Resilient Architectures via Collaborative Design: Maximizing Commodity Processor Performance in the Presence of Variations

Vijay Janapa Reddi, *Member, IEEE,* and David Brooks, *Member, IEEE*

***Abstract*—Unintended variations in circuit lithography and undesirable fluctuations in circuit operating parameters such as supply voltage and temperature are threatening the continuation of technology scaling that microprocessor evolution relies on. Although circuit-level solutions for some variation problems may be possible, they are prohibitively expensive and impractical for commodity processors, on which not only the consumer market but also an increasing segment of the business market now depends. Solutions at the microarchitecture level and even the software level, on the other hand, overcome some of these circuit-level challenges without significantly raising costs or lowering performance. Using examples drawn from our Alarms Project and related work, we illustrate how collaborative design that encompasses circuits, architecture, and chip-resident software leads to a cost-effective solution for inductive voltage noise, sometimes called the dI/dt problem. The strategy that we use for assuring correctness while preserving performance can be extended to other variation problems.**

***Index Terms*—Dynamic variation, error correction, error detection, error recovery, error resiliency, hw/sw co-design, inductive noise, power supply noise, reliability, resilient design, resilient microprocessor, timing error, variation, voltage droop.**

I. Introduction

THE LANDSCAPE of general-purpose microprocessor design is changing due to variations. Historical processor designs were driven by the goal of ever-higher performance. Lately, energy efficiency has emerged as an even more important design principle. Unfortunately, achieving these goals is becoming difficult in the presence of process, voltage, and thermal variations. Nowadays, processors tolerate these variations using guardbands or margins, trading lower power or higher performance for operational robustness; margins are a cheap and cost-effective solution for mass production. But as the industry moves forward toward smaller device feature sizes, these margins must grow, as circuit behavior susceptibility to variations increases at reduced feature sizes. As a consequence, building robust and thus reliable microprocessors comes at the expense of decreasing overall processor efficiency. Mitigating this inefficiency requires costly solutions that increase the price of a chip. Unfortunately, neither of these solutions are practical for commodity processors that must be optimized on a cost per unit of performance basis, not purely on performance per CPU unit only. In other words, they must operate under a good price-to-performance ratio.

Traditionally, designers of commodity processors have been tackling variation-associated reliability challenges at the circuit-level, masking the issues from the processor microarchitecture above and the software running on top of it. However, as these traditional solutions are not scaling well, future systems require adaptive processor design techniques. The underlying microarchitecture must dynamically detect and recover from reliability errors, or *emergencies*, in the field.

In this paper, we discuss an emerging collaborative machine organization, where both hardware and software play an integral role to diminish the detrimental effects of variations. We envision 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 Fig. 1. The figure presents an overview of the three levels and some of the information that flows between these levels when detecting and resolving emergencies. The bottom layer includes low-level hardware and circuit blocks that signal sensed critical information such as voltage or temperature emergencies. The microarchitectural layer collects this sensor data, filtering it and combining it with runtime activity history, such as the current thread and its code along with microarchitectural state information, before passing it up to a chip-management software layer that enables transparent and error-free application-level software execution.

Two distinct methods exist for dealing with emergencies in this architecture. First, the circuits and microarchitectural layer are responsible for guaranteeing reliable operation without the assistance of software. The software layer seeks to eliminate these exceptional events from recurring in the future through emergency-specific dynamic optimizations. But as a first line of defense, the circuits and microarchitectural layer operate independently of software to throttle circuit execution/behavior to guarantee correctness. An advantage of this multilayered approach is that it allows the hardware to focus on guaranteeing correct operation for the initial exceptional event, while the software focuses on eliminating or reducing the performance impact of the future events in the steady state.

Manuscript received June 28, 2011; accepted July 13, 2011. Date of current version September 21, 2011. This paper was recommended by Associate Editor V. Narayanan.

V. J. Reddi is with the Department of Electrical and Computer Engineering, University of Texas, Austin, TX 78712 USA (e-mail: vj@ece.utexas.edu).

D. Brooks is with the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138 USA (e-mail: dbrooks@eecs.harvard.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2011.2163635

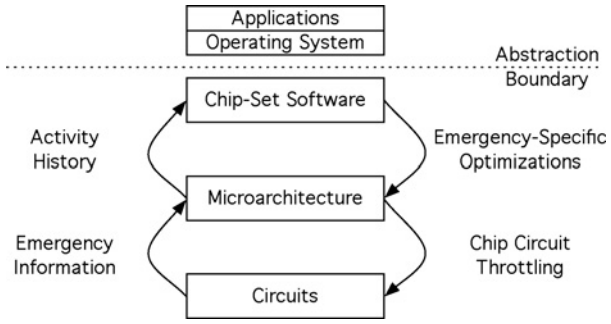

Fig. 1. Abstract overview of exposing circuit-level reliability challenges to the higher levels of execution.

The goal of such a multilevel approach is to eliminate the penalties to performance that arise in the use of circuit techniques and microarchitectural changes that lower power, price, or in general attempt to optimize a design for criteria other than performance. Toward this end, the work investigates what it takes to design and build commodity computing systems that achieve both high performance and low cost today and in the future. Cost is a generic term we use as a placeholder for whatever other design criteria that matters beyond performance (e.g., power, packaging costs, power supply costs, and so on). A holistic, integrated, and collaborative solution enables cost-effective processor design and operation even in the presence of variations. By having the higher layers influence or mitigate problems at the circuit layer, the resulting costs and efficiencies help track future increases in sustained performance by maintaining the price-to-performance ratio—an important principle in the commercial sector.

In order to demonstrate that hardware and software codesign dampen the impact of variations successfully, this paper specifically examines voltage variation as a case study. Shrinking feature size and diminishing supply voltage make circuits more sensitive to supply voltage fluctuations within the core, caused by workload activity changes. If left unattended, large voltage fluctuations can lead to timing violations or even transistor lifetime issues. Designers are faced with either increasing the cost of the chips by engineering them so that the hardware (including packaging and power supplies) tolerates sustained peak execution performance under extreme operating conditions, or foregoing such costs by lowering the operating efficiency of the processor. The alternative we see is to instead include hardware-based (circuit and microarchitecture) throttling mechanisms that sacrifice performance but only when operating conditions stray too far from nominal. The approach heads in the right direction in that it provides hardware guarantees that catastrophic events will never occur. However, hardware-only solutions are reactive, lack global perspective, and may be difficult to implement efficiently. Thus, the system relies on the hardware for immediate (albeit suboptimal) reaction to emergencies and relies on the global view provided by the chipset level software to eliminate repeated occurrences, which is a much more efficient and long-term solution.

The findings and discussion in this paper are a summary of prior work. This includes work done by others in the field, as well as our own research efforts geared toward resilient architecture designs in the presence of voltage variation. The rest of this paper is structured as follows. We begin with a brief introduction and background to parameter variations in Section II. We discuss how industry builds robust processors even in the presence of variations. We go onto discussing the challenges of sustaining what the industry does today in future nodes. Then Section III explains why dealing with variation at higher levels of abstraction mitigates impending challenges. In each of the sections that follow, Sections IV–VI, we explain the importance of tolerance, avoidance, and elimination, respectively. Tolerance, avoidance, and elimination are the principles behind the success of our system. We believe these are generalizable constructs that can extend over to other disciples in reliability as well. Section VII broadens our discussion to other reliability challenges, identifying remaining work to be done, and thus providing a path for future work along this direction. Finally, Section VIII summarizes our vision and concludes this paper.

II. 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 variation problems. Parameter variations can be broadly classified into device variations incurred due to imperfections in the manufacturing process and environmental variations due to fluctuations in on-die temperature and supply voltage. These variations greatly affect the speed of circuits in a chip; delay paths may slow down or speed up due to these variations. Consequently, they affect manufacturing yield and runtime efficiency.

A. Variations

There are three categories of variations: process, voltage, and thermal (PVT) 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. Therefore, it is static. Voltage and thermal variations are more dynamic, affecting chip operation at runtime. 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 hot spots within the microprocessor. Pronounced hot spots within a core can cause a chip to wear out early and stop functioning correctly. Voltage variation or voltage noise results from non-ideal power distribution. 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.

B. Worst-Case Design

Traditionally, designers have coped with parameter variations through careful design and testing, allocating sufficiently

large margins or tolerance guardbands. Margins compromise 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.

In an effort to better understand the impact of worst-case design, here we examine voltage noise (also referred to as dI/dt) as a specific case study. Efforts at reducing processor power, and improving performance-per-watt, have the unfortunate side effect of causing large current variations within the processor. Clock gating is one example of such an innovation. Roughly speaking, clock gating is promoted as a technique for reducing the power requirements of modern processors. Unfortunately, it involves sudden and large current transitions. Due to parasitic inductance in the power-supply network, these current oscillations may lead to undesirable swings in the microprocessor's core supply voltage. The voltage noise problem can result in supply voltages that violate the minimum or maximum voltage thresholds for the processor. This can potentially cause timing problems in a microprocessor and result in incorrect calculations [1]. Designers must take several precautions to ensure such voltage emergencies never occur. Current designs prevent dangerous voltage emergencies via careful allocation of large voltage margins, placement of decoupling capacitors, and advanced floorplanning.

1) *Voltage Margins:* Today's production processors use operating voltage margins that are nearly 20% of nominal supply voltage [2]. However, conservative designs either lower the operating frequency or sacrifice power efficiency. As feature sizes shrink and nominal supply voltage scales down gradually with limited threshold voltage scaling, circuit delay sensitivity to margins increases with each technology node.

Fig. 2 plots peak frequency at different voltage margins across four PTM [3] technology nodes (45 nm, 32 nm, 22 nm, and 16 nm) based on detailed circuit-level simulations of an 11-stage ring oscillator consisting of fanout-of-4 inverters. The plot shows that at today's 32 nm node, a 20% voltage margin translates to a 33% frequency degradation, and at future technology nodes the situation gets much worse. Practical limitations on reducing power delivery impedance combined with large current fluctuations make margin-based solutions unsustainable.

