Predicting Architectural Vulnerability on Multithreaded Processors under Resource Contention and Sharing

Lide Duan, Lu Peng, *Member*, *IEEE Computer Society*, and Bin Li

Abstract—Architectural vulnerability factor (AVF) characterizes a processor's vulnerability to soft errors. Interthread resource contention and sharing on a multithreaded processor (e.g., SMT, CMP) shows nonuniform impact on a program's AVF when it is co-scheduled with different programs. However, measuring the AVF is extremely expensive in terms of hardware and computation. This paper proposes a scalable two-level predictive mechanism capable of predicting a program's AVF on a SMT/CMP architecture from easily measured metrics. Essentially, the first-level model correlates the AVF in a contention-free environment with important performance metrics and the processor configuration, while the second-level model captures the interthread resource contention and sharing via processor structures' occupancies. By utilizing the proposed scheme, we can accurately estimate any unseen program's soft error vulnerability under resource contention and sharing with any other program(s), on an arbitrarily configured multithreaded processor. In practice, the proposed model can be used to find soft error resilient thread-to-core scheduling for multithreaded processors.

Index Terms—Hardware reliability, modeling and prediction, modeling of computer architecture

1 Introduction

Soft errors have been significantly degrading the reliability of current high-performance processors. They occur mainly due to the electronic noises caused by energetic nuclear particles (e.g., alpha particles, neutrons, and pions) from the environment [50]. These particles may invert the state of a logic device (from "0" to "1," or from "1" to "0") when the resulted charge has been accumulated to a sufficient amount, introducing soft errors (i.e., transient faults) into the system. With the feature size and supply voltage scaling down to extremely small values, current processors become highly vulnerable to soft errors [17], [27], [29], [30], [34], [40], [44], [47], [48]. In the past decades, many companies have observed severe damage caused by soft errors on large servers [50].

However, not all soft errors will affect the final output of the program. For example, a bit flip in an invalid (empty) reorder buffer (ROB) entry will not have any effect in the program execution; similarly, overwriting a corrupted register before its erroneous value can be used by other instructions prevents the error propagation. Based on this observation, architectural vulnerability factor (AVF) [28], [4] was proposed to quantify the probability that a soft error finally produces a visible error in the program output. A higher AVF value indicates higher vulnerability to soft errors, so the AVF is used by computer designers as an important reliability metric at the architectural level. In this paper, we characterize and predict the AVF when a program[1] runs under resource contention and sharing with other program(s) on multithreaded processors, including simultaneous multithreading (SMT) and chip-multiprocessor (CMP) architectures.

Motivation. The AVF provides useful guidelines in designing reliable processors, but its measurement is extremely expensive in terms of hardware and computation. To measure the AVF, one can use statistical fault injection (SFI) [43], [24] or architecturally correct execution (ACE) analysis [28]. The former requires a large number of experiments that randomly inject errors into program execution; while the latter needs to implement a postcommit analysis window [28], [15] to identify the hardware bits that are required for correct execution. Regardless, either of these two methods results in costly overhead and significant performance degradation.

The ACE analysis can be also applied to multithreaded processors to measure the AVF. However, the measurement is even more involved than in single-threaded processors because multiple instruction streams from different threads need to be traced. For multithreaded workloads with data sharing, a system-wide, much bigger analysis window must be implemented. Therefore, an accurate AVF prediction in place of expensive AVF measurement can eliminate a great amount of overheads.

- *L. Duan is with AMD, Inc., 3014 W. William Cannon Dr. Apt. 1823, Austin, TX 78745. E-mail: lide.duan@amd.com.*
- *L. Peng is with the Division of Electrical and Computer Engineering, School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA 70803. E-mail: lpeng@lsu.edu.*
- *B. Li is with the Department of Experimental Statistics, Louisiana State University, Room 173, Martin D. Woodin Hall, Baton Rouge, LA 70803. E-mail: bli@lsu.edu.*

Manuscript received 7 Feb. 2012; revised 27 Aug. 2012; accepted 12 Nov. 2012; published online 27 Nov. 2012.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2012-02-0018. Digital Object Identifier no. 10.1109/TDSC.2012.87.

1. In this paper, a program refers to a single-threaded program. We use "program" and "thread" interchangeably. A multiprogrammed workload refers to a program combination, and a multithreaded workload refers to a program with multiple threads that have data sharing.

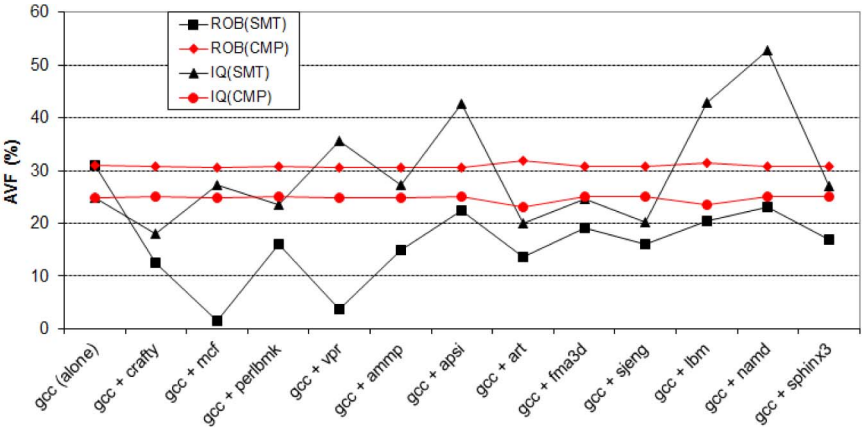

Fig. 1. The AVF variations of *gcc* (SPEC 2000) when it is coscheduled with different benchmarks on SMT/CMP architectures.

Difficulties. Our previous work [11] predicted the AVF for a single-threaded superscalar processor from its performance metrics. In contrast, the work in this paper extends the prediction to multithreaded architectures, thus facing a variety of difficulties. First, interthread resource contention and sharing significantly and nonuniformly affect the AVF.[2] Fig. 1 shows how the AVF of reorder buffer (ROB) and issue queue (IQ) would vary when *gcc* runs alone and against different benchmarks on a two-way SMT and a dual-core CMP (which are based on the same processor configuration for comparison purposes). In the SMT processor, ROB is private to each thread, and IQ is shared between threads. We can see that the impact is relatively small in CMP, whose interthread contention is mainly located in the shared cache but barely affects processor structures' AVFs. On the other hand, the contention resulted from pipeline resource sharing in a SMT processor significantly and nonuniformly affects the AVFs when *gcc* is coscheduled with different benchmarks. In addition, despite the strong variation, ROB AVF of SMT is consistently lower than that when *gcc* runs alone, but IQ AVF may be higher (e.g., *gcc* + *apsi*) or lower (e.g., *gcc* + *crafty*). This interesting observation introduces new issues into AVF behavior (compared to contention-free superscalar's AVF [11]), indicating higher difficulty in accurately predicting the multithreaded processor's AVF.

Second, the problem complexity and scope are significantly enlarged in the context of multithreading. AVF reflects soft error masking effect at both program level [37] and machine level [38], so AVF prediction should take into account both the application and the processor configuration. Walcott et al. [41] performed the prediction across SPEC CPU 2000 benchmarks on a fixed machine configuration; our previous work [11] extended the prediction to be across a very small set of configurations by changing only four parameters. Nevertheless, both of these two works were restricted to single-threaded processors with certain simplifications. In contrast, this work intends to correlate two important (but complex) problems: the processor configuration being from a statistically large design space, and the prediction effectiveness being across different multiprogrammed and multithreaded workloads.

Consequently, the interthread contention and the interaction between software program and hardware configuration significantly increase the complexity of predictive modeling. As a result, the approaches adopted in prior studies [11], [41] fail in making accurate AVF prediction for multithreaded processors (details can be referred to Section 3.3). To tackle these difficulties, this paper proposes a novel scheme by decoupling the prediction into two levels.

Our proposal: two-level predictive modeling. In this work, we propose a scalable two-level predictive mechanism capable of accurately predicting key processor structures' soft error vulnerability on multithreaded processors under resource contention and sharing. At the first level, a cross-program model is trained to predict the contention-free AVF on a single-threaded processor. The inputs to the first-level model include a few important performance measurements (e.g., structure occupancies, cache miss rates) from the contention-free execution and the corresponding configuration parameters. The output of the first-level model, along with key processor structures' occupancies measured when the program runs against other program(s) on a multithreaded processor, are inputted to the second-level model, which finally predicts the program's AVF under resource contention and sharing with others.

Essentially, the first-level model uses key parameters and simple performance measurements to characterize underlying hardware configuration and the software program, respectively. This level takes into account the software-hardware interaction, providing a contention-free baseline to the second level. On top of it, the second-level model captures the interthread resource contention and sharing via examining multiprogrammed executions. By employing the proposed two-level prediction, we are capable of accurately predicting the AVF for an unseen program when it is coscheduled with any other program(s) on an arbitrarily configured SMT/CMP architecture.

