# Comprehensive and Efficient Design Parameter Selection for Soft Error Resilient Processors via Universal Rules

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

**Abstract**—Soft errors have been significantly degrading the reliability of current processors whose feature sizes and supply voltages are fast scaling down. In this paper, we propose two effective approaches to characterize processor reliability against soft errors at pre-silicon stage. By utilizing a rule search strategy named Patient Rule Induction Method (PRIM), we are capable of generating a set of selective rules on key design parameters. These rules quantify the design space subregion with the lowest effective soft error rate (SER), thus providing useful guidelines in designing reliable processors. Furthermore, we also propose to use Classification and Regression Trees (CART) to partition the design space into a number of small subregions each being associated with a representative SER value. This gives the processor designer a global view of the SER distribution, enabling a comprehensive analysis over the entire design space. More importantly, both approaches generate "universal" models whose effectiveness is validated with a set of test programs unseen to training. Compared to traditional application-specific design space studies, our models' cross-program capability can save great training effort in the era of multithreading. Finally, a case study on multiprocessors is performed to simultaneously balance multiple design metrics, including reliability, performance, and power.

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

## 1 Introduction

Soft errors have become an important factor in degrading the reliability of current high-performance processors. They occur mainly due to the electronic noises caused by energetic nuclear particles, such as alpha-particles, neutrons, and pions, from the environment [46]. 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 (or 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 [4][20][30][32][33][39][42][44].

Not all soft errors will affect the final output of the program. For instance, a bit flip in the branch predictor may cause a wrong prediction but doesn't have any impact on program correctness. To characterize a processor's reliability against soft errors, one should look at its effective Soft Error Rate (SER), which can be calculated from the following equation [17]:

$$\mathbf{SER = FIT * AVF}$$

FIT (Failures in Time) quantifies the number of *raw* soft errors occurring in a time unit on the processor, depending on circuit level properties such as the area. Architectural Vulnerability Factor (AVF) [31][6] is the probability that a raw soft error finally produces a visible error in the program output. The AVF is an important architecture-level reliability metric reflecting soft error masking effect, depending on both the software program in execution [37] and the underlying hardware configuration [38]. Therefore, AVF has gained strong interests from processor architects in recent years, resulting in a significant number of studies [10][14][18][25][26][35][40][41][44][45]. In this paper, we first investigate the impact of design parameter selection on AVF, then focus on minimizing

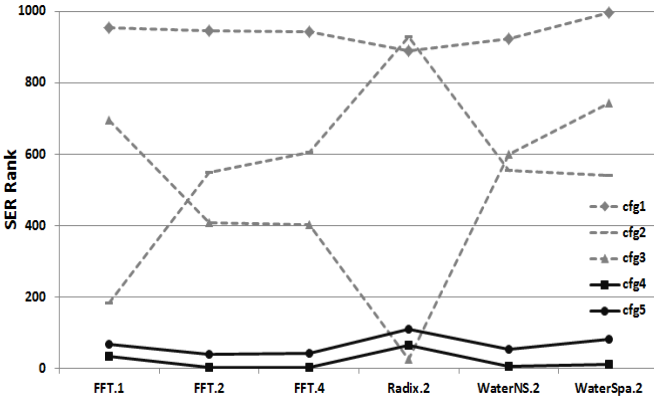



Fig. 1. The SER rank of a certain configuration varies significantly across different multi-threaded workloads. The number at the end of each workload name indicates the number of threads generated when that workload is run.


- *L. Duan is with AMD, Inc. Email: lide.duan@amd.com. This work was performed during his Ph.D. study at LSU.*
- *Y. Zhang is with the Division of Electrical and Computer Engineering, School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA. E-mail: yzhan29@ lsu.edu.*
- *B. Li is with the Department of Experimental Statistics, Louisiana State University, Baton Rouge, LA. E-mail: bli@lsu.edu.*
- *L. Peng is with the Division of Electrical and Computer Engineering, School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA. E-mail: lpeng@ lsu.edu.*


*Manuscript received (insert date of submission if desired).*

xxxx-xxxx/0x/$xx.00 © 200x IEEE

the SER to achieve soft error resilient designs for the entire processor.

**Motivation**. The SER of a processor may drastically change if it runs different threads or uses different configuration parameters. In particular, the AVF (so is the SER) of a multiprocessor is affected by the data contention and sharing in the low level cache [45]. Fig. 1 shows how the system SER would vary when different workloads (multi-threaded benchmarks from SPLASH2 [43]) execute on a number of multiprocessor configurations. These 5 configurations, cfg1 to cfg5, are representative examples chosen from 1,000 design points randomly sampled from a large multiprocessor design space (illustrated in Section 5). For each workload, the 1,000 sampled configurations are ranked in terms of their SER values, with the lowest value as rank 1 and the largest value as rank 1,000. Therefore, a lower rank indicates a more reliable design. We can see that, for the same configuration, its SER rank could be significantly different across different programs (or the same program with different numbers of threads, i.e. FFT). For cfg1-3, there exists at least one workload suffering from a high SER rank. A reliable processor design would favor configurations (such as cfg4 and cfg5 in this example) whose SER ranks are *consistently* low in different applications. Hence, this work aims at effectively and efficiently identifying such designs from a statistically large design space.

**Our proposal: efficient and comprehensive design parameter selection**. In this paper, we propose two effective approaches to characterize the SER over the design space. First, we propose to use *Patient Rule Induction Method* (PRIM) [16] to efficiently generate a set of selective rules on key design parameters. Applying these rules on the design space effectively identifies the design space subregion that has the lowest response values (i.e. "valley seeking"). Therefore, the configurations selected by these rules demonstrate the lowest SER, thus being inherently reliable to soft errors. Second, we utilize *Classification and Regression Trees* (CART) [7] to partition the design space into a set of subregions, each being fitted with a representative SER value. By doing so, we can obtain a global and comprehensive view of the SER distribution over the entire design space. Consequently, both techniques can provide computer architects with useful guidelines to design soft error resilient processors at pre-silicon stage.

More importantly, the parameter selection results generated by our proposed methods are "universal" across different programs. In other words, only a single model is needed for all the programs in consideration; the model effectiveness is also validated on other programs not used in training. This is in contrast to traditional design space explorations which build a separate model for each program: in those studies, the training cost would become intractable with the quickly increasing number of workloads; in particular, the number of multi-programmed workloads increases super-linearly with the number of threads. As a result, our models' cross-program capability is very useful in such situations. Note, however, that by "universal" we don't refer to rules working for all programs; instead, we manage to identify the rules with workload-independent effectiveness for SPEC CPU and SPLASH2 benchmark suites used in this work. Regardless, these commonly used benchmarks well represent real-world applications.

Finally, the proposed approaches are inherently generic, so they can be used to optimize various processor design metrics. A case study performed in Section 5 utilizes the proposed universal design parameter selection to achieve a multi-objective optimization of reliability, performance, and power for multiprocessors.

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

- **Analysis of design parameter selection on AVF.** By investigating the rules generated for AVF, we quantitatively demonstrate that: (1) minimizing the AVF for different processor structures has different impacts on the performance; (2) reducing the AVF of a single structure may increase the AVF of others.
- **Universal rules generation and validation using PRIM.** We propose to use an advanced rule search strategy (PRIM) to efficiently extract the design space subregion with universally optimal soft error reliability. The effectiveness of the universal rule set is further validated using a well-developed statistical method ("Bootstrap" [15]) on a set of test programs that are unseen during model training.
- **Comprehensive design space characterization using CART**. We propose to employ a regression tree fitting process (CART) to obtain a global picture of the SER distribution on the entire design space. CART provides useful guidelines in designing reliable processors, especially in the cases when the optimal designs cannot be achieved.
- **Balancing reliability, performance, and power for multiprocessors.** We perform a study on multiprocessors using the proposed universal modeling scheme to simultaneously balance multiple design metrics. We quantitatively identify suitable trade-offs among reliability, performance, and power when running multi-threaded workloads.

## 2 METHODOLOGY

### 2.1 Patient Rule Induction Method (PRIM)

Patient Rule Induction Method (PRIM) [16] identifies a subregion in the input space (composed of configuration parameters in this paper) that gives relatively low values for the output response (e.g. the SER). The identified input space subregion (or "box") is described as a set of simple "selective rules" in the form of $B = \bigcap_{j=1}^{p} (x_j \in s_j)$. $x_j$ represents the j$^{th}$ input variable, and $S_j$ is a subset of all possible values for the j$^{th}$ variable. Hence, the identified region $B$ is the intersection of $p$ subsets, each being from one of the $p$ input variables.

The box construction of PRIM consists of two phases: patient successive top-down peeling and bottom-up recursive pasting. Fig. 2 visualizes this procedure. The top-down peeling starts from the entire input space. At each

iteration, we have the following operations: a small subbox $b$ within the current box $B$ is removed; we calculate the output mean for the elements remaining in $B - b = \{x \mid x \in B \ \& \ x \notin b\}$, trying this operation in each dimension (i.e. try removing a different subbox from each input variable); finally we choose the one that yields the smallest output mean value for the next box $B$-$b$. The above procedure is iteratively applied until the proportion of the data points remaining in the current box (called the support) is below a chosen threshold $\beta$ Note that for a categorical variable, an eligible subbox $b$ contains only one element of the possible values of the variable in the current box $B$. For example, suppose three possible values for LSQ size remain in the current box, i.e. $s_3$={16,40,64}, there are three eligible subboxes: {$x_3$=16}, {$x_3$=40} and {$x_3$=64} in this dimension. They are also the possible candidates to be removed in the next iteration.

