Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors

Lide Duan[1], Ying Zhang[1], Bin Li[2], and Lu Peng[1]
[1]Department of Electrical and Computer Engineering
[2]Department of Experimental Statistics
Louisiana State University
Baton Rouge, LA 70803
{lduan1, yzhan29, bli, lpeng}@lsu.edu

Abstract — High-performance processors suffer from soft error vulnerability due to the increasing on-chip transistor density, shrinking processor feature size, lower threshold voltage, etc. In this paper, we propose to use a rule search strategy, i.e. Patient Rule Induction Method (PRIM), to optimize processor soft error robustness. By exploring a huge microarchitectural design space on the Architectural Vulnerability Factor (AVF), we are capable of generating a set of selective rules on key design parameters. Applying these rules at early design stage effectively identifies the configurations that are inherently reliable to soft errors. Furthermore, we propose a generic approach capable of generating a set of "universal" rules that achieves the optimization of the output variable for different programs in execution. The effectiveness of the universal rule set is validated on programs that are not used in training. This cross-program capability is very useful in the era of multi-threading. Finally, the proposed scheme is extended to multiprocessors where multiple design metrics including reliability, performance and power are balanced. Our proposed methodology is able to generate quantitative and universal solutions for both uniprocessors and multiprocessors.

Keywords-soft error; architectural vulnerability factor; cross-program; design parameter selection; multi-objective optimization;

I. INTRODUCTION

Current high-performance processors suffer from soft error susceptibility issues which are generated in twofold aspects. The electronic noises, which usually come from large power supplies, strong radiation, or high-energy particle strikes, may invert the logic bits of processor structures, introducing transient faults (i.e. soft errors) into the system [36][1][11][20][22][23][26][30][33]. On the other hand, there is a strong decreasing tendency in the development of processor feature size and supply voltage, but the on-chip transistor density is fast increasing. Collectively, these factors make current processors extremely vulnerable to soft errors.

To characterize a processor's soft error reliability, one should look at its effective soft error rate, i.e. the amount of actual errors resulted from raw soft errors in a time unit. Hence, the effective soft error rate (SER) is the product of raw SER and the probability that a soft error produces a visible error in the program output. The former is dependent on many factors at the circuit level including the critical charge of the circuits, processor area, temperature, etc.; while the latter is quantified at the architectural level by the Architectural Vulnerability Factor (AVF) [21][3]. The AVF was proposed based on the observation that a large amount of raw soft errors are masked at different layers in a computer system. In this paper, we focus on minimizing the AVF for soft error resilient designs at the architectural level. The term "reliability" used in this paper (as well as many prior publications) refers to the processor robustness to *raw* soft errors.

The AVF is a combinational behavior of hardware and software [28][29]. A raw soft error can be masked in either the hardware system (e.g. idle bits in processor structures) or the program in execution (e.g. dynamically dead instructions). Consequently, a system's AVF value can span a wide range when the same program executes on different processor configurations (whose parameters are from a design space), while different programs may demonstrate completely different AVF behaviors on the same processor. The situation is even more complicated in a multiprocessor where different cores share a low level cache. The AVF of one core may be affected by the other cores through the contention from the shared cache [35]. Figure 1 shows how the system AVF of certain configurations (cfg1 to cfg5) would vary when different workloads (multi-

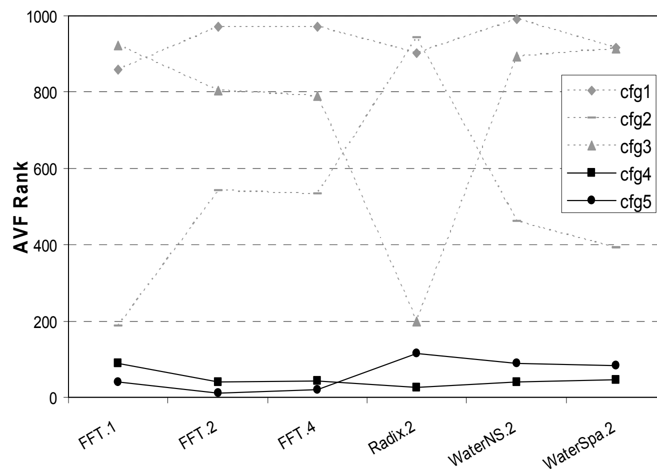

Figure 1. The AVF rank of a certain configuration varies significantly across different multi-threaded workloads. A lower rank indicates a lower AVF value that is favored in a reliable processor.

threaded benchmarks from SPLASH2 suite [34]) execute on a multi-core processor. The number at the end of a benchmark's name indicates the number of threads generated for that benchmark. cfg1 to cfg5 are chosen from a huge multiprocessor design space (illustrated in Section IV), from which 1,000 design points are randomly sampled and simulated for each workload in analysis. Only multiprocessors with homogeneous cores are considered here. The 1,000 sampled configurations are ranked in each workload in terms of their AVF values, with the lowest value as rank 1 and the largest value as rank 1000. We can see that, for the same configuration, its AVF rank could be significantly different across different programs or the same program with different numbers of threads (i.e. FFT). For cfg1-3, there exist one or more workloads suffering from significant reliability degradation, i.e. a high AVF rank. A reliable processor design would favor configurations (such as cfg4 and cfg5 in this example) whose AVF ranks are consistently low in different applications. However, the difficulty is how to quantitatively and efficiently identify such designs from a huge design space.

