) and target maximum performance (IPC). We also schedule for minimal emergencies (Droops).

We simulate the scheduling policies, rather than evaluate them in real hardware.

Existing hardware does not support dynamic feedback to software regarding voltage noise. For instance, there is no way to study the effects of a scheduler that dynamically selects the pool of threads to co-schedule based on run-time voltage emergency feedback from hardware. As a consequence, rather than implement the different policies in a real operating system scheduler, we model the behavior of different scheduling policies.

Due to the lack of feedback from hardware, scheduling for any metric involving Droops requires a priori knowledge of emergency activity. Therefore, we do a pre-run to gather all the data, and subsequently use that data to actually model the policies. Miscellaneous modeling details are available in Section B.2.3.

In order to identify the difference between scheduling for voltage noise versus performance, in Figure 5.19 we plot performance in terms of instructions per cycle (IPC) versus droops we observe over the course of our batch schedule. We measure IPC using VTune [1]. Both the y- and x-axis of the graph are normalized to SPECrate, which acts as a baseline. SPECrate assumes two instances of the same program are running together at the same time. We do this to get rid of inherent IPC differences between the benchmarks, allowing us to focus in on only the effects of co-scheduling. Each marker in the graph corresponds to one simulation. We ran a 100 random simulations.

The four quadrants in Figure 5.19 (Q1 through Q4) help us draw different conclusions. Ideally, we want results in quadrant Q1, which indicates that the scheduling policy lowers emergencies, in addition to improving performance. Quadrant Q2 is good, but only from a performance standpoint. Q2 suffers from an increase in emer-

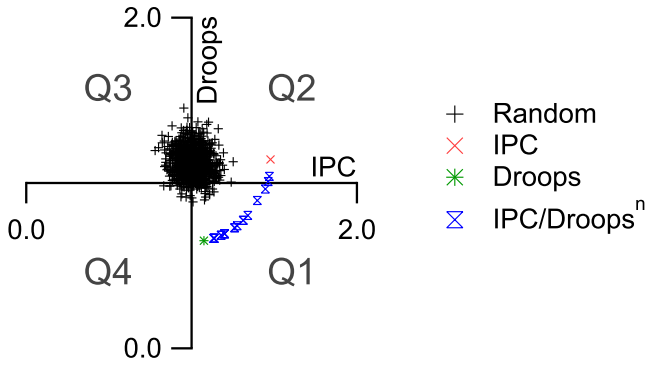


Figure 5.19: Proof that scheduling for voltage noise is different from scheduling for performance. Scheduling for performance causes more emergencies, which upon factoring emergency tolerance rollback costs can actually result in performance degradation. Noise-aware schedulers are necessary in our architecture.

gencies. Results in Q3 are bad, since performance degrades and emergencies go up. Lastly, results in Q4 imply a reduction in emergencies at the expense of some performance.

By today's standards, our random simulation is representative of production operating systems. The POSIX 2010 policies includes simple policies like round-robin and first-in, first-out that are effectively random in behavior. From observing data in Figure 5.19 we can conclude that random schedules lead to more voltage emergencies. Additionally, there are no guarantees about performance.

By comparison, a performance centric scheduler achieves best performance, as expected. However, such a scheduler is unaware of voltage emergency activity occurring as a result of its scheduling decisions. In Figure 5.19 the IPC marker is in quadrant Q2, indicating that on aggregate more emergencies occur than our baseline. Although improving performance implicitly leads to fewer execution stalls, this data indicates that reducing stalls alone is insufficient to reduce emergencies in a multi-core system.

Interactions across threads (or cores) impact the amount of voltage noise we observe. Therefore, a noise-aware scheduler is necessary.

Consider the Droops metric, or noise-aware scheduling, whose data point resides in quadrant Q4. The noise-aware scheduler focuses on emergency activity and is therefore able to minimize emergencies across all 50 jobs. It does this without adversely affecting performance.

A noise-aware scheduler can be adapted to not only reduce emergencies, but also improve performance. To achieve this we propose a new scheduling metric: $\mathsf{IPC}/\mathsf{Droops}^n$. Droops are weighted by some factor n that determines how costly emergencies are to tolerate. The value of n is small if recovering from emergencies is cheap, costing only few tens of clock cycles. Otherwise, n is large. A thread scheduler can use this value, n, to balance the penalty of tolerating emergencies as it attempts to maximize performance. The arc of markers in quadrant Q2 of Figure 5.19 illustrate the range of opportunity over different values of n.

Chapter 6

Conclusion

Continuing technology advances amplify the importance of reliability in modern high-performance processors. Shrinking feature size and diminishing supply voltage are making circuits ever more sensitive to transient errors, stemming from process, voltage and thermal variations. A paradigm shift is necessary in our thinking to continue producing processors that maximize performance under strict dollar and power budgets in the presence of these variations.

Power-constrained processor design is impacting processor reliability. Techniques that target power reduction like clock gating, when aggressively applied to constrain power consumption, are leading to large current swings in the processor. When coupled with the non-zero impedance of a power-delivery subsystem, these current swings can cause voltage to fluctuate beyond safe operating margins. Such dangerous fluctuations, called voltage emergencies, have traditionally been dealt with by optimizing for the worst-case voltage flux, allocating sufficiently large voltage margins to avoid any timing violations.