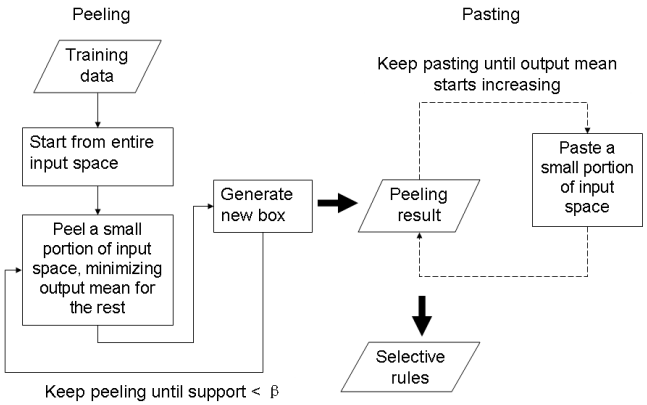



Fig. 2. PRIM training procedure, including peeling and pasting.

**CART(D, cp)**
{
1. Start from the entire input space, and initialize the current depth $d$ to 1.

2. For each input variable $j$ and each split point $s$ (i.e. a possible value) of $j$:
2.1. Partition the current space into two parts (X is any data point in current space):

$r_1(j,s) = \{X \mid X_j <= s\}$ and $r_2(j,s) = \{X \mid X_j > s\}$

2.2. Choose the $j$ and $s$ that solve:

$$\min_{j,s} [\sum_{Xi \in r1} (y_i - c_1)^2 + \sum_{Xi \in r2} (y_i - c_2)^2]$$

where $c_1 = ave\ (y_i \mid X_i \in r_1(j,s))$ and

$c_2 = ave\ (y_i \mid X_i \in r_2(j,s))$

3. Partition the current space into $R_1$ and $R_2$ with the chosen $j$ and $s$ from Step 2.
3.1. Model each subspace with the average response value of the points in it.
3.2. Increment $d$ by 1.

4. For each of the two subspaces $R_1$ and $R_2$:
4.1. Repeat Step 2.
4.2. If either ($d$ > **D**) or (the relative decrease of the Mean Square Error < **cp**):
The partition of current subspace is stopped.
Otherwise:
Repeat Step 3 and Step 4.
}

Fig. 3. The CART algorithm used in this paper.

The pasting algorithm is simply the inverse of the peeling procedure. The reason for pasting is that at each peeling iteration we only look one step ahead. The box boundary is thereby determined without knowledge of later iterations. Consequently, we may peel too much from the input space, and the final box can sometimes be improved by readjusting its boundaries. From the peeling result, the current box $B$ is iteratively enlarged by pasting onto it a small subbox that minimizes the output mean in the new larger box. The subbox being pasted is chosen in the same manner as in peeling. The bottom-up pasting is iteratively applied, successively enlarging the current box until the addition of the next subbox causes the output mean start increasing.

Regarding the complexity of PRIM, the first peel requires at most $n * (\sum_{j=1}^{p} C_j)$ operations, where $n$ is the number of observations, $p$ is the number of input variables, and $C_j$ is the number of possible values for the $j^{th}$ variable. The number of operations for each peel will decrease since fewer and fewer samples will be left during peeling. On the other hand, PRIM performs approximately $-\log(n)/\log(1-\alpha)$ peeling steps, where α is the percentage of points that is peeled off at each iteration.

An advantage of PRIM over greedy methods is its patience. For example, a binary tree partitions the data quickly because of its binary splits, while in PRIM each time only a small proportion (α) of data is peeled off. Hence, the solution of PRIM is usually more stable: if the data are slightly changed, a tree structure may change drastically but the PRIM solution is less affected. Moreover, if the optimal subspace is not connected, PRIM can generate a sequence of hyper-boxes instead of just one. Namely, after getting the first hyper-box, the PRIM procedure can be repeated on the remaining dataset. As a result, a disconnected subspace can also be covered. However, in practice the leading one hyper-box usually covers most of the points with the smallest response values. Therefore, we only apply PRIM once to identify the desired subregion in the following sections. Finally, the threshold $\beta$ indicates the percentage of data points remaining in the final hyper-box. As can be seen in the following sections, if 2% design points are extracted in the final results, they are usually within the top 5% - 15% optima of the entire design space for the metric in analysis.

## 2.2 Classification and Regression Trees (CART)

While PRIM is an efficient mechanism to effectively identify the "valley" of the input space, Classification and Regression Trees (CART) provides a comprehensive view

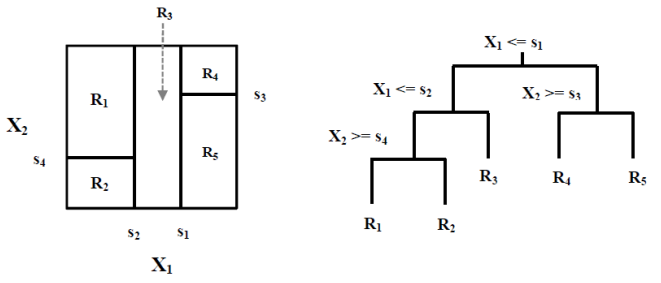



Fig. 4. Illustration of CART in a two dimensional input space.

of the entire space. CART partitions the input space into a set of rectangles, fitting a simple model (usually a constant) in each of them. By applying CART, we characterize different portions of the input space with simple but representative response values, thus enabling direct comparisons among different subregions. Consequently, we can obtain a global and comprehensive understanding of the response distribution over the entire input space.

CART is a recursive binary partitioning algorithm that provides an alternative to traditional parametric methods for regression problems. We use the **rpart** package from the R language [1] to train the CART model; the detailed algorithm is illustrated in Fig. 3. Initially, it splits the space into two regions $R_1$ and $R_2$ via choosing the optimal variable and split-point that can achieve the best fit (i.e. minimizing the Mean Square Error as shown in Step 2.2) in the current iteration. Then each of these two regions is further split into two more regions, and this process is continued until some stopping criteria are reached. In this algorithm, the split of a subspace is stopped when the current depth of the tree exceeds a threshold **D** or the relative decrease of the Mean Square Error due to the new split is lower than a complexity parameter **cp**. This recursive approach essentially builds a binary tree whose leaves correspond to the resulting input space subregions. Each of these subregions is modeled with the average of the response values of the data points in this subregion. By looking at these values, we can have a comprehensive view of the distribution of the response in different input space subregions.

As an illustrative example, Fig. 4 shows the result of the CART process in a two-dimensional ($X_1$ and $X_2$) input space. In this example, we first split the space at $X_1 = s_1$; then the region $X_1 <= s_1$ is split at $X_1 = s_2$, and the region $X_1 > s_1$ is split at $X_2 = s_3$; finally the region $X_1 <= s_2$ is split at $X_2 = s_4$. Therefore, the entire space is partitioned into five regions $R_1$ to $R_5$ as shown in the left panel of this figure. The right panel is the binary tree representation of the same model. The full dataset sits at the top of the tree; observations satisfying the condition at each junction are assigned to the left branch; the leaves of the tree correspond to the final resulting regions. $R_1$ and $R_2$ are not further partitioned because they have reached the threshold depth (assuming that **D** = 3), and the splits of the other regions are stopped early because they are not worthy compared to the introduced complexity.

A key advantage of CART is its interpretability. The partition of the feature space can be fully described by a single binary tree, which provides an intuitive visualization to high-dimensional datasets. Second, CART is inherently nonparametric and can easily handle input variables with mixed types. Therefore, it doesn't need to make any assumptions on the value distributions of the inputs, avoiding the effort in examining and preprocessing training data. Furthermore, CART is adept at capturing non-additive behavior such as the complex interaction between different input variables; it is also insensitive to outliers. On the other hand, a tree model is usually not very stable: small changes in the training data could result in a very different series of splits. Nevertheless, the PRIM model (Section 2.1) is complementary to CART in terms of stability. In the later sections, we will apply both schemes to conduct a comprehensive yet stable investigation on the problem studied in this paper.

## 3 Analysis of Design Parameter Selection on AVF

This section investigates the impact of design parameter selection on AVF, rather than SER, due to the following reasons. First, this section focuses on analyzing an individual processor structure (e.g. ROB), whose SER is directly related to its number of entries (i.e. its area). Therefore, for individual structures, the rules minimizing SER always tend to choose configurations with fewer entries, making it less interesting to be examined. Second and more importantly, AVF itself, as the most important portion of SER, depends on both the software and hardware, drawing strong interests from computer designers. Many prior works [8][9][18][19][40][10][13] had focused on merely investigating AVF itself. As a result, it's interesting and worthy to discuss the correlation of AVF and configuration as part of this paper, particularly in the context of generic rule summary over a design space.