In this work, we propose an effective approach to identify the configurations that have consistently low AVF values from a huge design space. Those identified configurations are inherently reliable to soft errors. Specifically, we characterize the design space using *Patient Rule Induction Method* (PRIM) [8] to generate a set of selective rules on key design parameters. Applying these rules on the design space effectively identifies the subregion of the design space within which the output variable is considerably smaller (i.e. "valley seeking") than its average value over the entire design space. Therefore, the design configurations selected by the generated rules are inherently resilient to potential soft errors. This technique provides computer architects with useful guidelines to design reliable processors at pre-silicon stage. More importantly, we are able to apply PRIM skillfully to derive a method that can generate "universal" rules effective across different applications. This cross-program capability is extremely useful in the era of multi-threading since the number of multi-threaded programs boosts quickly nowadays. For instance, the number of multi-programmed workloads increases super-linearly with the increases of the number of threads and the number of benchmarks in consideration. Traditional application-specific design space studies lack the scalability to multiprocessors due to the intractable training costs required by the greatly enlarged workload set. In contrast, our universal rules guided design parameter selection can provide workload-independent guidelines which are also validated to be effective for unseen applications not used in training. Note, however, that by "universal" we don't refer to rules working for ALL programs (which may not even exist); instead, we manage to identify the rules with cross-program effectiveness for SPEC CPU and SPLASH2 benchmark suites used in this work. Nevertheless, these commonly used benchmarks are good representatives of real-world applications. Finally, the proposed approach is inherently generic, so it can be applied to optimize other processor design metrics, such as power and performance. A case study performed in Section IV utilizes the universal design parameter selection method to achieve a multi-objective optimization of reliability, performance, and power for multiprocessors.

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

- **Design parameter selection for reliable processors:** We propose to use an advanced rule search strategy (PRIM) to extract the design space subregion showing lowest AVF values. By using this technique, we quantitatively demonstrate that (1) minimizing the AVF for different processor structures may degrade or improve the performance; (2) reducing the AVF of a single structure may increase the AVF of others in the processor. This addresses the demand for a holistic reliability optimization.
- **Universal rules generation and validation:** We propose a generic approach capable of generating a set of "universal rules" that achieves the optimization of the output variable across different programs in execution. The effectiveness of the universal rule set is further validated using a well-developed statistical method (bootstrapping [7]) on programs that are not used in training.
- **Balancing reliability, performance and power for multiprocessors:** We perform a study using the proposed universal modeling scheme to simultaneously balance multiple design metrics for multiprocessors. We quantitatively identify proper trade-offs of reliability, performance and power for a multi-core processor running multi-threaded workloads.

II. METHODOLOGY

The objective of PRIM, which was originally proposed by Friedman and Fisher [8], is to find 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 AVF. The identified input space subregion (or "box") is described as a set of simple "selective rules" in a form of

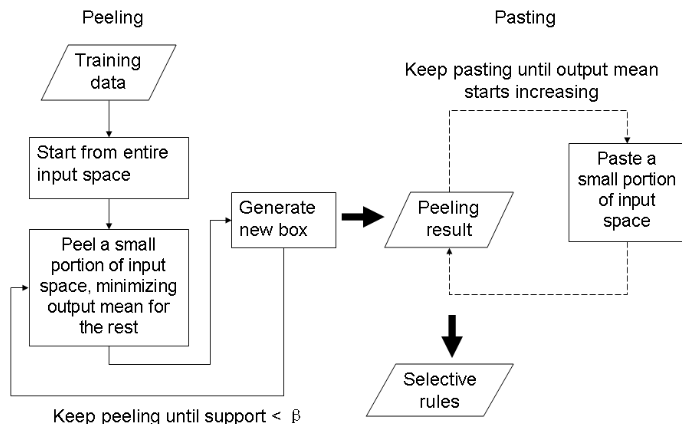

Figure 2. PRIM training procedure, including peeling and pasting.

$B = \bigcap_{j=1}^{p} (x_j \in s_j)$. x_j represents the jth input variable, and s_j is a subset of all possible values for the jth variable. Specifically in our work, x_j can be one of the eight configuration parameters (**P_1** to **P_8**) listed in Table 2, in which the third column lists all possible values for each input variable. Hence, the identified region *B* is the intersection of p subsets, each being from one input variable. For example, s_3={16,40,64} is a valid subset for **P_3** (LSQ size).