Contributions. In summary, the main contributions of this paper are as follows:

- *Universal prediction of the AVF on single-threaded processors*. The first-level model accurately predicts the contention-free AVF on any given processor

2. In this paper, AVF always refers to the AVF of hardware structure(s), which can be private to some thread or shared between threads. Sometimes, we use a thread's (program's) AVF to refer to the AVF of processor structures that are private to the thread.

configuration from a design space. This model is universal across different programs, and also validated for unseen programs not used in training.

- *Universal prediction of the AVF under contention across multiprogrammed workloads*. The second-level model takes the knowledge of the contention-free AVF from the first-level model, performing an accurate prediction of the AVF under resource contention for any program combination in analysis. This combined capability is extremely useful in the era of multithreading.
- *Analysis and dynamic prediction on multithreaded workloads with data sharing*. Section 6 investigates the impact of memory allocation and program parallelism on AVF and cumulative system vulnerability, performing a dynamic AVF prediction for multithreaded workloads during runtime.
- *A case study of soft error resilient thread-to-core scheduling*. By utilizing the proposed AVF prediction, Section 7 presents a case study that identifies the optimal thread-to-core assignment that minimizes the AVF of a chip multithreaded (CMT) processor.

2 Methodology: Boosted Regression Trees (BRT)

A nonparametric tree-based predictive scheme named boosted regression trees (BRT) serves as building blocks to our two-level model. In this section, we introduce BRT and its interpretations.

BRT [14] is an ensemble technique that employs two algorithms: Regression trees from classification and regression trees (CART) [6] and Boosting that builds and combines a collection of trees. CART is a recursive binary splitting algorithm that partitions the input space into a set of rectangles (i.e., the leaves of the tree); Boosting is an enhancement to tree-based methods, iteratively fitting a number of regression trees based on the training data and gradually increasing the emphasis on the observations modeled poorly by the current model.

The detailed BRT algorithm used in this paper is described in Fig. 2. We consider a problem with n observations $\{y_i, X_i\}, i = 1, 2, \ldots, n$, where X_i is a p-dimensional input vector and y_i is the response. Step 3 is the inner loop that recursively constructs a binary tree; in particular, Step 3.4 splits a region (a node in the tree) into two subregions such that the best fit (i.e., the minimal Mean Square Error) in the current iteration can be achieved among all possible split points. The tree construction terminates when its depth reaches a certain number; in this work, we set the maximum tree depth to 3, indicating that at most eight leaves will be generated for each tree. The outer loop (Steps 2 to 4) is the boosting procedure that combines a large number of tree models to improve the prediction accuracy. In Step 3.3, I (.) is an indicator function which returns 1 (otherwise 0) if its argument is satisfied; ν is a parameter between 0 and 1, controlling the learning rate of the procedure. Empirical results have shown that smaller values of ν always lead to better generalization errors [14], so we fix ν at 0.01 in this study.

BRT is inherently nonparametric and can well handle mixed types of input variables. It does not make any assumption on the distribution of the input variable, thus avoiding the transformations used in preprocessing the training data. BRT is also capable of capturing complex behavior using a relatively small number of inputs. This is in contrast to some other multivariate nonlinear modeling techniques, in which cases extensive inputs from the analyst, analysis of interim results, and subsequent modifications of the method are required. Besides, BRT is insensitive to outliers, and unaffected by monotone transformations and different scales of input measurements. Prior studies [14] have shown the superior performance of BRT over traditional prediction techniques under various types of data. Some other works [7], [46] provided insights to the theoretical properties and practical aspects of Boosting, such as the efficiency, convergence, and consistency. Our prior work [22] quantitatively showed that BRT is more stable than linear regression.

In addition to accurate prediction, BRT also provides visualized model interpretations. The *input variable importance* measures the relative importance of an input variable to the response via accumulating the number of times the variable (dimension) is selected for splitting a region during all the iterations. Every increment to this number is weighted by the improvement in the MSE as a result of the corresponding split. Furthermore, the *partial dependence plot*

Boosted Regression Trees (BRT)

{

1. Initialize the prediction function $\hat{f}_0(X_i) = \bar{y}$, where $\bar{y}$ is the average for $\{y_i\}, i = 1,2,\ldots, n$. Initialize the iteration index m to 1.

2. Compute the current residuals:

$$r_{im} = y_i - \hat{f}_{m-1}(X_i),\ i = 1,\ldots,n.$$

3. For each input variable j and each split point s (i.e. a possible value) of j:

3.1. Partition the current space into two parts (X is any data point in current space):

$$p_{1m}(j,s) = \{X \mid X_j <= s\} \text{ and } p_{2m}(j,s) = \{X \mid X_j > s\}$$

3.2. For each part, compute its constant fit:

$$\gamma_{hm} = \arg\min_{\gamma} \sum_{X_i \in p_{hm}} (r_{im} - \gamma)^2, h = 1,2.$$

3.3. Update the fitted model:

$$\hat{f}_m(X) = \hat{f}_{m-1}(X) + \nu \times \sum_h \gamma_{hm} I(X \in p_{hm})$$

3.4. Choose the j and s whose updated $\hat{f}_m(X)$ results in the lowest Mean Square Error (MSE) for current data points. Partition the current space into R_1 and R_2 with the chosen j and s, and update the prediction function as in Step 3.3.

3.5. For each of R_1 and R_2, repeat Step 3 unless the tree depth reaches a threshold or the improvement in the MSE is smaller than a certain value.

4. Increment m by 1. Repeat Step 2 to 4 unless m reaches a threshold.

}

Fig. 2. The BRT algorithm used in this paper.

TABLE 1
The Inputs and Outputs of the Proposed Model at Different Levels

Model Level	Inputs	Output
Level 1	7 contention-free performance metrics (IPC, L1 DCache miss rate, L2 cache miss rate, the occupancies of ROB, LSQ, IQ, and RF) + 9 configuration parameters (P_1 to P_9 in Table 3)	contention-free AVF
Level 2	contention-free AVF + 9 structure occupancies under contention (ROB, LSQ, IQ, RF, I/D TLBs, L1 I/D caches, and L2 cache)	AVF under contention

shows the effect of a subset of input variables on the response after accounting for the average effect of all the other input variables in the model. Given any subset $\mathbf{x}_s$ of the input variables indexed by $\mathbf{s} \subset \{1, \ldots, p\}$, the partial dependence of $f(\mathbf{x})$ is defined as $F_s(\mathbf{x_s}) = E_{\mathbf{x}_{\setminus s}}[f(\mathbf{x})]$, where $E_{\mathbf{x}_{\setminus s}}[\cdot]$ refers to the expectation over the joint distribution of all the input variables with indices not in $\mathbf{S}$. In practice, the partial dependence can be estimated from the training data by $\hat{F}_{\mathbf{s}}(\mathbf{x_s}) = (1/n)\sum_{i=1}^{n} \hat{f}(\mathbf{x_s}, \mathbf{x}_{i\setminus \mathbf{s}})$, where $\{\mathbf{x}_{i\setminus \mathbf{s}}\}_1^n$ are the data values of $\mathbf{x}_{\setminus s}$.

3 Two Level AVF Prediction

3.1 Description

The ultimate goal of this work is to predict the AVF of a program in contention with other program(s) running simultaneously on a multithreaded processor (e.g., SMT, CMP) without AVF measurement mechanisms. We will show that it can be predicted from the AVF when this program runs alone with no contention, and a group of important performance metrics reflecting the occupancy rates[3] of key processor structures under contention. The latter can be easily measured during program execution on a SMT/CMP architecture, but the former cannot. Therefore, we further predict the contention-free AVF for the underlying single-threaded processor. We organize the prediction in a two-level model in this section.

At the first level, a universal model is trained to predict the AVF in a contention-free single-threaded processor. We first sample a group of training configurations from a huge processor design space, simulating them for benchmarks in the training set. The AVF and a few performance measurements (e.g., IPC, cache miss rates, and structure occupancies) are measured after simulation. The model at this level is trained using these performance measurements and the corresponding configuration parameters as the inputs and the contention-free AVF as the response. Table 1 lists the inputs and the output at each level of our model. Fig. 3 illustrates the training procedure for Level 1. For brevity, only the inputs from a certain benchmark $\mathbf{B_k}$ are shown; but this procedure is actually performed for all the n training benchmarks.

For the second-level model involving interthread contention, Fig. 4 takes a benchmark combination $(\mathbf{B_i}, \mathbf{B_j})$ as an example to illustrate its training procedure. The AVF of $\mathbf{B_i}$ from its single-threaded execution and a few processor structure occupancies measured from the multiprogrammed execution where $\mathbf{B_i}$ competes with $\mathbf{B_j}$ serve as the inputs to the model; while the AVF of $\mathbf{B_i}$ in contention with $\mathbf{B_j}$ is the response to be predicted (see the third row of Table 1). We provide training samples from different benchmark combinations to make the second-level model universal across different benchmark combinations. The performance metrics used at this level reflect the interthread contention and sharing on multithreaded processors; in contrast, those at the first level are measured from contention-free single-threaded processors. Regardless, all of these performance metrics are easy to be measured in either simulators or real processors via performance counters.