In this section, we directly apply the PRIM method to build a separate AVF model for each benchmark. This application-specific approach generates the parameter selection rules that work best for each individual workload[1], but requires workload dependent model training whose overhead is significant when there are a large number of workloads. Moreover, the rules derived from a certain workload may not work well for another. Nonetheless, analysis of these rules still provides valuable insights on reliable processor design. In Section 4, we will extend to SER, generating universal rules effective across workloads.

### 3.1 Experimental Setup

We implement the AVF calculations [31][18] in an extended version of SimpleScalar3.0 [2] to simulate a detailed out-of-order multistage superscalar processor. The implemented AVF measurement is based on *Architecturally Correct Execution (ACE)* analysis [31], which identifies the hardware bits (called ACE bits) that are required for correct program execution via a post-commit analysis window. Using this approach, our simulation framework

[1] In this paper, we use "application", "workload" and "program" interchangeably.

Table 1. SPEC benchmarks used in this paper. The partitioning of training and test sets will be used in Section 4.

| Train (24) | Test (12) |
|---|---|
| *applu, apsi, art, bzip2, crafty, equake, fma3d, mcf, mesa, perlbmk, vortex, astar, 06bzip2, gobmk, hmmer, libquantum, parser, lbm, lucas, 06mcf, milc, namd, sjeng, sphinx3* | *gcc, twolf, ammp, eon, gap, gzip, vpr, facerec, galgel, mgrid, swim, wupwise* |

Table 2. The uniprocessor design space composed of 8 parameters. Branch predictors are renamed to *a* to *h* for easy reference.

| | Parameter | Selected Values | # Options |
|---|---|---|---|
| $\mathbf{P_1}$ | Processor width | 2, 4, 8 | 6 |
| | Fetch queue size | 2, 4, 8 (vary with processor width) | |
| | # of Integer ALUs / # of FP ALUs | 1/1, 2/1-associated with processor width 2<br>2/1, 2/2-associated with processor width 4<br>2/2, 4/4-associated with processor width 8 | |
| $\mathbf{P_2}$ | ROB size | 64, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160 | 11 |
| $\mathbf{P_3}$ | LSQ size | 16, 24, 32, 40, 48, 56, 64 | 7 |
| $\mathbf{P_4}$ | L1 I/D cache size (L1CS) | 16, 32, 64, 128 KB (32B block, 2-way) | 4 |
| | L1 cache latency | 1, 2, 3, 4 cycles (vary with L1 cache size) | |
| $\mathbf{P_5}$ | L2 cache size (L2CS) | 512, 1024, 2048, 4096 KB (64B block) | 4 |
| | L2 cache latency | 8, 12, 16, 20 cycles (vary with L2 cache size) | |
| $\mathbf{P_6}$ | L2 cache associativity (L2CA) | 4, 8 | 2 |
| $\mathbf{P_7}$ | Branch predictor (BP) | 2lev/1/4096 (*a*), 2lev/1/8192 (*b*), 2lev/2/4096 (*c*), 2lev/2/8192 (*d*), 2lev/4/4096 (*e*), 2lev/4/8192 (*f*), bimod/4096 (*g*), bimod/8192 (*h*) | 8 |
| $\mathbf{P_8}$ | BTB | 1024/4, 1024/8, 2048/2, 2048/4 | 4 |

Table 3. Workload-dependent rules for optimizing individual structure's AVF. "Width/ALUs" is the combination of processor width, # of integer ALUs, and # of FP ALUs; "&" refers to "AND"; "||" refers to "OR".

| Benchmark | ROB AVF | LSQ AVF | Functional Unit AVF | Register File AVF |
|---|---|---|---|---|
| *mcf* | (ROB > 130) & (LSQ < 64) & (BP!= *g*) & (BP!=*h*) | (LSQ > 48) & (L2CS < 4096) & (BP!=*g*) & (BP!=*h*) | Width/ALUs=8/4/4 | (Width/ALUs!=2/1/1) & (LSQ > 24) & (BP=*g*\|\|*h*) |
| *applu* | (ROB > 90) & (LSQ < 32) | (ROB < 110) & (LSQ > 40) | (Width/ALUs=4/2/2\|\|8/2/2\|\|8/4/4) & (LSQ < 32) | (Width/ALUs=4/2/2\|\|8/2/2\|\|8/4/4) & (ROB > 80) & (LSQ > 24) & (L1CS > 16) |
| *fma3d* | (Width/ALUs=2/2/1) & (ROB > 90) | (Width/ALUs=2/2/1\|\|4/2/1\|\|8/4/4) & (LSQ > 40) & (L1CS < 128) | (Width/ALUs=2/1/1\|\|8/4/4) & (L1CS < 64) | Width/ALUs=8/4/4 |
| *gobmk* | (ROB>110) & (L1CS<128) & (L2CS > 512) & (BP!=*g*) & (BP!=*h*) | (LSQ > 40) & (L1CS < 64) & (BP!=*g*) & (BP!=*h*) | (Width/ALUs=2/2/1\|\|8/4/4) & (L1CS < 64) | (Width/ALUs!=2/1/1) & (Width/ALUs!=2/2/1) & (L1CS > 32) & (L2CS > 512) & (BP!=*a*) & (BP!=*c*) & (BP!=*e*) |
| *milc* | (ROB>90) & (LSQ<32) | (ROB < 110) & (LSQ > 40) | Width/ALUs=8/4/4 | (Width/ALUs!=2/1/1) & (ROB > 110) & (LSQ > 40) |

measures the AVF of major microarchitectural components including Reorder Buffer (ROB), Load Store Queue (LSQ), Functional Units (FU), and Register File (RF).

We use a mixed set of benchmarks from SPEC CPU 2000 and 2006 suits as listed in Table 1. When generating the universal rules in Section 4, 24 of these benchmarks form the training set while the rest 12 are used for test. In this section, we only select some of the representative ones to report their application-specific rules. In order to have a comprehensive evaluation, the entire SPEC CPU 2000 suite (except *sixtrack*) is included. For the other SPEC CPU benchmarks not used in this work, we were not able to compile them into Alpha binaries runnable in the SimpleScalar simulator. Each benchmark is simulated for 100 million instructions in details after being fast forwarded to a representative phase derived from SimPoint Toolkit [34]. The AVF measurements along with performance data are outputted at the end of each simulation.

We construct a uniprocessor design space composed of 8 configuration parameters ($\mathbf{P_1}$ to $\mathbf{P_8}$ in Table 2). For register files and caches, we assume no ECC protection since adding ECC protection effectively reduces AVF to 0. The design space size is 473,088, from which we randomly and uniformly sample 2,000 configurations. This amount of sample size turns out to be sufficient for training accurate models and also cost-efficient for simulation.

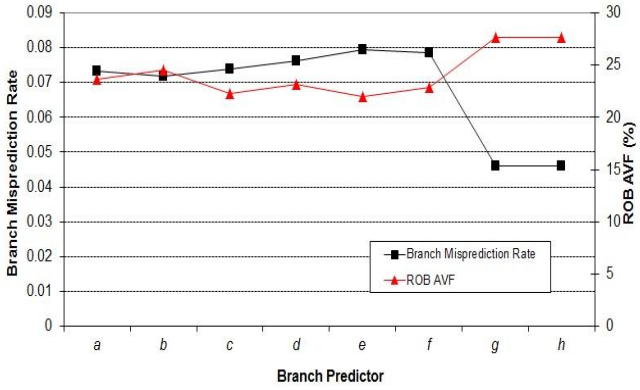



Fig. 5. ROB AVF of *mcf* (SPEC 2000) varies with different branch predictors.

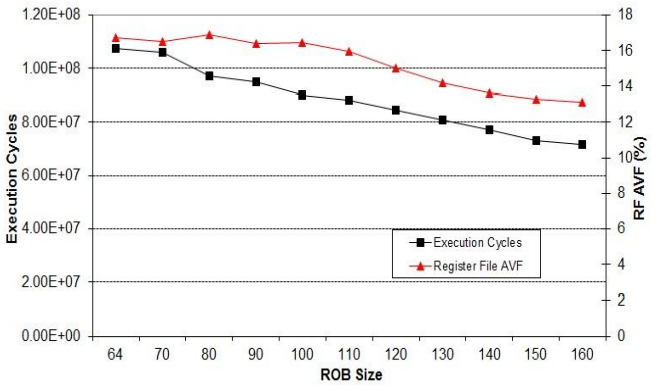



Fig. 6. Register File AVF of *milc* (SPEC 2006) varies with different ROB sizes.

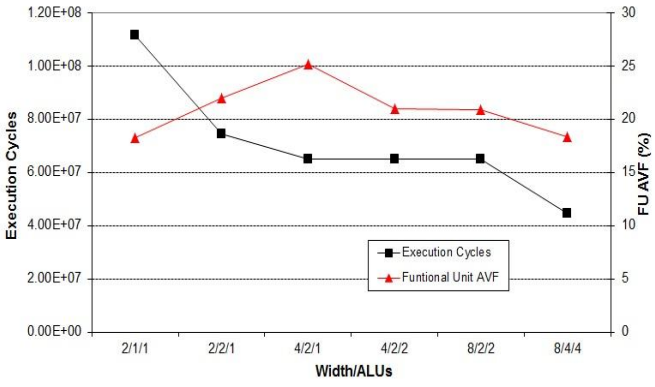



Fig. 7. Functional Units AVF of *fma3d* (SPEC 2000) varies with different combinations of processor width and # of ALUs.