With continued technology scaling, the voltage noise problem is an increasingly important design challenge. Prior method of avoiding emergencies altogether incurs severe performance penalties. Enabling aggressive operating margins is critical at the risk of voltage emergencies is important, since worst-case margins cripple performance-per-watt efficiency in microprocessor designs. The overall goal of this work is a full system design, implementation and evaluation of a hardware-software collaborative approach to the voltage noise problem.

We demonstrate our contributions toward this hardware and software combination that enables aggressive operating voltage margins: we allow voltage emergencies to

occur at the expense of rolling back execution while relying on a feedback-driven software layer to permanently eliminate recurring emergencies. We base this design on characterizing emergencies in terms of code behavior, which enables us to predict them intelligently, and even eliminate them completely. Our collaborative design is a more holistic technique for handling voltage emergencies, as compared to existing and prior work in this area. Therefore, our solution allows us to more easily harness the benefits of improved energy efficiency or performance improvement that aggressive margins enable.

Rather than proactively avoiding any voltage emergencies from occurring, we exploit voltage emergency *tolerance*. By tolerating emergencies we demonstrate that designing for the absolute worst-case severely penalizes the maximum efficiency we can extract from our processor chips. Most programs do not need the large voltage margins under typical case conditions, like the 14% margin in use by the Intel Core^{TM}2 Duo processor or the 20% margin that the designers of the POWER6 processor put in place. An aggressive 4% margin suffices for the common case. A noise-tolerant system can handle those rare cases using a fail-safe recovery mechanism, like checkpoint-recovery. An architecture for tolerating emergencies also allows us to identify leading indicators of activity that induces emergencies. We found a strong relationship between microarchitectural events and current and voltage fluctuations within the microprocessor. We consider several microarchitectural parameters, such as the number of entries in the re-order buffer, the instruction fetch queue, and the load/store queue, along with microarchitectural events like cache misses and pipeline flushes, showing that it is a confluence of stalling activity that induces emer-

gencies. We demonstrate a means of capturing the interleaving of program path with microarchitectural events that generates a representative snapshot of emergency activity. These voltage emergency signatures reflect corresponding dynamic current and voltage activity resulting from program interactions with the underlying microarchitecture leading to emergencies.

Tolerating emergencies is prohibitively expensive, therefore we invented the voltage emergency predictor as a mechanism for emergency *avoidance*. Our avoidance mechanism uses voltage emergency signatures to anticipate emergencies and proactively avoid them via throttling, while relying on the general-purpose checkpoint-recovery logic already available in todays production systems to train itself. Our signature-based voltage emergency predictor operates independently of sensor delays, package characteristics, and microarchitecture details, and it enables operation at aggressive voltage margins without compromising correctness. Instead of using a conservative 14% voltage margin, the predictor improves performance by 13.5% at an aggressive 4% voltage margin, which is very competitive to the 14.2% improvement we can achieve using an idealized oracle-based throttling mechanism at the same setting.

Having shown that voltage emergencies are recurring using emergency signatures, and thus predictable, we extend our work further to use software for voltage emergency *elimination*. In our collaborative architecture, software reduces hardware penalties to either tolerate or avoid voltage emergencies by permanently fixing code regions responsible for those emergencies. Additionally, since we found that co-scheduling of threads impacts the number of emergencies at run-time, we propose and demonstrate novel operating system scheduling policies that specifically reduce voltage emergencies

based on feedback from hardware.

Since hardware always offers a fail-safe route, software uses dynamic feedback from hardware to decide where, when and how to optimize. We show that through code transformation techniques a dynamic compiler eliminates over 60% of all emergencies, and therefore dramatical reduces the recurring burden on hardware. Similarly, our noise-aware thread scheduling policy demonstrates that co-scheduling threads in a multi-core environment can mitigate global checkpoint recovery overheads.

Optimizing away voltage emergencies is analogous to removing cache misses or branch mispredictions to achieve better performance or lower power consumption. Considering the impact of voltage noise on processor efficiency, aggressive operating voltage margins are inevitable. As feature size shrinking continues reliability problems like voltage emergencies will continue to emerge more forcefully, requiring us to rethink traditional processor design involving software as an essential fabric in the production of future processors.

Going forward, building robust microprocessors that deliver maximum performance under strict power and cost budgets is going to become ever more important in the presence of variations. We hope that the three principles we introduce in this thesis: tolerance, avoidance and elimination act as guiding principles for building resilient systems in the future.

Appendix A

Measuring Voltage Noise in Production Processors

Contents

A.1 Measurement and Validation

Power supply design requires robust analysis and engineering effort to prevent transient voltage droops. Such voltage noise can cause the processor to malfunction, due to slower operating circuits. Consequently, processor designers and motherboard regulator engineers use custom industrial toolkits to carefully measure, test and validate voltage variation with a microprocessor given a specific test-bed (or motherboard).