By combining the two levels, we are able to predict the AVF on a SMT/CMP architecture running any benchmark combination. As shown in Fig. 5, to predict the AVF of a benchmark $\mathbf{B_k}$ (in particular, $\mathbf{B_k}$ can be any unseen benchmark not in the training set) running against another benchmark, say $\mathbf{B_t}$, on any given processor configuration, we need to follow the below steps:

1. Run the single-threaded simulation for $\mathbf{B_k}$ on the processor configuration in analysis; collect the performance measurements after simulation.

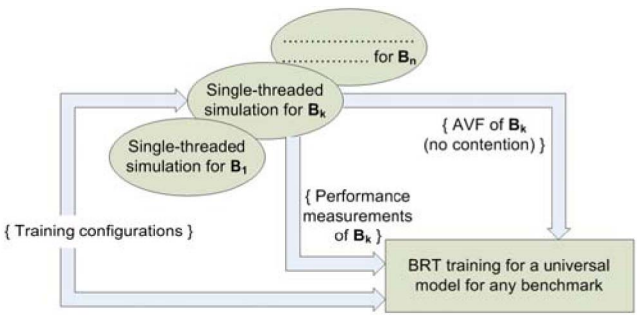

Fig. 3. The first level universal model training using BRT. Only the inputs from a certain benchmark $\mathbf{B_k}$ are shown.

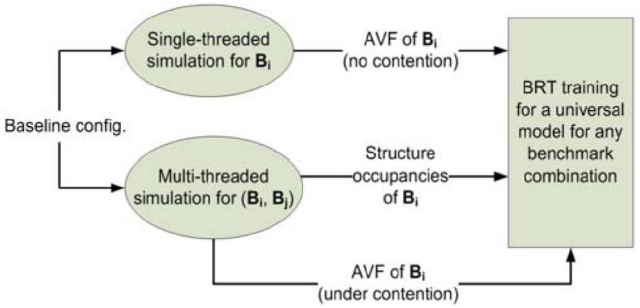

Fig. 4. The second level universal model training using BRT.

3. The occupancy rate of a processor structure is calculated as its proportion of entries that are in use.

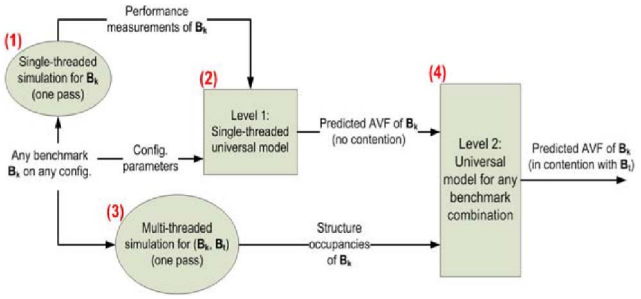

Fig. 5. An overview of the two-level AVF prediction on a multi-threaded processor.

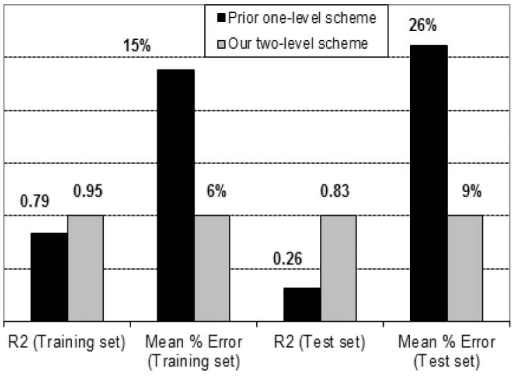

Fig. 6. Comparison between the prior one-level scheme and the proposed two-level scheme. Higher R-Square (R2) and lower mean percent error are better.

2. Give the above performance measurements and the configuration parameters as inputs to the first-level model; predict the contention-free AVF of $\mathbf{B_k}$.
3. Run the multiprogrammed simulation for the benchmark combination $(\mathbf{B_k}, \mathbf{B_t})$; measure key processor structure occupancy rates under contention.
4. Provide the predicted contention-free AVF and the measured structure occupancies as inputs to the second-level model; finally predict the AVF of $\mathbf{B_k}$ under resource contention with $\mathbf{B_t}$. Note that, to make prediction for an unseen benchmark $\mathbf{B_k}$, the above approach only needs two passes of simulation: one for $\mathbf{B_k}$ itself with no contention, and one for $\mathbf{B_k}$ and its competitor with contention.

3.2 Reasoning

Fundamentally, two levels are utilized in our model because a single-level model cannot well handle the high complexity introduced by the issues targeted in this work. First, our model explores the configuration design space: Using the same model, we can make prediction for any arbitrarily configured processor in a statistically large design space. Second, our model is effective across different programs or program combinations: we can make prediction for even unseen programs using the same model. Third (and most importantly), our model takes into account the contention and sharing in multiprogrammed/multithreaded executions: We can make prediction for SMT/CMP-based architectures. Consequently, the complex interactions among these issues raise higher demands on the predictive model whose functionality can no longer perform well within a single level. Previously, one-level models were only used to target a single aspect, either the design space [9], [10] or across programs [41], [11]. Lee et al. [21] investigated both the design space and the multiprogrammed contention, in the performance domain though, also using a model with two levels that are composed together.

As an example, we use register file (RF) to demonstrate the necessity of the two levels in AVF prediction. We trained a direct one-level model [11], [41] to predict the SMT processor's RF AVF. The same prediction technique as in [11] was used since it demonstrated better performance than linear regression used in [41]; the inputs to this model are the various performance measurements when different threads are run simultaneously. For a fair comparison, the training and test sets (described in Section 4.1) are the same for these two approaches. Fig. 6 compares the prediction accuracy between these two models in terms of R-Square (higher is better with 1 as the maximum) and mean percentage error (lower is better with 0 percent as the minimum). We can see that the one-level scheme shows more than two times higher error rates than the two-level scheme. Especially, for the test set, the mean percentage error reaches a very high value (26 percent) while the R-Square is unacceptably low (0.26). Therefore, two levels are necessary in making accurate predictions for complex scenarios. Decoupling the prediction into two levels reduces the model complexity at each level, also improving prediction accuracy.

Taking a closer look at the above example, we highlight the importance of contention-free AVF for the RF. Program-level behavior plays an important role in the RF AVF: For instance, those registers occupied by the dynamic dead code have valid but unACE bits. These are reflected in the contention-free AVF, which serves as the baseline in our two-level model to later interact with the contention inputs. However, the one-level scheme did not precisely generate this information, thus performing poorly in making the final prediction as a whole. In Section 4.3, we will (Fig. 8) show the quantitative importance of the contention-free AVF inputted at the second level.

3.3 Discussions

Other than eliminating the measurement overheads, the biggest advantage of our AVF prediction is its effectiveness across multiprogrammed workloads. In other words, once the model is trained, it can be applied to different program combinations on multithreaded processors. This is in contrast to traditional application-specific design space studies that build a separate model for each workload. As the multiprogrammed workloads have combinatorial growth in the number of possible program combinations, our "universal" predictive approach can save a great amount of training costs, thus being extremely useful in the era of multithreading.

On the other hand, in order to make prediction for a program under contention, our model needs the corresponding contention-free execution of the same program. This is easily available when the program is repeatable (e.g., when using known benchmarks). However, in a real system, re-executing an arbitrary application with no contention may be impractical. In such cases, we can still make prediction for certain structures after simplifying the model. From Section 4, we will see that for queue-based

structures such as ROB, LSQ, and IQ, the AVF under contention largely depends on the performance measurements under contention; contention-free AVF has very little impact in these cases. Therefore, for these structures we can remove the first-level model and retrain the second-level model only using the occupancies. The resulted simplified model, which does not need contention-free execution anymore, can still make accurate predictions for those queues. A similar example can be referred to Section 6.2.

4 Impact of Multiprogrammed Resource Contention on AVF

4.1 Experimental Setup

We implement the AVF measurements [28], [4], [15] in M-Sim3.0 [31] to model the soft error vulnerability of key processor structures. In M-Sim's SMT model, each hardware thread has its own ROB and load store queue (LSQ), but other structures including IQ, functional units (FU), and Physical RF are shared among threads. Basically, an ICOUNT fetcher [39] fetches instructions for each thread, storing them in the corresponding ROB/LSQ after decode and rename; each thread dispatches instructions in a round-robin manner from its own ROB into a shared IQ in contention with other threads; the interthread contention exists in the remaining pipeline stages until commit. On the other hand, the CMP model creates a separate core along with the private L1 I/D caches for each thread, but the L2 unified cache is shared among cores.

In this work, we measure and study the AVF of the following five structures: ROB, LSQ, IQ, FU, and Physical RF. To measure the AVF, all the committed instructions need to go through a 40K-entry postcommit analysis window that determines the type of the instruction. The major types include:

1. first-level dynamically dead, whose result is not used by any younger instruction;
2. transitively dynamically dead, whose result is only used by instructions that are also dynamically dead;
3. ACE, whose bits are critical for correct execution;
4. unknown; and
5. NOP and prefetch instructions, and so on.