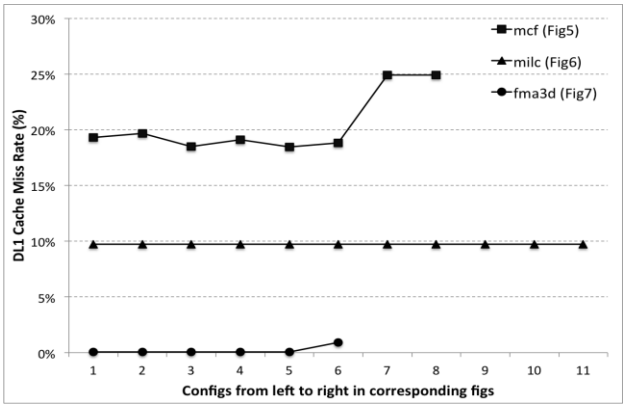



Fig. 8. The level 1 data cache miss rates for the workloads running on the configurations shown in Fig. 5 - 7.

### 3.2 Results and Discussion

Application-specific rules achieve the AVF optimization for individual workloads, enabling insightful analysis of design parameter selection that optimizes processor soft error reliability. Table 3 lists the rule sets for a number of benchmarks. The results for other benchmarks used in this work show similar behavior, thus being omitted in the paper. From these results, we observe that minimizing the AVF of different processor structures have different impacts on the performance.

First, some rules tend to degrade the performance considerably when minimizing the AVF. Specifically, the rules for *mcf* to optimize the ROB AVF introduce some restriction on the branch predictor selection. Fig. 5 shows the variation of branch misprediction rate and the ROB AVF when running *mcf* on configurations with different branch predictors *a* to *h*. The other parameters are identical for these configurations. Clearly, the ROB AVF varies contrarily with respect to the variation of branch misprediction rate. If we exclude the two (*g* and *h*) with the lowest misprediction rates as the rules suggest, one can expect a significant performance loss. Similar observation can be made from the rules for minimizing the LSQ AVF. For example, in *gobmk*, the restriction on L1 cache size to have smaller values (< 64KB) will result in a longer execution time. The reason that queue-based structures like ROB and LSQ usually have lower AVF with worse performance is intuitive: the AVF of these structures is directly related to their occupancy rates, so a low AVF value indicates a low queue occupancy which in turn implies degraded performance.

In contrast, optimizing the Register File AVF simultaneously improves the performance. Therefore, the rules generated for minimizing the Register File AVF clearly select the designs achieving better performance via employing larger queues or caches, wider CPUs, or more accurate branch predictors. For example, the rules for *milc* favor a ROB size larger than 110. Fig. 6 demonstrates how the Register File AVF and the performance would vary when different ROB sizes are used. It is easy to see that a larger ROB size results in better performance and a more reliable Register File as well. This is because more pipeline resources (e.g. a larger ROB) usually improve performance via making instructions pass through the pipeline more quickly. This effectively shortens the write-read interval for a certain register, thus reducing the ACE cycles and lowering the AVF for the Register File.

The correlation between the AVF and performance is fuzzier for the Functional Units (FU). A very wide CPU usually incurs a low FU AVF because it has sufficient ALUs to execute the instructions more quickly (thus fewer ACE cycles); a very narrow CPU significantly degrades performance, but its FU AVF may still be low due to the inefficient usage of ALUs. This can be verified from the rules for optimizing the FU AVF of many benchmarks.

For instance, Fig. 7 illustrates that for *fma3d* the execution cycles consistently decrease with the increasing CPU width and number of ALUs, but the FU AVF increases initially and decreases later. Consequently, the rules for *fma3d* shown in Table 3 choose the two extreme settings (either widest or narrowest) when optimizing the FU AVF. If performance is taken into consideration, one should choose a wide CPU.

On the other hand, the AVFs are also related to the cache miss rates. Fig. 8 lists the level 1 data cache miss rates for all the configurations presented in Fig. 5 to 7. As can be seen, *mcf* shows a variation of data cache miss rate that is very close to its ROB AVF variation shown in Fig. 5. A higher cache miss rate indicates more severe congestion in the pipeline, so there are more microarchitectural bits that are vulnerable to soft errors. In contrast, *milc* has a constant miss rate among the configurations showing decreasing RF AVF. This workload's memory behavior is insensitive to the ROB size changes. Similarly, *fma3d* demonstrates near-zero miss rates for the different widths being examined. In spite of a slight increase in the miss rate for the widest configuration, the FU AVF still decreases because of the reduction in the vulnerable cycles incurred by more ALUs and much wider pipeline.

To summarize, minimizing the individual processor structure's AVF may degrade the performance (e.g. in ROB and LSQ), or improve the performance (e.g. in Register File), or result in either way (e.g. in Functional Unit). Furthermore, in Table 3 we also observe many contradictions between the rule sets optimizing different structures' AVF for the same workload. For example, *applu* requires a small LSQ size (< 32) but a large one (> 40) in optimizing the AVF for ROB and LSQ, respectively. Fig. 9 illustrates this contradiction. We can see that the LSQ AVF decreases with the increase of LSQ size, but meanwhile the ROB AVF quickly boosts to a very high value. Therefore, if the rules for optimizing the LSQ AVF are adopted in the processor design, ROB will become extremely vulnerable. Consequently, reducing the AVF of one processor structure may increase the AVF of others. When designing a reliable processor, one can easily make a mistake by transferring the soft error vulnerability to other parts of the processor, instead of really reducing it. Therefore, the overall AVF (and SER) of the entire processor should be considered to achieve a holistically reliable solution.

## 4 Universal Rules Guided Design Parameter Selection

### 4.1 Efficient Generation Using PRIM

We extend the modeling in the previous section in two aspects here. First, the effective SER is used as the model response in this section, as SER is a more direct measurement of soft error vulnerability. Since we only care about the relative SER values, we use (AVF * area) to estimate the SER of a processor, assuming a constant FIT per area unit. For the area of each microarchitectural structure, we use its number of bits that can be available in a recent AVF simulation framework [18][3]. The processor overall AVF is the ratio between the number of the entire processor's ACE bits and the whole processor size. It can be calculated as the summation of different structures' AVFs weighted by the corresponding structures' sizes. The area of the processor is the summation of all the structures' areas.

Second, we generate rules effective across programs in this section. The rule sets generated in the previous section work well for their corresponding applications, but may differ from each other. In practice, these results may not be very useful if the rule sets for different applications significantly disagree. Processor architects are more interested in generic guidelines that can achieve universal reliability across different applications. Fortunately, from Fig. 1 we can see that there exists some consensus among different programs about what configurations are reliable. Therefore, it is possible to extract a "universal" rule set that works well across different programs.

Consequently, we propose to select parameters that minimize the average SER rank across all the training benchmarks. First, we need to rank the configurations in each benchmark in terms of the SER measurements. This is because the absolute SER value ranges significantly differ in different benchmarks. For the 2K configurations used in this work, the one with the lowest SER value is ranked 1 while the one with the highest SER is ranked 2000. Hence, a certain configuration would have 24 different ranks for the 24 training benchmarks in Table 1, respectively. Second, we use the average of these ranks as

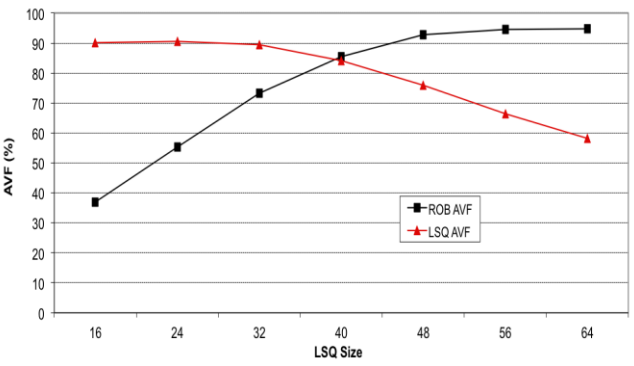



Fig. 9. ROB and LSQ AVFs of *applu* (SPEC 2000) vary with different LSQ sizes.

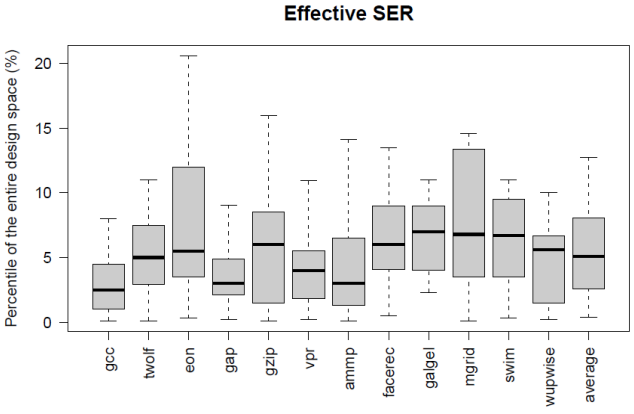



Fig. 10. Validation of Rule Set I on the test benchmarks.

the output response to train a PRIM model. The generated rule set contains the design points that are universally reliable for all training benchmarks. In practice, we found that minimizing the average of the cube of the ranks (i.e. mean(rank$^3$)) is very effective in identifying the universal rules, as this tends to balance the ranks across different benchmarks. For example, suppose we have 5 benchmarks and need to compare two cases with ranks (2, 2, 2, 2, 2) and (1, 1, 1, 1, 6), respectively. If the average of ranks is used, the two cases are considered as the same; but if the cube of ranks is used, the first case is better than the second one.