Current processor to platform verification and validation is done using electronic loads called Voltage Transient Test (VTT) tools [2]. These tools allow voltage droop characterization in the time domain series. These test environments enable characterization of noise phenomena like the resonance under manual external stimuli. However, they typically require additional hardware that is custom designed, and as such is not publicly available for academic use.

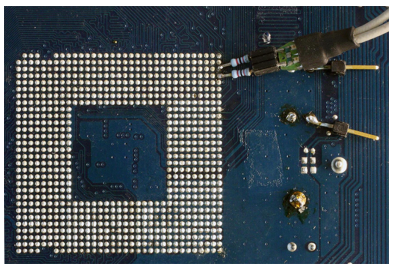
A.1.1 Using Off-the-shelf Components

VTT tools are impractical for our experimentation and measurement purposes. First, VTT rely on external stimuli. Generally designers use worst-case peak to peak current swings to investigate platform voltage noise characteristics. Unfortunately, such current swings are not representative of actual program activity. By comparison, we want to characterize and study voltage droop in response to program-specific current draw activity. This is important to us since we are interested in studying the typical case voltage swing because of program activity. Therefore, we must be able to observe processor voltage uninstrusively as it is running in a typical environment.

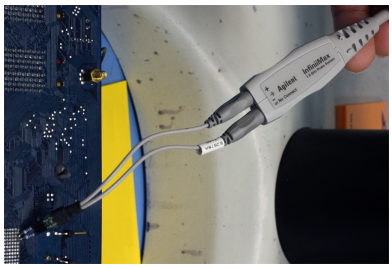
We discovered a new approach of measuring processor voltage in a production setup using off-the-shelf components. The setup is illustrated in Figure A.1, going from (a) through (d) in a sequential process. The specifics of the host system and equipment are as follows: we measure core supply voltage fluctuations using a Gigabyte GA-945GM-S2 motherboard that exposes processor package pins corresponding to core VCC*sense* and VSS*sense*. These pins provide an isolated low impedance connection to the supply voltage, VCC and VSS power plane, within the core. To ensure we do not introduce measurement error and that we maintain high signal fidelity, we use a InifiniiMax 1.5GHz 1130A differential probe to sense voltage. A DSA91304A Infiniium oscilloscope gathers probe readings at a high frequency. We configure the scope to take one reading per clock tick, allowing us to closely relate program activity to voltage fluctuations. A host system gathers these scope measurements remotely over the network.

The specific processor we use is a Intel® Core™2 Duo Desktop Processor E6300 and its specification number is SL9SA. This is a dual core processor, comprising of two cores on a single die. Although our experiments are reserved to this one processor, the measurement methodology is extensible to any processor the motherboard supports. The motherboard supplies power to both processing units using a single voltage regulator module, therefore any transient voltage droop affects both cores.

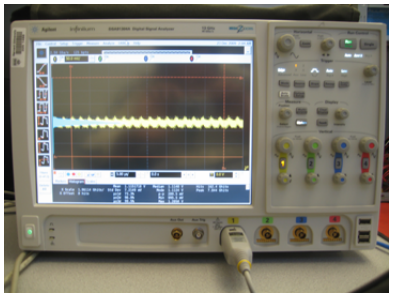
The benefits of our setup are that we can measure voltage fluctuations within the processor without requiring any special experimental toolkit. Even more importantly, we can now run entire suites of real programs to completion, rather than simulation and observing activity only over few millions of instructions. Moreover, most simu-



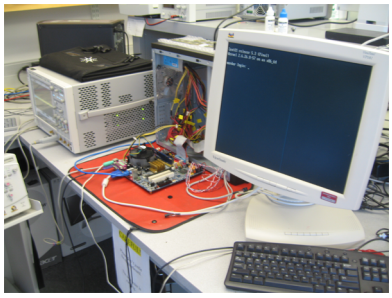
(a) Intel provides VCC_{sense} and VSS_{sense} pins that allow us to sense silicon voltage and ground, respectively, using the processor's low impedance land pins.



(b) To ensure high signal fidelity and prevent measurement induced noise, we measure voltage with ultra low loading using the Inifini-iMax 1.5GHz 1130A differential probe.



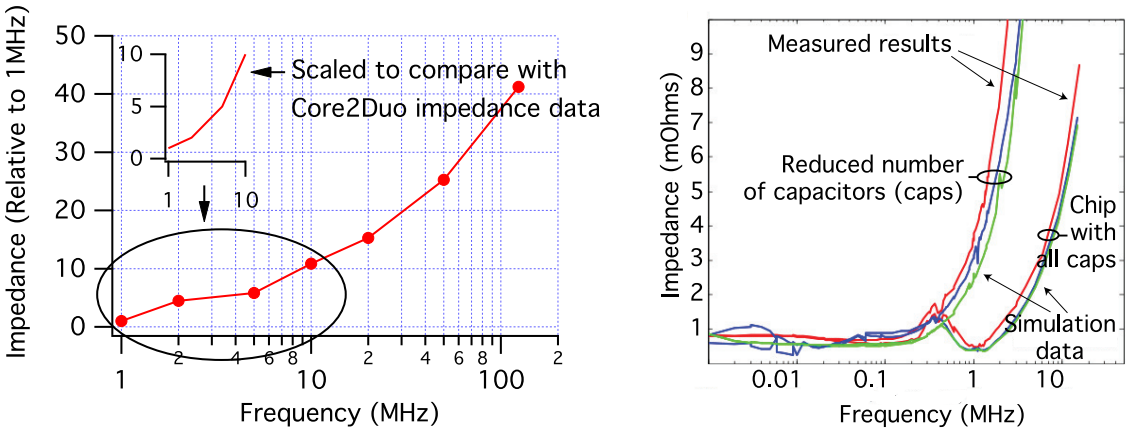
(c) We measure and collect the sensed voltage data from the differential probes using an Agilent DSA91304A Infiniium oscilloscope.



