AVF Analysis Acceleration via Hierarchical Fault Pruning

Michail Maniatakos
EE Department
Yale University
michail.maniatakos@yale.edu

Chandra Tirumurti
Design and Test Solutions
Intel Corporation
chandra.tirumurti@intel.com

Abhijit Jas
Design and Test Solutions
Intel Corporation
abhijit.jas@intel.com

Yiorgos Makris
EE & CS Departments
Yale University
yiorgos.makris@yale.edu

***Abstract*—The notion of Architectural Vulnerability Factor (AVF) has been extensively used by designers to evaluate various aspects of design robustness. While AVF is a very accurate way of assessing element resiliency, its calculation requires rigorous and extremely time-consuming experiments. In response, designers have introduced various methodologies that allow AVF calculation within reasonable time, at the cost of some loss of accuracy. In this paper, we present a method for calculating the AVF of design elements -using Statistical Fault Injection (SFI)- with equal accuracy but several orders of magnitude faster than traditional SFI techniques. Our method partitions the design into various hierarchical levels and systematically performs incremental fault injections to generate the AVF numbers. The presented method has been applied on an Intel microprocessor, where experimental results corroborate its ability to achieve great speed-up while maintaining perfect accuracy in calculating AVF.**

I. Introduction

The increasing threat of soft errors in nanometer technologies has resulted in a plethora of design solutions for protecting latches from Single Event Upsets (SEUs) [1], [2], as well as combinational logic from Single Event Transients (SETs) [3], [4]. Despite the demonstrated effectiveness of these solutions, applying them blindly across an entire design incurs prohibitive cost. As a result, various methods for assessing the susceptibility of individual latches or logic gates have also been proposed [5], [6], in order to support partial hardening approaches [7], [8], [9], [10], [11]. Susceptibility evaluation and the corresponding ranking of latches or logic gates typically takes into account a number of factors, including electrical device characteristics, timing issues, as well the actual logic function implemented. These factors reflect the circuit-level and gate-level reasons that may prevent an SEU or an SET from causing a soft error in a circuit. However, they are unable to capture error masking causes at higher levels and, therefore, they prove rather insufficient when applied to modern microprocessors.

Modern microprocessors exhibit a high degree of architectural-level and application-level masking, resulting in many errors being suppressed or having a low probability of affecting the workloads that are typically executed. Indeed, the multitude of functional units and stages in the deeply-pipelined superscalar microprocessors, along with advanced architectural features such as dynamic scheduling and speculative execution, imply that rather complex conditions need to be satisfied in order for an error to affect the architectural state of the microprocessor or the outcome of an application. In an effort to capture these additional masking factors, vulnerability analysis methods have been developed specifically for microprocessors [12], [13], [14], [15], [16]. These methods typically employ statistical fault injection (SFI) and simulation of actual workload using an architectural performance model, a Register Transfer (RT-) or a gate-level model of the microprocessor and aim to assess the probability that a transient error in a state element will affect workload execution, commonly known as the Architectural Vulnerability Factor (AVF). As we discuss in the next section, however, the use of performance models limits the accuracy of the vulnerability analysis, while the use of RT- or gate-level models of an entire microprocessor requires prohibitive simulation time.

In this paper, we propose a new method for calculating AVF in modern microprocessors using Hierarchical Fault Pruning (HFP). The proposed method is faster, yet maintains the accuracy of the traditional SFI approach, which employs the entire microprocessor all at once. HFP leverages the high masking factors of microprocessor modules to quickly prune the fault list as it progresses towards larger partitions where simulation is slower and, thereby, accelerate AVF analysis. The remainder of the paper is structured as follows. Previous architectural vulnerability analysis methods are discussed in Section II. The proposed method is presented in Section III. The experimental setup employed to perform a comparative evaluation of the proposed method is introduced in Section IV and the results are discussed in Section V.

