Orchestrator: Guarding Against Voltage Emergencies in Multithreaded Applications

Xing Hu, Guihai Yan, Member, IEEE, Yu Hu, Member, IEEE, and Xiaowei Li, Senior Member, IEEE

Abstract—Voltage emergency (VE) has become a critical challenge with decreasing feature size and increasing power capacity. Destructive core interference is one main source of VE in multicore processors. We observed that the applications following single program and multiple data programming model tend to spark domain-wide destructive core interference because multiple threads exhibit similar power activity. We analyze and quantify this effect and propose one low-cost solution, Orchestrator, to avoid voltage droop synergy among cores. Orchestrator leverages the thread diversity to smooth voltage droops in multicore architectures based on thread scheduling. The thread migration impact on performance is also considered. Experimental results show that Orchestrator can significantly reduce VEs, thereby improving performance.

Index Terms—Multicore, multithreaded application, single program multiple data (SPMD), voltage emergencies (VEs).

I. INTRODUCTION

Voltage emergency (VE) is a big issue for power integrity, which may incur timing failures and ruin system states. It comes from the current variation and parasitical impedance in the power delivery network (PDN). These two factors become even worse with increasing power capacity and decreasing feature size, which makes VE more troublesome [1].

The traditional solution for VE elimination is to ensure safe timing even in the worst case of voltage droops by reserving sufficient voltage margin [2]. However, prior study shows that this conservative voltage margin solution will dramatically degrade performance for future multicore processors [3]. In view of the limitation of overdesign, prior studies proposed architectural solutions to mitigate the VEs [4], [5].

In multicore processors, core-to-core voltage interferences become the main challenge to voltage integrity [3], [6], since a cluster of cores usually share one power domain for the concern of implementation and power management.

Manuscript received May 31, 2013; revised November 10, 2013; accepted December 3, 2013. Date of publication February 6, 2014; date of current version November 20, 2014. This work was supported in part by the National Natural Science Foundation of China under Grant 61076018, Grant 61276036, Grant 61100016, Grant 61076037, and Grant 60921002, and in part by the National Basic Research Program of China (973 Program) under Grant 2011CB302503. X. Hu is with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: huxing@ict.ac.cn).

G. Yan, Y. Hu, and X. Li are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: yan_guahai@ict.ac.cn; huuyu@ict.ac.cn; lxw@ict.ac.cn).

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/TVLSI.2013.2296787

For multithreaded applications, Miller et al. [5] observe that core-to-core interferences are usually produced by synchronized pipeline wake-up activities when threads confront barriers. This paper eliminates VE by staggering threads into a barrier and sequentially stepping over it. We observe another prominent cause for VE: single program and multiple data (SPMD) programming model which is dominant for large-scale distributed-memory machines [7]. In this model, multiple threads sharing the same function body leads to similar power activities of cores, which possibly evokes core resonances. Compared with the barriers, SPMD model can spark intercore voltage resonances more frequently during thread life cycle. In addition, the staggering solution is less effective for SPMD-induced VE because of following two reasons: 1) it is hard to figure out a guaranteed phase staggering solution that can fit for the whole execution epoch due to the phase variability in thread lifetime and 2) staggering solution will penalize performance because of the thread serialization, even if such a solution existed.

To solve the problem, we propose Orchestrator which focuses on the intercore voltage interactions caused by similar program activities and smooth the voltage variations by thread scheduling. The key operational principle is to coordinate diversified power characteristics among neighbor cores. Enhanced by Orchestrator, systems with fail-safe mechanisms can perform better with less voltage margin. Overall, we make the following contributions.

1) We observe that applications following SPMD programming model have similar interthread execution phases and may frequently incur destructive thread interferences.

2) We propose a metric, intrinsic droop intensity (IDI), to quantify thread interference effect. Then, we correlate IDI with program activities based on a regression model.

3) We propose Orchestrator, a low-cost thread scheduling scheme to smooth VEs. We first analyze and predict IDI of threads from runtime statistics, and then give the appropriate thread locations to avoid the voltage droop synergy.

4) We enhance Orchestrator with a migration-aware algorithm, in viewing that migration in the nonuniform cache access (NUCA) hierarchy may perform great variability. Therefore, Orchestrator can provide good compatibility with those advanced NUCA techniques by taking migration impact into consideration. In addition, we also discuss the effectiveness of Orchestrator in simultaneous multithreading (SMT) processor.
The rest of this paper is organized as follows. Section II gives the background and motivational experiments, followed by detailed description of the proposed Orchestrator scheme in Section III. Section IV elaborates the implementation. Sections V and VI present the experimental setup and results. The related works and conclusion are given in Sections VII and VIII.

II. BACKGROUND AND MOTIVATION

A. PDN Design in Multicore Processors

In multicore processors, PDN is usually divided into several clusters with independent voltage domains because of the following two reasons. First, arranging every core in a different power domain is irrational, since it is impractical to decap every core independently because of the difficulty for scaling down the capacitance size. Second, single voltage domain is also unreasonable due to the difficulty of supporting intelligent power management. This clustered design style takes both implementation cost and power management friendliness into consideration [8], which is not only scalable, but also supports advanced power management strategies. In this paper, we use a 16-core processor divided into four domains as the baseline, as shown in Fig. 1. Each domain has an independent power source, as [8] and [9] assumed. Thus, cores in different power domains would be independent of each other while cores in the same power domain would interfere with each other.