(d) Gathering data using an external system that streams measurements from the oscilloscope during execution.

Figure A.1: Setup illustrating how we sense and measure voltage fluctuations within the processor during execution time.

lators can only support a limited set of programs, even further constraining robust evaluation and characterization of voltage noise.



(a) Measured. (b) Intel impedance data [6].

Figure A.2: Validating our measurements by comparing impedance we derive from our experimental setup to Intel's published results.

A.1.2 Comparing Impedance

We validated our experimental setup by constructing the impedance profile of our platform and comparing it with Intel's published data. Figure A.2a corresponds to the impedance profile we constructed on our system using a technique previously described in literature [11]. However, we make a minor modification to this prior technique which we will describe later. Figure A.2b corresponds to Intel data. We extracted this figure from an Intel voltage regulator manual corresponding to dual core processor designs [6]. We edited the original graph with markers and labels to present our arguments clearly.

There are two important validation points in our graph. First, impedance peaks at around 100MHz, which matches with prior results describing typical power delivery network characteristics [30, 12, 53, 27]. Second, our data matches Intel Core^{TM}2 Duo data even more closely between the 1MHz and 10MHz. The small graph embedded within Figure A.2a corresponds to Measured results in Figure A.2b for a Chip with all

caps. Other data is unimportant and irrelevant for our purposes.

In the beginning of this section we mentioned making a slight alteration to an already existing methodology that we used to construct the relative impedance of our system. In prior work, the authors need to create a square pulse train of current activity in order to measure the voltage droop response of the platform at a certain frequency. So they drive the clock tree of the processor using an external clock signal generator at the required frequency. They claim this creates the representative current stimulus necessary within the processor, at which point they proceed to estimate the voltage droop. This process, repeated over a range of frequencies, eventually allows them to construct the impedance profile we see in Figure A.2b.

In our effort, we do not rely on an external source to create the current stimulus, rather we use software. In this way we can use a production setup for our measurements of on-die voltage. We constructed a user-level program that alternates between high power and low power instruction sequences repeatedly in a loop. Instructions for the high power sequence were from CPUBurn [40]. The lower power sequence is made up of no-operation instructions. The duty cycle for alternating between these two extreme power consumption paths is a command line argument to our program. Modulating activity in this way allows us to create current stimuli with varying frequencies within the processor, similar to driving the clock tree, but without requiring a custom and proprietary hardware setup.

A.2 Determining the Worst-case Voltage Margin

We can determine the voltage margin of a processor at a certain frequency by undervolting the core's nominal supply voltage. Reducing the supply voltage slows down circuit operation speed. Consequently, peak operational frequency of the processor logic should drop. Otherwise, the clock of the processor is ticking much faster than the rate at which processor logic is capable of operating at. Therefore, undervolting theoretically should force the processor into functional errors. But in production settings this does not happen immediately, as there are voltage margins or guardbands protecting against *transient* worst-case voltage droops.

Nevertheless, at some point as we continue to lower voltage the processor will run into functional errors, causing the system to hang or in effect "crash". At this voltage our processor is unable to meet timing constraints necessary to operate correctly at the current frequency setting. Circuits operation has finally become slow enough to reveal the minimum voltage necessary to ensure correctness of execution. Therefore, we have just forced our processor into a voltage emergency. The difference between the original nominal voltage setting and this current voltage setting is the amount of guardband in place to tolerate those transient voltage swings.

We did this analysis for our Intel Core^{TM}2 Duo processor and found the worst-case voltage margin to be 14%. Determining the margin this way requires motherboard support. Most motherboards do not allow undervolting. Therefore, we specifically used the Gigabyte 965P-DS3 motherboard for this experiment. This motherboard is publicly available for purchase and it allows us to undervolt the processor very aggressively, well below the processor's minimum operating voltage margin.

Appendix B

Framework for Evaluating New Techniques to Lower Voltage Noise

Contents

Simulation and modeling frameworks are essential to any study. This is especially true when we are working on hardware related issues, such as voltage noise. Production hardware systems are highly optimized and proprietary, leaving little room for introspection. We need a framework that allows us to carefully study and understand a problem in great detail. And as we better understand the problem, the framework should allow us to evaluate the effect of novel solutions we propose to solve the problem. To this end, in this chapter we explain how we model, study, and evaluate solutions to voltage noise and provide specifics about the tools we use.