II. Existing AVF analysis methods

The notion of Architectural Vulnerability Factor has been extensively used in the past to rank state elements based on their criticality to program execution correctness. AVF expresses the probability of a bit-flip resulting in a visible system error. Previously proposed methods for performing AVF analysis employ either performance models [12], [17], [18] or RT-Level models [13], [14], [15] of a microprocessor. As we explain below, the former enable fast AVF estimation but suffer in terms of accuracy, while the latter offer far more accurate results but require prohibitive simulation times.

The performance model-based method described in [12] introduces the concept of Architecturally Correct Execution (ACE) and defines ACE bits as those that can cause corruption in the final output of the program. In contrast, un-ACE bits are those which under no conditions may produce a discrepancy in the final program outcome (i.e. branch predictor bits). ACE analysis is performed and evaluated on an IA-64 performance simulator, using which the authors can generate deterministic AVF estimates for the ACE bits and rank the corresponding state elements based on their criticality. For this purpose, workload is simulated to completion and the impact of faults in these state elements is analyzed in a single simulation pass, which is performed rapidly. Furthermore, another major benefit of ACE analysis is that it can be performed early in the design cycle. The major drawback of ACE analysis and AVF estimation using the architectural performance model, however, is the lack of detail about the actual hardware structures of the microprocessor. Therefore, the analysis is only performed for the modeled components, which in the case of [12] includes only components that affect the performance of a microprocessor. This results in significant loss of accuracy in AVF estimation. Furthermore, extensive manual effort is required to classify a bit as ACE or un-ACE.

The methods described in [13], [14], [15] resolve the AVF estimation accuracy problem by performing SFI in the RT-Level model, which reflects the actual hardware structures of the microprocessor. This accuracy, however, comes at a cost: RT-Level simulations are far slower than performance models and fault simulation tools are not readily available at this level. Furthermore, a rigorous transient fault injection campaign requires excessive simulation times in order to provide statistically significant results. In [13], the authors provide a qualitative comparison of the AVF estimation accuracy of their extensive RT-Level simulations to the ACE analysis presented in [12]. Their findings conclude that ACE analysis overestimates soft error vulnerability by about 3.5x and that this discrepancy stems from the model's lack of hardware detail and the single-pass simulation methodology. In [18], the authors of [13] replied that added detail to the ACE analysis can lead to tighter AVF bounds. Still, the limited details available at the performance model remain the main contributor to AVF overestimation.

III. Hierarchical Fault Pruning Methodology

In this paper we propose a methodology that employs statistical fault injection in a hierarchical manner in order to perform AVF analysis. In this context, by "hierarchical" we imply that fault simulation is incrementally performed in gradually expanding partitions of the design. Assume, for example, that we are interested in computing the AVF of the instruction scheduler module of the Aplha 21264 processor, shown in Figure 1. In this case, a possible hierarchy could involve the instruction scheduler by itself in Level 1, the out-of-order execution cluster, which includes the instruction scheduler, in Level 2, and the entire microprocessor in Level 3. As can be observed, successive levels always include the preceding ones, therefore fault simulation becomes increasingly more expensive. At the same time, the size of the fault list that needs to be simulated is drastically reduced due to masking, as we proceed from each level to the next. Indeed, modern microprocessors exhibit high masking factors [19], [20], reported to be up to 98% (AVF=2%) [19]. Based on these two observations, HFP aims to accelerate AVF computation by reducing the simulation effort without sacrificing accuracy.

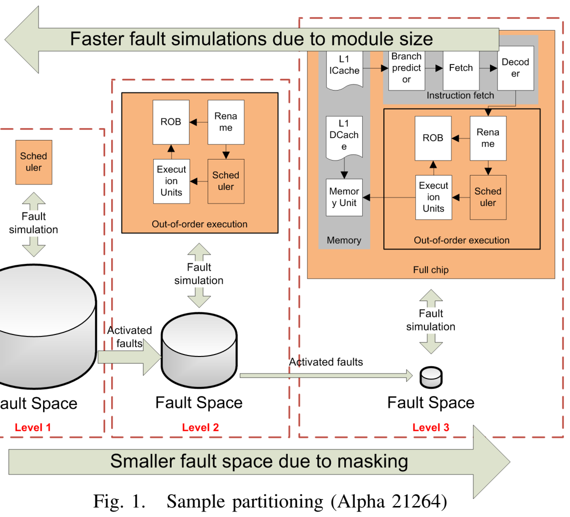

Fig. 1. Sample partitioning (Alpha 21264)