B. Program Behavior of Multithreaded Applications

Synchronization mechanisms may affect pipeline activities, which possibly exerts influence on VE. The synchronization mechanisms are classified into two categories in terms of functionality: 1) data synchronization and 2) operational synchronization. Data synchronization is used to ensure the coherence of shared data and the operational synchronization is to ensure the correctness of control dependence between threads. We mainly focus on operational synchronization because it coordinates activities of threads and has strong correlation with VEs. Taking barriers for example, designers turn off cores to save power when threads are waiting in barriers and turn on them after stepping over barriers. In such a power-saving design, the simultaneous reexecution after barriers may draw large current which results in VEs [5]. In addition to barriers, thread spawning is also one source of VE. Applications which create a set of worker threads during initialization would possibly cause large voltage droops, since all threads are launched simultaneously.

Barrier and thread spawning happen rarely in most programs, because they have been minimized during software designing to decrease performance degradation. Compared with barrier and thread spawning, the SPMD programming model would induce VEs more often. Because this programming style gets threads into similar execution traces [10]–[13], which possibly incurs destructive thread interferences. Such thread interferences would exist during long period of thread life cycle. Thus, we highlight such interactions which are the main culprit for VEs in multicore processors. In the next section, we demonstrate the reinforced intercore interactions during execution of multithreaded applications, called core resonance.

C. Core Resonance

There are two preconditions for core resonance: 1) the cores in the same domain show similar power profiles in a close phase and 2) the variation strength of profiles is strong enough. We find that for multithreaded applications, the first requirement is usually satisfied. As shown in Fig. 2, there are four threads running on core1, core2, core3, and core4, in the same power domain. They exhibit similar current profiles, thus easily resonate and incur large voltage droops at point a, b, and c. Fig. 3 shows the impact of core resonances in a voltage domain, i.e., to show how many cores will suffer from the emergencies (voltage dropping below 4.5%) concurrently. The result shows that on average, up to 73% of emergencies jeopardize more than two cores simultaneously. About 30% of emergencies can put all cores in danger. Hence, it is crucial to mitigate such destructive interferences in multicore PDN.

Prediction-throttling is an effective way to mitigate the voltage droops for monolithic processors. However, its efficacy is impaired in multicore processors, because single thread prediction mechanism is not accurate. Taking Fig. 2 for example, core4 is distinctly different from the others, because core4 is running the operating system (OS) while the other three cores are running a similar function body. We highlight four time points: a, b, c, and d. In point a, b, and c, current curve of core4 is smooth, but we still spot significant voltage droops in core4 because it is infected by synchronized current swings of core1, core2, and core3. In point d, although the current of core4 fluctuates severely, the voltage droop is not

Fig. 1. Four-domain processor baseline and PDN model.
very significant because the other cores are in mild phases and offers core4 some current relief. Single thread prediction mechanism cannot distinguish such intercore interference effect.

In view of these limitations of prior work, we propose a new architectural approach, Orchestrator, which is designed to be fully aware of the intercore interferences. Orchestrator mitigates VEs by avoiding possible core resonances. In addition, it is easy to be combined with fail-safe mechanisms.

III. ORCHESTRATOR PRINCIPLE

The key principle of Orchestrator is to avoid domain-wide destructive noise by orchestrating the corunning threads in the same power domain. There are two main points: 1) quantifying and predicting run-time thread interferences and 2) developing an algorithm to guide thread scheduling that can minimize voltage resonance. The following sections will address these two issues.

A. Voltage Feature Quantification

1) IDI of Thread: In multithreaded scenario, the voltage variations of a thread are affected by both the internal property of itself and the external interferences from threads nearby. If a thread occasionally stalls and restarts pipeline, it may frequently incur the VE by itself, which also acts as a current drawer and brings voltage interferences to other threads. Thus, we use IDI to characterize how frequently a thread would induce voltage fluctuations when stand along. Threads with higher IDI are more voltage-violent which suffer from voltage variation more frequently and are more likely to infect other threads. On the other hand, threads with lower IDI are voltage-mild and can offer relief for neighbourhood cores. In this paper, the IDI is quantified as the occurrences of voltage droops (larger than 3% of nominal voltage) when running along in a monitoring window (one million cycles in this paper).

The effect can be exemplified by Fig. 4. There are four applications (barnes, conocean, watersp, and cholesky) running on the processor. Experimental setup
is introduced in Section V. Each application has four threads, b1, . . . , b4 for barnes, c1, . . . , c4 for conocean, w1, . . . , w4 for watersp, and h1, . . . , h4 for cholesky. Each marker corresponds to a specific thread. We measure the IDI of each threads in advance. In this process, there is only one thread running in one power domain, thus, the voltage interference would be eliminated. Then, we apply different strategies to arrange threads in the processor. The x-axis represents the IDI of threads and the y-axis represents the actual VE occurrences of threads while executing with different corunners. Initially, the threads forked by the same parent are scheduled in the same domain, as the solid markers show. Taking black ■ for example, there are three out of four barnes threads showing high IDI. Unsurprisingly, all barnes threads suffer from intensive VE. The same situation also applies on watersp threads in Domain3. In contrast, conocean threads and cholesky threads show low IDI, resulting in nonintensive VE occurrences.