Our solution involves co-designing hardware and software to mitigate voltage noise. The hardware tolerates emergencies and avoids emergencies intelligently when possible while software attempts to eliminate them altogether. Therefore, we require both hardware and software components. The hardware components include a microarchitectural simulator, a power consumption model and a system that models the power delivery subsystem. All of these are necessary to model voltage noise. The microarchitectural simulator also allows us to investigate new hardware-based solutions. The software components include the programs we are studying and a run-time layer that allows us to investigate new techniques that can eliminate voltage emergencies.

Figure B.1 illustrates each of the components in our experimental framework, showing information flow from one component to another. The Processor Microarchitecture component models a specific processor implementation. The Power Consumption Model component tracks the simulated microarchitecture's activity to determine how much current the processor is drawing each cycle. The Power Delivery Subsystem component then uses this cycle by cycle current information to compute the

instantaneous voltage by convolving it with the impulse response characteristics of the power delivery network it is modeling. There are sensors within the microarchitecture simulator that monitor voltage to detect voltage emergencies. The microarchitectural simulator also includes the checkpoint-recovery mechanism necessary to tolerate emergencies (Chapter 3). The simulator also serves as the basis for investigating different avoidance mechanisms (Chapter 4). When the sensors detect an emergency, the microarchitecture simulator invokes software callbacks that notify the Run-time System of the emergency (Chapter 5). The run-time system then smoothes or transforms the code corresponding to the running Benchmark. This is done as the program is executing. In the case of thread scheduling, we do not rely on any of the Hardware Simulators. Multiple Benchmarks (one per core) are run natively on the real hardware. That is a different experimental setup. Please refer to Appendix A for those hardware setup details, including how we measure voltage noise in a real chip. Operating system thread scheduler details follow in this section.

We investigate our hardware and software co-design solutions using simulations of a representative superscalar microprocessor in which voltage noise fluctuations beyond 4 percent of nominal voltage are treated as voltage emergencies. Our modified 8-way superscalar x86 SimpleScalar gathers detailed cycle-accurate current profiles using Wattch [18]. Voltage variations are calculated by convolving the simulated current profiles with an impulse response of the power delivery subsystem [43, 34]. We use a power delivery subsystem model based on the characteristics of the Pentium 4 package [11], which exhibits a mid-frequency resonance at 100MHz with a peak impedance of 5mΩ, assuming peak current swings between 16A and 50A.

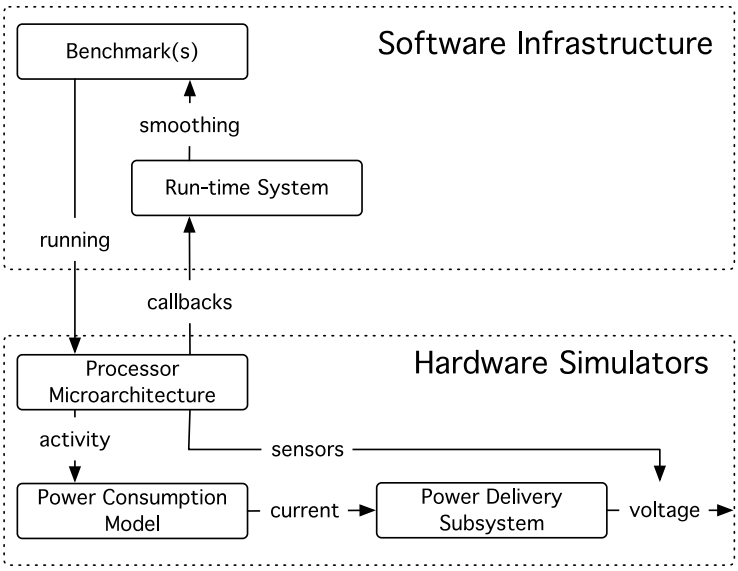


Figure B.1: Simulation framework for studying new voltage noise techniques.

B.1 Hardware Simulators

In this section, we provide details for each of the hardware components in Figure B.1.

B.1.1 Processor Microarchitecture

We study voltage noise assuming an out-of-order superscalar processor. The simulator we use is a x86 version of SimpleScalar [20]. We are grateful to Brad Calder's research group from the University of California at San Diego for this infrastructure.

Modifications. We made several extensions to the simulator. These extensions allow the simulator to (1) fast-forward execution more efficiently to study only the interesting parts of a program, (2) more robustly support a wide variety of programs and (3) provide voltage noise feedback to the software at run-time.

Forwarding. Fast-forwarding execution in simulation frameworks is very important to quickly focus in on only the interesting parts of a program. This is key since simulating the microarchitecture makes full program analysis impossible. Therefore, being able to get to the interesting parts of a program can dramatically speed up experimentation. Typically, simulators use compressed trace files or checkpoints that allow very efficient fast-forwarding. However, this limits the breadth of programs we can study, since any program we wish to study requires we have these checkpointing files.

To allow efficient fast-forwarding without the need for checkpoint files, we replaced the simulator code for fetching instructions with a Pin Tool [3]. Pin is a just-in-time (JIT) compiler that allows us to run the program at nearly full speed on the baremetal hardware, but with the ability to interject execution as necessary. We leverage this capability to fast-forward execution more efficiently in the simulator.