By using the above approach, we are able to generate a universal rule set that optimizes the overall SER of a uniprocessor. It is shown as Rule Set I as follows:

**Rule Set I (Optimizing Uniprocessor SER):**

**(Width/ALUs != 8/2/2) & (Width/ALUs != 8/4/4)**
**& (ROB < 150) & (LSQ < 24)**
**& (L1CS < 64kB) & (BP = *a* || *c* || *e* || *f*)**

Rule Set I provides useful guidelines in designing a holistically and universally reliable processor. It has upper bounds on ROB and LSQ sizes because smaller structures have less exposure to raw soft errors; other factors somehow degrade the performance, validating our previous observation that a contradiction exists between optimizing performance and some structures' AVF.

Compared to a number of prior design space studies [21][22][8][9], our PRIM-based approach is more efficient. First, traditional design space studies usually need to train a separate model for each benchmark. The training cost would become intractable when the number of workloads enlarges quickly. In contrast, our approach only builds a single model that is effective across different workloads. Second, the prior studies propose predictive models while PRIM quantifies the desired parameter value ranges. It's also possible for predictive models to derive these ranges (e.g. via exhaustive prediction like Pareto Analysis [23]), but PRIM is a direct and more efficient one-step search that avoids exhaustive prediction on the entire design space.

### 4.2 Universal Rules Validation

In this subsection, we apply Rule Set I on the 12 test benchmarks (see Table 1) to validate its effectiveness in identifying reliable design configurations. For each of these benchmarks being tested, the validation consists of the following steps:

(1) Simulate the test benchmark on the 2,000 configurations randomly and uniformly sampled from the entire design space. These simulations are used to approximate the whole design space whose exhaustive simulation is intractable.
(2) Identify what configurations among the 2,000 ones are selected by Rule Set I. When Rule Set I was generated above, $\beta$ was set to 2%. Therefore, there are approximately 40 points selected by this rule set.
(3) Identify in which part of the design space the points selected by Rule Set I are actually located.

The main difficulty of the above approach is in (3), because for each benchmark in the test set we intend to know where those configurations selected by the rule set are located in the entire design space (not just the sampled 2K configurations!). In other words, we intend to know what percentile (say $p$) of the design space that the values of these selected points are below. The $p$-percentile for the whole space indicates the value that is greater than $p$% of all the data points but less than the rest.

In order to make inference based on the entire input space, we use the bootstrapping method [15]. Specifically, we first sample (with replacement) 1,000 bootstrap samples from the 2,000 configurations. Note that each bootstrap sample also contains 2,000 design points though some of them may be repetitive due to the replacement in sampling. We then compute a confidence interval estimate of the entire design space's $p$-percentile based on these bootstrap samples. Specifically, for each bootstrap sample, we calculate its $p$-percentile. This gives us a total of 1,000 values for $p$-percentiles (one for each bootstrap sample). Among these 1,000 values, we further calculate their 5-percentile (say W). By doing so, we have 95% confidence that the $p$-percentile of the entire design space is larger than or equal to W. Finally, we adjust the $p$ value

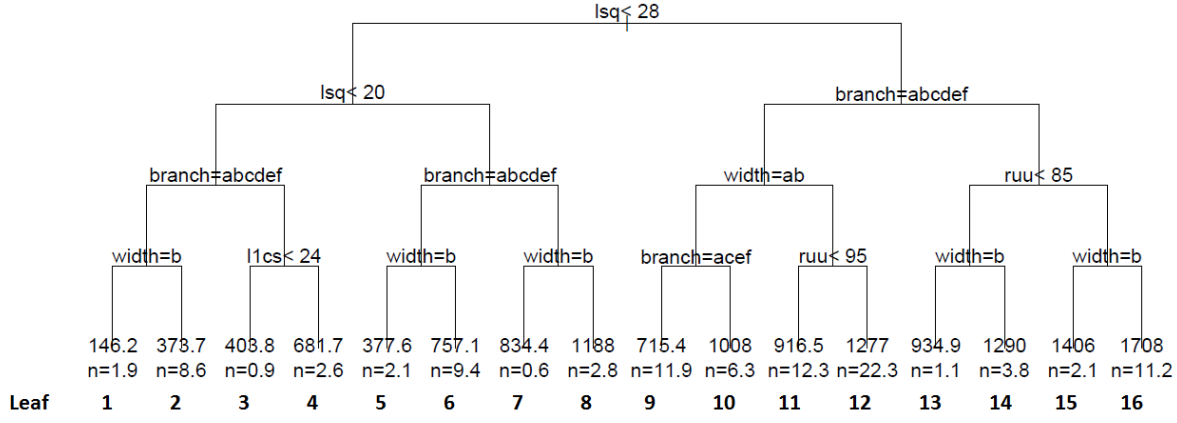



Fig. 11. The regression tree generated by CART on the training benchmarks.

(by repeating the above steps) to have the derived W slightly larger than the largest value of the selected points. Therefore, the final determined $p$ value is the percentile that all the selected points are below. Note that this approach is conservative since the exact $p$-percentile of the entire design space could be much larger than W.

For each benchmark being tested, we first calculate the minimum, lower quartile, median, upper quartile, and maximum of the design points selected by the rule set; after that, for each of these five values, we calculate the corresponding percentile of the entire design space that it is below (using the bootstrapping method). We use boxplot to demonstrate the validation results in Fig. 10. In this boxplot, the upper and lower boundaries of the central gray box correspond to the upper and lower quartiles; the highlighted horizontal line within the box is at the median; the vertical dotted line drawn from the box boundaries extend to the minimum and maximum. The vertical axis shows the percentile of the entire design space that the selected points are below. For example, for *gcc*, the maximum of the points selected by Rule Set I corresponds to a value of 8% in the vertical axis, meaning that in this benchmark all selected points are within the top 8% optima of the entire design space. We can see that Rule Set I is very effective in finding the optima for all test benchmarks. On average, the design points quantified by Rule Set I achieve the top 12% optima of the entire design space. Again, as clarified in Section 1, we don't intend to locate the design space subregion that is reliable independent of all programs, but demonstrate that the rules generated using our proposed methodology work well across SPEC CPU benchmarks. These rules would be effective for other programs outside SPEC provided that SPEC CPU suites well represent real-world applications.

### 4.3 Comprehensive Analysis Using CART

PRIM efficiently identifies the "valley" of the design space that has the lowest SER, but it doesn't reveal any information on the SER distribution outside the valley. Sometimes, the optimal designs may not be achieved due to budget issues; in such cases, processor architects are more interested in obtaining a global picture of the response distribution over the entire design space. CART can provide such information via fitting a regression tree. The detailed description of CART can be found in Section 2.2. In this subsection, we apply the CART process on the training data used in previous subsections to generate a comprehensive tree-structured model. The inputs to this model are the configuration parameters, and the output is the average SER rank over training benchmarks.

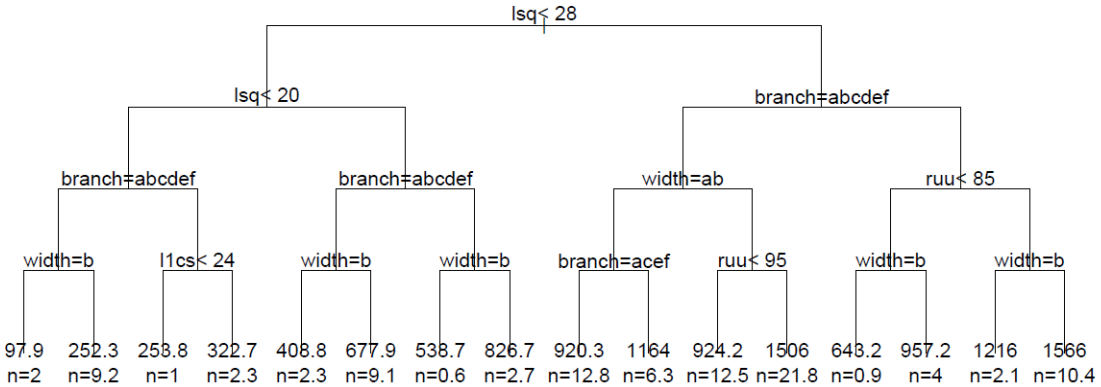



Fig. 12. The application of the generated CART tree on the test benchmarks.