Then, we schedule the threads with different levels of IDI into one power domain. The hollow markers show the situation after scheduling. The thread arrangement information is shown by the table on the top left corner in Fig. 4. Taking the four watersp threads for example, initially these four threads in domain 3 are confronted large sum of VEs, as shown by the solid purple markers ●. After scheduling, they are running with threads which have lower IDI. At this time, watersp threads incur much less VEs, as shown by the hollow purple markers ○, which are vertically beneath the solid markers. This figure has two implications: 1) thread VE occurrences are closely related to its corunners and 2) IDI is a faithful metric to quantify aggressiveness of threads.

2) IDI Estimation During Running Time: The previous section demonstrates that it is effective to mitigate voltage resonance by interleaving voltage-violent and voltage-mild threads in one domain. However IDI, as defined, is determined by intrinsic feature of a thread and cannot be obtained by directly measuring during runtime since the interthread interference is inevitable. Hence, the key implementation issue is how to get IDI of each thread at runtime.

Reference [3] proves that the voltage characteristics tightly correlate with several kinds of microarchitectural events, such as branch misprediction, cache miss, and table lookup bypass (TLB) miss. Because these events may incur pipeline stall and potentially induce voltage variations. Hence, we use these activity patterns as proxies to estimate IDI. These runtime statistics are nonlinearly correlated, therefore, the regression tree is an ideal approach to cope with such a relationship. The detail about IDI estimation is in Section IV.

3) Thread Interference and Domain Emergency Level (EL): We quantify the destructiveness of thread interferences based on their IDI. Each thread in a monitoring window is associated with an IDI, so its aggressiveness in phases can be expressed by an IDI vector $V_{\text{IDI}}$ which records IDI in several continuous windows. Suppose that there are 10 monitoring windows, then $V_{\text{IDI}}$ of thread $T_i$ is a 10-element vector denoted by $[\text{IDI}_1, \text{IDI}_2, \ldots, \text{IDI}_{10}]$. Given the threads with similar strong voltage fluctuation are more likely to resonate with large voltage droops, inner product between $V_{\text{IDI}}$ of threads can be used to quantify the voltage interference destructiveness. Larger inner product indicates that these two threads have greater chance to resonate, hence to generate large voltage droops. In contrast, small inner product indicates that the two threads are likely to help with each other. The interferences between thread $i$ and thread $j$, denoted by $I(i, j)$, is defined as follows:

$$I(i, j) = V_{\text{IDI}}^i \cdot V_{\text{IDI}}^j.$$ 

We use the domain $EL$ to predict droop intensity of each domain. It can be defined as the aggregated interference $I$ of the threads running in one domain. For example, the $EL$ of domain $S$ is

$$EL = \sum_{i \in S, j \in S} I(i, j).$$

B. Problem Description

The scheduling algorithm is to determine the locations of threads, which can minimize the sum of domain $EL$. To simplify the discussion, we assume that cores do not support SMT, so each core can only execute one thread at a time.

The formal description of this problem is as follows. Suppose there are $d$ domains in processor and $n$ threads for scheduling. $V_{\text{IDI}}$ of each thread is known. We aim to figure out the best thread-to-core mapping $M$ which minimizes the sum of domains’ $EL$. $M$ consists of triples which record location information of threads, like $(\#\text{thread}, \#\text{domain}, \#\text{core})$.

This problem is analogous to $k$-way networks partition [14]. $k$-way network partition problem is to separate network node into $k$ groups, each of which has the equal node count. In addition, the network streaming weight of each group is maximized. If we interpret the opposite number of thread interference into network streaming weight, our problem is analogous to it. $k$-way networks partition is a classic NP problem, which is too time consuming to find the optimal solution. The time complexity of exhaustive search is $C_n^d \ast C_{n-d}^d \ast C_{n-2d}^d \ast \ldots \ast C_{d/d}^d$, where $C_n^d$ is the combination, $d$ is the number of domains, and $n$ is the number of cores. This combinational complexity is at the magnitude of $n!$, and can hardly support on-the-fly implementation. Thus, we propose a suboptimal solution with a greedy algorithm.

C. Orchestrator Algorithms

Fig. 5 shows the process of the algorithm. Its principle is to schedule the most aggressive thread to the mildest domain first. The detail of this algorithm is elaborated in the following section.

1) VE Reduction-Oriented Scheduling Algorithm (Orch-VEO): The proposed algorithm consists of four steps, pseudocodes of which are shown in Algorithm 1.

Step 1 {line1–line3}: Determine the scheduling priority of threads and the initial order of Domain Sequence Buffer. There are $n$ threads waiting to be scheduled into $d$ power domains ($n$ is less than or equal to the count of available cores in the processor). The $EL$ of each domain is initialized to zero. To determine the scheduling priority, we use
Algorithm 1 VE-Oriented Scheduling

Input: $n$ threads’ $V_{ID1}$ and $d$ domains;
Output: Array $M[n][2]$ recording $\{domain_{ID},core_{ID}\}$ of $n$ threads;

1: Calculate Priority Stack according to $|V_{ID1}|$
2: Initiate the EL array of domains $EL[d]=\{0,0,0,0\}$
3: $tmp\_min = 0$
4: while Priority Stack is not empty; do
5:   Pop out stack top thread $t$
6:   //Arrange thread in domain $tmp\_min$
7:   $M[t][0] = tmp\_min$
8:   //Arrange thread in the first available core in domain $tmp\_min$
9:   $M[t][1] = core[t] + +$
10: //Update EL of domain $tmp\_min$
11: $EL_{tmp\_min} = + \sum _{i\in tmp\_min} I(j,i)$
12: $tmp\_min = domain \_ID$ with the lowest $EL$
13: end while

Fig. 5. Orch-VEO.