Trends indicate that margins will need to grow to accommodate worsening peak-to-peak voltage swings. Consequently, as we go into the future, designers must increasingly compromise peak performance for growing worst-case delays. Fig. 3 shows the worst-case peak-to-peak swing in future generations relative to today's 45 nm process technology. This data is based on simulations of a Pentium 4 power delivery package model [5], assuming nominal voltage gradually scales according to ITRS projections from 1 V in 45 nm to 0.6 V in 11 nm [6]. To study package response, current stimulus goes from 50 A to

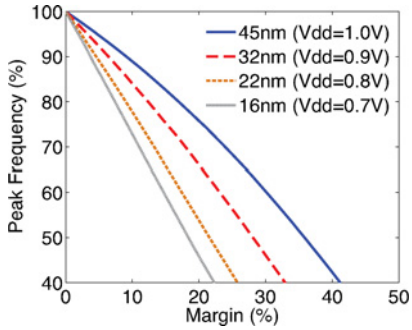

Fig. 2. Worst-case margins are a growing source of processor inefficiency [4].

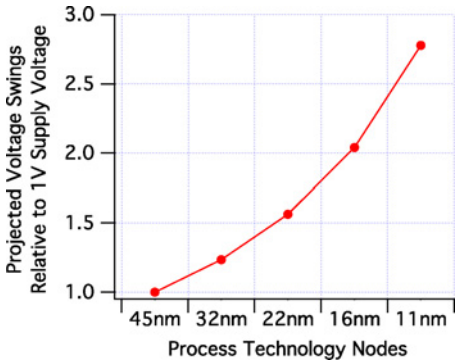

Fig. 3. Voltage noise is a growing problem in future process generations, as the peak-to-peak swings are increasing [7].

100 A in 45 nm. Subsequent stimuli in newer generations is inversely proportional to Vdd at the same power budget. Voltage swing doubles by the 16 nm technology node. Future processor performance and power efficiency will suffer to an even greater extent than in today's systems.

2) *Decoupling Capacitors:* To dampen peak-to-peak voltage swings and to keep voltage margins within some reasonable bounds, processor designers rely on package and on-chip decoupling capacitance. These capacitors attempt to maintain low impedance over a range of frequencies. Bulk capacitors on the motherboard dampen low-frequency noise, while package capacitors target mid-frequency noise between 50 and 200 MHz that is caused by impedance in the power delivery network. Lastly, on-chip decoupling capacitance targets high frequency noise caused by sudden sharp changes in current due to dynamic clock gating of idle functional units. Fig. 4 illustrates the distribution of these capacitors over the different types of voltage droops.

Traditionally, designers have been using oxide capacitors, but industry is making advances toward integrating deep-trench decoupling capacitors into logic circuits. Deep-trench capacitors provide significantly more capacitance per unit area than oxide capacitors. However, their primary drawback is the cost and amount of area necessary. For example, the Alpha 21264 reports that roughly 15%–20% of the die area is occupied by decaps [8]. Deep-trench capacitors will also add to the cost of a chip due to the costly manufacturing process and will exacerbate the already problematic leakage power problem in processors.

3) *Floorplanning:* Modules within the processor do not exert uniform current demands. Some modules consume sig-

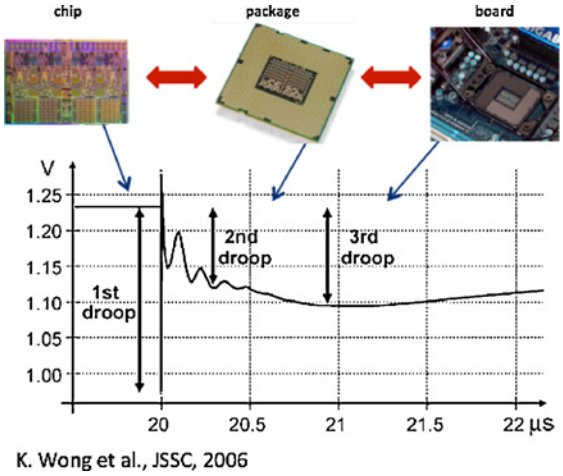

Fig. 4. Sources of capacitance for the three primary types of voltage droops.

nificantly more power than others. It is critical to ensure that such modules have a low impedance path on the power delivery grid. Similarly, it is also important that designers do not place high power modules that are likely to simultaneously switch on or off close together. Such placement could lead to a sudden large current swing in a short amount of time, causing a voltage emergency.

It is possible to engineer a floorplan that is resistive to such voltage noise by distributing the current demand of modules more regularly across the processor [9]–[11]. Because modules within the processor do not have uniform current demand, designers can exploit this information to place high-current modules spatially far apart from one another by pairing them with low power modules.

Overall, margins, decoupling capacitance, and floorplanning all help make the processor robust against voltage noise. However, such static solutions require careful preplanning. For instance, at present the only quantifiable methodology that strongly establishes the amount of decoupling capacitance required involves planning for the worst-case voltage swing. Such pessimistic design lowers the overall operating efficiency of the processor. The traditional means of dealing with voltage noise discussed in this section are already being stretched to their limits. Continued scaling trends will only make voltage noise a more serious problem for the community to address. As performance, power efficiency, area, and cost become more important, new and more cost-effective solutions will become necessary to cope with all forms of variations.

III. Resilient Architectures via Collaborative Design

The problem with addressing emergencies using the circuit-level techniques discussed in the previous section is that such techniques are inflexible. The solutions that designers put in place are not adaptable once the chip leaves 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 design strategies lead to worst-case design that is not representative of typical case execution. Worst-case design overly penalizes chip performance and power efficiency for infrequent corner cases, rather than optimizing the chip for the typical case behavior.

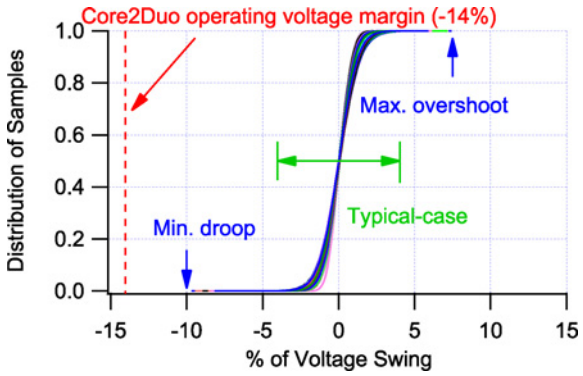

Fig. 5. Cumulative distribution of voltage samples across 881 different executions [7].

Consider our case study from the previous section—worst-case design for voltage noise. The worst-case operating voltage margin that the industry uses today is overly conservative. Prior work determines this from 881 benchmarking runs on an Intel Core2 Duo, as they unintrusively sense voltage near the silicon via isolated low-impedance processor pin connections [7]; the processor package exposes two pins for this purpose called VCC_{sense} and VSS_{sense}. On the basis of undervolting experiments, this processor is estimated to have a worst-case voltage margin of 14%. The experiments include a spectrum of workload characteristics: 29 single-threaded SPEC CPU2006 workloads, 11 Parsec [12] programs and 29×29 multiprogram workload combinations from CPU2006. Therefore, we believe that the conclusions drawn from this comprehensive investigation are representative of production systems and not biased toward a favorable outcome.

Fig. 5 shows a cumulative histogram distribution of voltage samples for the Core 2 Duo processor. The figure shows the deviation of each voltage sample relative to the nominal supply voltage. Each line within the graph corresponds to a run. Runtime voltage droops are as large as 9.6% (see min. droop marker). Therefore, the 14% worst-case margin is necessary. However, they occur very infrequently. Most of the voltage samples are within 4% of the nominal voltage. The typical-case marker in Fig. 5 identifies this range. Only a small fraction of samples (0.06%) lie beyond this typical-case region. Therefore, 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. Abstracting Circuit-Level Challenges to the Architecture

An alternative approach to worst-case design is to design the processor for typical-case operating conditions and to add a fail-safe hardware mechanism that guarantees correctness in case of behavior outside of typical case conditions. Handling emergencies using such a strategy can improve performance, assuming the cost of using the fail-safe mechanism is not too high. For example, using a 4% typical case voltage margin for the Core 2 Duo processor under test in Fig. 5 would translate to 15% improvement in clock frequency, assuming a $1.5\times$ voltage to frequency scaling factor [13]. As technology scaling

continues, these scaling factors will increase (see Fig. 2), thereby offering even better performance improvements as we tighten the voltage margin.

However, active emergency prevention is necessary if aggressive operating margins are sought to push performance. Solutions involving the processor architecture layer are better than pure circuit techniques because architectural techniques are more dynamic. Architecture techniques are feedback driven, observing runtime 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 as a consequence, reap better efficiency from the chips.

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 is incorrect.

B. Involving the Software Layers

Although architecture-level solutions are dynamic and enable design for typical case operation, they lack the global perspective of software. Software can reconfigure 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 coscheduled together in a multicore chip. Therefore, in addition to innovating solutions at the architecture layer, software can mitigate emergencies as well.

In the context of this paper, we constrain our definition of software to those systems that operate directly below all other code, including the operating system (OS) and the BIOS. These software systems do not make any assumptions about the higher-level code, nor do they rely on any external assistance. They dynamically translate the instruction stream online as the target software executes, thus they are completely transparent to the conventional software stack, like OS and application-level code. Such software systems already exist in production environments. Examples include Transmeta's Code Morphing Software [14] and IBM's DAISY [15].

Eliminating emergencies via software improves the overall performance of the processor. The architecture needs to recover and roll back less frequently owing 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 runtime performance.

Another reason for relying on software is that the improvement we observe through architecture-level solutions is a function of emergency frequency. As technology scaling trends continue, variation trends are likely to worsen, so emergencies will be more pronounced. Therefore, more aggressive utilization of the architecture's dynamic fail-safe mechanism will be necessary. Because the fail-safe mechanism is a runtime feature, performance at execution time will suffer if the number of emergencies increases. 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 [16]. Google engineers assume that hardware failures are inevitable and write their code accordingly. System failures do not affect their application's correctness or quality of service. The 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 reliability is simply an extension of present-day application-level reliability as in the case of Google.