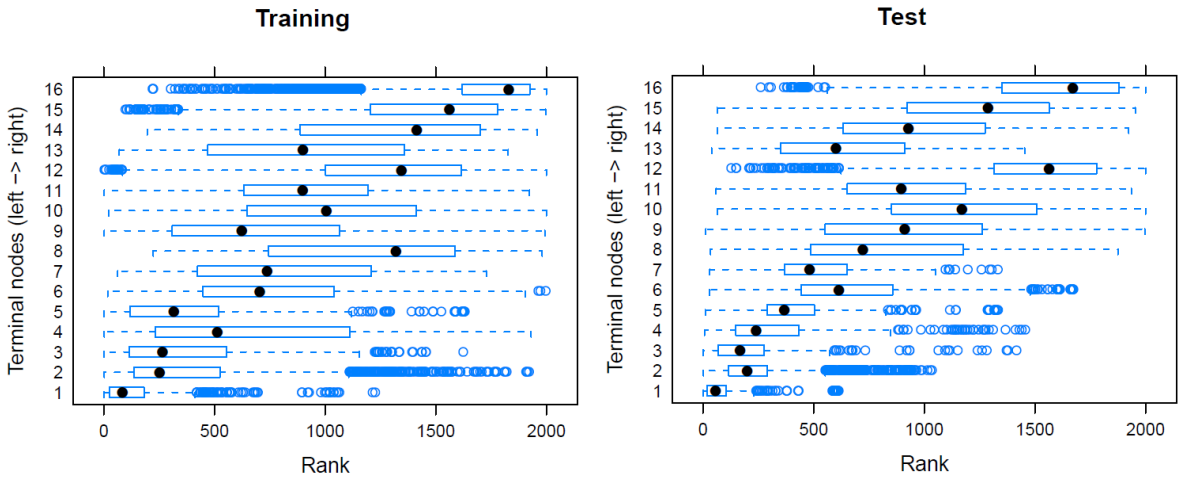



Fig. 13. The rank distribution of the data points in each of the 16 tree leaves on training (left) and test (right) data.

Fig. 11 shows the regression tree generated based on the training data. When generating this tree, we set the threshold depth **D** to 4 and the complexity parameter **cp** to 0.1%. As can be seen, the design space is partitioned into 16 subregions (i.e. the leaves in this tree); in each of them, the upper number is the average SER rank of the data points belonging to this subregion, and the bottom number (indicated by "n=") is the percentage (%) of the data points in this subregion over all data points. We summarize two typical leaves (the leftmost one and the rightmost one) as the following rule sets. Note that design points satisfying the condition at each junction are assigned to the left branch.

**Leaf 1 (the subregion with the lowest SER ranks):**

**(LSQ < 20) & (BP != *g*) & (BP != *h*) & (Width/ALUs = 2/2/1)**

**Leaf 16 (the subregion with the highest SER ranks):**

**(LSQ > 28) & (BP = *g* || *h*) & (ROB > 85) & (Width/ALUs != 2/2/1)**

We have a number of interesting findings. First, Leaf 1 quantifies the subregion with the lowest SER ranks among all the leaves. This is consistent with Rule Set I discovered by PRIM: both Rule Set I and Leaf 1 cover ~2% of the input space with most of the selected points overlapped. This part is the lowest "valley" of the whole space. On the other hand, Leaf 16 is the subregion with the highest SER ranks. When designing a reliable processor, processor architects should avoid any designs falling into this subregion. Second, from this global view we can see that the best subregions (in terms of SER ranks) are usually small and separated, while the worst subregions are usually large and connected. For instance, Leaf 1, 2, 3, 5 have the lowest SER ranks but only cover less than 13.5% of the space in total; in contrast, Leaf 12 and 16 show extremely high ranks, each covering 22.3% and 11.2% of the space, respectively. This actually indicates the high difficulty of seeking the optimal design subregions. In this sense, the PRIM scheme proposed in this paper is very effective and useful. Third, the tree hierarchy suggests the importance of different configuration parameters. For example, the LSQ size has the most significant impact on the processor SER: it appears at the top levels since both FIT and AVF are affected by the LSQ size. At the lower levels, branch predictor, processor width, and ROB size further partition the subspaces.

As a validation, we apply the obtained tree model to the test benchmarks. In Fig. 12, the tree structure is the same as in Fig. 11, but the numbers in the leaves are different since the test data is being used now. As can be seen, the overall patterns (in terms of average SER ranks) of the two trees agree; Leaf 1 and 16 still have the lowest and highest average ranks, respectively. Furthermore, Fig. 13 demonstrates how the SER ranks would distribute

Table 4. The multiprocessor design space composed of 9 parameters. Only multiprocessors with homogeneous cores are considered. The entire space size is 1,458,000.

| | Parameter | Selected Values | # Options |
|---|---|---|---|
| $M_1$ | Processor width | 2, 4, 8 | 6 |
| | # of Integer ALUs / # of FP ALUs | 1/1, 2/1-associated with processor width 2<br>2/2, 4/3-associated with processor width 4<br>4/3, 6/4-associated with processor width 8 | |
| $M_2$ | ROB size | 72, 84, 96, 108, 120, 132, 144, 156, 168 | 9 |
| $M_3$ | LQ/SQ sizes | 16, 20, 24, 28, 32 | 5 |
| $M_4$ | IQ size | 32, 40, 48, 56, 64, 72 | 6 |
| $M_5$ | Phys. Int/FP reg. file sizes | 100, 120, 140, 160, 180 | 5 |
| $M_6$ | BTB | 1024, 2048, 4096 | 3 |
| $M_7$ | RAS | 8, 12, 16 | 3 |
| $M_8$ | L1 I/D cache sizes | 16, 32, 64, 128 KB (64B block, 2-way assoc.) | 4 |
| | L1 cache latency | 1, 2, 3, 4 cycles (vary with L1 cache size) | |
| $M_9$ | (Shared) L2 cache size | 512, 1024, 2048, 4096, 8192 KB (64B block, 8-way assoc.) | 5 |
| | (Shared) L2 cache latency | 10, 12, 14, 16, 18 cycles (vary with L2 cache size) | |

Table 5. Universal rule sets optimizing different metrics for multiprocessors.

| | |
|---|---|
| **Rule Set II (Optimizing SER)** | (Width/ALUs=4/2/2\|\|4/4/3\|\|8/6/4) & (ROB<156) & (LSQ<32) & (IQ<40) & (L1CS>32KB) & (RF != 120) |
| **Rule Set III (Optimizing Throughput $^{-1}$)** | (Width/ALUs=8/4/3\|\|8/6/4) & (IQ>64) |
| **Rule Set IV (Optimizing Power)** | (Width/ALUs=8/6/4) & (ROB<156) & (16<LSQ<32) & (IQ<72) & (Phy. Reg. File<140) & (16KB<L1CS<128KB) & (L2CS !=1MB) |
| **Rule Set V (Optimizing SER $^{0.3}$ * Throughput $^{-0.4}$ * Power $^{0.3}$)** | (Width/ALUs=8/4/3\|\|8/6/4) & (LSQ < 24) & (IQ<48) & (L1CS<128KB) & (L2CS < 4MB) |
| **Rule Set VI (Optimizing SER $^{0.2}$ * Throughput $^{-0.6}$ * Power $^{0.2}$)** | (Width/ALUs=8/4/3\|\|8/6/4) & (ROB<108) & (LSQ != 24) & (IQ<40) & (L1CS<64KB) & (L2CS = 2MB\|\|8MB) |

in each of the 16 leaf subregions. This is a boxplot whose description can be found in Fig. 10 (Section 4.2); in this figure, circles are added to indicate outliers. It's easy to see that the rank distributions (in particular, the central rectangles) are very similar for the training and test benchmarks. Consequently, the effectiveness of the CART model is validated on (unseen) test programs.

## 5 Balancing Reliability, Performance, and Power for Multiprocessors

In this section, we further extend our universal PRIM modeling scheme to multiprocessors, demonstrating that universally soft error resilient design configurations still exist for a homogeneous multi-core processor running multi-threaded workloads. By varying the number of cores and the number of application threads, Soundararajan et al. [36] concluded that the configurations optimizing soft error reliability of different multi-threaded applications were not straightforward. Nevertheless, our proposed scheme can still quantify and validate the optimal design subspace for multiprocessors. Moreover, we also perform a multi-objective optimization that concurrently balances multiple design metrics (reliability, performance, and power) for a multiprocessor.

### 5.1 Experimental Setup

All experiments in this section are run using the M5 simulator [5] capable of simulating multi-threaded programs. Because of the inter-thread data sharing, a dynamically dead instruction whose result is not used by other instructions in its own thread may become an ACE instruction (thus being "vulnerable") in the system if its result is accessed by instructions from another thread. Therefore, in order to calculate the AVF for multi-threaded programs, a system-wide post-commit analysis window needs to be maintained. The committed instructions from different threads are inserted into this unified window, and their types can be determined after they reach the other end of the window. The AVF can then be calculated from such information. By following this approach, we implement the AVF measurements for ROB, Load Queue, Store Queue, and Issue Queue for multiprocessors with Alpha 21264-like CPUs. The SER of the multiprocessor is calculated based on the AVF and areas of these structures.

Six benchmarks (*Cholesky, FFT, Radix, OceanContiguous, WaterNSquared,* and *WaterSpatial*) from SPLASH2 [43] suite are evaluated, each being measured with 1 thread, 2 threads, and 4 threads enabled on single-core, dual-core, and quad-core processors, respectively. All cores in our multiprocessor model have their 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. Multi-threaded workloads explore thread-level parallelism. The multiple threads running simultaneously show contention as well as constructive behavior in the shared memory hierarchy. Therefore, the SER, performance, and power of one thread can be affected by its competing threads.