Fig. 6. Performance impact of thread migration.

modular operation to transform the high-dimension vector $V_{ID1}$ to the scalar $|V_{ID1}|$. Then, threads are sorted in a descending order according to $|V_{ID1}|$ and stored in Priority Stack. The thread with the greatest $|V_{ID1}|$ has the highest priority to be scheduled. As shown in Fig. 5, the thread sequence is $T_{14}, T_6, T_{12}, \ldots, T_5$, so $T_{14}$ is the first thread to be scheduled.

Step 2 {line4–line9}: Check whether the Priority Stack is empty. If not, pop out the thread with highest priority $thread_h$ and schedule it to the mildest domain. In this case, the mildest one is domain3. Therefore, $thread_h$ is scheduled to the first available core in domain3.

Step 3 {line10–line11}: Update EL of newly allocated domain. After arranging the selected thread in mildest domain, the EL of this targeted domain should be recalculated. The calculation formula is $EL_{T_{14}, T_6, T_{12}, \ldots, T_5}$ is the first thread to be scheduled.

Performance Impact of Thread Migration: Thread scheduling is an important supporting technique in Orchestrator. However, it may initiate thread migration and incur performance overhead caused by context switching and changes of data distribution. Following penalties are introduced during context switching: transferring register state and TLB state, recreating branch predictor state. However, previous work [15] observes that the context switching overhead is amortized in real schemes when the migration interval is long enough. Hence, we conduct experiments to evaluate performance degradation brought by the latter factor, in two different hierarchies: private last level cache (LLC) and shared LLC. As shown in Fig. 6, migration overhead is measured with migration interval of every $10^6$ cycles. Two hundred workloads are randomly chosen from SPLASH-2 Benchmark suite.

1) Private LLC: In this scenario, every core uses private-own last level cache. We evaluate the system performance (in million instructions per second) normalized to the scenario without migration. The black line in Fig. 6 shows that most workloads perform well. Less than 2% of cases bring more than 5% of performance degradation, proving that 1M cycles are sufficient for most applications to warm up caches and branch predictors.

2) Shared LLC: In this scenario, 16 cores share one last level cache, like Processor Niagara. Prior work [16] witnessed that migration between cores sharing a LLC usually imposes small negative impact on performance. Such results are true for dual-core processors. For processors with more cores, the increasing complexity of cache topologies makes it hard to predict performance. Fig. 6 shows the results which are sorted according to the performance. In most cases, the performance overhead of migration is negligible. However, there are some cases where migration results in worse or surprisingly better performance. This is because the shared LLC cache, we implemented is static-NUCA [17]. In static-NUCA, data is statically mapped. Remapping one thread may change the distance between the core and its data in LLC. Some data originally remote becomes local, hence improving performance; while some data originally local becomes remote, hence degrading performance. It would exert big influence on performance only when most of data becomes remote or local for long period of program execution.
In shared LLC scenario, migration may bring large performance variability because of changes in thread distributions. There are several techniques to handle this issue: data block migration [18], [19] and thread migration [20]. They make prediction of the appropriate thread locations which can reduce the performance degradation brought by remote data accessing. Using location information provided by these excellent works, Orchestrator can mitigate VEs and reduce performance degradation of migration as well.

Orch-MA: To diminish the performance degradation of migration, designers wish some threads running in specific cores to reduce remote data accesses [20]. When thread needs to be tacked onto one specific core, we can still mitigate VEs by scheduling the corunning threads.

We refer the collection of threads with specified location as Pinning Set. Suppose there are $p$ threads pinned up on cores, locations of which are recorded in Pinning Map $M_p$, consisting of $p$ triples like $\{#thread_{ID},#domain_{ID},#core_{ID}\}$. The objective is to minimize sum of domain EL with one more additional constraint: threads belong to Pinning Set should be initially scheduled in accordance with $M_p$. We achieve this goal with modification on Step 1 of Orch-VEO, by replacing the line 1, 2, and 3 of the pseudocode.

Orch-MA Step 1: We sort the unpinned $n-p$ threads (out of Pinning Set) in a descending order according to their $|V_{IDI}|$ and store such sequence in the Priority Stack. The initial EL of domain $S$ is aggregated interference between pinned-up threads, calculated as follows:

$$EL = \sum_{i,j \in S, i,j \in Pinning Set} I(i,j).$$

The pinned-up threads are eliminated from the scheduling list. The other threads are still scheduled under Orch-VEO algorithm, using Orch-VEO Steps 2–4. The more pinned-up threads in one domain, the less Orchestrator can do to mitigate VEs.

IV. IMPLEMENTATION

The Orchestrator can be implemented as hardware circuitry in chips, which costs little area and performance overhead.

A. Working Mechanisms

As shown in Fig. 7, Orchestrator consists of two components: Monitor and Scheduler. Monitor is responsible for estimating and recording IDI of each thread in every monitoring window. It uses pretrained regression model to estimate IDI of each thread, taking performance counter statistics as input. The predicted IDI information will be recorded in the IDI table. At the end of a running window, Scheduler will be triggered to determine the new thread mapping based on the input of the IDI table and the Pinning Map, then it initiates thread scheduling for the next running window.

