Orchestrator: a Low-cost Solution to Reduce Voltage Emergencies for Multi-threaded Applications

Xing Hu*†, Guihai Yan*, Yu Hu*, and Xiaowei Li*
*State Key Laboratory of Computer Architecture,
Institute of Computing Technology, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
{huxing,yan_guihai,huyu,lxw}@ict.ac.cn

***Abstract*—Voltage emergencies have become a major challenge to multi-core processors because core-to-core resonance may put all cores into danger which jeopardizes system reliability. We observed that the applications following SPMD (Single Program and Multiple Data) programming model tend to spark domain-wide voltage resonance because multiple threads sharing the same function body exhibit similar power activity. When threads are judiciously relocated among the cores, the voltage droops can be greatly reduced. We propose "Orchestrator", a sensor-free non-intrusive scheme for multi-core architectures to smooth the voltage droops. Orchestrator focuses on the inter-core voltage interactions, and maximally leverages the thread diversity to avoid voltage droops synergy among cores. Experimental results show that Orchestrator can reduce up to 64% voltage emergencies on average, meanwhile improving performance.**

I. Introduction

Inductive noise is a big challenge to power integrity. Imperfect parasitic parameters of PDN contribute to voltage variations which cause timing failures, thereby degrading the system reliability, availability, and serviceability. Traditional PDN design approaches rely on conservative voltage margin, i.e. over-design, to ensure safe timing even in the worst case of voltage droops [1]. However, with the decreasing working voltage and increasing density of integration, the over-design paradigm has been proven to be highly unsustainable [2].

In view of the limitation of over-design, prior studies proposed many solutions aiming to tighten the voltage margin, such as 1) avoid voltage emergencies (VE) by pipeline throttling[3], thread scheduling [4] or software scheduling [5], 2) use sensors or predictors to detect or predict the occurrence of VE [6][7], and 3) handle VE by additional circuitry for rapid [8][9] or efficient recovery [10]. However, these solutions focus on PDN of monolithic single-core processors and become less effective for multi-core processors. When core counts scale up, core-to-core voltage interferences will be the main challenge to voltage integrity [2][11].

Miller et al. observes that barriers could cause destructive core-to-core interferences during the execution of multi-threaded applications [12]. This work eliminates the voltage emergencies by staggering the threads into a barrier and sequentially stepping over it. In our work, we observe another prominent culprit for VE: SPMD (Single Program and Multiple Data) programming model.

* Corresponding author: Yu HU, E-mail: huyu@ict.ac.cn. This work is supported in part by National Natural Science Foundation of China (NSFC Program) under Grant No.(61076018, 61274030, 61100016, 61076037 and 60921002), and in part by National Basic Research Program of China (973 Program) under Grant No.2011CB302503.

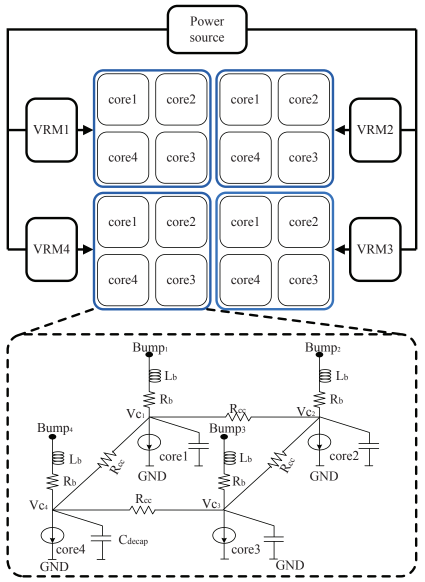

Fig. 1. Four-domain processor baseline and PDN model

Compared to the barriers, SPMD model can spark inter-core voltage resonance more frequently, because multiple threads share the same function body leading to similar power activities of cores. Staggering solution is less effective to SPMD-induced VE because of two reasons: 1) given the phase of a thread is varying over time, it's hard to figure out a guaranteed phase staggering solution that can fit for the whole execution epoch; 2) even with such a staggering solution, it will inevitably penalize performance because some threads will be held for a while to shift their power phases.

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

1) We observe that the applications following SPMD programming model can frequently arouse large voltage fluctuations.