M5 simulates Alpha 21264-like out-of-order CPUs, whose important parameters are tuned and form a new design space shown in Table 4. In this study, 1,000 configurations are randomly sampled from the multiprocessor design space and simulated for each benchmark. Note that a separate core (with the corresponding configuration from the design space) is created for each of the threads enabled in the simulated program. The detailed simulation starts after the program's sequential initialization, and stops when the fastest thread finishes a certain amount of instructions.

### 5.2 Results and Discussion

Before simultaneously balancing the three metrics, we separately optimize each of them first. The multiprocessor's soft error reliability can be characterized by its aggregated SER: in our case, it's the total area times the average of all cores' AVFs (because of the core homogeneity). The reciprocal of the system throughput, i.e. $1/\text{Throughput} = (\Sigma IPC_i)^{-1}$, where $0<=i<n$, is used to represent a n-core processor's performance. Finally, the total

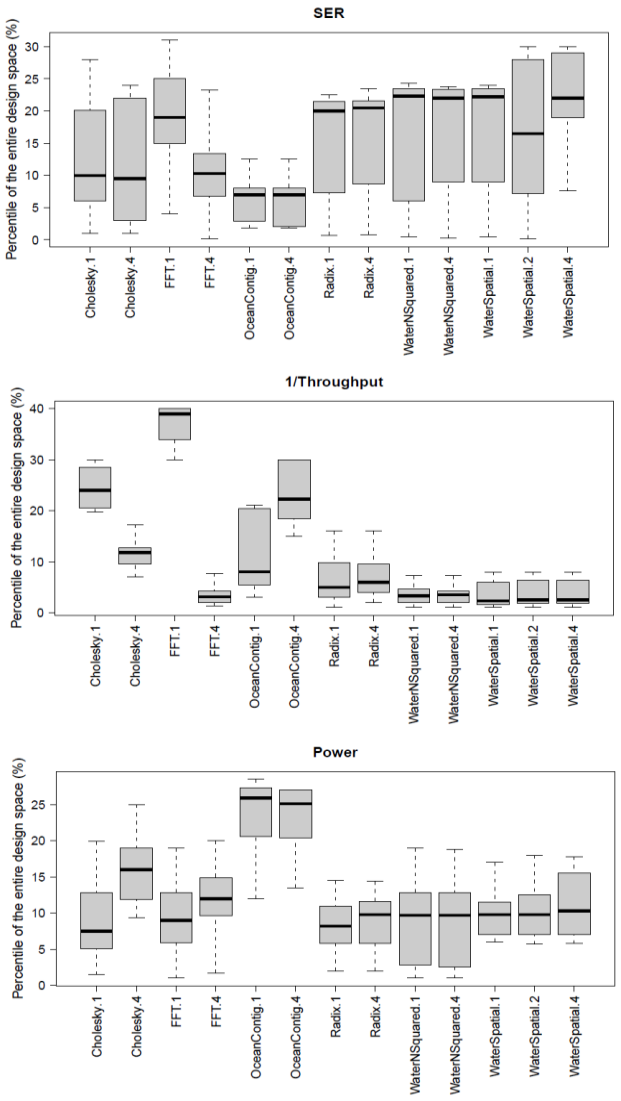



Fig. 14. Validation of Rule Set II, III, IV on the test multi-threaded benchmarks. The number at the end of a benchmark's name indicates the number of threads generated when that benchmark is run.

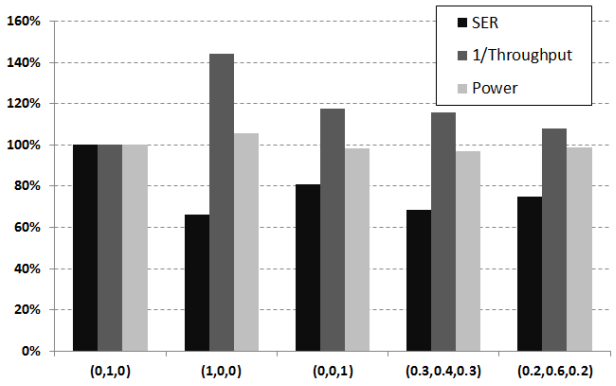



Fig. 15. The comparison (in terms of the three metrics) of different assignments of (a, b, c) in the proposed objective function f. All cases are normalized to the corresponding values in (0,1,0) that optimizes the performance.

power is the summation of all cores' power. Note that all three metrics favor lower values. We follow the same approach described in Section 4.1 (i.e. configuration parameters as the inputs and average rank as the output) to generate the universal rule sets respectively optimizing the three metrics. Specifically in this work, we put the 2-threaded runs of 5 benchmarks (except *WaterSpatial*) in the training set, and validate the generated rules on the other runs. In particular, *WaterSpatial* is chosen to have all configurations (including 1-threaded, 2-threaded, and 4-threaded) in the test set in order to validate the generated model's effectiveness across SPLASH2 benchmarks and different numbers of threads. The three rule sets (Rule Set II, III, IV) are listed in Table 5. Not surprisingly, a multiprocessor with wider CPUs and more pipeline resources usually demonstrates better performance, while a power-efficient design often selects parameters at the lower end of the range. In contrast, Rule Set II which minimizes the multiprocessor SER shows upper bounds on a few important parameters, e.g. ROB, LSQ, and IQ, but favors a bigger L1 cache. The validation of these three rule sets are shown in Fig. 14. We can see that most identified design points achieve the top 20% optima.

Simultaneously balancing the three metrics is essentially a multi-objective optimization problem that requires a reasonable objective function. We propose to minimize the following function $f$ to achieve a good trade-off among different conflicting metrics:

$$f = SER^{a} * (1/Throughput)^{b} * Power^{c}$$

**where a, b, c >= 0, and a+b+c = 1**

The exponentials a, b, and c are weight factors controlled by the designer. Formulating the objective function in this way results in an optimization process in proportional to the relative change of different metrics, ensuring more fairness than other objective functions such as normalized summation. Consequently, the designer can give more importance to a certain metric by enlarging its weight factor.

Rule Set II, III, and IV are actually special cases where one of the three weight factors equals 1 and the other two equal 0. That said, Rule Set II – IV merely optimize a certain metric without taking the other two into account. Therefore, one can expect large degradations in the other two metrics for each of the three rule sets. Fig. 15 shows the comparison of different weight factor assignments in terms of SER, performance, and power. A separate rule set is generated for each weight factor assignment shown in this figure. Each column in this figure corresponds to the average response of the design points selected by the corresponding rule set; the value of the column is normalized to the case that merely optimizes the performance, i.e. the leftmost case where (a, b, c) = (0, 1, 0). For example, the rule set merely optimizing SER with weights (1, 0, 0) demonstrates 34% improvement in SER but 44% loss in performance. Hence, none of the three rule sets (II, III, and IV) provides a good balance of reliability, performance, and power. This is expected because they individually optimize only one of the three conflicting metrics. Therefore, we need to tune the weight factors to achieve better trade-offs among the metrics. (0.3, 0.4, 0.3) is a well-balanced assignment which shows 32% improvement, 15% loss, and 3% improvement in SER, performance, and power, respectively. One can further enlarge the middle weight factor to mitigate the performance loss. For instance, (0.2, 0.6, 0.2) is another assignment that achieves SER improvement of 25% with only 7% performance degradation. Rule sets for these two assignments are also listed in Table 5 as Rule Set V and VI.

## 6 Related Work

The concept of AVF was originally proposed in [31], and Biswas et al. [6] extended it to address-based structures. A common approach to calculate the AVF is via Architectural Correct Execution (ACE) analysis [31] which provides a tight lower bound on the soft error reliability of various processor structures. A unified framework named Sim-SODA [18] to study the superscalar processor's AVF was released. Soundararajan et al. [35] described a simple infrastructure to estimate an upper bound of the ROB AVF. Zhang et al. [44] characterized the AVF on SMT architectures by examining the impact from workloads, fetch policies, etc. On the other hand, Statistical Fault Injection (SFI) [27][41] provides another approach to calculate the AVF. 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.

For the correlation between the AVF and configuration parameters, Cho et al. [8][9] predicted the dynamics of power, CPI, and the AVF using a combination of wavelets and neural networks. Our work differs from theirs in that we provide simple but useful guidelines to conduct reliable processor design. We also quantitatively analyze the effect of optimizing holistic reliability and identify the trade-off of reliability, performance, and power for multiprocessors.

A series of studies [28][29] discussed design space exploration on performance and/or power. Ipek et al. [21]

predicted performance of memory hierarchy, CPU and CMP design spaces using Artificial Neural Networks (ANNs); similarly, Lee et al. [22][24] proposed to use spline-based regression to predict performance and power from a large design space. It's also possible to derive optimal points based on their predictive models (e.g. via exhaustive prediction in Pareto Analysis [23]), but our method is a one-step search that is more efficient and direct. Besides, PRIM can provide highly interpretable selective rules. More importantly, we demonstrated in this paper that the PRIM-generated rules are effective across SPEC and SPLASH2 benchmarks. This is in contrast to traditional application-specific design space studies.