B. Monitor

The Monitor has two parts: IDI predictor and IDI table which take charge of IDI estimation and recording, respectively. The IDI predictor takes the performance counter statistics as input and estimates the IDI of each thread as output. Then, the outputs are stored in the corresponding entry in the IDI table. The IDI table is a 2-D array, every row of which records $V_{IDI}$ of a thread. The row number of the IDI table is related to thread number and the column number of the IDI table is related to timing interval ID. For example, at the end of $i$th monitoring window $W_i$, the IDI of thread $j$ would be stored to the $i$th column and $j$th row in the IDI table.

For the ease of implementation for the IDI table and Scheduler, we simplify the IDI recording by classifying the IDI value into four ranks: Rank1, Rank2, Rank3, and Rank4, so that only two bits are needed for every entry.

1) IDI Predictor: One important structure in Monitor is the IDI predictor. As prementioned, we take branch misprediction intensity, L1 and L2 cache miss intensity, and TLB miss intensity as the input variables to estimate IDI. These statistics can be easily obtained by a set of performance counters. Since the relationship between input and output is complex and nonlinear, we use regression tree to describe it with a rule-based model [21]. The regression tree model, widely being used in data mining, is composed of a group of rulers, each having two parts, a condition and a simple multivariant regression model. When condition is satisfied, the associated fitting model is chosen to calculate the predicted value. Regression tree makes prediction in almost real-time because it just looks up constants in the tree instead of complicated calculations. In this paper, is used to fit the relationship between the performance counter information and IDI.

First, we have to train the regression tree with a set of training samples. The training stage is conducted off line. During the training period, performance counter information and corresponding droop intensity information of threads are gathered as the training set to generate regression tree. To avoid interferences, during the process of obtaining droop intensity information, there is only one thread running in a
power domain at a time. When Orchestrator comes to industry, the droop intensity can be measured by using sensors such as critical path monitor (CPM) [22], output of which indicates the timing margin of monitored critical paths. The CPM can be calibrated after fabrication. In POWER7, it shows that a movement of one-bit position in CPM indicates a variation of 17 mV and 48 MHz, or 8.6 °C. The temperature is changing far more sluggish than voltage variation. Thus transient changing value in CPM can be interpreted as voltage variation. The training stage is over when the regression tree is become reaching a stable state. Then, we build it into the target chips to acquire IDI traces.

We linearly separate IDI value with the minimal scale of 50 K. The IDI between 0 and 50 K is classified to Rank1, between 50 to 100 K is in Rank2, between 100 to 150 K is in Rank3, and more than 150 K is in Rank4. Statistically, VE occurrences of power domain running Rank1-threads, Rank2-threads, and Rank3-threads are 3%, 10%, and 51% normalized to Rank4-threads. Thus, such separation retains the effectiveness of IDI which can distinguish voltage aggressiveness of threads. After ranking, the IDI can be stored within two bits, thus the IDI table, computing circuitry and sorting logic can be significantly simplified.

We evaluate the prediction accuracy based on simulation, detailed experimental setup of which is in Section V. SPLASH-2 benchmarks are selected as the workloads. One hundred samples of 10^6-cycle execution slices are randomly taken as the training set. Then, we apply the regression tree to other 250 samples as the testing set to evaluate the accuracy. The result is shown in Fig. 8. In this figure, the x-axis denotes the actual thread droop intensity and y-axis denotes the predicted droop intensity. Our prediction model can correctly predict 87% of samples. The overall rank prediction miss is about 13%, but no more than 3% of samples make severe miss (i.e., to confuse voltage-mild thread with voltage-violent one). The regression tree is very cost-effective. The regression tree in this paper only needs to implement a set of if-else conditions for given predictors and costs little hardware overhead.

C. Scheduler

The Scheduler implements the Orch-VEO and Orch-MA algorithms. Scheduler first sorts threads according to their \( |\text{IDI}| \) and stores the sequence in the priority stack, then records ID of the topmost thread, ID of the mildest domain and ID of the first available core in this domain. To reduce time complexity of scheduler, we can use sorting circuitry for acceleration.

We implement Orchestrator in Verilog and use a commercial logic synthesis tool Synopsys design compiler (DC) to evaluate the area and power overhead at frontend. The Orchestrator consists of several hardware components: a regression tree predictor to estimate IDI, an IDI table and a scheduler which is mainly comprised by sorting circuitry. The scale of the IDI predictor is fixed while the IDI table and the sorting circuitry scale with the core number. The IDI predictor and the IDI table updates every million of cycles, and the scheduler is triggered every ten millions of cycles. Therefore, the average dynamic power of Orchestrator is very low and the static power becomes the dominant source. When the core number is 16, Orchestrator circuit has equivalent area of 8617 two-input NAND gates under the 28 nm, 0.8 V technology library tsmc28hp of Taiwan Semiconductor Manufacturing Company. The total power is 0.22 mw and the critical path delay is 0.06 ns. When the core number is 64, the area of Orchestrator is equal to that of 23248 two-input NAND gates. The power is 0.61 mW and the critical path delay is 1.06 ns. Orchestrator incurs negligible power and area overhead. In addition, it would not incur much performance overhead since it is not on the critical path of processor.