2) Based on the observation, we propose "Orchestrator" , one low-cost, sensor-free scheme to smooth voltage variations. We first analyze and predict the variation intensity of running threads from runtime statistics, and then figure out the optimal thread mapping for each thread to avoid the voltage droop synergy. All these operations are conducted on the fly.

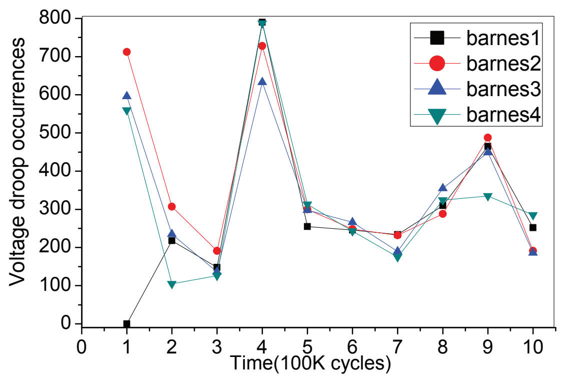

Fig. 2. Voltage emergency traces of `barnes`

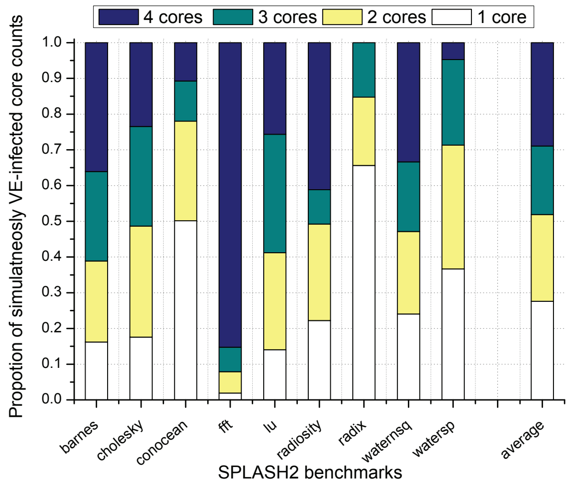

Fig. 3. Simultaneously VE-infected core counts

The rest of this paper is organized as follows: Section II gives the motivation of this paper, followed by the detailed description of proposed Orchestrator scheme in Section III. Section IV and Section V present the experimental setup and results. The conclusions are given in Section VI.

II. Background and Motivation

A. Multi-core PDN baseline

It's a popular design style for multi-core processors to divide the PDN into multiple voltage domains [13]. Such "divided-and-conquer" PDN design is scalable which supports advanced power management efficiently as the core counts scale up [14][15]. The domains are physically independent from each other and the inter-domain voltage interferences can be minimized. In this paper, we assume such a baseline PDN with multiple independent domains. Without loss of generality, we use a 16-core processor divided into four domains as the baseline. Each domain has an independent power source, as prior work [14][13] assumed.

B. Voltage variation of SPMD applications

There are two preconditions for core resonance: 1) the cores in the same domain show similar power profiles in the same phases. 2) the variation strength of profiles is strong enough. We find that for multi-threaded applications, the first requirement is usually satisfied. Take `barnes` (a SPLASH benchmark) for example, we break down the voltage emergency traces of four `barnes` threads running in four independent domains, counting voltage emergencies (degrade by 4% of nominal voltage) in every 100K cycles. The voltage emergency traces of the four threads (`barnes1`,..., `barnes4`) turn out to be almost at the same pace, as shown in Figure 2, which create a big chance for strong core resonance if the four threads are scheduled into the same domain. The results shown in Figure 3 confirm this argument.

Figure 3 illustrates the impact of core resonance in a voltage domain, i.e. to show how many cores will suffer from the emergencies concurrently. The result shows that on average up to 73% emergencies jeopardize more than two cores simultaneously, and about 30% emergencies can put all cores in danger. Hence, it's crucial to mitigate core resonance in multi-core PDN.

III. Orchestrator Principle

The key principle of Orchestrator is to avoid domain-wide destructive noise by orchestrating the co-running threads, i.e. to judiciously depart the threads with similar power profiles into different domain, thereby minimizing the resonance.

To do so, we need to 1) quantify each thread's aggressiveness to the other threads, and 2) develop an algorithm to guide thread scheduling that can minimize voltage resonance. The following section is devoted to these two issues.