C. Alarm-Based Computing

As we have seen from the introduction of this section, most programs experience minor voltage swings relative to the extreme operating voltage margin. Designing the processor for those infrequent corner cases severely penalizes the common case. Therefore, compared to the industry-wide route, we advocate taking a radial route to the problem of allowing emergencies to occur. In an alarm-based computing platform, an emergency acts as an indicator to the system, signaling it that dangerous activity is occurring at the lowest level of the execution engine. The alarm-based computing system tolerates emergencies infrequently, while eliminating frequently recurring emergencies using patterns in emergency behavior of a running code. This vision is implemented in both hardware and software. In the process of building a system, we answer several fundamental research questions that demonstrate and validate the contributions of our collaborative design argument, such as the following.

1) What information should the circuit layer provide to the microarchitecture layer?
2) How should we design the microarchitecture so that it can tolerate emergencies gracefully without severe penalties?
3) What information should the microarchitecture propagate to the software layer?
4) What should the emergency elimination software layer look like?
5) What techniques should the software utilize to eliminate emergencies?

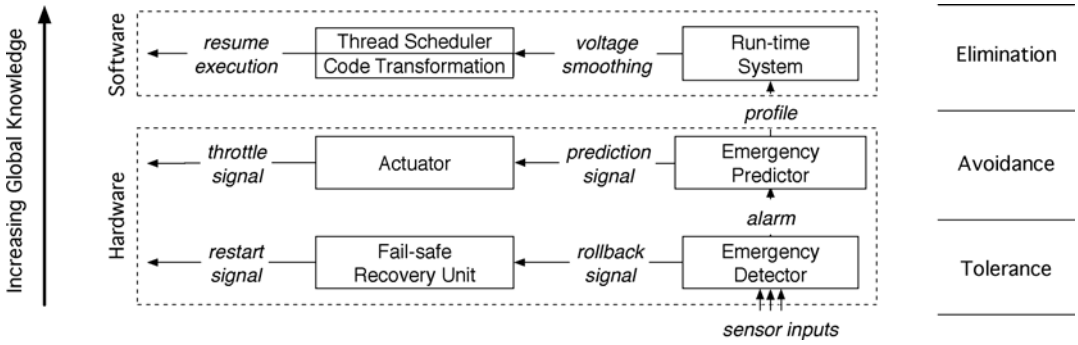

Fig. 6. Our alarm-based computing system is comprised of three paradigms: tolerance, avoidance, and elimination. Various techniques that mitigate voltage emergencies exist either at the hardware or software only layers, or that rely on both. A large body of prior work also falls into this general structure.

Fig. 6 illustrates the overall design of our system for coping with voltage variation. The system has an emergency detector (hardware) that triggers a fail-safe recovery unit (hardware) to rollback execution whenever it detects an emergency. The detector is simply a distributed set of voltage sensors, monitoring for voltage emergencies. Upon detecting a margin violation, the detector raises an alarm signaling an emergency predictor (hardware) with 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 runtime system (software) that eliminates recurrences of that emergency. The runtime software layer eliminates the emergency either via code transformation (using compiler techniques) or by invoking a thread scheduler to coschedule the suffering thread with an alternative program. The latter is useful in multicore 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. It acts as a low-penalty fail-safe.

This scheme is a three-tiered approach, including tolerance, avoidance and elimination. As we go up the layers, there is more global knowledge of underlying activity. This knowledge increases the system's ability to fix emergencies effectively and more permanently at each layer. Because these three principles are abstract notions independent of any specific implementation, in the following sections we present various techniques that exist in literature at each layer. We summarize the contributions and provide critical insights.

IV. Tolerance

Tolerating, or allowing emergencies to occur, is useful both for tightening margins, and observing the emergency behavior of running code. To enable this, the architecture must support a built-in mechanism that allows voltage emergencies to occur, but when they do, it recovers processor state and resumes executions. By tolerating emergencies initially, we can subsequently learn to avoid, as well as eliminate them altogether. In this section, we empirically understand the activity leading to emergencies. We also discuss related work in this context and discuss what tolerance allows us to learn about emergencies.

A. Mechanism for Allowing Emergencies

Razor [17] is one of the most well-known approaches for tolerating variation errors in the field. It relies on a combination of architecture and circuit techniques to detect and recover from errors. In Razor, each pipeline stage is shadowed by another latch that triggers after some constant delay, such that the shadow latch always captures the correct logic value. Razor signals an emergency when the main latch value differs from the shadow latch, in which case execution is replayed safely using the value from the shadow latch to ensure forward progress. Because of this ability to detect and recover from errors, designers can optionally reduce the supply voltage to significantly lower power consumption, or alternatively boost up the clock frequency for better performance, either way maximizing performance per watt. However, Razor's shortcoming is that it is severely intrusive, requiring intimate knowledge of critical path delays and changes to traditional microarchitectural structures at the expense of burning additional power. Moreover, it adds area and cost overheads, making design and validation even more complicated than they already are in today's systems.

Alternatively, checkpoint-rollback mechanisms have been proposed to guarantee fail-safe execution. Checkpoint-rollback mechanisms have been proposed for handling soft errors [18]–[20]. They support execution rollback in the presence of an error. They are already available in existing production systems [21], [22], and more novel applications of this general-purpose hardware are continuously emerging [20], [23]–[27]. Extending their service to voltage noise is yet another addition.

General purpose checkpoint recovery is good from a reusability standpoint. However, the cost of relying only on coarse-grained checkpoint recovery is prohibitively expensive. Therefore, tolerating emergencies is not always possible. Even assuming optimistic checkpointing intervals (between 100 and 1000 cycles), traditional checkpoint-recovery schemes translate to unacceptable performance penalties [28]. Therefore, it is only feasible to rely on this mechanism infrequently.

Gupta *et al.* [28] proposed an alternative recovery scheme to more effectively handle voltage emergencies. Their implicit checkpointing scheme tolerates all voltage emergencies

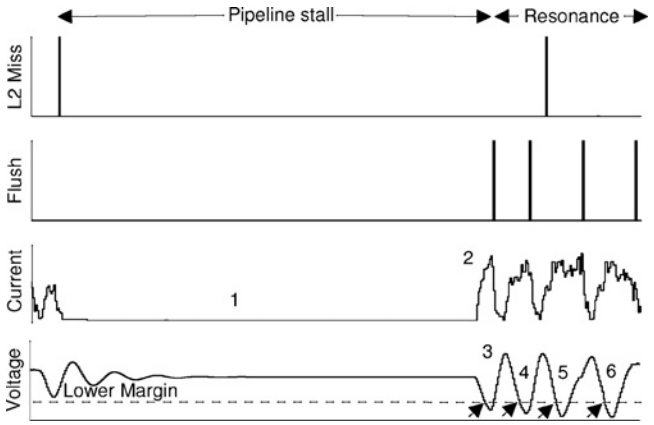

Fig. 7. Snapshot of *art* from the CPU2000 benchmark suite over 430 cycles. The snapshot illustrates how microarchitectural event induced pipeline stalling and resonance activity can lead to emergencies (indicated using arrows) [32].

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).

Similar to Razor, such a scheme is intrusive and forces changes to predesigned and validated microarchitectural structures, thus requiring additional design, testing, and revalidation of prior logic blocks. General-purpose design considerations are important because we are targeting the commodity processor market segment, where the cost of a processor can have implications on market share.

B. Learning from Emergencies

To be able to avoid and eliminate emergencies, we need to find leading indicators of dangerous voltage fluctuations. Prior work considers several microarchitectural parameters, such as the number of entries in the reorder buffer, the instruction fetch queue, and the load/store queue, along with microarchitectural events such as cache misses and pipeline flushes [29]. On the basis of their findings, here we summarize the perturbation effects of microarchitectural events on processor activity using real program examples and show they can lead to voltage emergencies. We also discuss patterns in activity that allow us to not only identify emergencies uniquely, but also predict their recurring occurrences [29]–[31].

First, we discuss the effect of individual microarchitectural events on current and voltage. Fig. 7 shows a snapshot of pipeline activity for benchmark *art* from SPEC CPU2000 over 430 cycles. Microarchitectural events for the cache and branch predictor are illustrated along with current and voltage activity of the processor. In the pipeline stall part of the figure, we observe an L2 cache miss (illustrated by a vertical bar in the L2 Miss sub-graph). During the time it takes to service the L2 miss, pipeline activity ramps down, as seen in the current profile (marker 1). However, after the L2 miss data is available, functional units become busy and there is a sudden increase in current activity (marker 2). This steep increase in current causes the voltage to temporarily dip below the voltage margin (marker 3) because of parasitic inductance in the power delivery subsystem.

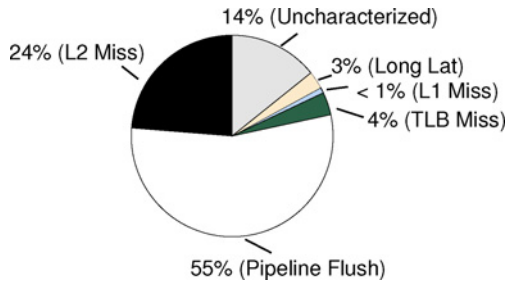

Fig. 8. Distribution of processor events and operations that cause voltage emergencies in the CPU2000 suite [33].

Additionally, microarchitectural events that cause periodic high-current and low-current activity can cause a resonance buildup of voltage, if the period coincides with the resonance frequency of the power delivery subsystem. The resonance portion of Fig. 7 illustrates multiple pipeline flush events occurring in close proximity to one another. Pipeline flush events cause a sudden decrease in activity and are followed by a rush of activity as instructions are rapidly issued along the correct program. The close distribution of these events leads to a resonating effect that results in rapid fluctuations in current. These successive fluctuations not only cause the voltage to swing, but also progressively increase in amplitude from one event to another (markers 4, 5, and 6).