The box construction of PRIM consists of two phases: patient successive top-down peeling and bottom-up recursive pasting. Figure 2 visualizes this procedure. The top-down peeling starts from the entire design 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 $B - b = \{x \mid x \in B \,\&\, x \notin b\}$, and try 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 support of the current box *B* is below a chosen threshold *β*, which is the proportion of the design points remaining in the identified area. 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 that 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 possible candidates to be removed in the next iteration.

The pasting algorithm is simply the inverse of the peeling procedure. The reason for pasting is that at each iteration of peeling 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*p* operations, where *n* is the number of observations and *p* is the number of input variables. The number of operations for each peel will decrease since fewer and fewer samples are left during peeling. On the other hand, PRIM performs approximately *–log(n)/log(1-α)* peeling steps, where *α* is the portion that is peeled off at each peeling iteration.

An advantage of PRIM over greedy methods such as tree-based 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 (hyper-boxes) is usually much more stable than the tree models. In other words, if the data are slightly changed, a tree structure may change dramatically while 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

Table 1. SPEC benchmarks used in the uniprocessor study. The partitioning of training and test sets will be used in Section III (B).

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

Table 2. The uniprocessor design space is composed of parameters **P_1** to **P_8**. Branch predictors are renamed to *BP1* to *BP8* for later use.

	Parameter	Selected Values	# Options
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 2/1, 2/2-associated with processor width 4 2/2, 4/4-associated with processor width 8	
P_2	ROB size	64, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160	11
P_3	LSQ size	16, 24, 32, 40, 48, 56, 64	7
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)	
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)	
P_6	L2 cache associativity (L2CA)	4, 8	2
P_7	Branch predictor (BP)	bimod/4096 (*BP1*), bimod/8192 (*BP2*), 2lev/1/4096 (*BP3*), 2lev/2/4096 (*BP4*), 2lev/4/4096 (*BP5*), 2lev/1/8192 (*BP6*), 2lev/2/8192 (*BP7*), 2lev/4/8192 (*BP8*)	8
P_8	BTB	1024/4, 2048/2, 1024/8, 2048/4	4

on the remaining dataset. As a result, the disconnected subspace can also be covered. However, we found that 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 β 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.

III. DESIGN PARAMETER SELECTION FOR UNIPROCESSORS

This section utilizes the proposed methodology to select design parameters for a soft error resilient uniprocessor. We implement the AVF calculations [21][9] in an extended version of SimpleScalar3.0 [25] to simulate a detailed out-of-order multistage superscalar. Our simulation framework measures the AVF of major microarchitectural components in a uniprocessor, including ROB, LSQ, Functional Unit, and Register File. The processor overall AVF is the ratio between the number of ACE bits of the entire processor and the processor size in bits. It can be calculated as the summation of different structure's AVFs weighted by the corresponding structures' sizes.

We use a mixed set of benchmarks from SPEC CPU2000 and 2006 suits. As listed in Table 1, 24 of them form the training set in generating universal rules (Section III(B)), while the rest 12 are used for test (Section III(C)). In order to have a complete evaluation, the entire SPEC CPU2000 suite (except *sixtrack*) is included. For the other SPEC CPU benchmarks not used in this work, we are not able to compile them into Alpha binaries runnable in SimpleScalar. Each benchmark is simulated for 100 million instructions in details after being fast forwarded to a representative phase derived from SimPoint Toolkit [24]. The AVF values along with performance data are outputted at the end of each simulation.

For this study on uniprocessors, we construct a design space composed of eight configuration parameters ($\mathbf{P_1}$ to $\mathbf{P_8}$ in Table 2). Note that the ROB and LSQ sizes are configured in a fine-grained manner because our scheme intends to quantify the value ranges of the parameters in the optimal configurations. Given the possible values of each parameter, the total number of points in the design space is 473,088. For each benchmark, we simulate 2,000 configurations randomly and uniformly sampled from the entire design space.

A. *Application-Specific Design Parameter Selection*

In this subsection, we directly apply the PRIM method and build a separate model for each benchmark to minimize its AVF. In other words, the derived rules are towards the optimization of the AVF for a specific application. For each benchmark, the rule sets minimizing the AVF of different structures are generated. From these results, we observe that minimizing the AVF of different structures have different impacts on the performance. Table 3 lists the results for a set of benchmarks. The rules for other benchmarks show similar behaviors, thus being omitted.

First, some rules tend to degrade the performance considerably when minimizing the AVF. Specifically, the rules for *mcf* to optimize the ROB AVF introduce restrictions on branch predictor selection. Figure 3 shows the variation of branch misprediction rate and the ROB AVF when different branch predictors *BP1* to *BP8* are used in the configurations running *mcf*. The other parameters are the same for these configurations. Clearly, the ROB AVF varies contrarily with respect to the variation of branch misprediction rate. If we exclude *BP1* and *BP2* 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 will result in a larger execution time.