A. Quantify threads' aggressiveness

We use Intrinsic Droop Intensity (IDI) to characterize how aggressive a thread can arouse the voltage fluctuations when stand-along. A thread's aggressiveness is an intrinsic property of the thread, and won't change with different co-running threads. A thread's IDI is quantified as the occurrences of large voltage droops when running alone in a given time window (1 million cycles in this paper), which is estimated by micro-architectural events during running time (detailed implementation is in Session IV.B). The rationale behind IDI is based on the fact that threads with higher IDI are more voltage-violent and more likely to suffer from voltage emergencies and "infect" the neighbouring cores. On the other hand, threads with lower IDI are voltage-mild and can offer current relief for neighbouring cores.

The effect can be exemplified by Figure 4. There are four applications (`barnes`, `conocean`, `watersp`, `cholesky`) running on the processor. 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. The X axis represents the threads' IDI and the Y axis represents the threads' actual VE occurrences in one million cycles with different co-runners. Initially, the threads forked by the same parent are scheduled in the same domain, as the solid markers such as "■" show. There are three out of four `barnes` threads show high IDI. Unsurprisingly, all `barnes`'s threads suffer from intensive VE. The same situation also applies on Domain3 hosting `watersp`. By contrast, the Domain2 hosting `conocean` and Domain4 hosting `cholesky` show low IDI which results in non-intensive VE.

After re-mapping the threads, as the hollow markers such as "□" show, the VE occurrences of the original VE-intensive domains can be greatly reduced. But those VE-non-intensive domains hardly exacerbate, such as Domain2 denoted by "♢" and Domain4 by "▽". These results show that the IDI is a faithful metric to quantify threads' aggressiveness.

B. Scheduling algorithms

The scheduling algorithm is to determine the location of each thread, on which the detrimental inter-core interferences can be

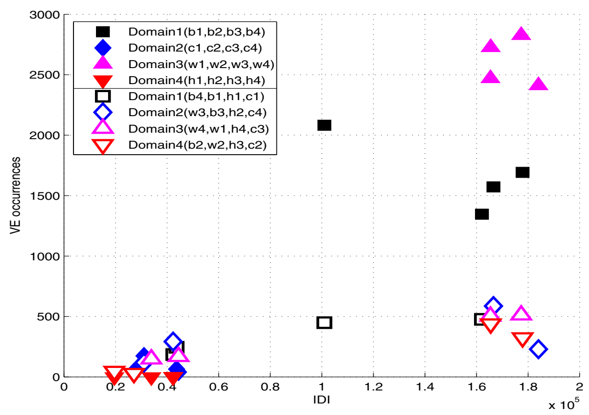

Fig. 4. Relationship between IDI and VE intensity

minimized. To simplify the discussion, we assume that cores do not support simultaneous multi-threading, so each core can only run one thread at a time. Each thread in a given scheduling window is associated with an IDI vector, denoted by V_{IDI}. Suppose the running window is ten millions' cycles, for example, then thread T_i's V^i_{IDI} is a ten-element vector denoted by $[IDI_1, IDI_2, \ldots, IDI_{10}]$. Given the threads with similar voltage traces are more likely to resonate with large voltage droops, we use the inner product between V_{IDI}s to quantify the intensity of voltage interferences. Large inner product between two threads' indicates that the two threads have greater chance to resonate with large voltage droops. By contrast, small inner product implies that the two threads exhibit the different voltage traces and unlikely resonate with each other. The interferences between thread i and thread j, denoted by $I(i,j)$, is defined as follows:

$$I(i,j) = V^i_{IDI} \cdot V^j_{IDI}$$

The larger I, the stronger interferences. The emergency level EL of a voltage domain (S) is defined as the aggregated I of the threads in the same domain, i.e,

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

To sum up, the objective can be described as follows, given the n-domain processors and the V_{IDI} of each thread. Our algorithm is to figure out a thread mapping M which minimizes the arithmetic mean of EL.

To make optimal arrangement of threads is very timing consuming. The time complexity of exhaustive search is $C^n_t * C^n_{t-n} * C^n_{t-2n} * \ldots * C^n_n / n!$, where C is the combination, n is the number of domains and t is the number of cores. This combinational complexity is at the magnitude of t^t, and can hardly support on-the-fly implementation. Thus, we propose a heuristic algorithm.