Associating a specific microarchitectural event with an emergency requires us to apply our knowledge of how microarchitectural events affect processor activity. Identifying the root cause of an emergency is not as simple as merely looking at the most recent microarchitectural event prior to the emergency. Consider the emergency at marker 5, which occurs slightly past an L2 miss event. Unlike the L2 miss event under pipeline stall, this L2 miss event is not responsible for the emergency. L2 miss events take a few hundred cycles to complete, and instructions pending on that data are not issued until the event completes. Therefore, the burst of current activity does not correspond to this pending L2 event; rather, it corresponds to the pipeline flush preceding the L2 event.

Prior work devised an algorithm to identify root causes [29]. The algorithm scans recent processor events in a fixed-priority order looking for event completion times that coincide with the time of the emergency. It scans down the list of L2 misses, translation lookaside buffer (TLB) misses, pipeline flushes, L1 misses, and long latency operations, in that order. To show the strength of the relationship between these processor events/operations and emergencies, Fig. 8 shows the percentage of emergencies caused by different root causes for the SPEC CPU2000 benchmark suite. A majority of the emergencies are caused by pipeline flushes and L2 misses. The uncharacterized 14% is due to emergencies that cannot be uniquely attributed to a single root cause, such as the resonance case illustrated in Fig. 7.

Whether an event at a particular location causes an emergency may depend on activity just before and after the event. Even a small loop like that in Fig. 9(b), extracted from benchmark *gcc* of SPEC CPU20006, can have behavior phases with markedly different activity patterns. Fig. 9(a) is a snapshot of activity within that loop over 880 cycles. It

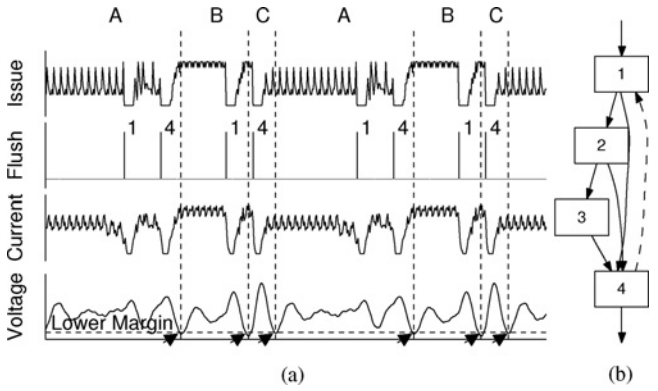

Fig. 9. (a) Voltage emergencies are associated with recurring activity (phases A, B, and C) over 880 cycles. The numbers next to the vertical bars in the flush graph correspond to the basic block number in (b) containing the mispredicted branch. (b) Emergency-prone loop from function `init_regs` in *gcc* from CPU20006 benchmark suite. Its activity snapshot is shown in (a) [4].

shows three repeating phases of the loop. Phase A uses paths $1 \rightarrow 4$ and $1 \rightarrow 2 \rightarrow 4$, while phase B uses only path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. The issue rate of phase A is relatively low, while that of phase B is quite high. The flush events labeled 1 are caused by branch mispredictions at the end of basic block 1. Those events in phase B, where the issue rate is high, always cause emergencies. Those same events in phase A never do. Therefore, tracking program flow and microarchitectural events yields a proxy for the activity leading to emergencies. Our findings are similar for events in Phase C that correspond to events in Phase A.

V. Avoidance

In addition to tolerating emergencies, researchers have proposed means for avoiding emergencies by using heuristics that *predict* emergencies. When a prediction is made and an emergency is impending, the predictor triggers an actuation mechanism that throttles machine execution, allowing processor voltage to smoothly recover back to nominal conditions. In this way, these predictors avoid emergencies. Some schemes use simple heuristics, such as a threshold crossing, while others use history, including microarchitectural and program state, to anticipate emergencies. Here we describe these schemes, addressing their limitations and feasibility, starting from simpler and moving toward more complex schemes.

A. Sensor-Based Schemes

A number of throttling mechanisms have been proposed to dampen sudden current swings, including frequency throttling, pipeline freezing, pipeline firing, issue ramping, and changing the number of the available memory ports [34]–[37]. These typical sensor-based proposals rely on a tight feedback loop like that shown in Fig. 10(a). 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," or a predictor. Crossing that threshold means that voltage is approaching its lower margin, so the actuator predictively turns on throttling until the crisis is past.

However, such mechanisms require a tight feedback loop that detects an imminent violation and then activates a throttling mechanism to avoid the violation. The detectors are either current sensors or voltage sensors that trigger when a soft threshold is crossed, indicating a violation is likely to occur. 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.

Fig. 10(b) 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 Fig. 10(b), the voltage starts to recover under throttling, but not in time to avoid crossing the lower operating margin.

Fig. 11(a) 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. The baseline system resembles a Pentium 4 configuration. 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. Yet even a 2-cycle delay causes 50% of all soft threshold crossings to violate the simulated microprocessor's minimum operating margin specification. This is even when we assume all levels of decoupling capacitance (i.e., on-chip, package, and board capacitance).

In the absence of a fail-safe mechanism to tolerate emergencies, these simpler predictors are ineffective because of sensor delays. A single emergency could lead to catastrophic core failure. However, it is possible to leverage this simple generic scheme by backing it up with fail-safe checkpoint recovery mechanism. Unfortunately, because of the significantly large number of missed emergencies, performance improvement will poor because fail-safe recovery penalties diminish potential gains.

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 Fig. 10(c) gives the throttling system more time to prevent an emergency. But as the uncorrected execution waveform in Fig. 10(c) 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. Fig. 11(b) shows that this penalty can be quite large. Assuming an ideal sensor with no

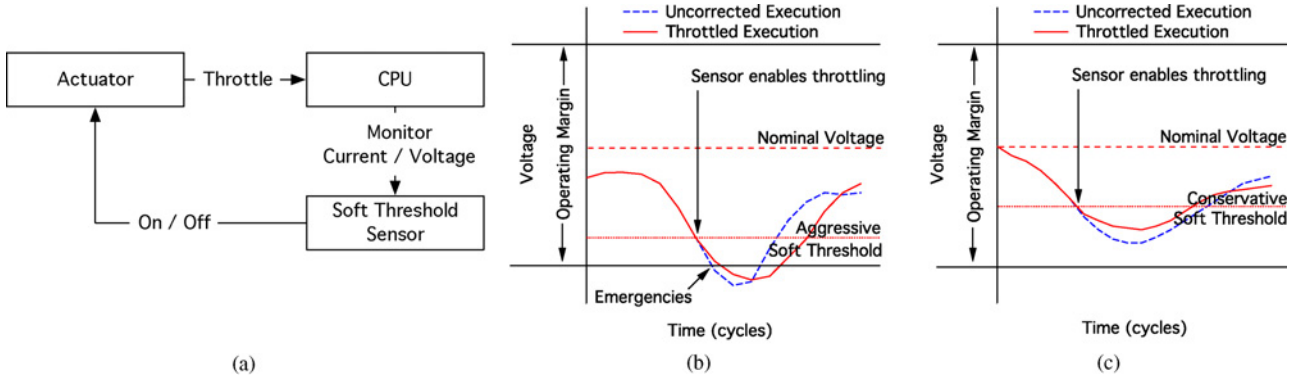

Fig. 10. Sensor-based throttling. (a) Feedback loop is intended to detect and prevent emergencies. (b) Aggressive soft thresholds allow too little time to prevent emergencies. (c) Conservative soft thresholds trigger unnecessary throttling [31].

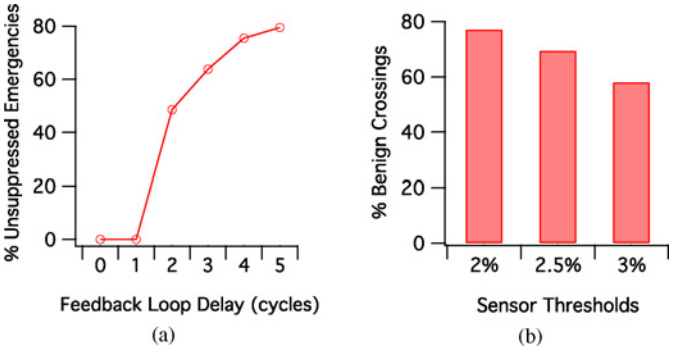

Fig. 11. Implications of feedback loop delay and soft threshold settings on correctness and performance. (a) Large percentage of emergencies are not detected with sufficient time to prevent them due to feedback loop delays. (b) Even assuming a 0-cycle feedback loop delay, the number of soft threshold crossings that do not violate the minimum operating margin (i.e., benign crossings) is so large that performance suffers due to unnecessary throttling [31].

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

B. Event-Guided Predictors

Sensor-based predictors operate independently of program or microarchitectural state. Such higher-level information has the intrinsic property that it relates program/machine activity to power supply behavior. The intuition behind this relationship is that processor current draw depends on the set of functional blocks that are active and consuming power during each cycle. The activity of these functional blocks depends on the set of instructions in flight through the core's pipeline, thus relating current draw and consequently voltage flux to higher-level program instruction sequences.

Therefore, an alternative approach is to eliminate the feedback loop associated with previous proposals and instead monitor specific microarchitectural events as indicators of processor activity that can lead to voltage emergencies. In essence, this predictor replaces current and voltage sensing in Fig. 10(a) with microarchitectural event detection. Such an event-driven mechanism triggers corrective action when it detects certain emergency-prone events (L2 cache misses and branch flushes, as they are the events associated with most of the emergencies). A naive implementation might take preventive measures at every such event (e.g., to activate a throttling mechanism at every L2 miss). That would be overly conservative, however; because most such events do not give rise to emergencies, such a system lends itself to a high false-alarm rate of 71% [33].

Instead, by tracking specific instructions associated with events (L2 misses or pipeline flushes) that have caused emergencies, and maintaining contextual information for each event and emergency, it is possible to reduce the false rate significantly. Reacting only to events associated with emergencies results in much less overhead than the naive implementation. Fig. 12 shows a cumulative distribution graph plotting the number of unique program addresses that trigger emergencies and their contribution to the total number of emergencies during execution. Each benchmark except for parser, gcc, twolf, and crafty has fewer than 15 unique program addresses that cause over 90% of runtime emergencies. Thus, the state that an instruction-specific event-guided mechanism needs to maintain can be stored in just a few bytes.