This information is then used to calculate the AVF. For FU, we also take into account the logic-masking effect in the operands, e.g., a multiply instruction with one source operand being zero can tolerate errors in other source operands. Furthermore, we use physical memory addresses and physical register numbers to track the data dependencies. This is critical for accurate AVF measurement of multithreaded programs that have data sharing. Overall, the program-level impact [37], [38] on the AVF has been taken into account in this work.

Thirty-eight benchmarks from SPEC CPU 2000 and 2006 suites are evaluated. For the other SPEC 2000/2006 benchmarks not included here, we could not compile them into Alpha binaries runnable in the M-Sim simulator. We will first study two-way SMT and dual-core CMP, and then discuss model scalability to more than two threads. All the benchmarks are simulated on our single-threaded baseline configuration first (whose parameters are shown in bold in Table 3), and then categorized in Table 2 according to the measured IPC and L2 cache miss rates. We use SimPoint toolkit [32] to

TABLE 2
Benchmarks Used in Multiprogrammed Workloads

IPC	L2 Cache Miss Rate	Training Benchmarks	Test Benchmarks
> 1.0	< 30%	*gcc(135), sphinx3(12), apsi(382), eon(201), gzip(372), h264ref(272), hmmer(342), mesa(322), perlbmk(5), gap(324), gromacs(285), sjeng(498), 06bzip2(633), fma3d(150)*	*calculix(200), namd(45), vortex(570)*
< 1.0	> 30%	*mcf(142), art(23), swim(234), lucas(597), equake(408), vpr(292), ammp(283), astar(449), lbm(69)*	*milc(85), mgrid(236)*
> 1.0	> 30%	*wupwise(95), libquantum(200), applu(281)*	*facerec(208)*
< 1.0	< 30%	*galgel(415), crafty(113), bzip2(152), gobmk(217)*	*parser(270), twolf(176)*

The number of fast-forwarded instructions (unit: 100 M) is shown after the benchmark name.

TABLE 3
Processor Configuration Design Space Composed of Parameters $\mathbf{P_1}$ to $\mathbf{P_9}$

	Parameter	Selected Values	# Options
$\mathbf{P_1}$	Processor width	2, **4**, 8	5
	# Integer ALUs / # FP ALUs	1/1-associated with processor width 2 2/2, **4/4**-associated with processor width 4 4/4, 8/8-associated with processor width 8	
$\mathbf{P_2}$	ROB size	72, 84, **96**, 108, 120, 132, 144, 156, 168	9
$\mathbf{P_3}$	LSQ size	24, **32**, 40, 48, 56, 64	6
$\mathbf{P_4}$	IQ size	32, 40, 48, 56, **64**, 72	6
$\mathbf{P_5}$	L1 I/D cache size	**16**, 32, 64, 128 KB (64B block, 4-way)	4
	L1 cache latency	**1**, 2, 3, 4 cycles (vary with L1 cache size)	
$\mathbf{P_6}$	L2 cache size	**512**, 1024, 2048, 4096 KB (64B block)	4
	L2 cache latency	**12**, 14, 16, 18 cycles (vary with L2 cache size)	
$\mathbf{P_7}$	L2 cache assoc.	4, **8**	2
$\mathbf{P_8}$	Branch predictor	bimod(1024), bimod(2048), **gshare(10-bit L1 width, 1024 L2 entries)**, gshare(11-bit L1 width, 2048 L2 entries), combined(1024), combined(2048)	6
$\mathbf{P_9}$	BTB	**512/4**, 512/8, 1024/4, 1024/8	4

The values shown in bold are used in our baseline setting.

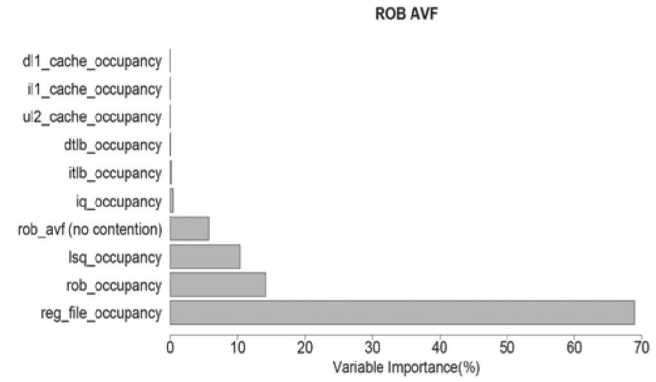

Fig. 7. Input variable importance of ROB AVF in SMT processors.

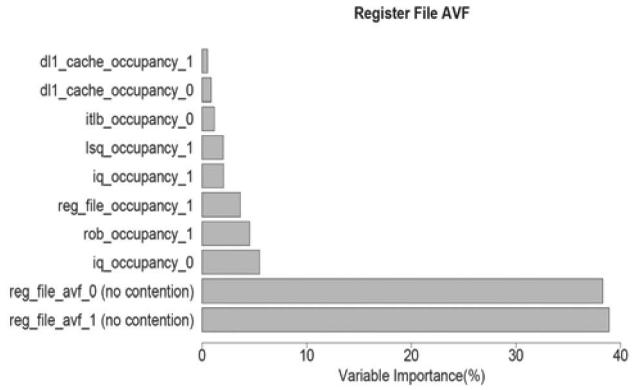

Fig. 8. Input variable importance of RF AVF in SMT processors.

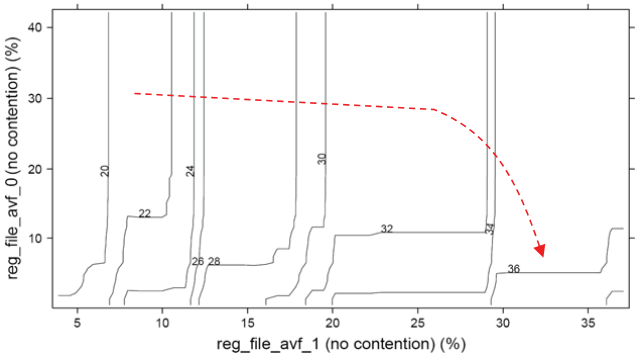

Fig. 9. Partial dependence plot of RF AVF on the two most important input variables in SMT processors.

derive a representative 100-million instruction phase for detailed simulation for each benchmark; the number of fast forwarded instructions is shown after the benchmark name in this table. In order to capture different CPU/memory behavior, we randomly choose some benchmarks from each of the four categories for test (the fourth column of Table 2). Consequently, the 435 two-threaded combinations generated from the 30 training benchmarks (i.e., $C_{30}^2 = 435$) will be used as the training samples; the trained predictor will be tested with the other 268 combinations, in which cases at least one benchmark is from the test set (i.e., $C_8^2 + 30 \times 8 = 268$).

In this section, we will discuss the second-level model first because we are more interested in the AVF under contention. All the training data used in this section are directly measured from the baseline configuration. In Section 5, this model will be combined with the first-level model to demonstrate the effectiveness of the entire two-level predictive mechanism.

4.2 Impact on SMT Private Structures

ROB and LSQ are private to each thread in our SMT model, only containing instructions from a single thread, no matter whether the thread runs alone or competes with another thread. However, the interthread contention from the shared pipeline resources strongly affects their AVF. We discuss the ROB AVF as an example in this section. Fig. 7 demonstrates the input variable importance derived from the ROB AVF predictor. Interestingly, we can see that the ROB AVF largely depends on the thread's RF occupancy (rather than the occupancy of ROB itself). RF plays an important role in the ROB AVF because in a multiway SMT processor the physical registers available for each thread are limited. The processing of the instructions in the pipeline is thereby more sensitive to the RF usage, and so is the AVF. In addition, we observe that the ROB AVF of a program shows strong variation when it competes with different programs, but is consistently lower than that when the program runs alone (as exemplified in Fig. 1). This is because the shared resource contention reduces the thread's RF occupancy to different extents in different benchmark combinations.

4.3 Impact on SMT Shared Structures

The other three SMT structures: IQ, FU, and RF are shared among different threads. For example, the instructions from both threads coexist in the IQ of a two-way SMT processor, simultaneously affecting the IQ AVF. Consequently, we need to provide the predictor with inputs from both threads during training. In this section, each of the two programs in a benchmark combination is simulated for 100 million instructions. We suffix the input variables from the faster thread (i.e., the one finishes earlier) with "0," and those from the slower thread with "1." For instance, "iq_occupancy_0" refers to the average IQ occupancy rate of the faster thread.