First we transform high-dimension vectors to scalar values by the modular operation. Then sort threads in a descending order according to the norm of V_{IDI}: $T_{m_1}, T_{m_2}, T_{m_3} \ldots T_{m_t}$. Threads with higher V_{IDI} norm have the higher priority to select neighbor threads. Take T_{m_i} for example. When it's thread T_{m_i}'s turn to choose neighbor threads, we first calculate inner product of T_{m_i} with every available threads, and then choose the $n-1$ neighbor threads that have the smallest inner product and mark the selected threads as unavailable.

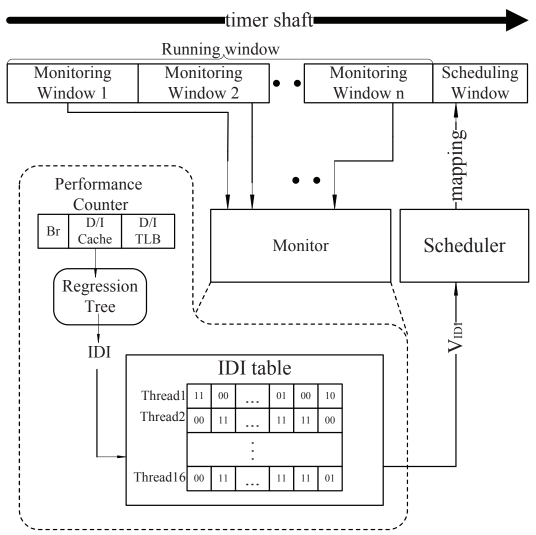

Fig. 5. Organization of Orchestrator

IV. Implementation

A. Hardware components

As shown in Figure 5, Orchestrator includes two components: *Monitor* and *Scheduler*. During every monitoring window, *Monitor* is responsible for estimating and recording IDI of each thread. It estimates IDI based on performance counter statistics. Threads' IDI information will be recorded in IDI table. At the end of a running window, *Scheduler* will be triggered to compute the new thread mapping based on the information recorded in IDI table, then it initiates thread scheduling for the next running window.

The input of *Scheduler* is the vector of threads' IDI which ranges from 0 to about 200K according to 400 randomly selected program slices. In order to simplify the hardware implementation and filter the noise of IDI, we linearly separate them with the minimal scale of 50K. The IDI between 0 and 50K is classified to Rank 1, between 50K to 100K is in Rank 2, between 100K to 150K is in Rank 3, and more than 150K is in Rank 4. The separation must retain the resolution capability of IDI. Statistically, voltage emergency occurrences of power domain running Rank1-threads, Rank2-threads, and Rank3-threads are 3%, 10%, 51% normalized to Rank4-threads, respectively. Thus , threads in different ranks perform different voltage characteristics and have different infection capability. After ranking, the entry of IDI table can be designed as two bit-width which causes little area overhead and reduces the hardware complexity.

B. IDI acquisition

The key implementation issue is how to get IDI of every thread in runtime. Orchestrator relies on IDI to guide scheduling. However IDI, as defined, is an intrinsic feature of a thread and cannot be obtained by direct measuring since the inter-thread interference is inevitable at runtime. We find that IDI is tightly correlated with some microarchitecture activities, while these activities of different threads show much less interferences than their voltage implication, which motivates a regression-based IDI-acquisition approach.

1) IDI regression model: The voltage characteristics have been proven to tightly correlate with several microarchitectural activities such as branch misprediction, cache miss and TLB miss [16][2]. These activity patterns can serve as a proxy to measure IDI. How-