While there are just a few instructions, in a prior work we demonstrated that event predictors are a poor heuristic for whether voltage emergencies are likely in the next few clock cycles [31]. Prediction accuracy can be poor. Single-event prediction accuracy is only about 10%. As we discussed in the tolerance section, the history of activity leading to voltage emergencies is important. A single event predictor does not capture sufficient history for accurate emergency anticipation. Gupta *et al.* [33] also confirmed these results, showing that throttling is a poor adaptation mechanism for the event-guided scheme described in this paper.

C. Signature Predictor

To overcome the limitations of event-based predictors, researchers have proposed a voltage emergency predictor that identifies when emergencies are imminent and prevents their occurrence by predicting them using signatures [31]. An emergency predictor predicts voltage emergencies using emergency signatures and throttles machine execution to

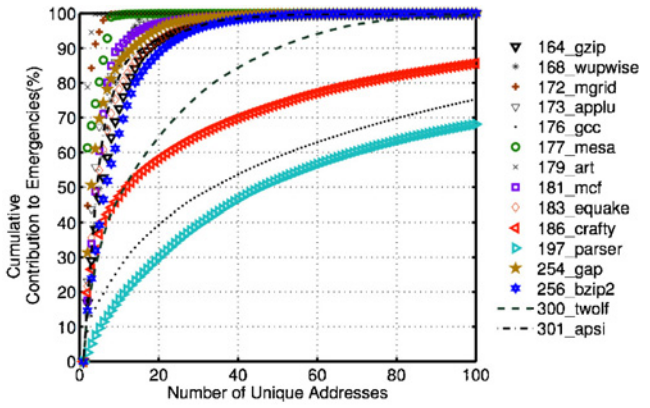

Fig. 12. Number of unique instructions causing emergencies and their corresponding contribution to the total number of emergencies [33].

prevent them. 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. Tolerance is an integral part of this predictor. Post emergency, a fail-safe checkpoint-recovery mechanism 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 to maximize performance, even in the presence of emergencies. The cost of signature-based throttling is fewer than ten cycles, which is much cheaper than the thousands of cycles that it costs to rely on the general-purpose checkpoint-recovery mechanism.

For this predictor to be effective, the predictor must anticipate an emergency accurately and do so with sufficient lead time for throttling to take effect. Signatures predict emergencies with an average accuracy of 93% across the entire spectrum of CPU2006 benchmarks, as illustrated via the 0-cycle lead time bar in Fig. 13. A lead time of 0 cycles optimistically assumes there is no delay to actuate throttling to prevent an emergency, thus representing an ideal scenario. However, real systems require non-zero lead times to account for circuit delays, as we discussed in the previous section. As lead time increases, Fig. 13 shows that accuracy degrades just slightly from 93%. Even with 16 cycles of lead time, which is ample time to predict and prevent an emergency, accuracy remains high at 90%.

Throttling cannot prevent all emergencies, even if prediction accuracy is high. In such cases, the fail-safe checkpoint-recovery mechanism recovers processor state, albeit at much higher penalty. But the authors find that the number of such emergencies is only 1% of all emergencies that occur without throttling. Therefore, the associated total recovery penalty is very low in our quantitative analysis.

An aggressive reduction in operating voltage margins translates to higher performance or better energy efficiency. However, benefits are offset to some degree because of throttling penalties to prevent emergencies and checkpoint-recovery rollbacks to train the predictor. In simulations of a representative superscalar microprocessor in which fluctuations beyond 4% of nominal voltage are treated as voltage emergencies, a signature-based predictors shows great promise. Based on a 1.5× relationship between voltage and frequency at the PTM 32 nm node [3], we observe an ideal performance gain of 14.2% using an oracle throttling scheme (see Fig. 14). By comparison, the voltage emergency predictor comes to within 0.7 percentage points of the ideal scheme, assuming infinite or unbounded resources to implement the predictor. But even under strict physical resource constraints, an intelligent bloom filter-based predictor ranging in size between 4 kB and 32 kB delivers substantial gains.

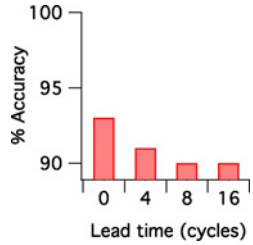

Fig. 13. Prediction accuracy of a signature-based predictor is high even when predicting cycles ahead of time [31].

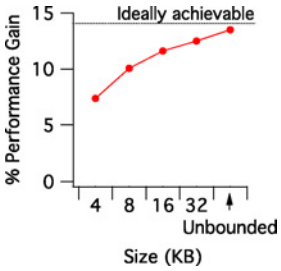

Fig. 14. Signature-based predictor enables substantial gains using an aggressive 4% voltage margin [38].

An added benefit of signature-based emergency prediction is that the predictor does not require fine tuning based on specifics of the microarchitecture nor the power delivery subsystem, as is the case with sensor-based predictors. The current and voltage activity within a microprocessor are products of machine utilization that are specific to running workload dynamic demands. Capturing that activity in the form of emergency signatures allows the predictor to dynamically adapt to the 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.

VI. Elimination

Software can eliminate emergencies altogether. Joseph *et al.* [35] pointed out that the most problematic processor current profiles include successive periods of high and low processor activity. It is when these high and low durations approach the resonant frequency of the power-supply network that the problem becomes more serious. Joseph *et al.* demonstrated this by developing an artificial application that was hand tuned to simulate periods of high and low activity that matched the

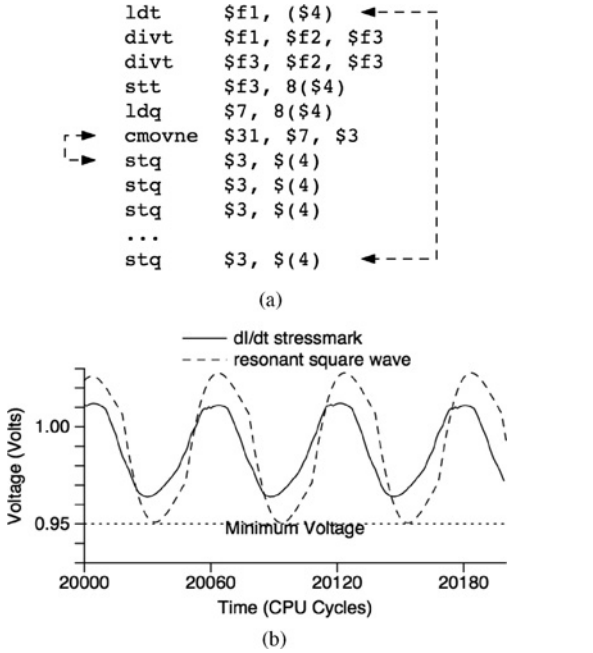

Fig. 15. (a) dI/dt stressmark. (b) Voltage swing of the dI/dt stressmark versus peak swings at resonance [35].

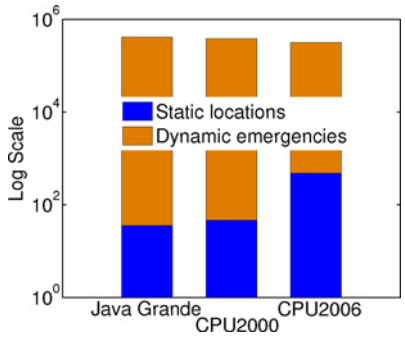

Fig. 16. Few hundred static program locations cause hundreds of thousands of emergencies.

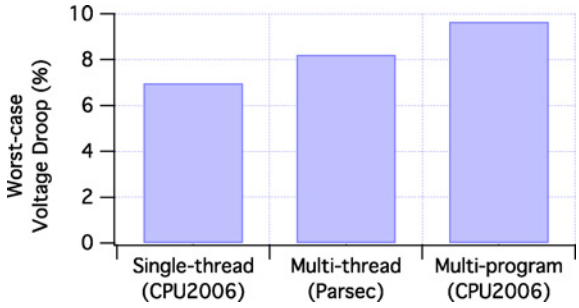

Fig. 17. Worst-case droop increases during multithreaded and multiprogram execution compared to single-thread execution.

resonant frequency of the processor's power-supply network. This synthetic benchmark, shown in Fig. 15(a), contains only a single loop body, yet it consistently causes voltage swings dangerously large enough to violate the minimum voltage margin [as shown in Fig. 15(b)]. The loop body oscillates between very low current activity (because divt produces long stalls) and high current activity (in which dependent instructions store divt result to memory, reread it, and restore it to registers). This software code loop provides motivation for software-based solutions because if such loops exist in real applications, it is logical to apply a permanent solution at the application level, thereby limiting the performance penalty of activating control hardware. Such cases have been found in real programs [39].

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. As an example, consider Fig. 16. It shows the number of distinct instructions responsible for emergencies, and the total number of emergencies they cause over the lifetime of a program across three different benchmark suites. In each case, only a few hundred instructions on average are responsible for hundreds of thousands of emergencies. So, a few emergency-prone code regions are responsible for almost the whole emergency problem. Prior hardware techniques must enable their mechanisms at least once per dynamic emergency. Software, in turn, can exploit the fact that there are so few emergency-prone locations, and that emergency-prone behavior is so repetitive. By using the history of activity that leads to emergencies at these locations, compilers can intelligently reshape code to eliminate recurring emergencies altogether.

In the context of multicore systems, intelligently scheduling threads to dampen voltage swings is likely to become important. As the number of cores sharing a power supply source increases, the absolutely peak-to-peak voltage swings may also increase due to interfering microarchitectural activity across hardware contexts. Fig. 17 quantitatively confirms this likelihood using data gathered on a Core 2 Duo processor. The figure shows the magnitude of the worst-case voltage droop is larger during multithreaded and multiprogram execution than during single-threaded execution. Therefore, in this section we also discuss thread scheduling for voltage noise.

A. Compiler Techniques

We claim that the hardware-based solutions work well for intermittent voltage emergencies, but a loop incurring repeated voltage deviations is best handled by a compiler. Consider a frequently executing loop that experiences recurring emergencies during every iteration of the loop simply because the program is taking the same error-prone code path every iteration. 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.