Fig. 8 shows the 10 most important inputs to the RF AVF. As shown, both threads simultaneously affect the soft error vulnerability of a shared structure. In the case of RF, the AVFs when the two threads execute independently with no contention have the highest importance. This actually indicates the necessity of our two-level model because it internally quantifies these important variables. Fig. 9 depicts the partial dependence of the RF AVF on the two contention-free AVFs. As shown, the RF AVF increases with respect to the increase of the slower thread's AVF and the decrease of the faster thread's. This indicates that the RF AVF would increase if one thread dominates the RF. In this case, more dependencies exist among registers (since they are from the same thread), and the RF thereby contains more ACE bits that are vulnerable to soft errors. Note that the dominance in RF by one thread is an artifact of our experimental setup where the faster thread simply terminates execution after finishing 100M instructions. However, the observation and reasoning derived from the dominance still remains true. Furthermore, different than SMT private structures, the AVF of a shared structure can be higher or lower than the corresponding contention-free AVF (e.g., IQ AVF in Fig. 1). The shared structure accommodates both threads, so there is

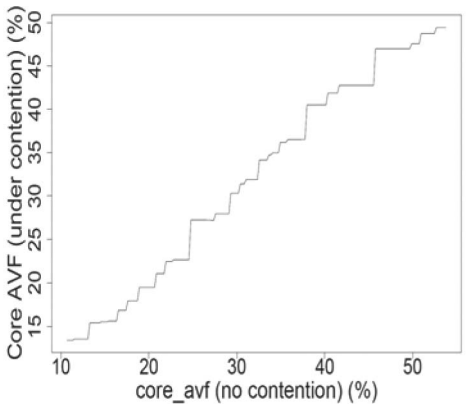

Fig. 10. Partial dependence plot of CMP core AVF on its most important input.

always a certain portion of the structure that is occupied by a different thread and demonstrates different AVF behavior than that when only one thread is run.

4.4 Impact on CMPs

In our dual-core CMP model, the two cores compete with each other via the shared L2 cache. Therefore, all the five processor structures in consideration are private to each core. We generally follow the same approach as above to train the predictors. The core AVF is the ratio between the number of the entire core's ACE bits and the core size. It can be calculated by summing up the individual components' AVFs weighted by the corresponding structures' sizes. We focus on the core AVF in the CMP study.

From our experiments, we find that the AVF of a CMP core running a certain benchmark varies in a very small range when different benchmarks execute on the other core. In other words, the shared cache contention has relatively little impact on a CMP core's soft error vulnerability. This is in contrast to the situation in an SMT processor where the AVF shows significant variation. Fig. 10 verifies the above finding by showing that the core AVF under contention is highly correlated with the core AVF with no contention.

5 Two-Level AVF Prediction for Multiprogrammed Workloads

The model obtained in Section 4 assumes the awareness of the program's AVF on a single-threaded processor with no contention. This contention-free AVF is also predicted in our model at the first level. This section first validates the first-level model, and then combines the two levels to demonstrate accurate predictions.

5.1 Single-Threaded Universal Model Validation

We tune several important parameters to form a large processor design space (Table 3) for the first-level model. The values used in the baseline configuration (i.e., the one based on which we trained the second-level model above) are shown in bold. The design space size is 1,244,160, from which we randomly and uniformly simulate 400 points for each of the benchmarks. After simulation, a few important but easily measured performance metrics are collected. We use 300 configurations of the 30 training benchmarks (the third column of Table 2) to train the model, which is then tested with the other 100 configurations of the 30 training benchmarks and all 400 configurations of the eight test benchmarks.

Fig. 11 shows the prediction results for the core AVF. We use percentage error (i.e., $|$predicted value $-$ true value$|$/true value * 100 percent) to report the prediction accuracy. Fig. 11 is a boxplot showing the distribution of percentage errors for different benchmarks. In a boxplot, the upper and lower boundaries of the central gray box correspond to the upper and lower quartiles of all the errors; the highlighted horizontal line within the box is at the median; the vertical dotted line drawn from the box boundaries extend to the border lines for outliers. The top

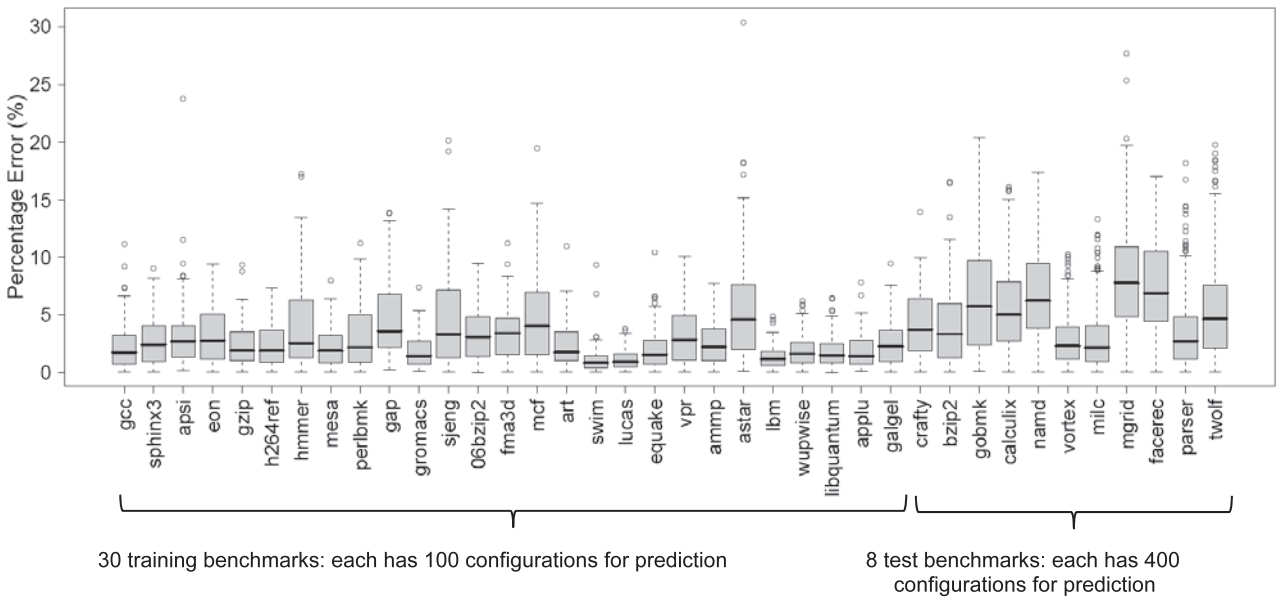

Fig. 11. Prediction accuracy (core AVF) of the first-level model. This model predicts 100 configurations for each of the 30 benchmarks in the training set (left) and 400 configurations for each of the eight benchmarks in the test set (right).

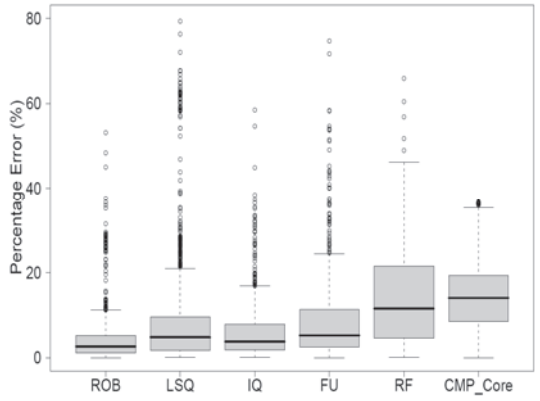

Fig. 12. Prediction accuracy from two-level prediction.

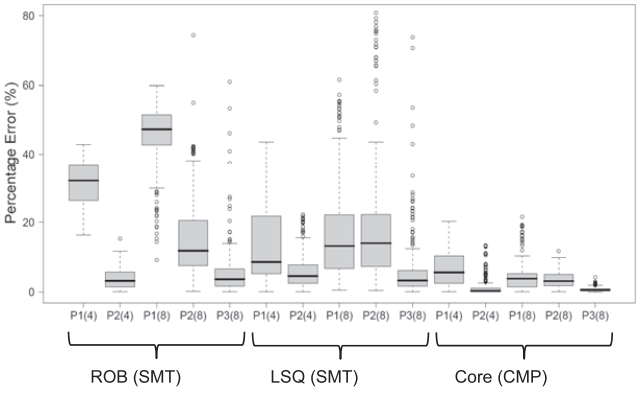

Fig. 13. Prediction accuracy when scaling to 4 and 8 threads.

border line is $Q3 + 1.5^{*}IQR$ and the bottom one is $Q1 - 1.5^{*}IQR$, where Q1 and Q3 are the first and third quartiles and IQR is the interquartile range Q3-Q1. Any observation outside that range is considered as an outlier, and denoted by a circle. From this figure, we can see that the BRT-based universal model is very accurate for all benchmarks, achieving a 2.94 percent error rate, on average. Note that only one single model is constructed here and tested across different benchmarks.

5.2 Combining the Two Levels

Upon obtaining the models at both levels, we are able to combine them to perform the entire prediction. Fig. 12 demonstrates the prediction accuracy in terms of percentage errors from the two-level prediction. It achieves high accuracy with median percentage errors of 2.64, 4.80, 3.76, 5.21, 11.56, and 14.03 percent for the five structures in SMT processors and the core AVF of CMPs, respectively. Note that the simulation results of the eight test benchmarks (the fourth column of Table 2) are not used in model training at either level, validating the applicability of our predictive scheme to unseen programs. Besides, the two-level predictive model can work for both homogeneous and heterogeneous multithreaded processors as long as the contention-related performance metrics can be measured. In reality, this can be easily satisfied since hardware performance counters for key processor structures are already available in most commercial processors.