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 high performance, e.g. larger ROB or LSQ, larger caches, wider CPUs, or more accurate branch predictors. For example, the rules for *milc* favor a ROB size larger than 110.

Table 3. Rules for optimizing individual structure's AVF for a set of benchmarks. "*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!= *BP1*) & (BP!=*BP2*)	(LSQ > 48) & (L2CS < 4096) & (BP!=*BP1*) & (BP!=*BP2*)	Width/ALUs=8/4/4	(Width/ALUs!=2/1/1) & (LSQ > 24) & (BP=*BP1*||*BP2*)
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!=*BP2*) & (BP!=*BP1*)	(LSQ > 40) & (L1CS < 64) & (BP!=*BP1*) & (BP!=*BP2*)	(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!=*BP5*) & (BP!=*BP3*) & (BP!=*BP4*)
milc	(ROB>90) & (LSQ<32)	(ROB < 110) & (LSQ > 40)	Width/ALUs=8/4/4	(Width/ALUs!=2/1/1) & (ROB > 110) & (LSQ > 40)

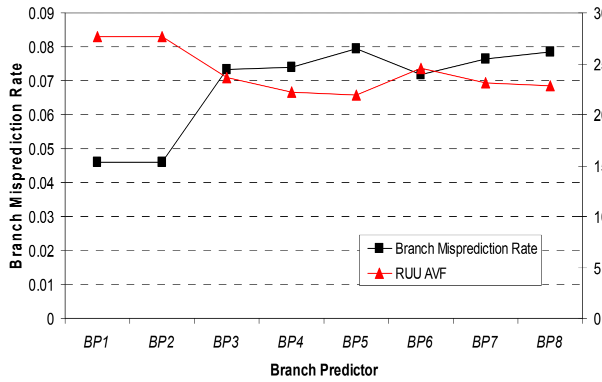

Figure 3. ROB AVF of *mcf* (SPEC 2000) varies with different branch predictors.

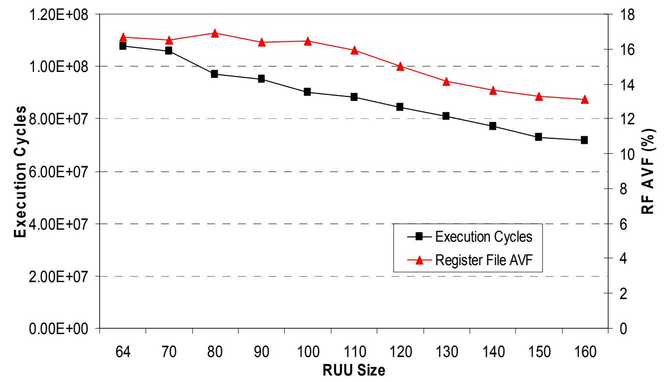

Figure 4. Register File AVF of *milc* (SPEC 2006) varies with different ROB sizes.

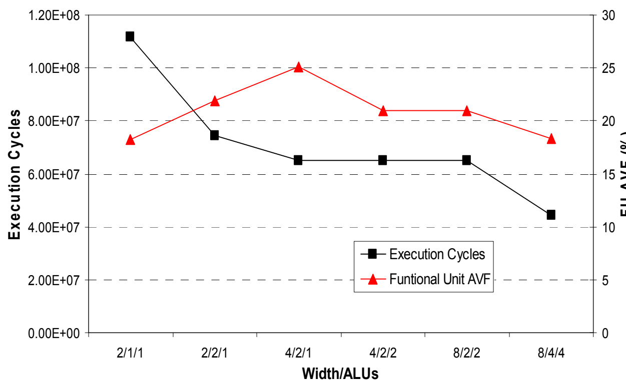

Figure 5. Functional Unit AVF of *fma3d* (SPEC 2000) varies with different combinations of processor width and # of ALUs.

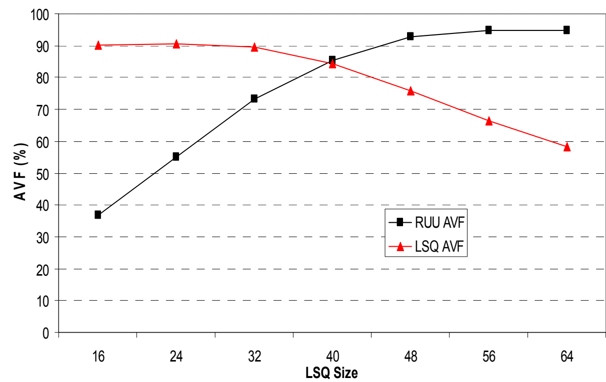

Figure 6. ROB and LSQ AVFs of *applu* (SPEC 2000) vary with different LSQ sizes.