V. EXPERIMENTAL SETUP

A 16-core processor is simulated for evaluation, core configuration of which is listed in Table I. Orchestrator is implemented on it with full-system simulations using Virtutech Simics with Multifacet general execution-driven multiprocessor simulator extension [23]. GEMS has integrated power simulation tools Watch inside [24]. We use them to generate power traces which can be transferred to current traces. Then, current traces are fed into HSPICE PDN model and HSPICE simulates the voltage traces of each core. The PDN for the baseline 16-core processor comprises of four-independent domains, as shown in Fig. 1. Each domain resembles that of Intel Xeon...
5500 processors [25]. In each power domain, each core is modeled as a timing-varying current source with decoupling capacitance and the intracore connections are modeled as resistive paths. The detailed parameters are the same with prior work [26].

Nine scientific applications in SPLASH-2 benchmarks suite are used in our simulations which cover most computation and communication patterns: barnes (8192 bodies, four-time steps), cholesky (tk16.O), conocean (258 × 258 ocean), fft (256 K complex doubles), lu, radix (1 M keys, 1024 radix), radiosity, water-nsq (512 mol, four-time steps), and water-sp. Every application, invoking four threads, has been executed for 1 B instructions which are cut into program slices of 1M cycles. Every sample selects 16 program slices from four applications to build the start point of workload.

Note that Orchestrator is an architectural solution to reduce the voltages emergencies, but cannot totally eliminate the VE. We therefore assume the target processor is equipped with fail-safe mechanisms which will be engaged on when a failure is about to happen or already happened. The fail-safe mechanisms may be either rolling back for recovery or dynamic frequency tuning, such as Razor [27] and fast digital phase lock loop (DPLL) enhanced method [28].

VI. EXPERIMENTAL RESULT

In this section, we first evaluate the effectiveness of both Orch-VEO and Orch-MA on VE reduction. Then, we show the performance improvements brought by these two algorithms.

A. Orchestrator Effectiveness

1) VE Reduction: We validate VE reduction effectiveness of Orch-VEO algorithm on several applications. Fig. 9 shows a specific case study on voltage-mild dominant, voltage-violent dominant, and mixed workloads listed in Table II. The Oracle assumes Orch-VEO with perfect inference accuracy, while Actual represents the actual scenarios with imperfect prediction accuracy. For workload 2, 3, 4, and 5, the emergency reduction is more than 80%. The reduction for workload 1, 7, 8, and 10 is ranging from 50% to 80%, not as dramatic as the former cases, because the workload is either mild-thread dominated, such as workload 1, or violent-thread dominated, such as workload 7, 8, and 10. Overall, the results show that Orch-VEO can effectively reduce the VEs. In addition, we can see the prediction accuracy of 87% works well in our scheme. The emergency reduction with actual Orchestrator degrades no more than 4% compared with its oracle counterpart. Unless otherwise specified, the following results are based on the actual Orchestrator.

We would like to further show the statistical validation of Orch-VEO for 100 workloads which are randomly constituted from SPLASH-2 benchmarks, under different migration intervals. Fig. 10 gives the result under interval of 1, 5, 10, and 20M cycles, using boxplot to show the median (the middle notch) and dispersion (the upper and lower bar) of a group of values; the outliers are denoted by the x. Generally, the emergency reduction declines with larger running window. We also find that in rare corner cases, Orch-VEO may increase the emergency rate, as shown in outliers of this figure. Because the thread power phases change so dramatically that the inference gets missed. However, the average reduction is still about 64% even at the largest interval of 20M cycles. In the following performance evaluation, we use the Orchestrator with migration interval of 10M cycles.

We also validate the effectiveness of Orch-MA algorithm which makes thread scheduling under thread pinning constrains. To anchor the influence of pinning, we use 100 stressmarks in which more than 75% of threads are voltage violent. As shown in Fig. 11, the workloads are sorted in an ascending order according to their VE occurrences based on Orch-VEO scheduling, and the black line represents this
scenario with no constraints. The solid red line represents the scenario with two pinned-up threads in each domain. While for the dot blue line, the pinned-up number is 3. All of them record VE occurrences normalized to baseline. They all can reduce the occurrences of VE to some extent, compared with baseline. When more threads are pinned up in specific cores, Orch-MA have less freedom to schedule threads, thus more VEs would occur. There are some outliers when pinned-up case drops below nonpinned-up case. This is caused by the limitation of thread interference $I$ which is arbitrary to predict the thread resonance effect in some corner cases. However, in most cases, threads with larger inner product of $V_{IDI}$ would incur more VEs. On the average, VEs can be mitigated by 55% even under scenario with three pinned-up threads in each domain.

B. Performance Improvement

Orchestrator is an architectural solution to mitigate VE. It can work in alliance with fail-safe mechanisms to provide a complete VE tolerant solution. We study two representative fail-safe mechanisms cooperating with Orchestrator: Razor [27] and fast DPLL [28]. In the following section, we first evaluate the net performance improvement by Orch-VEO without migration impact. Then, the overall performance improvement is shown with taking migration impact into account.