5.3 Model Scalability

Our predictive mechanism is scalable to more than two threads. Additional training to the predictor may be needed for some structures. To discuss the scalability issue, we further perform a set of 4-threaded (on four-way SMT and quad-core CMP) and 8-threaded (on eight-way SMT and eight-core CMP) multiprogrammed simulations. For 4-threaded workloads, we evaluate the 70 combinations generated from the eight test benchmarks (i.e., $C_8^4 = 70$); for 8-threaded workloads, we randomly choose 45 combinations from all the benchmarks. In addition to more hardware threads, the size of physical RF in four-way SMT processors is doubled compared to the two-way ones; while in eight-way SMT, more parameters are reasonably enlarged.

As an example, Fig. 13 shows the prediction results for ROB/LSQ AVFs in SMT processors and the core AVF in CMP. For each of them, three predictors are trained: P1 is the one from Section 4 (only trained with 2-threaded samples); P2 is trained with both 2-threaded and 4-threaded samples (we put 25 percent 4-threaded simulation results in the training set); P3 is trained with 2-threaded, 4-threaded, and 8-threaded samples (similarly, 25 percent 8-threaded simulation results are used in training). We evaluate the 4-threaded test samples with P1 and P2, and the 8-threaded test samples with all three predictors, respectively. For instance, P2(4) indicates the prediction accuracy for 4-threaded test samples using P2. As a result, there are five bars shown for each structure in this figure.

For ROB AVF, we observe a big improvement in the prediction accuracy using the predictors with additional training (i.e., P2 and P3). In Section 4, we illustrated that the RF occupancy significantly affects ROB AVF in 2-threaded SMT processors. However, in the 4-threaded case, since the RF has been enlarged in our simulations, it is no longer the bottleneck. Therefore, the ROB occupancy becomes the most influential factor. P1 did not capture this behavior very well, thus performing poorly in 4-threaded and 8-threaded predictions. In contrast, in the case of CMP core AVF, even P1 can make very accurate predictions when scaling to more threads. This is because the correlation between AVF and input variables does not change drastically in CMPs with more than two cores. To summarize, the predictors may need more training when scaling to more threads, but the proposed approach itself is scalable.

6 Analysis on Multithreaded Workloads

Multithreaded workloads partition the computation work among multiple software threads, exploring thread-level parallelism. The multiple threads generated from the same program not only experience contention in pipeline resources, but also show constructive behavior in memory hierarchy due to data sharing. However, the impact of data sharing on soft error vulnerability is still largely unexplored for multithreaded workloads. This section first analyzes the impact of memory allocation and program parallelism in multithreaded programs, and then conducts the dynamic AVF prediction for them.

All experiments in this section are run using the M5 simulator [3], which is capable of simulating multithreaded benchmarks. We implement the AVF measurements for

TABLE 4
Some Performance Measurements of the Four SPLASH2 Programs

	LU (Contig.)	*LU* (Noncontig.)	*Ocean* (Contig.)	*Ocean* (Noncontig.)
Execution Time (ticks)	3.16E+011	5.50E+011	5.15E+011	7.74E+011
L1D Cache Miss Rate	1.5%	14.0%	22.5%	23.6%
Branch Misses Per 1K Insts.	9.75	10.49	1.11	6.48

ROB, Load Queue, Store Queue, and IQ for multicores with Alpha 21264-like CPUs. A set of multithreaded benchmarks from SPLASH2 [45] suite are evaluated.

6.1 Impact of Multithreaded Implementation

A different implementation of the same algorithm or a different solution to the same problem could make a big difference for a multithreaded program's performance as well as its soft error vulnerability. In this section, we examine two pairs of SPLASH2 benchmarks (*LU* and *Ocean*) to investigate the impact resulted from different memory usages (contiguous versus noncontiguous). The *LU* kernel factorizes a dense matrix into the product of a lower triangular matrix and an upper triangular matrix; the *Ocean* application studies large-scale ocean movements based on eddy and boundary currents. For each of these two benchmarks, there are contiguous and noncontiguous versions that allocate data differently in the memory. The contiguous version arranges contiguous data in the memory of each thread, thus improving data locality and having better performance.

Table 4 shows some performance measurements for the four programs in analysis. As expected, the contiguous *LU* and *Ocean* have shorter execution time than their noncontiguous counterparts. However, the left panel of Fig. 14 demonstrates that the core AVF of the noncontiguous implementation can be higher (in *LU*) or lower (in *Ocean*) than the contiguous one. Noncontiguous *LU* stores the 2D matrix in a 2D array, which results in a noncontiguous memory fragment for each block in this matrix. Consequently, the L1 data cache miss rate (Table 4) increases to 14 percent for noncontiguous *LU* compared to 1.5 percent for its contiguous version. Because of the high cache miss rate, a large number of instructions congest the pipeline, resulting in higher utilization of pipeline resources and higher AVF. On the other hand, the cache miss rate does not show significant difference for *Ocean*; however, noncontiguous *Ocean* encounters much more branch mispredictions (6.48 versus 1.11 per 1K instructions). A higher branch misprediction rate slows down the program execution, but also lowers the AVF because the squashed instructions are not vulnerable to soft errors. Hence, from these two examples, we can see that merely optimizing the performance for multithreaded programs may degrade the soft error reliability.

While increasing the number of concurrent threads usually reduces the execution time, it has nondeterministic impact on the core AVF. This can be seen from the left panel of Fig. 14 where the 1-, 2-, and 4-threaded results are shown. Note that the values shown in this panel are the average AVF of different cores due to core homogeneity. On the other hand, system vulnerability [35], [1] has been proposed to characterize the *cumulative* soft error vulnerability over time for multicores (while the AVF reflects the per-cycle average soft error rate). Suppose the AVF of core i in an n-core processor is $\mathrm{AVF_i}$ ($0<=\mathrm{i}<\mathrm{n}$), the processor's system vulnerability is $(\sum_{i=0}^{n-1} AVF_i) * ExecutionTime$. The right panel of Fig. 14 shows this measurement for the four programs with different numbers of threads. We can see that increasing the number of threads usually increases the cumulative system vulnerability. In particular, there is an apparent increase between 2-threaded and 4-threaded runs. This is because the reduction in the execution time due to more threads cannot overcome the increase in the AVF summation of all cores.

6.2 Dynamic AVF Prediction

Since the multithreaded program's AVF significantly varies in different situations, an effective AVF prediction across programs and different numbers of threads is useful and necessary. In this section, we dynamically predict multithreaded workloads' AVF from processor structure occupancies and data cache coherency states. This prediction is performed during program runtime.

Eight benchmarks (*cholesky*, *fft*, *radix*, *barnes*, *ocean.contiguous*, *ocean.noncontiguous*, *water nsquared*, and *water spatial*)

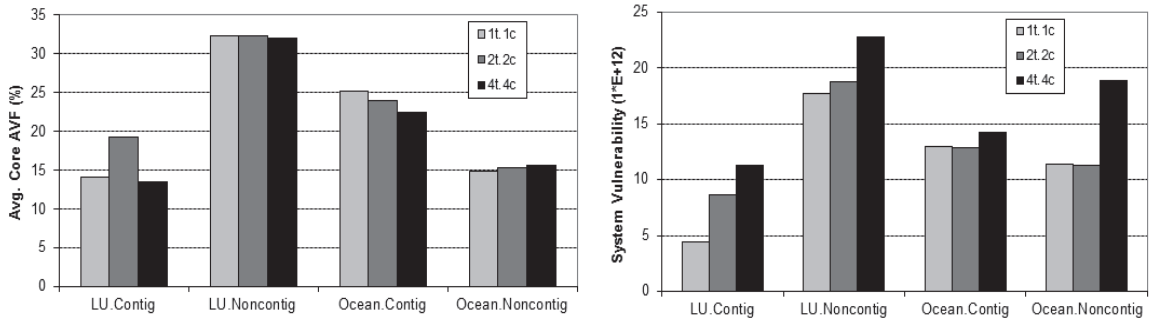

Fig. 14. The average core AVF (left) and cumulative System Vulnerability (right) of the four SPLASH2 programs running with one, two, and four threads on single-, dual-, and quad-core processors, respectively.

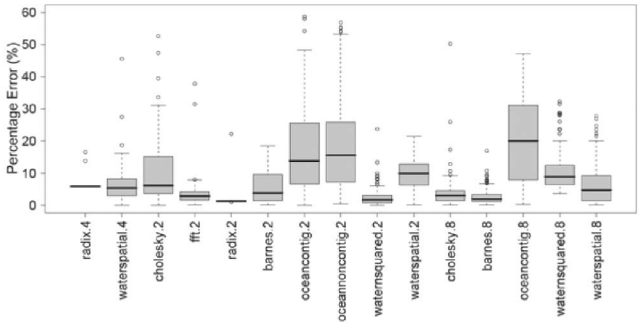

Fig. 15. AVF prediction accuracy of SPLASH2 programs. The number at the end of a program name indicates the number of threads enabled.

in SPLASH2 suite are simulated with two, four, and eight threads enabled on dual-core, quad-core, and eight-core processors, respectively. All cores in our CMP model have private L1 I/D caches and share a unified L2 cache. The data coherencies among different L1 caches are maintained using a MOESI snooping protocol. We dump the measurements every 500K cycles (called a phase) after program initialization, collecting about 100 phases from each program simulation. Since different cores' AVFs are very similar in the same program phase during runtime, we use the measurements from core 0 to represent the system in the following predictor training and testing.