Figure 4 shows the RF AVF and performance variation trends under different ROB sizes. It is easy to see that a larger ROB size results in better performance and a more reliable Register File as well. More pipeline resources will improve performance, making instructions pass through the pipeline more quickly. This will shorten the write-read interval for a certain register, thus reducing ACE cycles and decreasing the register file AVF.

The correlation between the AVF and performance is fuzzier in Functional Units. A wide processor usually incurs a low FU AVF because it has more ALUs which will execute the instructions more quickly (thus fewer ACE cycles); a narrow processor significantly degrades performance, but its FU AVF may be still 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, Figure 5 illustrates that in *fma3d* the execution cycles consistently decrease with increased processor 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) in optimizing the FU AVF. If performance is taken into consideration, one should choose a wide processor.

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).

B. Universal Rules Guided Design Parameter Selection

In Table 3, we observe many contradictions between the rule sets for optimizing the AVFs of different structures. For example, *applu* requires a small LSQ size (<32) but a large one (>40) in optimizing the AVF for ROB and LSQ, respectively. Figure 6 illustrates this contradiction. We can see that the LSQ AVF decreases with the increase of LSQ size, but in the mean time the ROB AVF quickly boosts to a very high value. Therefore, if the rules for optimizing the LSQ AVF are adopted in a 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 of the entire processor should be considered to achieve a holistically reliable solution.

However, all the rule sets in Table 3 are generated specifically for a certain benchmark. In other words, PRIM is directly applied to the AVF measurements of that particular benchmark to summarize the rules. Consequently, the rule sets work well for their corresponding applications, but differ from each other. From Figure 1 and Table 3, 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. However, directly combining the AVF measurements from different benchmarks for training is not feasible since the absolute AVF value ranges significantly differ in different benchmarks. Instead, we propose to rank the configurations in each benchmark in terms of the AVF measurements. For the 2K configurations used in this work, the one with the lowest AVF value is ranked 1 while the one with the largest AVF is ranked 2000. If so, a certain configuration would have 24 different ranks for the 24 training benchmarks in Table 1, respectively. We use the average of these ranks as the output response to train PRIM models. The generated rule set contains the design points that are universally reliable for all training benchmarks. An alternative is to look at the maximum of the ranks, but the results generated in this way are more conservative. 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 from the training benchmarks that optimizes the overall AVF of a uniprocessor. It is shown as Rule Set I as follows:

Rule Set I (Optimizing Uniprocessor AVF):

(Width/ALUs!=8/2/2) & (ROB>130) & (LSQ<24) & (L1CS<128kB) & (BP!=*BP1*) & (BP!=*BP2*)

Rule Set I provides useful guidelines in designing a holistically and universally reliable processor. It favors a large ROB size because ROB has the largest contribution to the core AVF; other factors somehow degrade the performance, validating our previous observation that a contradiction exists between optimizing performance and some structures' AVF. The next subsection will test this rule set on other benchmarks to validate its effectiveness on unseen programs.

C. *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 benchmark being tested, the validation consists of the following steps:

(1) Simulate 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, β 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 design points but less than the rest. In order to make inference based on the entire design space, we use the bootstrapping method [7]. Specifically, we first sample (with replacement) 1000 bootstrap samples for the 2000 configurations. Note that each bootstrap sample also contains 2000 design points. We then compute a confidence interval estimate of the p-percentile of the entire design space based on these samples. Specifically, for each bootstrap sample, we calculate its p-percentile. This gives us a total of 1000 values for p-percentiles (one for each bootstrap sample). Among these 1000 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 (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 it is below (using the bootstrapping method). We use boxplot to demonstrate the validation results in Figure 7. In a boxplot, the upper and lower

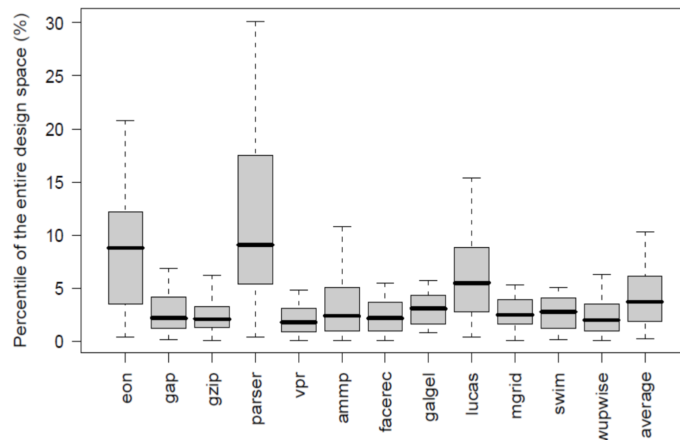

Figure 7. Validation of Rule Set I on the test benchmarks.

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 *vpr*, the maximum of the points selected by Rule Set I corresponds to a value of 5% in the vertical axis, meaning that in this benchmark all selected points are within the top 5% 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 10% optima of the entire design space. Again, as clarified in Section I, 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 benchmark suites well represent real-world applications.