1) Net Performance Improvement by Orchestrator: Razor

augments pipeline with shadow latches and control lines for in situ error detection and correction, which offers a fail-safe mechanism for dynamically voltage margin tuning. The voltage tuning interval of Razor is more than 10 $\mu$s [27], which is comparable with Orchestrator’s running window, so it’s reasonable to assume fixed voltage margin for one running window. In this experiment, we study the Razor scheme with voltage margin of 4%, while the margin for conservative worst-case-marginal solution (baseline) is 18% (a little lower than Power6’s). Assuming a 1.5$\times$ voltage to frequency scaling factor [1], [29], i.e., $A\%$ of voltage increase would transfer to 1.5 $\times$ $A\%$ of frequency increase. Then, the frequency of Razor scheme is 21% higher than baseline, and the frequency gap between these two schemes will become larger with technology scaling. However, cores will roll back for recovery if voltage droop exceed the 4% margin in Razor scheme, while the conservative solution need not to roll back. The recovery penalty is about 10-cycle period, which is comparable with the pipeline flushing cycles in modern processors. We show the detail of performance comparison between Razor-alone and Razor + Orch-VEO scheme in Fig. 12. MIPS of these two schemes are normalized to conservative worst-case-marginal scheme, and the workloads are sorted in an ascending order according to VE occurrences under Razor-alone scheme.

We can see that Razor-alone scheme performs well for those voltage-mild threads which are confronted few VEs and can run safely within tighter voltage margin. However, when comes to voltage-violent threads, the performance improvement will be rapidly shaved by VE tolerant penalties. We can see that in the right part of figure, the Razor-alone scheme performs much worse than the conservative margin scheme because Razor cannot do rapid voltage tuning when threads turn out to be voltage-violent and the recovery penalties soon whittle away the performance improvement. In cooperation with Orchestrator, the performance of voltage-violent threads can be significantly improved, shown as the line with markers. Because Orchestrator makes voltage feature prediction for threads and proactively schedules threads to minimize core resonances and creates much more chances for mild cores to offer relief to violent cores.

Fast DPLL is deployed in IBM’s Power7 processor which aims to handle unexpected voltage droops. Fast DPLL can decrease frequency by 7% in several cycles, which can assure timing margin at the onset of voltage droops [28]. After thousands of cycles, when voltage goes to the normal level, the slow DPLL would tune the frequency back to the normal frequency. Even if every core is equipped with one fast DPLL, cores still run in the same pace with identical frequency most often. Little frequency tuning in small period of time would not significantly change the situation of core interferences. We assume that 10 mV of voltage droops would bring about 100 MHz (5% of peak frequency) of frequency degradation. Our experimental results show that fast DPLL-alone can make 7% performance improvement over the baseline. By incorporating with Orch-VEO, the performance increases to 13%.
2) **Overall Performance Improvement by Orchestrator:** The overall performance is also evaluated, taking migration impact into account. We validate the compatibility of Orchestrator to work with other advanced S-NUCA solutions [20]. Using information given by those works, we can decide whether a thread should be migrated or stand still to do remote data access. Fig. 13 shows the performance comparison between Orch-VEO and Orch-MA. The curve with markers • represents the Orch-VEO cases and the curve with markers * represents the Orch-MA cases. We can see the performance of Orch-VEO is occasionally drooping below 70% of the baseline due to the migration impact. Regarding to the Orch-MA solution, since threads are pinned up on cores in knowing that migration may bring more than 6% of performance degradation, as a result, the bad cases will be largely improved by Orch-MA solution. As shown in the makers *, the bad cases are barely below 90%. Overall, Orchestrator is a good architectural solution to maintain performance in cooperating with the circuit-level fail-safe solutions.

**C. Thread Life Cycle Influence**

Generally, threads have several life stages during their execution: new, runnable, running, yield, and killed. When new thread is forked, it is set to runnable state and waiting to be scheduled. When the OS scheduler decides it is its turn to get CPU time, the thread goes into running state. When this thread occupies CPU for enough time, or a thread with higher priority shows up, this thread would be put to yield by OS scheduler. The scheduling algorithms of Orchestrator can be functioning properly when there are more threads than cores. The Orchestrator scheduler only takes care of threads which are in running state. OS tells Orchestrator the running threads ID, and then the Orchestrator monitors IDI of these threads. Once threads transfer from runnable status to running status, Orchestrator makes decision on thread-core affinities based on history IDI information. The thread-core affinities will be given back to OS to do thread scheduling. During thread switching, the IDI history information would be checkpointed as well as the other context information. Therefore, with the help of OS, Orchestrator can support the real cases that threads are generated and killed dynamically.

**D. Supporting for SMT**

If the processor supports SMT, multiple threads are running in a core simultaneously, the thread scheduling becomes more complicated since there are thread interferences inside one core. Orchestrator can still give a hint in this case. Previous work [30] discovered that VE in SMT processors is mainly caused by resource contention. To be more specific, if the stalled thread occupies pipeline resource such as reorder, load store queue or function unit, the other thread cannot execute either, which stalls the entire pipeline and incurs current fluctuation. Their work mitigates VE by using FLUSH fetch strategy which flushes the stalled thread out of pipeline to avoid resource contention. Orchestrator can alleviate this issue in an orthogonal way. It schedules threads into cores while taking into consideration of the possibility that the two threads are both blocked by cache miss, TLB miss or branch misprediction. A major concern is that the IDI prediction in SMT processors. Orchestrator can make IDI estimation if processors support recording performance counter information for each thread independently, which is possible since processor designers want to offer interfaces to assist programmers in finding the bottleneck of software. We conduct experiments in two-way SMT processors to validate the prediction accuracy for SMT-enabled scenarios. We find that it is less accurate than that in the SMT-disabled scenarios. This is because of the following reason: in SMT cores, threads share some pipeline resources, such as cache, reorder buffer, load/store queue, and register alias table of branch predictor. Therefore, threads interact with each other, leading to less accuracy of predicting IDI for every thread. The more threads in one core, the less IDI accuracy it may get. The prediction accuracy is shown in Fig. 14. As similar as Fig. 8, the x-axis denotes the actual thread droop intensity and y-axis denotes the predicted droop intensity. Our prediction model can predict 74% of samples. The overall rank prediction miss is about 26%, but only 14% of samples make severe miss (i.e., to confuse voltage-mild thread
in the Rank1 or Rank2 with voltage-violent one in Rank3 or Rank4. Hence, Orchestrator can still give guidance to some extent in two-way SMT processors.