While the program is running through parts of a program that are of no interest, we fast-forward execution by bypassing the simulator. The program is run directly on the underlying hardware. Therefore, the program is fast-forwarding at the speed of native execution. However, we regain control whenever the program reaches an interesting point. From that point forward, we redirect every instruction the program is executing into the microarchitecture simulator. We achieve this by writing a Pin Tool that tracks the dynamic control flow path of a program, which at every instruction boundary feeds the simulator with the bytes corresponding to that instruction, so that the simulator itself can execute the instruction within its virtual infrastructure.

However, feeding instructions into the simulator via a Pin Tool proves challenging

because of branch speculation. Pin allows a tool to observe only the true or committed dynamic path of a program. It cannot trace instructions down the speculative path because of branch mispredictions. But this is an important aspect of execution that affects voltage noise. Therefore, it is important we allow our simulator to speculatively execute instructions and capture those effects on voltage noise.

We allow speculative execution in our simulator despite a non-speculative instruction fetch engine by stalling the Pin instruction feeder whenever the simulator's branch predictor predicts an outcome that does not align with the true program path. Pin knows the program's true path before executing the branch instruction, since the branch direction is known by looking at values present in the condition code register. While Pin is stalled at the mispredicted branch, the simulator continues executing instructions speculatively starting from the first instruction along the wrong path. Eventually the simulator realizes that it mispredicted the branch and flushes its pipeline. At this point, execution resumes down the correct path of the program, and we resume instruction feeding via Pin. Since the simulator decodes instructions by itself, its instruction set architecture decoder must be robust.

Robustness. To make the simulator more robust, we replaced the simulator's native instruction encoder and decoder. The original encoder and decoder could not handle all instruction types. So instead we use XED [4], which is a software library for encoding and decoding x86 (IA-32 instruction set and Intel 64 instruction set) instructions. XED is more up to date and supports the x86 instruction set architecture (ISA) far more robustly. While this allows us to run many more instruction types through the simulator, corresponding extensions are also necessary to the simulator's

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 B.1: Architecture parameters for SimpleScalar.

x86 microcode engine. These extensions all put together allow us to run the whole suite of CPU2006 benchmarks more robustly on the simulator.

Feedback. Our work involves hardware and software co-design to mitigate voltage noise. For this purpose, we extended the simulator with support for callbacks. These hooks allow the software to register for events occurring within the simulated microarchitecture. For instance, the software can register a function that the microarchitectural simulator calls whenever an emergency occurs. In this way we evaluate what information hardware should provide the Run-time System layer to perform voltage smoothing. Moreover, the feedback loop between the hardware and software layers allows us to evaluate the effectiveness of voltage smoothing as the program is running, thus creating a dynamic feedback-driven system.

Configuration. We configure the simulator to be representative of a traditional superscalar processor. Table B.1 details the set of configuration parameters we use for all experiments discussed in this work.

B.1.2 Power Consumption Model

We use Wattch [18] to model core power consumption. Each cycle the power consumption model generates active current draw based upon microarchitectural switch-

ing activity that it observes within the simulator. We use the default set of configuration parameters.

B.1.3 Power Delivery Subsystem

A power delivery package model computes voltage variations by convolving the simulated current profiles from the power consumption model above with an impulse response of the power delivery subsystem [43, 34]. In parts of this work we evaluate three different power delivery subsystem package models. Details here pertain to each one of those packages. These models have been used and studied extensively [27, 28, 30]. The packages we use are labeled Pkg 1, Pkg 2 and Pkg 3 and their details are shown in Table B.2. Our baseline package for all experiments is Pkg 1 unless stated as otherwise.

Quality factor (Q) is the ratio of the resonant frequency to the rate at which the package dissipates its energy. A larger Q gives rise to larger voltage swings for currents oscillating within the resonance band of frequencies. Applications with current fluctuations in the resonance band therefore suffer more from inductive noise with a high-Q package.

Pkg 1 closely resembles characteristics of the Pentium 4 package [32]. Pkg 2 is representative of the package used in an earlier study [34], and its parameters are based on the Alpha 21264/21364 package. For comparisons, we also include Pkg 3, which represents a bad package with very large quality factor.

Package	Peak Impedance (mOhm)	Current (A)	Quality Factor	Resonance Cycles	Comment
Pkg 1	5	16–50	3	30	Pentium 4 [11]
Pkg 2	2	30–70	2	60	Used in [34]
Pkg 3	17	16–50	6	30	Worst package

Table B.2: Characteristics of the power delivery subsystem packages we use in our study. By default and unless stated as otherwise, we model voltage noise using Pkg 1.

B.2 Software Infrastructure

In this section, we explain details pertaining to the software pieces of our experimental framework. We explain the compiler tool-chain that we use to study code transformation techniques, the benchmarks we use and also how we model the behavior of an operating system thread scheduler.

B.2.1 Benchmarks

We use a combination of C, C++, Fortran and C# benchmarks. With the exception of C# benchmarks, all programs come from the SPEC CPU2006 benchmark suite [5]. These benchmarks were linked statically, and compiled using gcc 4.0 at the `O3` optimization level.