IV. 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 application threads, Soundararajan et al. [27] concluded that the configurations optimizing soft error reliability of different multi-threaded applications are not straightforward. Our proposed scheme can still quantify and validate the optimal subspace for multiprocessors. Furthermore, we perform a multi-objective optimization in this section that concurrently balances multiple design metrics (reliability, performance and power) for a multiprocessor.

All experiments in this section are run using the M5 simulator [2] capable of simulating multi-threaded benchmarks that have data sharing among threads. Consequently, the result of an instruction that is "dynamically dead" in one thread may be used in another thread, making it "vulnerable" as well. Therefore, in order to calculate the AVF for multi-threaded workloads, a system-wise 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 reaching the other end of the window. The AVF can then be calculated from such information. We implement the AVF measurements for ROB, Load Queue, Store Queue and Issue Queue for multiprocessors with Alpha 21264-like CPUs. 6 benchmarks (*Cholesky, FFT, Radix, OceanContiguous, WaterNSquared,* and *WaterSpatial*) in SPLASH2 [34] 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 multipro-

Table 4. The multiprocessor design space is composed of parameters $\mathbf{M_1}$ to $\mathbf{M_9}$. Only multiprocessors with homogeneous cores are considered. The entire space size is 1,458,000.

	Parameter	Selected Values	# Options
$\mathbf{M_1}$	Processor width	2, 4, 8	6
	# of Integer ALUs / # of FP ALUs	1/1, 2/1-associated with processor width 2 2/2, 4/3-associated with processor width 4 4/3, 6/4-associated with processor width 8	
$\mathbf{M_2}$	ROB size	72, 84, 96, 108, 120, 132, 144, 156, 168	9
$\mathbf{M_3}$	LQ/SQ sizes	16, 20, 24, 28, 32	5
$\mathbf{M_4}$	IQ size	32, 40, 48, 56, 64, 72	6
$\mathbf{M_5}$	Phys. Int/FP reg. file sizes	100, 120, 140, 160, 180	5
$\mathbf{M_6}$	BTB	1024, 2048, 4096	3
$\mathbf{M_7}$	RAS	8, 12, 16	3
$\mathbf{M_8}$	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)	
$\mathbf{M_9}$	(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 for optimizing different metrics for multiprocessors

Rule Set II (Optimizing AVF)	(Width/ALUs=4/2/2||4/4/3||8/4/3||8/6/4) & (ROB>132) & (LSQ<32) & (IQ<48) & (L1CS>32kB) & (L2CS<2MB)
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 AVF $^{0.3}$ * Throughput $^{-0.4}$ * Power $^{0.3}$)	(Width/ALUs=8/4/3||8/6/4) & (ROB>132) & (LSQ < 32) & (IQ<56) & (BTB>1024) & (L1CS<128kB) & (L2CS != 4MB)
Rule Set VI (Optimizing AVF $^{0.2}$ * Throughput $^{-0.6}$ * Power $^{0.2}$)	(Width/ALUs=8/4/3||8/6/4) & (ROB>120) & (LSQ < 32) & (IQ<48) & (L1CS<128kB) & (L2CS != 8MB)