Orchestrator scheduling strategy is to arrange the most violent thread to the mildest domain. Such scheduling algorithm may still work under the scenarios in which the number of threads is larger than that of cores. All the threads are sorted in an ascending order and stored in priority stack one by one. Then thread at the stack top is scheduled to the idle core in the mildest domain. When all cores are occupied by one thread and the priority stack is not empty, the thread at stack top will be scheduled to the mildest core in the mildest domain.

VII. RELATED WORKS

The VE tolerance in a tighter voltage margin condition is usually conducted in these two technical routines: low-penalty fail-safe mechanism and VE occurrence reduction. Some work focused on lightweight fail-safe mechanisms with low performance penalty. Razor [27] enhances pipeline with shadow latches and control lines which can execute error detection and recovery in cycles of time. Decor [31] uses delay committing techniques to ensure the correctness of committed instructions. Pan et al. [32] make efficient recovery based on vulnerability analysis and rolls back only when VEs truly incur timing errors. Some other work focused on VE mitigation. Previous work discovered that VE has a strong relationship with pipeline activities [33], such as pipeline flushing and long latency operation. Thus, some work eliminates VE based on prediction and throttling: using sensors or predictors to detect or predict the occurrences of VE and mitigate them by pipeline-throttling [34]–[36] or instruction scheduling [33]. For SMT processors, El-Essawy and Albonesi [37] mitigate VE by using fetching policy which prevents the stalled thread from occupying any more pipeline resources.

However, these solutions become less effective for multicore processors because of the increasing complexity of PDN and new program behaviors of applications. Reddi et al. [3] analyze the voltage noise in product multicore processors, which mainly focuses on the situation while running multiprogrammed workloads. They exploit oracle prior-known information to determine thread pairs, discovering that there is possibility to alleviate voltage droops by combining different workloads. We mainly focus on multithreaded programs. More specifically, the influence of SPMD programming model which may induce constructive interactions between cores and lead to burst voltage droops spatially and temporally. Based on this observation, we propose a feasible scheduling approach to mitigate voltage droop occurrence.

VIII. CONCLUSION

Voltage variation has become the dominant reliability problem in PDNs of multicore processors. Unlike monolithic single-core processors, light-weighted cores in the multicore architecture have intercore interference which is the key reason of significant voltage droops. Through analyzing the voltage interference of cores executing multithreaded applications, we observe that voltage variations are exacerbated during some similar, synchronized execution phases of multiple threads. When these threads are intelligently located among the cores, the amount of voltage droops can be greatly reduced. Based on that, Orchestrator, a sensor-free nonintrusive scheme for multicore architectures is proposed to smooth the voltage variations. Orchestrator focuses on the intercore power interactions. It makes prediction of threads interference effect based on performance counter information and allocates threads to avoid the voltage droops synergy among cores. Experimental results demonstrate that Orchestrator can mitigate VEs significantly and improve performance with fail-safe mechanisms.

REFERENCES


Guohai Yan (M’10) received the B. Sc. degree from Peking University, Beijing, China, in 2005, and the Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2011.

He is currently an Associate Professor with the Institute of Computing Technology, Chinese Academy of Sciences. His current research interests include heterogeneous architecture, low-power techniques, and network on chip.

Yu Hu (M’06) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Electronic Science and Technology of China, Chengdu, China, in 1997, 1999, and 2003, respectively.

She is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Her current research interests include architectural-level and circuit-level reliable design and fault diagnosis techniques.

Xiaowei Li (SM’04) received the B.Eng. and M. Eng. degrees in computer science from the Hefei University of Technology, Hefei, China, in 1985 and 1988, respectively, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1991.

He was an Associate Professor with the Department of Computer Science and Technology, Peking University, Beijing, from 1991 to 2000. In 2000, he joined ICT, CAS, as a Professor, where he is currently the Deputy Director of the State Key Laboratory of Computer Architecture. He has authored or co-authored over 150 papers in journals and international conferences, and he holds 21 patents and 29 software copyrights. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks.

Dr. Li has been the Chair of the Technical Committee on Fault Tolerant Computing, the China Computer Federation since 2008, the Vice Chair of the IEEE Asia Pacific Regional Test Technology Technical Council since 2004, and the Steering Committee Vice-Chair of the IEEE Asian Test Symposium since 2007. He was the Steering Committee Chair of the IEEE Workshop on RTL and High Level Testing. He is on the Technical Program Committee of several IEEE and ACM conferences, including VTS, DATE, ASP-DAC, and PRDC. He is a member on the editorial board of JCSCT, JOLPE, and JETTA.

Xing Hu received the B.S. degree from the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, in 2009. She is currently pursuing the Ph.D. degree in computer science with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

Her current research interests include computer architecture, particularly reliable design of power efficient architectures.