Currently, production static compilers do not account for voltage fluctuations when scheduling instruction sequences. While techniques can and have been developed to produce

power-efficient code by the static compiler, it would be difficult to extend these static optimizations to solve the voltage noise problem. There is a lack of comprehensive understanding about instruction sequences that result in voltage fluctuations.

However, Toburen [40] and Yun and Kim [41] made some initial progress in this direction. Toburen's approach builds an instruction schedule that limits processor power dissipation during each cycle. The power-aware scheduler places as many instructions as possible in a given VLIW instruction bundle until a given power threshold is reached. Often, high-energy instructions are not scheduled together because they can result in large and sudden current spikes. Such instructions are instead dispersed slightly from one another by exploiting scheduling slack, which is typically available if the compiler produces sufficiently large enough code regions. In this manner, the compiler generates a uniform dI/dt curve that decreases the processor's average peak power consumption each cycle. Similarly, Yun and Kim proposed a power-aware modulo scheduling algorithm for high-performance VLIW processors. Their proposed algorithm reduces both the step power (the effect that causes voltage noise) and peak power by constructing a more balanced parallel schedule that does not sacrifice performance.

Even if static compiler algorithms were developed for locating potentially dangerous instruction sequences, the decision on whether or not to intervene would depend on characteristics of the underlying power-supply network and the operating voltage range of the target processor, which typically are not known at static compile time. Therefore, these static optimizations are not easily retargetable across different combinations of platform and application. Finally, static techniques may not avoid all voltage emergencies, because many of the emergencies occur because of dynamic instruction sequencing, which is difficult to predict prior to program execution.

Dynamic optimization systems [42] are well suited for emergency-specific code transformations, especially in scenarios like "90% of the execution time is spent in 10% of the code." Fig. 16 shows similar behavior with respect to emergencies. By operating in a lazy optimization mode, the dynamic optimizer can wait until it is informed by the hardware of a voltage emergency (after the hardware activates control mechanisms to eliminate the emergency), and it can then reoptimize and cache a version of the code that exhibits more voltage stability. In the ideal case, only one iteration of a power-virus loop would require hardware intervention, and the remaining iterations would be executed from the software-based dynamically-optimized code cache. Because this scheme dynamically discovers emergency hot spots, it is more robust for wide-scale deployment. Hazelwood and Brooks [30] were the first to introduce the idea of a dynamic compiler-driven strategy to eliminate a large fraction of emergencies.

A compiler-based issue rate staggering technique has been shown to be effective at eliminating emergencies [43]. The technique reduces emergencies by applying transformations such as rescheduling existing code or injecting new code into the dynamic instruction stream of a program. The runtime code rescheduler uses the root-cause identification algorithm discussed in the tolerance section to decide the kind of code transformation that is best suited for eliminating a specific type of emergency (e.g., pipeline flush, L2 miss). The rescheduler combines this information with control flow extracted from voltage emergency signatures to apply transformations only along certain program paths (when possible) to eliminate the emergency.

Fig. 9(a) illustrates that voltage emergencies depend on the issue rate of the machine. Therefore, slowing the issue rate of the machine at the appropriate point can prevent voltage emergencies. One can achieve the same goal in software by altering the program code that gives rise to emergencies at execution time, and can do so without large performance penalties. The compiler technique that the authors propose exploits 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.

1) *NOP Injection:* A simple way for the compiler to slow the pipeline is to insert NOP instructions specified in the instruction set architecture into the dynamic instruction stream of a program. However, modern processors discard NOP instructions at the decode stage. Therefore, the instruction does not affect the issue rate of the machine. Instead of real NOPs, the compiler can generate a sequence of instructions containing RAW dependencies that have no effect. Because these *pseudo-NOP* instructions perform no useful work, this approach often degrades performance.

2) *Code Rescheduling:* A better way to smooth processor activity is to exploit RAW dependencies already existing in the original control flow graph of the program. This constrains the burst of activity when the machine resumes execution after the stall, which prevents the emergency. Whether the compiler can successfully move instructions to create a sequence of RAW dependencies depends on whether moving the code violates either control dependencies or data dependencies. From a high level, the compiler's instruction scheduler does not break data dependencies, but it works around control dependencies by cloning the required instructions and moving them around the control flow graph such that the original program semantics are still maintained.

Fig. 18 shows how the issue-rate smoothing technique works. The plot shows a slice of program activity corresponding to a loop within benchmark *Sieve* from the Java Grande suite. Fig. 18(a) shows that data dependence on a long-latency operation stalls all processor activity, so the current profile goes flat (marker 1). When the operation completes, the issue rate increases rapidly (marker 2) as several dependent instructions are successively released to functional units. This activity increases draw (marker 3), and as a result the voltage dips below the lower margin (marker 4).

Fig. 18(b) shows activity after the reschedule transforms the code slightly to reduce the issue rate. Because dependent

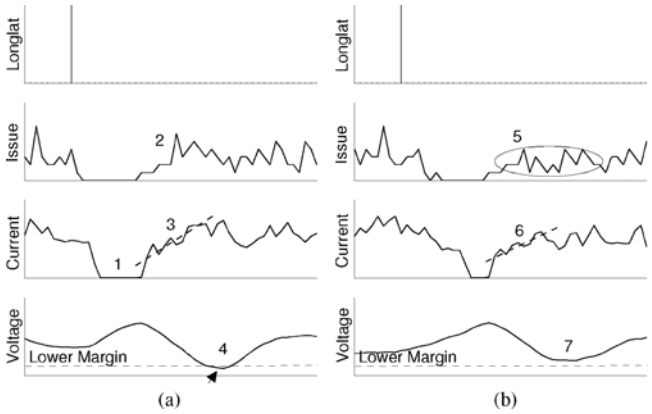

Fig. 18. 50-cycle execution snapshot of *Sieve* from the Java Grande benchmark suite. (a) Pipeline stall on a long latency operation triggers an emergency (indicated by an arrow) as the issue rate ramps up sharply once the operation completes. (b) Code rescheduling slows the issue rate just enough to prevent the emergency illustrated in (a).

instructions are packed more tightly, the issue rate in Fig. 18(b) does not spike as high as in Fig. 18(a) (marker 5). As a result, the processor now draws current less aggressively. The gradient at marker 6 is less steep compared to marker 3. Therefore, the original emergency at marker 4 is now permanently eliminated (marker 7).

Using this one issue-rate constraining technique, the compiler removes over 62% of all emergencies across the Java Grande suite. On average, only 20% of all root causes had to be rescheduled because they contribute to a large percentage (over 98%) of all emergencies. These results indicate that issue-rate smoothing works well for isolated emergencies like the cases illustrated in Figs. 18(a) and 7.

However, there are caveats to code rescheduling. Code rescheduling work best on in-order processors where machine behavior is predictable at the compiler-level. Out-of-order superscalar processors can render such compiler-level techniques ineffective because of low level hardware instruction scheduling. However, prior work indicates that making the RAW dependence chain as long as the issue width of the machine can overcome this hurdle effectively [44].

3) *Other Techniques:* Several other well-known compiler algorithms could also be applied to this problem [29], [30]. For example, when a static compiler schedules instructions, it often has several options for scheduling an instruction without affecting application performance. Thus, the compiler may inadvertently create regions of high and low processor activity simply due to its predefined settings for scheduling instructions in the event of a performance tie. By recognizing these schedule slips, a dynamic optimizer can later apply code motion to move instructions from high-processor to low-processor utilization regions. This can result in the removal of a voltage emergency without degrading the runtime performance of the application.

As we mentioned earlier, Joseph *et al.* [35] described a dI/dt stressmark that can, through a single-loop body of alternating periods of high and low activity, consistently cause voltage emergencies. In the stressmark, low activity is generated by a sequence of long-running sequential divide operations, and high activity is generated by a sequence of parallel operations that stress the functional units and on-chip memory hierarchy.

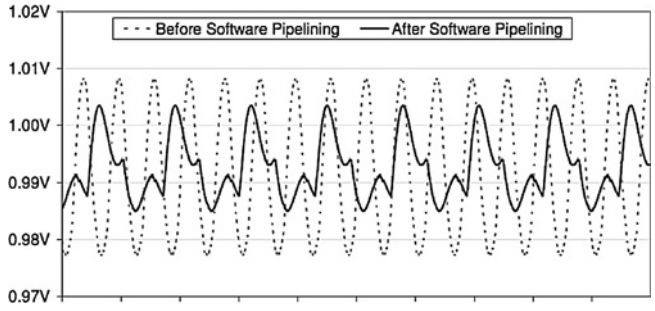

Fig. 19. Effect of compiler-guided software pipelining on a dI/dt stressmark. After software pipelining the system does not experience recurring large voltage swings.

It is possible to leverage existing compiler optimizations to smooth out these periods of high and low activity dynamically. A widely used compiler algorithm for increasing the instruction-level parallelism of cyclic code is software pipelining. By unrolling loops and overlapping the execution of instruction sequences from several loop iterations, the instructions can be scheduled more tightly. Typically, the result of software pipelining is that n-iterations of a loop will be combined to form one larger loop iteration. The nature of the software pipelining algorithm has two interesting side effects. First, the technique allows high-activity periods in one loop iteration to be combined with low-activity periods of the next loop iteration, potentially leading to a more stable sequence of instructions that will often complete faster than the original sequences. Second, by changing the amount of work done in a loop iteration, periods of high and low activity that fall on the resonant frequency will be disrupted. Fig. 19 depicts the result of applying software pipelining to the loop body of the dI/dt stressmark. By unrolling the loop body once, and therefore lengthening the period of low activity originally resulting from three subsequent divide operations, the stressmark shifts off of the resonant frequency. This reduces the resulting voltage fluctuations and eliminates numerous invocations of the fail-safe hardware.

B. Thread Scheduling

As the number of cores per processor continue to increase, and cores continue to share the same power supply source, increasingly one core can either constructively or destructively interfere with other cores, leading to more or less voltage noise. Fig. 20 illustrates an example of destructive interference, where the noise when two cores within a Core 2 Duo processor are simultaneously active is smaller than the noise during single-core execution. The figure also illustrates constructive interference, which is just the opposite. The figure shows aggregate droop activity where both cores, sharing a common power source, are simultaneously running two instances of *473.astar*. The x-axis of the graph refers to the start time offset between the two programs. In other words, the graph is a convolution of two execution windows.