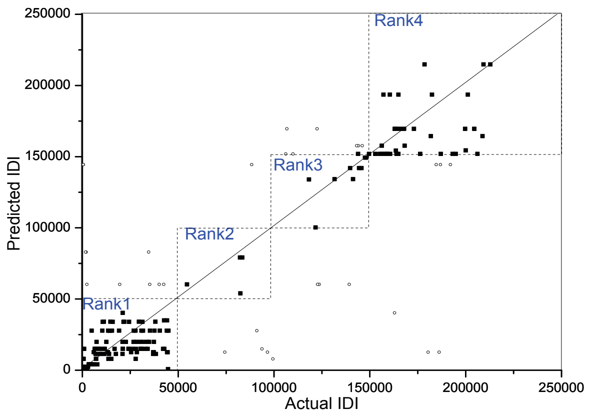

Fig. 6. Accuracy of regression tree

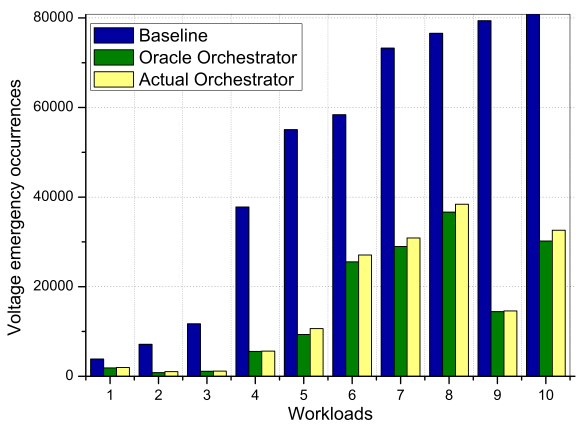

Fig. 7. Voltage emergency reduction of 10 workloads

ever, these runtime statistics are nonlinearly correlated, while the regression tree is an ideal approach to cope with such a relationship.

Regression tree describes a complex nonlinear relationship with a rule-based model [17], widely being used in data mining. The regression tree model is composed of a group of rulers, each having two parts, a condition and a simple multi-variant regression model. When condition is satisfied, the associated fitting model is chosen to calculate the predicted value. Regression tree makes prediction in almost realtime because it just looks up constants in the tree instead of complicated calculations. In this work, we use it to fit the relationship between the performance counter information and IDI. More specifically, we take branch misprediction intensity, L1 and L2 cache miss intensity, and TLB miss intensity as the input variables. They can be easily obtained by a set of performance counters. The output is the corresponding 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 of threads are gathered as the training set to generate regression tree. The droop intensity can be calculated by using online measuring sensors such as CPM [18]. To avoid interferences, during the training stage only one thread is running in a power domain at a time. When the regression tree is trained to reach a stable state, we build it into the target chips to acquire IDI traces.

2) Regression accuracy: We use the SPLASH-2 benchmarks to evaluate the prediction accuracy. SPLASH-2 benchmarks traces are cut into 1M-cycle samples, and 100 samples are randomly taken as the training set which is enough to train the regression tree to a stable state. Then we apply the regression tree to other 250 samples as the testing set to evaluate the accuracy. The result is shown in Figure 6. In the 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 Rank1 with Rank4). The regression tree is very cost-effective. The regression tree in this paper can be implemented with thousands of logic gates. So the hardware overhead is negligible.

V. Experimental Setup

We simulated a 16-core processor as the baseline. The core configuration is listed in Table V. We use the full-system simulator GEMS and Wattch [19] to generate power traces which transfer to current traces using constant voltage level. Current traces are fed into Hspice PDN model. Then Hspice simulates the voltage traces of each core.

TABLE I
Core configuration

Parameters	Configuration
Number of Cores	4
Clock Frequency	2.0 GHz
Fetch/Decode Width	4 instructions/cycle
Branch-Predictor Type	64 KB bimodal gshare/chooser, 1K entries
Reorder Buffer Size	128
Unified Load/Store Queue Size	64
Physical Register File	32-entry INT, 32-entry FP
INT ALU, INT Mul/Div, FP ALU, FP Mul/Div	4/2/4/2
L1 Data Cache	16KB, 2-way, 32B line-size, 1-cycle latency
L1 Instruction Cache	16KB, 2-way, 32B line-size, 1-cycle latency
L2 Unified Cache	1MB, 4-way, 64B line-size, 16-cycle latency
I-TLB/D-TLB	64-entry, fully-associative

Orchestrator are implemented with full-system simulations using Virtutech Simics with Multifacet GEMS extension [20]. We use SPLASH-2 benchmark suite [21] which covers most computation and communication patterns as workloads.

The PDN for the baseline 16-core processors comprises four independent domains, as shown in Figure 1. Each domain resembles that of Intel Xeon 5500 processors [22]. In each power domain, each core are modeled as a timing-varying current source with decoupling capacitance and the intra-core connections are modeled as resistive paths. The detailed parameters are the same with prior work [23].

Note that Orchestrator is an architectural solution to reduce the voltages emergencies, but cannot totally eliminates the VE. We therefore assume the target processor is equipped with fail-safe mechanisms which will be engaged when a failure is about to happen or already happened. The fail-safe mechanisms may be either in recovery style or dynamic frequency tuning style, such as Razor [8], and fast DPLL-enhanced method [24].