Several prior publications studied on-line AVF prediction. Fu et al. [19] observed a fuzzy correlation between the AVF and a few common performance metrics. Walcott et al. [40] 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 [10][25] further generalized this correlation to be across workloads, execution phases, and configurations. Alternatively, Li et al. [26] developed an online algorithm to estimate processor structures' vulnerability using a modified error injection and propagation scheme [27][41].

## 7 Conclusions

This paper proposes to use two statistical techniques to characterize processor reliability against soft errors via exploring a configuration design space. The first technique, Patient Rule Induction Method, generates selective rules on design parameters to identify the design space subregion showing optimal soft error reliability. By analyzing the generated rules, we find that minimizing the AVF for different processor structures has different impacts on the performance, and reducing the AVF of a single structure may increase the AVF of others. The second technique, Classification and Regression Trees, partitions the input space into small subregions, providing a global picture of the SER distribution. Via examining the regression tree, we find that the optimal subregions are usually small, separated, and thereby difficult to identify; the tree hierarchy also indicates the importance of different parameters. Finally, this work is extended to multiprocessors where multiple design metrics are simultaneously balanced. We find that tuning the weight factors of different metrics in our proposed objective function can achieve multi-objective optimizations among reliability, performance, and power.

## Acknowledgment

This work is supported in part by an 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. We acknowledge the computing resources provided by the Louisiana Optical Network Initiative (LONI) HPC team. Finally, we appreciate invaluable comments from anonymous reviewers.

## References

[1] http://www.r-project.org/
[2] http://www.simplescalar.com/
[3] http://www.ideal.ece.ufl.edu/main.php?action=simsoda
[4] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," In *International Symposium on Microarchitecture (MICRO)* 1999.
[5] N. Binkert et al, "The M5 simulator: Modeling networked systems," In *IEEE Micro*, vol. 26, no. 4, pp. 52-60, July-Aug 2006.
[6] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," In *ISCA* 2005.
[7] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees," Wadsworth International Group, Belmont, California, 1984.
[8] C. Cho, W. Zhang, and T. Li, "Informed Microarchitecture Design Space Exploration using Workload Dynamics," In *MICRO* 2007.
[9] C. Cho, W. Zhang and T. Li, "Modeling and Analyzing the Effect of Microarchitecture Design Space Parameters on Microprocessor Soft Error Vulnerability," In *MASCOTS* 2008.
[10] L. Duan, B. Li and L. Peng, "Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics," In *HPCA* 2009.
[11] L. Duan, B. Li and L. Peng, "Reliability-Constrained Processor Performance Optimization via Design Parameter Selection," In *PACT* 2009 poster session.
[12] L. Duan, L. Peng, and B. Li, "Two-Level Soft Error Vulnerability Prediction on SMT/CMP Architectures," In IEEE International Symposium on Workload Characterization (*IISWC*), 2011.
[13] L. Duan, L. Peng, and B. Li, "Predicting Architectural Vulnerability on Multi-Threaded Processors under Resource Contention and Sharing," In *IEEE Transactions on Dependable and Secure Computing*, Nov. 2012 (Preprint).
[14] L. Duan, Y. Zhang, B. Li, and L. Peng, "Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors," In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.
[15] B. Efron and R. Tibshirani, "An Introduction to the Bootstrap," Chapman & Hall/CRC. 1994
[16] J. Friedman and N. Fisher, "Bump Hunting in High-dimensional Data," In *Statistics and Computing*, 9, 123-143, 1999.
[17] X. Fu, T. Li, and J. Fortes, "Combined Circuit and Microarchitecture Techniques for Effective Soft Error Robustness in SMT Processors," In *DSN* 2008.
[18] X. Fu, T. Li, and J. Fortes, "Sim-SODA: A Unified Framework for Architectural Level Software Reliability Analysis," In *Workshop on Modeling, Benchmarking and Simulation* 2006.
[19] X. Fu, J. Poe, T. Li, and J. Fortes, "Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior," In *MASCOTS* 2006.
[20] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," In *ISCA* 2003.
[21] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana, "Efficiently Exploring Architectural Design Spaces via Predic-

tive Modeling," in *ASPLOS* 2006.

[22] B. Lee and D. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," In *ASPLOS* 2006.

[23] B. Lee and D. Brooks, "Illustrative Design Space Studies with Microarchitectural Regression Models," In *HPCA* 2007.

[24] B. Lee, J. Collins, H. Wang, and D. Brooks, "CPR: Composable Performance Regression for Scalable Multiprocessor Models," In *MICRO* 2008.

[25] B. Li, L. Duan and L. Peng, "Efficient Microarchitectural Vulnerabilities Prediction Using Boosted Regression Trees and Patient Rule Inductions," In *IEEE Transactions on Computers*, Vol 59(5), May 2010.

[26] X. Li, S. Adve, P. Bose, and J. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," In *ISCA* 2008.

[27] X. Li, S. Adve, P. Bose, and J. Rivers, "SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors," In *International Conference on Dependable Systems and Networks (DSN)* 2005.

[28] Y. Li, B. Lee, D. Brooks, Z. Hu and K. Skadron, "CMP Design Space Exploration Subject to Physical Constraints," In *HPCA* 2006.

[29] M. Monchiero, et al. "Design Space Exploration for Multicore Architectures: power/Performance/Thermal View," In *ICS* 2006.

[30] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," In *ISCA* 2002.

[31] 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," In *MICRO* 2003.

[32] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," In *ISCA* 2000.

[33] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor," In *FTCS* 1999.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behaviors," In *ASPLOS* 2002.

[35] N. Soundararajan, A. Parashar, and A. Sivasubramaniam, "Mechanisms for Bounding Vulnerabilities of Processor Structures," In *ISCA* 2007.

[36] N. Soundararajan, A. Sivasubramaniam, and V. Narayanan, "Characterizing the Soft Error Vulnerability of Multicores Running Multithreaded Applications," In *SIGMETRICS* 2010.

[37] V. Sridharan and D. Kaeli, "Eliminating Microarchitectural Dependency from Architecture Vulnerability," In *HPCA* 2009.

[38] V. Sridharan and D. Kaeli, "Using Hardware Vulnerability Factors to Enhance AVF Analysis," In *ISCA* 2010.

[39] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery Using Simultaneous Multithreading," In *ISCA* 2002.

[40] K. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic Prediction of Architectural Vulnerability from Microarchitectural State," In *ISCA* 2007.

[41] N. Wang, A. Mahesri, and S. Patel, "Examining ACE Analysis Reliability Estimates Using Fault-Injection," In *ISCA* 2007.

[42] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor," In *ISCA* 2004.

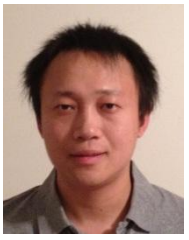
[43] S. C. Woo, et al. "The SPLASH-2 Programs: Characterizing and Methodological Considerations," In *ISCA* 1995.

[44] W. Zhang, X. Fu, T. Li, and J. Fortes, "An Analysis of Microarchitecture Vulnerability to Soft Errors on Simultaneous Multithreaded Architectures," In *ISPASS* 2007.

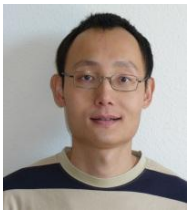
[45] W. Zhang and T. Li, "Managing Multi-Core Soft-Error Reliability Through Utility-driven Cross Domain Optimization," In *ASAP* 2008.

[46] J. Ziegler and et al. "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)," IBM Journal of Research and Development, Volume 40, Number 1, 1996.

**Lide Duan** received the BS degree in Computer Science and Engineering from Shanghai Jiao Tong University, China, and the Ph.D. degree in Electrical and Computer Engineering from Louisiana State University. He is currently working as a senior design engineer in core modeling for AMD's x86 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 Louisiana Optical Network Initiative (LONI) and the Dissertation Year Fellowship from LSU Graduate School during his Ph.D. study.



**Ying Zhang** received the Bachelor's and Master's degree in Electronics and Information Engineering from Huazhong University of Science and Technology, China, in June 2006 and 2008. He is currently a PhD student in Electrical and Computer Engineering, Louisiana State University. His research interests include GPU performance characterization and hard-error reliable processor design. He also has interests in energy-efficiency optimization for heterogeneous architectures.



**Bin Li** received his Bachelor's degree in Biophysics from Fudan University, China. He obtained his Master's degree in Biometrics (08/2002) and Ph.D. degree in Statistics (08/2006) 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 & 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 ASA on Bayesian Statistical Science in 2005. Dr. Li is a member of the Institute of Mathematical Statistics (IMS) and American Statistical Association (ASA).

**Lu Peng** is currently an Associate Professor with the Division of Electrical and Computer Engineering at Louisiana State University. He received the Bachelor's and Master's degrees in Computer Science and Engineering from Shanghai Jiao Tong University, China. He obtained his Ph.D. degree in Computer Engineering from the University of Florida in Gainesville in April 2005. His research focus on memory hierarchy system, reliability, power efficiency and other issues in CPU design. He also has interests in Network Processors. He received an ORAU Ralph E. Powe Junior Faculty Enhancement Awards in 2007 and a Best Paper Award from IEEE International Conference on Computer Design in 2001. Dr. Peng is a member of the ACM and the IEEE Computer Society.