In addition to structure occupancies and cache misses (as used in the above two-level predictor), we also include data cache coherency states as part of the predictor inputs to characterize the interthread data sharing. Specifically, the percentage of data cache blocks that are in each of the five states (i.e., "MOESI") is calculated. We train the predictor using the 4-threaded runs of six benchmarks, and test it with the other two; furthermore, the trained predictor is also validated with the 2-threaded and 8-threaded phases of all eight benchmarks (except a few 8-threaded simulations not runnable in M5). The prediction results are shown in Fig. 15. We can see that most predictions have median percentage errors lower than 10 percent. Therefore, our predictor's effectiveness is demonstrated across different workloads, architectures, and phases. Fig. 16 quantifies the input variable importance of this predictor. As shown, while IQ occupancy is the most influential factor, three coherency states (dcache.MODIFIED, dache.OWNED, and dcache.SHARED) also appear among the top 10 most important factors to the core AVF of multithreaded programs.

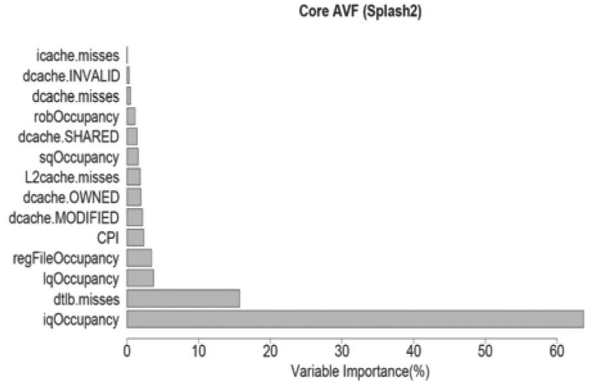

Fig. 16. Input variable importance of the predictor for SPLASH2 multithreaded programs.

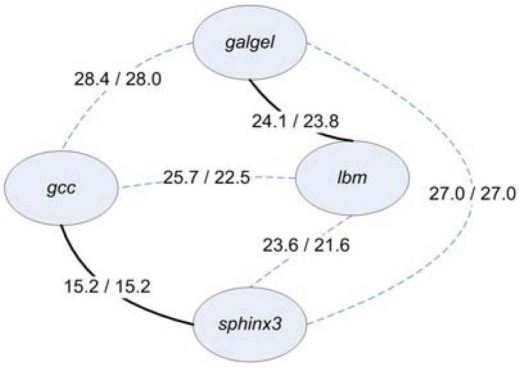

Fig. 17. An example of identifying the optimal thread-to-core scheduling on a two-way two-core CMT processor. The optimal scheduling is shown in solid lines.

7 Case Study: Soft Error Resilient Thread-to-Core Scheduling

This section presents a case study that identifies the optimal solution for soft error resilient thread-to-core scheduling on a chip multithreaded (CMT) processor [36]. A CMT processor provides a combination of SMT and CMP. Programs coscheduled on the same core (via SMT) compete for the shared pipeline resources, while different cores compete for the shared memory hierarchy. Therefore, different thread-to-core schedules may result in completely different behavior in performance, power, and also AVF. Suppose each of the n cores in a CMT processor can support k simultaneous hardware threads, our goal is to assign the $n * k$ programs to different cores of this CMT processor in a way that minimizes the overall AVF.

Fig. 17 is a graphic representation of the problem when $n = 2$ and $k = 2$. In this completely connected graph, vertices represent the programs to be scheduled, and the edge weight between two vertices is the core AVF when the two programs are coscheduled on the same core. Therefore, finding the optimal program-to-core assignment (when $k = 2$) that minimizes the system AVF is identical to solving the *minimum-weight perfect matching problem*[4] in this graph. Jiang et al. [18] proposed a polynomial-time algorithm to solve this problem when k equals 2 (n can be larger than 2). They also proved that this problem became NP-complete when k is greater than 2.

Consequently, as long as all the core AVF values (i.e., the weights of all the edges) are known when different program combinations are coscheduled, we can easily identify the optimal thread-to-core assignment using Jiang et al.'s methodology. The two-level AVF prediction proposed in this paper provides an efficient approach to obtain these AVF values. In Fig. 17, the two values shown on each edge are the measured/predicted core AVF when the corresponding two programs execute on the same core. As shown, both measured and predicted values make the same correct decision: in this example, the optimal schedule is the assignment of (*gcc*, *sphinx3*) on one core and (*galgel*, *lbm*) on the other core. We can see that a suboptimal schedule, such as (*gcc*, *lbm*) with (*galgel*, *sphinx3*), can enlarge the system AVF by over 34 percent.

4. A matching in a graph is a set of edges without common vertices. A perfect matching is a matching that covers all vertices of the graph.

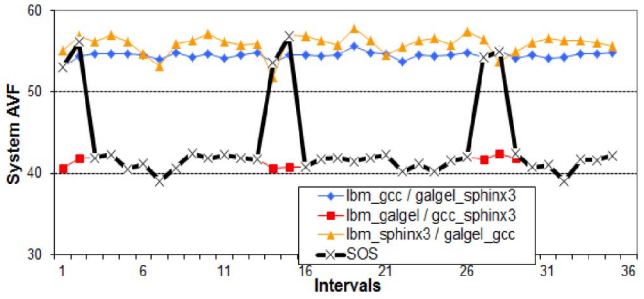

Fig. 18. The runtime AVF behavior of different schedules. The SOS curve indicates the online AVF variation when the "SOS" job scheduler is utilized.

Finally, to apply the above scheduling approach during program runtime, one can utilize Snavely et al.'s [33]. "Sample-Optimize-Symbios" (SOS) job scheduler. SOS first runs each possible schedule for a short phase (i.e., "Sample"), selects the one with the highest goodness level (i.e., "Optimize"), and then runs the selected schedule for a number of intervals (i.e., "Symbios"). SOS periodically repeats the above procedure to choose the optimal schedule adapting to program runtime behavior. Fig. 18 shows the runtime AVF of different schedules constructed from the previous four benchmarks. When SOS is applied, it simulates each of the three schedules for one interval, during which it collects the performance inputs and predicts the AVF. After the sample phase, SOS sticks to the schedule with the lowest AVF for the next 10 intervals, and then reenters the sample phase. From this figure, we can see that SOS is very effective in identifying runtime optimal schedules.

8 Related Work

The concept of AVF was originally proposed in [28], and Biswas et al. [4] extended it to address-based structures. A common approach to calculate the AVF is via ACE analysis [28] which provides a tight lower bound [5] on the soft error reliability of various processor structures. A unified framework named Sim-SODA [15] to study the superscalar processor's AVF was released. Soundararajan et al. [34] described a simple infrastructure to estimate an upper bound of the ROB AVF. Zhang et al. [47] characterized the AVF on SMT architectures by examining the impact from workloads, fetch policies, etc. On the other hand, Statistical Fault Injection (SFI) [43], [24] provides another approach to calculate the AVF. The comparison between these two methods have been made [42], [5]. Recently, Sridharan et al. proposed program vulnerability factor (PVF) [37] and hardware vulnerability factor (HVF) [38] to describe architecture-independent program vulnerability and software-independent hardware vulnerability to soft errors.

Several prior publications studied online AVF prediction. Fu et al. [16] observed a fuzzy correlation between the AVF and a few common performance metrics. Walcott et al. [41] extended the input metrics set and used linear regression to reexamine this correlation. They performed a very accurate prediction, proving the existence of the correlation between the AVF and various performance metrics. Our prior work [11], [12], [22] further generalized this correlation to be across workloads, execution phases, and processor configurations. Alternatively, Li et al. [23] developed an online algorithm to estimate processor structures' vulnerability using a modified error injection and propagation scheme [24]. For the correlation between the AVF and configuration parameters, Cho et al. [9], [10] predicted the dynamics of power, CPI, and the AVF using a combination of wavelets and neural networks. In contrast, the first-level model in this work is universal across different programs, taking performance measurements as part of the inputs. Another work of ours [13] utilized a rule search strategy to generate selective rules on design parameters that optimize cross-program soft error reliability.

Regarding the investigation of the interthread contention on multicore architectures, Lee et al. [21] proposed a composable performance regression (CPR) scheme to predict the performance of a benchmark combination on a multiprocessor from configuration parameters; Chandra et al. [8] predicted the impact of contention on the shared cache using three performance models. Our work differs from theirs in the following ways: 1) We study and predict the soft error vulnerability of processor structures; 2) we also evaluate the SMT structures where the interthread contention shows more significant impact; and 3) the statistical technique used in our work provides useful model interpretations. Moreover, by varying the numbers of cores and application threads, Soundararajan et al. [35] concluded that the configurations optimizing soft error reliability of different multithreaded applications are not straightforward. Zhang and Li [48] proposed useful schemes to manage multicore's soft error reliability.