Interfering microarchitectural activity across cores, such as pipeline flushes and cache misses, is the root cause of

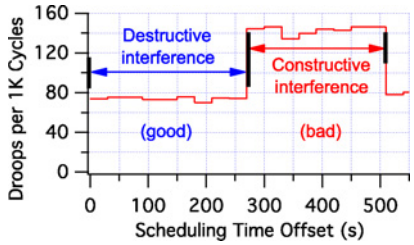

Fig. 20. In multicore systems we observe voltage noise interference [7].

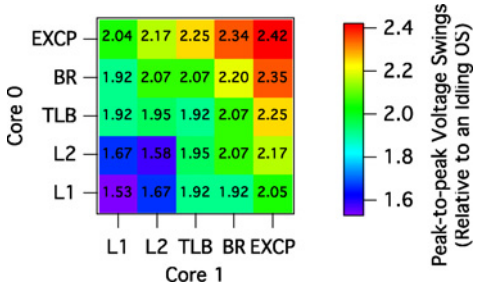

Fig. 21. Root cause of constructive and destructive interference in multicore systems is because of interfering microarchitectural activity [7].

constructive and destructive interference. We can confirm this effect by simultaneously running microbenchmarks on each processor core and capturing the net effect or the magnitude of the peak-to-peak voltage swing across the entire chip. The heatmap in Fig. 21 represents the interference on a Core 2 Duo processor. Both cores are subject to different microarchitectural activity: L1 (only) and L2 cache misses, TLB misses, branch mispredictions (BR) and exceptions (EXCP). The magnitudes of the chip-wide voltage swings are normalized relative to an idling machine. The y-axis corresponds to microarchitectural activity on Core 0 and the x-axis corresponds to activity on Core 1. Depending on the pair of events, we observe that the magnitude of the voltage swing changes.

In such a multicore scenario, a software solution larger than a compiler becomes necessary. Virtual machine monitors or operating systems become appealing, as these systems see and control all threads executing on the underlying 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.

Scheduling of threads has been 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 [45]–[51]. For instance, scheduling two cache intensive programs together is bad, because the cache resource becomes a bottleneck and both programs suffer. Similar to a processor's cache, processor supply voltage is a shared resource as well. In a resilient architecture design, where a fail-safe mechanism provides hardware guarantees, thread interference across cores could cause system-wide performance degradation, as cores tied to the same power plane are likely to share a single recovery unit. Consequently, thread scheduling is on the critical path for performance improvement, similar to cache contention.

Because multiple cores in most commodity modern processors share the power supply, threads could interfere with one another in a manner leading to frequent emergencies. A noise-aware thread scheduler can schedule threads intelligently to minimize the number of emergencies [7]. When activity on one core suddenly stalls, voltage swings due to a sharp and large drop in current draw. However, by maintaining continuous current-drawing activity on an adjacent core also connected to the same power supply, thread scheduling dampens the magnitude of that current swing. In this way, scheduling can prevent an emergency when either core stalls [52].

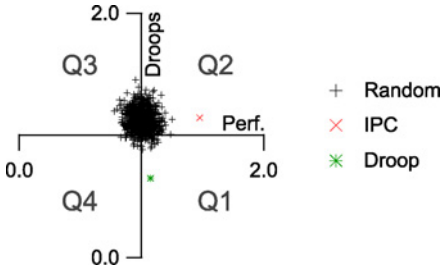

Fig. 22. Evidence that scheduling for voltage noise is different from scheduling for performance. Modified version from [7].

Coscheduling threads to reduce voltage emergencies differs from scheduling for performance. Recall that voltage swings occur primarily because of fluctuations in activity due to stalls. Although scheduling for performance includes eliminating stalls, that same metric does not necessarily guarantee fewer emergencies. Because of this explicit scheduling, voltage noise is necessary. To prove this point, Reddi *et al.* evaluated different operating system scheduling policies, measuring emergencies over the course of a batch job schedule consisting of 50 random jobs. For this selected set of programs, they evaluate a range of scheduling policies, from random selection (random) to maximum performance [instructions per cycle (IPC)] as well as a custom policy for minimizing emergencies (droop).

In Fig. 22, they show performance in terms of IPC versus droops observed over the course of the batch job schedule. Both the y-axis and x-axis of the graph are normalized to SPECrate, which assumes two instances of the same program are running together at the same time. Each marker in the graph is one batch simulation, and they ran 100 random simulations.

The four quadrants in Fig. 22 (Q1 through Q4) help us draw different conclusions from their analysis. 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 emergencies. 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 include simple policies, such as round robin and first-in, first-out, which are effectively random in behavior. From observing data in Fig. 22 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 Fig. 22 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 multicore system. Interactions across threads (or cores) impact the amount of voltage noise we observe. Therefore, a noise-aware scheduler is necessary. Consider the droop metric, or noise-aware scheduling, whose data point resides in quadrant Q4. The noise-aware scheduler focuses on emergency activity and can therefore minimize emergencies across all 50 jobs.

VII. Broader Impact

The general paradigm of dealing with exceptional conditions via hardware-guaranteed operation and software assistance will be applicable to many areas beyond voltage emergencies. For example, one could imagine the ideas and constructs that we presented in this paper also applying to thread deadlock and denial-of-service protection, hard and soft-error prevention, and bus contention in multiple-core microprocessors. As we look to microprocessors in the 5-year to 10-year time frame, we expect that many of these issues, in addition to PVT emergencies, will be pressing design issues that can be addressed with the multilayer design paradigm described in this paper. In the following first subsection, we describe the importance of thinking beyond just processor power and performance. Cost is an important factor. In the section that follows, we elaborate new opportunities for optimization that are enabled at the software layer by transitioning to a collaborative hardware and software resilient architecture.

A. Price-to-Performance Ratio

The need for collaborative hardware and software effort is now more important than ever before. The design of general-purpose microprocessors has long been primarily driven by the goal of ever-higher performance. While the industry has been extremely successful in this endeavor, we have begun to see the emergence of application domains, especially in the commercial sector, where energy efficiency and the price-to-performance ratio are considered more important design principles than peak performance alone. It is the use of commodity hardware in large-scale computing domains like datacenters, where efficiency at any cost is no longer an option, that specifically drives the need for collaborative effort. In such computing domains, price-to-performance is considered a more important design principle than peak performance alone.

These application domains contain enormous amounts of request/thread-level and application-level parallelism that encourages application architects to build computing clusters with large numbers of parallel commodity processors. Cheap hardware is lending itself to applications involving high-volume web services (e.g., search engines), biological and physical analysis, simulations (e.g., gene sequencing), and massively multi-player role-playing games. All these domains are continuously striving for higher availability and better performance, but with decreasing cost per system. Consequently, for a fixed level of performance per processor, application architects 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. The Google Cluster Architecture that runs their popular web search engine is a perfect example of an application where "price/performance beats peak performance" [53]. Barroso *et al.* described 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.

We can no longer cavalierly sacrifice energy efficiency or price to obtain new levels of performance. Instead, we must identify and develop machine organizations whose resulting costs and efficiencies track future increases in sustained performance rather than the more-quickly-growing (and yet almost never achieved) peak performance. While massively parallel application domains like datacenters offer obvious big savings, the benefits of hardware-software collaborative design also translate to desktop and workstation computing domains. Datacenters and their massively parallel applications 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. However, summing these individual savings on a national scale would yield large savings.

B. Emerging Opportunity for Software Optimization

Resilient architectures expose a new knob for tuning performance and/or power efficiency at the software layer. Assuming hardware provides a fail-safe guarantee against errors, software can in essence think of emergencies, such as voltage emergencies, as activity analogous to branch mispredictions or cache misses.

Over the past several years, a large body of optimization techniques and algorithms have emerged that specifically target microarchitectural event activity. Minimizing such activity frequently often leads to better runtime performance, owing to fewer execution stalls. Algorithms such as loop splitting or loop fusion improve runtime performance by laying out data such that there are fewer cache misses. Techniques that perform intelligent code layout help maintain steady instruction fetch bandwidth by reducing the number of branch mispredictions.

Likewise, we believe that resilient architectures will open up a new venue for code transformation explorations. As hardware provides a fail-safe, researchers could actively engage in developing new algorithms that specifically target the frequency and occurrence of emergencies. By reducing the number of emergencies, system performance improves, due to fewer hardware rollbacks. There are a large number of questions still pending answers. For instance, there are no cost models that guide the compiler toward a particular decision. Present-day work is driven by runtime heuristics, similar to profile-based application tuning. A more theoretical approach could lead to optimal results. Nevertheless, we see

this downside as an upside opportunity for long-term research in this area.

VIII. Conclusion

Modern applications benefit from an ever-increasing amount of performance, and thus microprocessor vendors continue to make advances in very large scale integration (VLSI) technology, circuits, and microarchitectures to address this need. However, as we demonstrated in this paper, we saw a growing gap between nominal operating conditions and peak operating conditions in modern and future microprocessor designs due to variations. To mitigate this effect, we described our own investigation, combined with prior effort, to design and build commodity computing systems that achieve both high performance and low cost today and in the future.

We advocated an approach that relies on the hardware for immediate suboptimal reaction to emergencies while software eliminates repeated occurrences, which is much more efficient and sustainable in the long-term. The work spans VLSI circuits, computer architecture, and software systems. In particular, we took a close look at voltage variation as an emerging dominant problem. We discussed details and findings that enable efficient hardware and software collaborative design, specifically within this context, showing not only hardware techniques, but also low-level software techniques to mitigate voltage variation. There is large room for improvement at both the architecture and software layers for innovative design and collaboration. Such collaboration will lead to products that yield good performance, but within reasonable costs.

References

[1] International Technology Roadmap for Semiconductors, *Process Integration, Devices and Structures*, 2002.