VI. Experimental Results

In this section, we first evaluate the reduction of voltage emergencies by Orchestrator. Then we analyze the impact of thread migration on performance. At the end we show the performance improvements brought by Orchestrator.

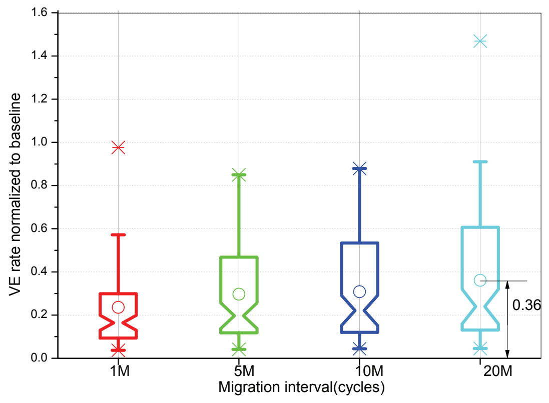

Fig. 8. VE rate normalized to baseline

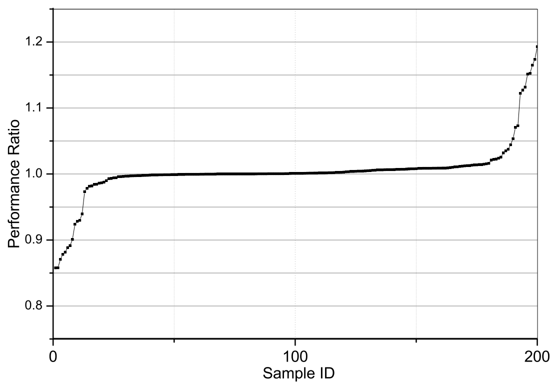

Fig. 9. Performance impact of thread re-mapping

A. Voltage emergency reduction

Figure 7 shows the voltage emergency reduction by Orchestrator-enabled processor, with voltage-mild dominant, voltage-violent dominant and mixed workloads shown in Table 2. The “Oracle” assumes Orchestrator with perfect inference accuracy, while “Actual” represents the actual scenarios with imperfect prediction accuracy of 87. For workload 2, 3, 4, and 5, the emergency reduction is more than 80%. The reduction for workload 1, 7, 8 and 10 is not as dramatic as the others, 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 Orchestrator can dramatically reduce the voltage emergencies. Also, 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.

Here we further show the statistical effectiveness of Orchestrator for 100 workloads which are randomly constituted from SPLASH2 benchmarks. Given the length of running windows also affect the effectiveness, we study the emergency reduction under 1M, 5M, 10M, and 20M cycles, respectively. Figure 8 gives the result using “boxplot” which shows 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. Even at the largest interval of 20M cycles, the average reduction is still about 64%. We also find that in rare corner cases, Orchestrator may increase the emergency rate, as the outliers show in the results. It is because of that the thread power phases change so dramatically that the inference gets missed.

TABLE II
WORKLOADS

ID	Benchmarks
1	`conocean,fft,lu,radix`
2	`lu,barnes,radix,fft`
3	`barnes,conocean,radix,watersp`
4	`fft,waternsq,conocean,cholesky`
5	`radix,watersp,barnes,cholesky`
6	`radix,waternsq,barnes,fft`
7	`radiosity,conocean,watersp,radix`
8	`barnes,cholesky,radiosity,waternsq`
9	`conocean,radiosity,waternsq,watersp`
10	`cholesky,radiosity,waternsq,watersp`

B. Thread re-mapping impact on performance

Orchestrator is based on thread scheduling which may potentially penalize performance. Thread migration initiates the context switching in cores and causes performance overhead due to lost of hot data or mandatory warm-up. The system performance (in billion instructions per second) is normalized to that of without migration. Our baseline is configured with a shared last-level cache (LLC), as most commercial multi-core processors adopted. Prior work [25] [26] witnessed that migration between cores sharing a LLC usually imposes small negative impact on performance. So we reduce the migration interval to 1M cycles, attempting to highlight the worst cases. Figure 9 shows the results for 200 workloads. The results 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 in shared LLC cache, the data is statically mapped. Re-mapping the threads may change the distance between one thread and its data in LLC. Some data originally remote becomes local data, 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. Overall, although Orchestrator needs to re-map the threads, the performance overhead won’t be an essential limitation.