For evaluating compiler-based voltage emergency elimination techniques (Section 5.3), we use the C# benchmarks from the Java Grande benchmark suite [19]. Table B.3 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.

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 B.3: Descriptions of C# benchmarks.

B.2.2 Compiler

We use the ILDJIT [21] 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. The compiler has access to metadata such as the complete control flow graph and data flow graph, all of which is useful for optimizing code at runtime.

Modifications. We extended the native ILDJIT compiler to include the code injection and scheduling algorithms described in chapters earlier on.

B.2.3 Operating System Thread Scheduler

Scheduling threads intelligently reduces voltage noise. We demonstrate this through our evaluation of different scheduling policies in Chapter 5.4, but defer details to this section for maintaining clarity of thought.

We evaluate scheduling effects on voltage noise in two phases: a profiling phase and an evaluation phase. During the profile phase we gather all the data necessary to

simulate policies, and during the evaluation phase we use metrics of interest to draw conclusions. The latter is done offline, as that provides us with flexibility to do more thorough and comprehensive analysis than doing it online. Simulating scheduling policies offline isolates the effects of co-scheduling certain threads together. Results are reproducible and we can easily evaluate the impact of the different scheduling policies on voltage noise.

Profiling Phase. For the purpose of our thread scheduling experiments, we pre-run all combinations of programs together; programs are our experimental granularity of threads. Sweeping this combination allows us to construct a matrix of data containing all the necessary information that we subsequently use to evaluate different schedules.

Evaluation Phase. We model a batch processing environment. The scheduler receives a queue of jobs that must all be run to completion. This queue consists of a large pool of threads from which the scheduler must select the next set of jobs to run each interval. The thread scheduling experiment completes when all threads finish executing. Results (overall IPC, or average number of emergencies per 1K cycles etc.) are analyzed at the end of an experiment.

Every wakeup call, or thread scheduling interval, the scheduler schedules n threads to run concurrently on our n-core system. The scheduling policy determines which set of threads are chosen for running together. Some scheduling policies do not require information from the profiling phase. For example, standard 2010 POSIX policies include round-robin or first-in, first-out. In our setup, these policies boil down to randomly picking a combination of threads from the pool and running them to com-

pletion. Some other policies require information from the profiling run. One such example could be enforcing a policy that requires the operating system to schedule for the least number of emergencies. The scheduler knows a priori from the profiling phase which set of threads that if scheduled together result in the least number of emergencies. It then schedules those threads together. At the end of a scheduling interval, the threads are removed from the scheduling queue if the threads have finished execution. This process repeats each interval.

Our thread scheduling interval is every 60 seconds. We justify this long scheduling interval using the cost and sensitivity of capturing voltage noise measurements. The experimental harness for measuring voltage noise is described in the previous chapter (see Appendix A). As per that setup, we cannot gather voltage measurements at more representative scheduling intervals like 100 milliseconds. Such fine-grained polling of data causes a lot of measurement noise, since the host system is both running the threads, as well as doing measurements. Both these tasks must be done on the same platform to synchronize when measurements begin and end and when the program calls the `main` and `exit` functions, respectively. We find that measuring and collecting data more frequently than 60 seconds leads to an "observer effect". However, we find no such measurement noise at 60 second intervals.

Bibliography

[1] http://software.intel.com/en-us/intel-vtune/.

[2] http://www.cascadesystems.net/lga775.htm.

[3] http://www.pintool.org/.

[4] http://www.pintool.org/docs/24110/xed/html/.

[5] http://www.spec.org/cpu2006/.

[6] Voltage regulator-down (vrd) 11.0. *Processor Power Delivery Design Guidelines For Desktop LGA775 Socket*, November 2006.

[7] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive Incremental Checkpointing for Massively Parallel Systems. In *International Conference on Supercomputing '04*, 2004.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Prentice Hall, 2006.

[9] H. Ando et al. A 1.3 ghz fifth-generation sparc64 microprocessor. In *Proceedings of Design Automation Conference*, 2003.

[10] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3ghz fifth generation SPARC64 microprocessor. *IEEE Journal of Solid-State Circuits*, 38, 2003.

[11] K. Aygun, M. J. Hill, K. Eilert, R. Radhakrishnan, and A. Levin. Power delivery for high-performance microprocessors. *Intel Technology Journal*, 9, November 2005.

[12] K. Aygun, M.J. Hill, K.D. Ellert, and K. Radhakrishnan. Measurement-to-modeling correlation of the power delivery network impedance of a microprocessor system. In *Electrical Performance of Electronic Packaging, 2004. IEEE 13th Topical Meeting on*, pages 221 – 224, oct. 2004.

[13] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Programming Language Design and Implementation*, pages 1–12, 2000.

[14] Luiz A. Barroso. The price of performance: An economic case for chip multiprocessing. *Queue, ACM*, 2005.

[15] Luiz A. Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The Google cluster architecture. *Micro, IEEE*, 2003.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[17] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. Lee, C. B. Wilkerson, S-L Lu, T. Karnik, and V. De. Energy-efficient and metastability-immune timing-error detection and instruction replay-based recovery circuits for dynamic variation tolerance. In *International Solid State Circuits Conference*, 2008.