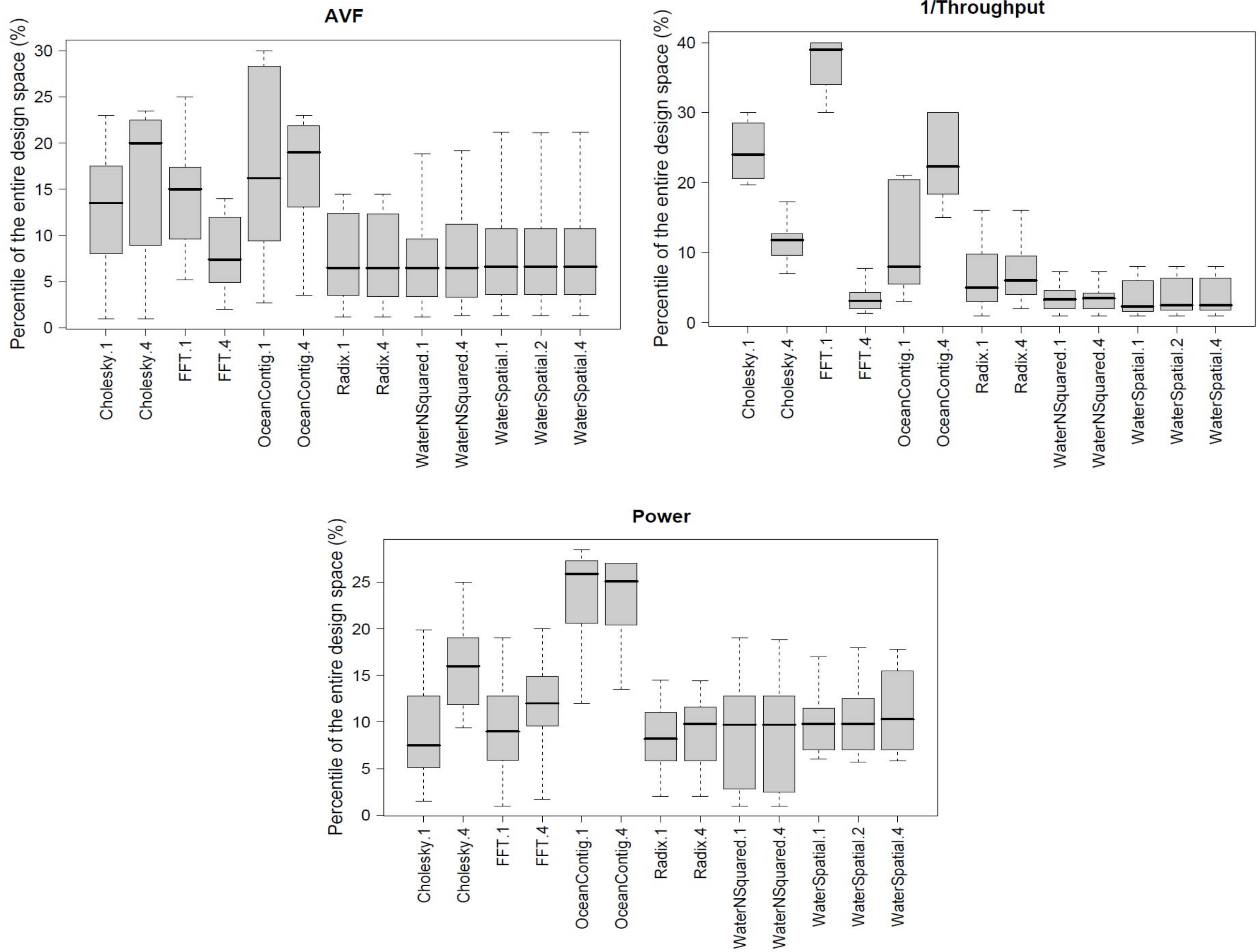

Figure 8. Validation of Rule Set II, III, IV on the test multi-threaded benchmarks. The number at the end of a benchmark's name indicates the number of threads for that benchmark.

cessor 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 protocol. Multi-threaded workloads explore thread-level parallelism. The multiple threads running simultaneously show contention as well as constructive behaviors in the shared memory hierarchy. Therefore, the AVF, performance and power of one thread can be affected by its resource competitors. 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 benchmark. The detailed simulation starts after the program's sequential initialization, and stops when the fastest thread finishes a certain amount of instructions.

Before simultaneously balancing the three metrics, we separately optimize each of them first. The multiprocessor's soft error vulnerability can be characterized by its aggregated AVF: in our case, it's 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 power is the summation of all cores' power. Note that all three metrics favor a lower value. We follow the same approach described in Section III(B) to generate universal rule sets optimizing the three metrics, respectively. Specifically in this work, we put the 2-thread runs of 5 benchmarks (except *WaterSpatial*) in the training set, and validate the generated rules with 1-thread and 4-thread runs. In particular, *WaterSpatial* is chosen to have all configurations (including 1-thread, 2-thread, and 4-thread) in the test set, validating the generated model's effectiveness across different SPLASH2 benchmarks and different numbers of threads. These 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 lower end of the range. In contrast, Rule Set II which minimizes the processor AVF favors a large value in some structures (e.g. ROB) but a small value in some others (e.g. LSQ, IQ). The validation of these three rule sets are shown in

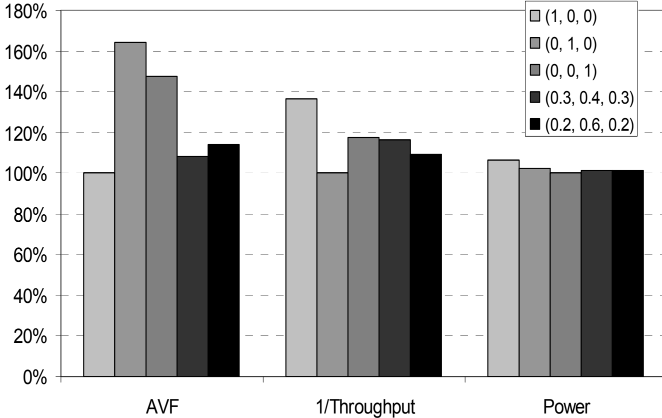

Figure 9. The comparison (in terms of the three metrics) of different assignments of (a, b, c) in the proposed objective function *f*. The values are normalized to the corresponding optimal assignment, i.e. (1, 0, 0) for AVF, (0, 1, 0) for performance, and (0, 0, 1) for power.

the top 20% optima.