C. Performance improvement

Orchestrator is an architectural solution to mitigate voltage emergency. It needs a fail-safe mechanism to recover from failures. We study two representative fail-safe mechanisms cooperating with Orchestrator: Razor [8] and fast DPLL(Digital Phase Lock Loop) [24].

Razor augments pipeline with shadow latches and control lines for in-situ error detection and correction, which offers an fail-safe mechanism for dynamically voltage margin tuning. The emergency rate depends on the voltage margin reserved. In this experiment, we study three margins: 4%, 4.5%, and 5%. The voltage tuning interval of Razor is more than 10 microseconds [8], which is comparable with Orchestrator’s running window. So it’s reasonable to assume fixed voltage margin for one running window. For Razor, the recovery penalty is about 10 cycles, which is comparable with the pipeline flushing cycles in modern processors.

Fast DPLL is the latest work which aims to handle unexpected voltage droops deployed in IBM’s Power7 processor. Fast DPLL can decrease frequency by 7% in several cycles, which can assure timing margin at the onset of voltage droops [24]. After thousands of cycles, when voltage goes to the normal level, the slow DPLL would tune the frequency back to the normal frequency. Although

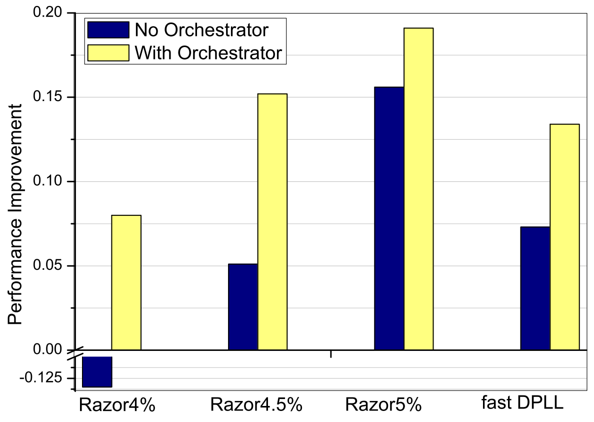

Fig. 10. Performance improvement comparison

every core is equipped with one fast DPLL, cores still run in the same pace with identical frequency most often. Little fixed tuning in one small window would not significantly change the situation of core interferences. We assume that 10mv of voltage droops can be transferring to 100Mhz (5% of peak frequency) of frequency degradation.

Figure 10 shows the result. Under tight margin (i.e. 4%) in this example, Razor-alone cannot improve performance, but degrades performance due to overly high failure rate. With the relaxed voltage margin, Razor can help gain better performance. However, Razor cooperating with Orchestrator can greatly improve performance, even with the tightest margin set. Fast DPLL-alone can make 7% performance improvement over the baseline. By incorporating Orchestrator, the performance increases to 13%. In other words, Orchestrator is an ideal architectural solution to cooperate with the circuit-level fail-safe solutions.

VII. Conclusion

We observe that the multi-threaded applications execution phases may exacerbate voltage fluctuations, because the threads forked by one program may cause voltage resonance in one power domain. Mixing diversified threads together in one power domain can greatly reduce voltage droops. We propose one metric of intrinsic droop intensity to characterize threads' voltage features and using performance counter information to estimate it on the fly based on a regression model. We also propose an online heuristic algorithm which dramatically reduces voltage emergencies. The experimental results show that Orchestrator can dramatically reduce voltage emergencies by 64% and thereby improve system's performance.

References

[1] N. James, P. Restle, J. Friedrich, B. Huott, and B. McCredie, "Comparison of split-versus connected-core supplies in the power6 microprocessor," in *International Solid-State Circuits Conference (ISSCC)*, pp. 298–604, Feb. 2007.

[2] V. Reddi, S. Kanev, W. Kim, S. Campanoni, M. Smith, G.-Y. Wei, and D. Brooks, "Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling," in *International Symposium on Microarchitecture (MICRO)*, pp. 77–88, Dec. 2010.

[3] M. Powell and T. Vijaykumar, "Pipeline damping: a microarchitectural technique to reduce inductive noise in supply voltage," in *International Symposium on Computer Architecture (ISCA)*, pp. 72–83, June 2003.