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

[19] Mark Bull, Lorna Smith, Martin Westhead, David Henty, and Robert Davey. Benchmarking java grande applications. In *The Practical Applications of Java*, 2000.

[20] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical report, 1996.

[21] Simone Campanoni, Giovanni Agosta, Stefano Crespi-Reghizzi, and Andrea Di Biagio. A highly flexible, parallel virtual machine: design and experience of ildjit. *Softw., Pract. Exper.*, 40(2):177–207, 2010.

[22] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, and Mateo Valero. Predictable performance in smt processors: Synergy between the os and smts. *IEEE Trans. Comput.*, 55(7):785–799, 2006.

[23] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation. In *40th International Symposium on Microarchitecture (MICRO-40)*, 2007.

[25] Alexandra Fedorova. *Operating system scheduling for chip multithreaded processors*. PhD thesis, Cambridge, MA, USA, 2006. Adviser-Seltzer, Margo I.

[26] E. Grochowski, D. Ayers, and V. Tiwari. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *Int'l Symposium on High-Performance Computer Architecture*, 2002.

[27] M. S. Gupta, J. L. Oatley, R. Joseph, G-Y Wei, and D. Brooks. Understanding voltage variations in chip multiprocessors using a distributed power-delivery network. In *Design, Automation and Testing in Europe, (DATE)*, 2007.

[28] M. S. Gupta, K. Rangan, M. D. Smith, G-Y Wei, and D. Brooks. Towards a Software Approach to Mitigate Voltage Emergencies. In *International Symposium on Low Power Electronics and Design, (ISLPED '07)*, 2007.

[29] Meeta S. Gupta, Vijay J. Reddi, Michael D. Smith, Gu-Yeon Wei, and David M. Brooks. An event-guided approach to handling inductive noise in processors. In *DATE*, 2009.

[30] Meeta Sharma Gupta, Krishna Rangan, Michael D. Smith, Gu-Yeon Wei, and David M. Brooks. DeCoR: A Delayed Commit and Rollback Mechanism for Handling Inductive Noise in Processors. In *HPCA '08*, 2008.

[31] Meeta Sharma Gupta, Krishna K. Rangan, Michael D. Smith, Gu-Yeon Wei, and David Brooks. DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors. In *HPCA-14*, 2008.

[32] Intel. Intel Pentium 4 processor in the 423 pin/package /Intel 850 chipset platform, February 2002.

[33] N. James, P. Restle, J. Friedrich, B. Huott, and B. McCredie. Comparison of split-versus connected-core supplies in the POWER6 microprocessor. In *International Solid State Circuits Conference 2007*, February 2007.

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

[35] W. Kim, M. S. Gupta, G-Y Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *14th International Symposium on High-Performance Computer Architecture (HPCA-14)*, 2007.

[36] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Checkpointed early load retirement. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[37] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.

[38] Jason Mars, Neil Vachharajani, Mary Lou Soffa, and Robert Hundt. Contention aware execution: Online contention detection and response. In *CGO '10: Proceedings of the 2010 International Symposium on Code Generation and Optimization*, Toronto, Canada, April 2010.

[39] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *35th International Symposium on Microarchitecture (MICRO-35)*, November 2002.

[40] Michael Mienik. Cpu burn-in homepage.

[41] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, 2005.

[42] Satish Narayanasamy, Bruce Carneal, and Brad Calder. Patching processor design errors. In *ICCD*, 2006.

[43] Michael Powell and T. N. Vijaykumar. Exploiting resonant behavior to reduce inductive noise. In *ISCA*, June 2004.

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

[45] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[46] Smruti Sarangi, Satish Narayanasamy, Bruce Carneal, Abhishek Tiwari, Brad Calder, and Josep Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro*, 2007.

[47] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. *ASPLOS-XII*, pages 73–82, 2006.

[48] Slegel et al. IBM's S/390 G5 microprocessor design. *Micro, IEEE*, 1999.

[49] Larry Smith, Raymond Anderson, Doug Forehand, Tom Pelc, and Tanmoy Roy. Power distribution system design methodology and capacitor selection for modern cmos technology". 1999.

[50] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 35(11):234–244, 2000.

[51] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Fast Checkpoint/Recovery to Support Kilo-instruction Speculation and Hardware Fault Tolerance. Computing science technical report, University of Wisconsin-Madison, 2000.

[52] Ilya Wagner, Valeria Bertacco, and Todd Austin. Shielding against design flaws with field repairable control logic. In *IEEE/ACM Design Automation Conference*, 2006.

[53] A. Waizman. Cpu power supply impedance profile measurement using fft and clock gating. In *Electrical Performance of Electronic Packaging, 2003*, pages 29 – 32, oct. 2003.

[54] N. J. Wang and S. J. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Trans. Dependable Secur. Comput.*, 3(3):188–201, 2006.

[55] Wei Zhao and Yu Cao. Predictive technology model for sub-45nm early design exploration. *ACM JETC*.

[56] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Architectural support for programming languages and operating systems*, pages 129–142, New York, NY, USA, 2010. ACM.