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

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

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 as above would result 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. The above discussions regarding 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. Figure 9 shows the comparison of different assignments of weight factors (a, b, c) in terms of AVF, performance and power. A separate rule set is generated for each weight factors assignment. Each column in this figure corresponds to the average response of the design points selected by the corresponding rule set, and this value is normalized to the case merely optimizing the response metric (i.e. one in Rule Set II, III, IV). For example, the AVF of the configurations selected by Rule Set III (weight factors (0, 1, 0)) demonstrates 64.5% degradation compared to those selected by Rule Set II which merely optimizes the AVF; in contrast, Rule Set II shows 36.6% degradation in performance than Rule Set III. Hence, none of the three rule sets (II, III, and IV) provides good balance of reliability, performance and power. This is in our expectations since they individually optimize only one of the three conflicting metrics. On the other hand, tuning the weight factors would result in better trade-offs among the metrics. (0.3, 0.4, 0.3) is a well balanced assignment which shows in the figure 8.4%, 16.3%, and 1.3% degradations in AVF, performance and power, respectively. One can further enlarge the performance's weight factor to mitigate the performance loss. For instance, (0.2, 0.6, 0.2) is another assignment that decreases performance degradation to 9.4% but comes with an increased AVF degradation to 14.3%. The rule sets for these two assignments are also listed in Table 5 as Rule Set V and VI.

V. RELATED WORK

For the correlation between the AVF and configuration parameters, Cho et al. [4] predicted the dynamics of power, CPI and the AVF using a combination of wavelets and neural networks. They also followed the same approach to predict the average soft error vulnerability and its tradeoff with performance [5]. Our work differs from theirs in that we provide simple but helpful 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 discussed design space exploration on performance and/or power [18][19]. Ipek et al. [12] predicted performance of memory hierarchy, CPU and CMP design spaces using Artificial Neural Networks (ANNs); Similarly, Lee et al. [13] 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 [14]), 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. [10] observed a fuzzy correlation between the AVF and a few common performance metrics. Walcott et al. [31] 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 processor performance metrics. Duan et al. [6] further generalized the correlation to be across workloads, execution phases and configurations. Alternatively, Li et al. [16] developed an online algorithm to estimate processor structures' vulnerability using a modified error injection and propagation scheme [17][32].

The PRIM-based statistical technique has been utilized in Section 5 in [6], but architecture-wise, our work is completely different. In [6] and [15], the authors performed the instantaneous AVF estimation based on the measurements of performance metrics. In contrast, our work guides to produce reliable processors via design parameter selection at the pre-silicon stage before the processor is manufactured.

VI. CONCLUSIONS

In this paper, we propose to use a rule search statistical technique to generate simple selective rules on design parameters. These rules quantify the design space subregion that contains the configurations optimized for soft error reliability or other design metrics, providing computer architects with valuable guidelines to design reliable and high performance processors at early design stage. We found that reducing the AVF of a single processor structure may increase the vulnerability of other structures, and merely minimizing a processor's AVF may degrade performance. Our proposed generic approach is capable of generating a set of "universal" rules that achieves the optimization of the output variable across different programs in execution. The effectiveness of the universal rule set is validated on programs that are not used in training. Finally, the extension to multiprocessors enables a multi-objective optimization of reliability, performance and power for multi-threaded workloads.

ACKNOWLEDGMENTS

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, LEQSF (2006-09)-RD-A-10, NSF (2009)-PFUND-136, and the Louisiana State University Research Council. Lide Duan is holding a Louisiana Optical Network Initiative (LONI) Graduate Fellowship and a George Reymond Scholarship. Ying Zhang is holding a Flagship Graduate Fellowship from the LSU graduate school. We acknowledge the computing resources provided by the LONI HPC team. Finally, we appreciate invaluable comments from the shepherd and anonymous reviewers which help us finalize the paper.

REFERENCES

[1] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," In *International Symposium on Microarchitecture (MICRO)* 1999.

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

[3] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," In *ISCA* 2005.

[4] C. Cho, W. Zhang, and T. Li, "Informed Microarchitecture Design Space Exploration using Workload Dynamics," In *MICRO* 2007.

[5] 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.

[6] L. Duan, B. Li and L. Peng, "Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics," In *HPCA* 2009.

[7] B. Efron and R. Tibshirani, "An Introduction to the Bootstrap," Chapman & Hall/CRC. 1994

[8] J. Friedman and N. Fisher, "Bump Hunting in High-dimensional Data," In *Statistics and Computing*, 9, 123-143, 1999.

[9] 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.

[10] X. Fu, J. Poe, T. Li, and J. Fortes, "Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior," In *MASCOTS* 2006.

[11] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," In *ISCA* 2003.

[12] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," in *ASPLOS* 2006.

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

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

[15] 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.

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

[17] 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.

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

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

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

[21] 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.

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

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

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

[25] SimpleScalar. www.simplescalar.com

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

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

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

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

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

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

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

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

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

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

[36] 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.