[4] W. El-Essawy and D. Albonesi, "Mitigating inductive noise in smt processors," in *International Symposium on Low Power Electronics and Design(ISLPED)*, pp. 332 –337, Aug. 2004.

[5] M. S. Gupta, K. K. Rangan, M. D. Smith, G.-Y. Wei, and D. Brooks, "Towards a software approach to mitigate voltage emergencies," in *International Symposium on Low Power Electronics and Design(ISLPED)*, pp. 123–128, Aug. 2007.

[6] V. Reddi, M. Gupta, G. Holloway, G.-Y. Wei, M. Smith, and D. Brooks, "Voltage emergency prediction: Using signatures to reduce operating margins," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 18–29, Feb. 2009.

[7] E. Grochowski, D. Ayers, and V. Tiwari, "Microarchitectural simulation and control of di/dt-induced power supply voltage variation," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 7–16, Feb. 2002.

[8] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge in *International Symposium on Microarchitecture (MICRO)*, Dec. 2003.

[9] M. Gupta, K. Rangan, M. Smith, G.-Y. Wei, and D. Brooks, "Decor: A delayed commit and rollback mechanism for handling inductive noise in processors," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 381–392, Feb. 2008.

[10] S. Pan, Y. Hu, X. Hu, and X. Li, "A cost-effective substantial-impact-filter based method to tolerate voltage emergencies," in *International Conference on Design, Automation and Test in Europe (DATE)*, pp. 311–315, Mar. 2011.

[11] 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," *ACM Trans. Archit. Code Optim.*, vol. 7, pp. 12:1–12:28, Oct. 2010.

[12] T. Miller, R. Thomas, X. Pan, and R. Teodorescu, "Vrsync: Characterizing and eliminating synchronization-induced voltage emergencies in many-core processors," in *International Symposium on Computer Architecture (ISCA)*, pp. 249 –260, June 2012.

[13] E. Rotem, A. Mendelson, R. Ginosar, and U. Weiser, "Multiple clock and voltage domains for chip multi processors," in *International Symposium on Microarchitecture (MICRO)*, pp. 459–468, Dec. 2009.

[14] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," vol. 37, pp. 290–301, June 2009.

[15] G. Yan, Y. Li, Y. Han, X. Li, M. Guo, and X. Liang, "Agileregulator: A hybrid voltage regulator scheme redeeming dark silicon for power efficiency in a multicore architecture," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–12, Feb. 2012.

[16] M. Gupta, V. Reddi, G. Holloway, G.-Y. Wei, and D. Brooks, "An event-guided approach to reducing voltage noise in processors," in *International Conference on Design, Automation and Test in Europe (DATE)*, pp. 160–165, Apr. 2009.

[17] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, "Classification and regression trees," 1984.

[18] A. Drake, R. Senger, H. Deogun, G. Carpenter, S. Ghiasi, T. Nguyen, N. James, M. Floyd, and V. Pokala, "A distributed critical-path timing monitor for a 65nm high-performance microprocessor," in *International Solid-State Circuits Conference (ISSCC)*, pp. 398–399, Feb. 2007.

[19] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *International Symposium on Computer Architecture (ISCA)*, pp. 83–94, May. 2000.

[20] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, pp. 92–99, Nov. 2005.

[21] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *International Symposium on Computer Architecture (ISCA)*, pp. 24–36, June 1995.

[22] A. Chandrasekhar, D. Ayers, F. Yahyaei-Moayyed, and C.-C. Huang, "Chip-package-board co-design of a 45nm 8-core enterprise xeon processor," in *Electronic Components and Technology Conference (ECTC)*, pp. 536–542, June 2010.

[23] G. Yan, X. Liang, Y. Han, and X. Li, "Leveraging the core-level complementary effects of pvt variations to reduce timing emergencies in multi-core processors," in *International Symposium on Computer Architecture (ISCA)*, pp. 485–496, June 2010.

[24] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, and J. B. Carter, "Active management of timing guardband to save energy in power7," in *International Symposium on Microarchitecture (MICRO)*, pp. 1–11, Dec. 2011.

[25] Q. Teng, P. Sweeney, and E. Duesterwald, "Understanding the cost of thread migration for multi-threaded java applications running on a multicore platform," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 123–132, Apr. 2009.

[26] J. Brown, L. Porter, and D. Tullsen, "Fast thread migration via cache working set prediction," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 193 –204, Feb. 2011.