[2] N. James, P. Restle, J. Friedrich, B. Huott, and B. McCredie, "Comparison of split-versus connected-core supplies in the POWER6 microprocessor," in *Proc. IEEE Int. Solid State Circuits Conf.*, Feb. 2007, pp. 298–604.

[3] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm early design exploration," *IEEE Trans. Electron Devices*, vol. 53, no. 11, pp. 2816–2823, 2006.

[4] V. J. Reddi, M. S. Gupta, K. K. Rangan, S. Campanoni, G. Holloway, M. D. Smith, G.-Y. Wei, and D. Brooks, "Voltage noise: Why its bad, and what to do about it," in *Workshop SELSE*, 2009.

[5] 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 *Proc. DATE*, Apr. 2007, pp. 1–6.

[6] International Technology Roadmap for Semiconductors, *Process Integration, Devices and Structures*, 2007.

[7] V. J. Reddi, S. Kanev, W. Kim, S. Campanoni, M. D. Smith, G.-Y. Wei, and D. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2010, pp. 77–88.

[8] M. K. Gowan, L. L. Biro, and D. B. Jackson, "Power considerations in the design of the Alpha 21264 microprocessor," in *Proc. 35th Annu. Des. Autom. Conf.*, Jun. 1998, pp. 726–731.

[9] F. Mohamood, M. B. Healy, S. K. Lim, and H.-H. S. Lee, "Noise-direct: A technique for power supply noise aware floorplanning using microarchitecture profiling," in *Proc. ASP-DAC*, Jan. 2007, pp. 786–791.

[10] M. D. Pant, P. Pant, and D. S. Wills, "On-chip decoupling capacitor optimization using architectural level prediction," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 10, no. 3, pp. 319–326, Jun. 2002.

[11] Y. Chen, K. Roy, and C.-K. Koh, "Current demand balancing: A technique for minimization of current surge in high performance clock gated microprocessors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 13, no. 1, pp. 75–85, Jan. 2005.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Architectures Compilation Tech.*, Oct. 2008, pp. 72–81.

[13] 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 *Proc. IEEE Int. Solid State Circuits Conf.*, Feb. 2008, pp. 402–623.

[14] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proc. 1st Annu. IEEE/ACM Int. Symp. Code Gener. Optimization*, Mar. 2003, pp. 15–24.

[15] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and optimization," *IEEE Trans. Comput.*, vol. 50, no. 6, pp. 529–548, Jun. 2001.

[16] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *Micro, IEEE*, vol. 23, no. 2, pp. 22–28, Mar.–Apr. 2003.

[17] D. Ernst, N. Kim, S. Das, S. Pant, R. Rao, T. Pham, K. F. C. Ziesler, D. Blaauw, T. Austin, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. 36th Int. Symp. Microarchitecture*, Dec. 2003, pp. 7–18.

[18] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Proc. Int. Conf. Supercomputing*, 2004, pp. 277–286.

[19] 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 1.3 GHz fifth generation SPARC64 microprocessor," *IEEE J. Solid-State Circuits*, vol. 38, no. 11, pp. 1896–1905, Nov. 2003.

[20] N. J. Wang and S. J. Patel, "ReStore: Symptom-based soft error detection in microprocessors," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 3, pp. 188–201, Jul.–Sep. 2006.

[21] 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 1.3 GHz fifth-generation sparc64 microprocessor," in *Proc. Design Autom. Conf.*, Jun. 2003, pp. 702–705.

[22] T. Slegel, I. Averill, R. M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb, "IBM's s/390 g5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar.–Apr. 1999.

[23] 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," Univ. Wisconsin-Madison, Madison, Comput. Sci. Tech. Rep., 2000.

[24] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *Proc. 35th Int. Symp. Microarchitecture*, Nov. 2002, pp. 3–14.

[25] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Checkpointed early load retirement," in *Proc. 11th Int. Symp. HPCA*, Feb. 2005, pp. 16–27.

[26] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultralow-cost defect protection for microprocessor pipelines," in *Proc. 12th ASPLOS*, 2006, pp. 73–82.

[27] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously recording program execution for deterministic replay debugging," in *Proc. 32nd Annu. ISCA*, 2005, pp. 284–295.

[28] M. S. Gupta, K. Rangan, M. D. Smith, G.-Y. Wei, and D. M. Brooks, "DeCoR: A delayed commit and rollback mechanism for handling inductive noise in processors," in *Proc. HPCA*, Feb. 2008, pp. 381–392.

[29] M. S. Gupta, K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, "Toward a software approach to mitigate voltage emergencies," in *Proc. ISLPED*, Aug. 2007, pp. 123–128.

[30] K. Hazelwood and D. Brooks, "Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization," in *Proc. ISPLED*, Aug. 2004, pp. 326–331.

[31] V. Reddi, M. Gupta, G. Holloway, G.-Y. Wei, M. Smith, and D. Brooks, "Voltage emergency prediction: Using signatures to reduce operating margins," in *Proc. IEEE 15th Int. Symp. HPCA*, Feb. 2009, pp. 18-–29.

[32] M. S. Gupta, "Variation-aware processor architectures with aggressive operating margins," Ph.D. dissertation, Harvard Univ., Cambridge, MA, 2009, adviser D. Brooks.

[33] M. Gupta, V. Reddi, G. Holloway, G.-Y. Wei, and D. Brooks, "An event guided approach to reducing voltage noise in processors," in *Proc. Des. Autom. Test Eur. Conf. Exhibit.*, Apr. 2009, pp. 160–165.
[34] E. Grochowski, D. Ayers, and V. Tiwari, "Microarchitectural simulation and control of di/dt-induced power supply voltage variation," in *Proc. Int. Symp. High-Performance Comput. Architecture*, 2002, pp. 7–16.
[35] R. Joseph, D. Brooks, and M. Martonosi, "Control techniques to eliminate voltage emergencies in high performance processors," in *Proc. HPCA*, Feb. 2003, pp. 79–90.
[36] M. D. Powell and T. N. Vijaykumar, "Pipeline muffling and *a priori* current ramping: Architectural techniques to reduce high-frequency inductive noise," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 2003, pp. 223–228.
[37] M. Powell and T. N. Vijaykumar, "Exploiting resonant behavior to reduce inductive noise," in *Proc. ISCA*, Jun. 2004, pp. 288–299.
[38] V. J. Reddi, M. Gupta, G. Holloway, M. D. Smith, G.-Y. Wei, and D. Brooks, "Predicting voltage droops using recurring program and microarchitectural event activity," *IEEE Micro*, vol. 30, no. 1, p. 110, Jan.–Feb. 2010.
[39] V. J. Reddi, "Software-assisted hardware reliability: Using run-time fedback from hardware and software to enable aggressive timing speculation," Ph.D. dissertation, Harvard Univ., Cambridge, MA, 2010, adviser D. Brooks.
[40] M. Toburen, "Power analysis and instruction scheduling for reduced di/dt in the execution core of high-performance microprocessors," Masters thesis, North Carolina State Univ., Raleigh, 1999.
[41] H.-S. Yun and J. Kim, "Power-aware modulo scheduling for highperformance VLIW processors," in *Proc. ISLPED*, 2001, pp. 40–45.
[42] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system," in *Proc. Programming Language Des. Implementation*, May 2000, pp. 1–12.
[43] V. Reddi, M. Gupta, M. Smith, G.-Y. Wei, D. Brooks, and S. Campanoni, "Software-assisted hardware reliability: Abstracting circuit-level challenges to the software stack," in *Proc. 46th Annu. DAC*, Jul. 2009, pp. 788–793.
[44] V. J. Reddi, S. Campanoni, M. S. Gupta, M. D. Smith, G.-Y. Wei, D. Brooks, and K. Hazelwood, *Eliminating Voltage Emergencies via Software-Guided Code Transformations*, vol. 7. New York, NY: ACM, Oct. 2010, pp. 12:1–12:28.
[45] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous mutlithreading processor," *SIGPLAN Not.*, vol. 35, no. 11, pp. 234–244, 2000.
[46] A. Fedorova, "Operating system scheduling for chip multithreaded processors," Ph.D. dissertation, Harvard Univ., Cambridge, MA, 2006, adviser M. I. Seltzer.
[47] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: Online contention detection and response," in *Proc. 8th Annu. IEEE/ACM Int. Symp. CGO*, 2010, pp. 257–265.
[48] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS observations to improve performance in multicore systems," *IEEE Micro*, vol. 28, no. 3, pp. 54–66, May–Jun. 2008.
[49] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. Architectural Support Programming Languages Operating Syst.*, 2010, pp. 129–142.
[50] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proc. 11th Int. Symp. HPCA*, 2005, pp. 340–351.
[51] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable performance in SMT processors: Synergy between the OS and SMTs," *IEEE Trans. Comput.*, vol. 55, no. 7, pp. 785–799, Jul. 2006.
[52] W. El-Essawy and D. Albonesi, "Mitigating inductive noise in SMT processors," in *Proc. ISLPED*, Aug. 2004, pp. 332–337.
[53] L. A. Barroso, "The price of performance: An economic case for chip multiprocessing," *Queue, ACM*, vol. 3, no. 7, pp. 48–53, Sep. 2005.

Vijay Janapa Reddi (M'10) received the B.S. degree in electrical and computer engineering from Santa Clara University, Santa Clara, CA, the M.S. degree from the University of Colorado, Boulder, and the Ph.D. degree in computer science from Harvard University, Cambridge, MA.

He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Texas, Austin. He explores new opportunities and synergies for cross-layer solutions that improve processor and system power, performance, and reliability. His background is in compilers, computer architecture, and virtual machine technologies. His current research interests include computer systems, focusing on the intersection between hardware and software.

David Brooks (M'02) received the B.S. degree in electrical engineering from the University of Southern California, Los Angeles, and the M.A. and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ.

He is currently a Gordon McKay Professor of Computer Science with the School of Engineering and Applied Sciences, Harvard University, Cambridge, MA. He joined Harvard University in 2002 after spending one year as a Research Staff Member with the IBM T. J. Watson Research Center, Yorktown Heights, NY. His current research interests include technology-aware computer design, with an emphasis on power-efficient computer architectures for high-performance and embedded systems.