9 Conclusions

In this paper, we propose a two-level predictive mechanism to accurately predict the soft error vulnerability of multithreaded processors under resource contention and sharing. The first-level model correlates the AVF in a contention-free environment with important performance metrics and the underlying processor configuration; the second-level model takes as inputs the output of the first-level model and a few performance measures under resource contention from multithreaded processors. The proposed scheme is scalable to architectures with more than two threads. Furthermore, a different multithreaded implementation (e.g., in memory allocation and program parallelism) has significant impact on multithreaded workloads' soft error reliability. In general, increasing the number of parallel threads increases the cumulative system vulnerability. Collectively, this work provides useful mechanisms and guidelines to achieve soft error resilient multithreaded processor designs.

Acknowledgments

This work was supported in part by a US National Science Foundation (NSF) grant CCF-1017961, the Louisiana Board of Regents grant NASA/LEQSF (2005-2010)-LaSPACE and NASA grant number NNG05GH22H, NASA(2011)-DART-46, LQESF(2011)-PFUND-238 and the Chevron Innovative Research Support (CIRS) Fund. The authors acknowledge the computing resources provided by the Louisiana Optical Network Initiative (LONI) HPC team. Finally, the authors appreciate invaluable comments from anonymous reviewers.

REFERENCES

[1] H. Asadi, V. Sridharan, M. Tahoori, and D. Kaeli, "Balancing Performance and Reliability in the Memory Hierarchy," *Proc. Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, 2005.

[2] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. Int'l Symp. Microarchitecture (MICRO)*, 1999.

[3] N. Binkert et al., "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52-60, July/Aug 2006.

[4] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," *Proc. 32nd Ann. Int'l Symp. Computer Architecture (ISCA)*, 2005.

[5] A. Biswas, P. Racunas, J. Emer, and S. Mukherjee, "Computing Accurate AVFs Using ACE Analysis on Performance Models: A Rebuttal," *Computer Architecture Letters*, vol. 7, no. 1, pp. 21-24, Jan. 2008.

[6] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees.* Wadsworth Int'l Group, 1984.

[7] P. Bühlmann, "Boosting for High-Dimensional Linear Models," *Ann. of Statistics*, vol. 34, pp. 559-583, 2006.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture," *Proc. 11th Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2005.

[9] C. Cho, W. Zhang, and T. Li, "Informed Microarchitecture Design Space Exploration Using Workload Dynamics," *Proc. IEEE/ACM 40th Ann. Int'l Symp. Microarchitecture (MICRO)*, 2007.

[10] C. Cho, W. Zhang, and T. Li, "Modeling and Analyzing the Effect of Microarchitecture Design Space Parameters on Microprocessor Soft Error Vulnerability," *Proc. Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS)*, 2008.

[11] L. Duan, B. Li, and L. Peng, "Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics," *Proc. 15th Int'l Conf. High-Performance Computer Architecture (HPCA)*, 2009.

[12] L. Duan, L. Peng, and B. Li, "Two-Level Soft Error Vulnerability Prediction on SMT/CMP Architectures," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC)*, 2011.

[13] L. Duan, Y. Zhang, B. Li, and L. Peng, "Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors," *Proc. Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, 2011.

[14] J. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine," *The Ann. of Statistics*, vol. 29, pp. 1189-1232, 2001.

[15] X. Fu, T. Li, and J. Fortes, "Sim-SODA: A Unified Framework for Architectural Level Software Reliability Analysis," *Proc. Workshop Modeling, Benchmarking and Simulation*, 2006.

[16] X. Fu, J. Poe, T. Li, and J. Fortes, "Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior," *Proc. IEEE 14th Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS)*, 2006.

[17] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," *Proc. 30th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2003.

[18] Y. Jiang, X. Shen, C. Jie, and R. Tripathi, "Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[19] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[20] B. Lee and D. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[21] B. Lee, J. Collins, H. Wang, and D. Brooks, "CPR: Composable Performance Regression for Scalable Multiprocessor Models," *Proc. IEEE/ACM 41st Ann. Int'l Symp. Microarchitecture (MICRO)*, 2008.

[22] B. Li, L. Duan, and L. Peng, "Efficient Microarchitectural Vulnerabilities Prediction Using Boosted Regression Trees and Patient Rule Inductions," *IEEE Trans. Computers*, vol. 59, no. 5, pp. 593-607, May 2010.

[23] X. Li, S. Adve, P. Bose, and J. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2008.

[24] X. Li, S. Adve, P. Bose, and J. Rivers, "SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors," *Proc. Int'l Conf. Dependable Systems and Networks (DSN)*, 2005.

[25] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron, "CMP Design Space Exploration Subject to Physical Constraints," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2006.

[26] M. Monchiero et al., "Design Space Exploration for Multicore Architectures: Power/Performance/Thermal View," *Proc. 20th Ann. Int'l Conf. Supercomputing (ICS)*, 2006.

[27] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2002.

[28] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. IEEE/ACM 36th Ann. Int'l Symp. Microarchitecture (MICRO)*, 2003.

[29] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2000.

[30] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," *Proc. 29th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS)*, 1999.

[31] J. Sharkey, D. Ponomarev, and K. Ghose, "M-SIM: A Flexible, Multithreaded Architectural Simulation Environment," Technical Report CS-TR-05-DP01, Dept. of Computer Science, SUNY at Binghamton, Oct. 2005.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behaviors," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[33] A. Snavely and D. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[34] N. Soundararajan, A. Parashar, and A. Sivasubramaniam, "Mechanisms for Bounding Vulnerabilities of Processor Structures," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2007.

[35] N. Soundararajan, A. Sivasubramaniam, and V. Narayanan, "Characterizing the Soft Error Vulnerability of Multicores Running Multithreaded Applications," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems*, 2010.

[36] L. Spracklen and S. Abraham, "Chip Multithreading: Opportunities and Challenges," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2005.

[37] V. Sridharan and D. Kaeli, "Eliminating Microarchitectural Dependency from Architecture Vulnerability," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA)*, 2009.

[38] V. Sridharan and D. Kaeli, "Using Hardware Vulnerability Factors to Enhance AVF Analysis," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2010.

[39] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 1996.

[40] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery Using Simultaneous Multithreading," *Proc. 23rd Int'l Symp. Computer Architecture (ISCA)*, 2002.

[41] K. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic Prediction of Architectural Vulnerability from Microarchitectural State," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2007.

[42] N. Wang, A. Mahesri, and S. Patel, "Examining ACE Analysis Reliability Estimates Using Fault-Injection," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2007.

[43] N. Wang, J. Quek, T. Rafacz, and S. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," *Proc. Int'l Conf. Dependable Systems and Networks (DSN)*, 2004.

[44] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 2004.

[45] S.C. Woo et al., "The SPLASH-2 Programs: Characterizing and Methodological Considerations," *Proc. Int'l Symp. Computer Architecture (ISCA)*, 1995.

[46] T. Zhang and B. Yu, "Boosting with early stopping: Convergence and Consistency," *Ann. of Statistics*, vol. 33, pp. 1538-1579, 2005.
[47] W. Zhang, X. Fu, T. Li, and J. Fortes, "An Analysis of Microarchitecture Vulnerability to Soft Errors on Simultaneous Multithreaded Architectures," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software (ISPASS)*, 2007.
[48] W. Zhang and T. Li, "Managing Multi-Core Soft-Error Reliability through Utility-Driven Cross Domain Optimization," *Proc. Int'l Conf. Application-Specific Systems, Architectures and Processors (ASAP)*, 2008.
[49] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.
[50] J. Ziegler et al., "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)," *IBM J. Research and Development*, vol. 40, no. 1, pp. 3-18, 1996.

Lide Duan received the BS degree in computer science and engineering from Shanghai Jiao Tong University, China, and the PhD degree in electrical and computer engineering from Louisiana State University. He is currently a senior design engineer for AMD's x86 core architectures. His research interests include computer architecture, soft error reliability analysis and prediction, and application-level error propagation prediction. He received a Graduate Fellowship from the Louisiana Optical Network Initiative (LONI) and the Dissertation Year Fellowship from the Louisiana State University Graduate School while working on his PhD degree.

Lu Peng received the bachelor's and master's degrees in computer science and engineering from Shanghai Jiao Tong University, China, and the PhD degree in computer engineering from the University of Florida, Gainesville, in April 2005. He is currently an associate professor with the Division of Electrical and Computer Engineering at Louisiana State University. His research focuses on memory hierarchy systems, reliability, power efficiency, and other issues in CPU design. He also has interest in network processors. He received an ORAU Ralph E. Powe Junior Faculty Enhancement Award in 2007 and a Best Paper Award from the IEEE International Conference on Computer Design in 2001. He is a member of the ACM and the IEEE Computer Society.

Bin Li received the bachelor's degree in biophysics from Fudan University, China, the master's degree in biometrics in August 2002, and the PhD degree in statistics in August 2006 from the Ohio State University. He is an associate professor with the Experimental Statistics Department at Louisiana State University. His research interests include statistical learning and data mining, statistical modeling on massive and complex data, and Bayesian statistics. He received the Ransom Marian Whitney Research Award in 2006 and a Student Paper Competition Award from the American Statistical Association on Bayesian Statistical Science in 2005. He is a member of the Institute of Mathematical Statistics and the American Statistical Association.