The HFP method is presented in detail in Algorithm 1. First, we partition the design into the various hierarchy levels and create the list of latches to be injected. Then, we generate a fault list with random transients for each latch in the first level. After initializing the algorithm, we fault simulate all faults in this fault list. At the end of this simulation, any fault which causes some discrepancy at the primary outputs (POs) of the first level is stored and set to be injected again at the next level. We define the fraction of the faults that appear at the POs of the partition as Module Vulnerability Factor (MVF), to avoid confusion with AVF which is defined based on faults propagating to the POs of the entire design. When all faults are simulated at the first level, all stored faults become the input fault list of the second level and the same process is applied. These faults are simulated again, since the next partition contains the previous one, but now the design is much larger and more faults will be masked and not appear at the POs of this new level. This process repeats for each of the defined levels. At the end of the algorithm, the AVF of each latch is calculated as the product of the MVFs of this latch across all the different levels.

For example, in the 3-level hierarchy shown in Figure 1, let us assume that the MVF of a latch in the scheduler (i.e., MVF_1) is 25% at the first level, i.e., 1 out of 4 transients in this latch makes it to the POs of the scheduler. Let us also assume that out of these faults, 25% make it to the POs of the out-of-order cluster (e.g., MVF_2=25%) and, out of those, 25% leave the full-chip model and are stored in memory (i.e., MVF_3=25%). Evidently, the probability that a fault will reach the POs of the second level (out-of-order cluster) is MVF_{1-2} = MVF_1*MVF_2 = 25%*25%=6.25%, while the probability that it will reach the POs of the third level (entire design) is AVF = MVF_{1-3} = MVF_1*MVF_2*MVF_3= 25%*25%*25% = 1.5%, consistent with what is reported in [19]. As a result, assuming an initial fault list size of 100K faults in the scheduler, only 25K faults will be simulated at level 2, and only 6.25K faults will be simulated at the entire design level. In contrast, without

Algorithm 1: Hierarchical Fault Pruning algorithm

```
1  Assume $T$ is the list of latches to be injected;
2  Assume $n$ is the number of hierarchical levels;
3  Initialize $Level = 1$;
4  foreach $Latch$ in $T$ do
5      Generate random fault list $F(Level, Latch)$ for $Level = 1$;
6  end
7  for $Level = 1$ to $n$ do
8      foreach $Latch$ in $T$ do
9          Initialize activated fault list $A(Level, Latch) = \emptyset$;
10         foreach $Fault$ in $F(Level, Latch)$ do
11             Fault simulate $Fault$;
12             if $Fault$ propagates to partition's POs then
13                 Add $Fault$ to $A(Level, Latch)$;
14             end
15         end
16         $MVF(Level, Latch) = |A(Level, Latch)|/|F(Level, Latch)|$;
17         $F(Level + 1, Latch) = A(Level, Latch)$;
18     end
19 end
20 foreach $Latch$ in $T$ do
21     $AVF(Latch) = MVF(1, Latch) * MVF(2, Latch) * ... * MVF(n, Latch)$
22 end
```

the proposed hierarchical approach, all 100K faults would have to be simulated at the entire design level.

Defining the design hierarchy is a key part of the process where designer expertise is important. A small number of large partitions require fewer iterations of the HFP algorithm, yet each iteration takes longer to complete. A larger number of smaller partitions results in faster iterations of the HFP algorithm, yet many faults are repeatedly injected at successive levels. A balanced approach used herein, which serves as a good starting point, is to define a hierarchy consisting of (i) sub-block level, (ii) layout block level, and (iii) full-chip level.

IV. Experimental Setup

In order to evaluate the effectiveness of HFP, we performed extensive experiments on a contemporary Intel microprocessor implementing the P6 architecture[1]. Figure 2 shows the basic P6 architecture [21]. Instructions flow from the instruction cache to the decoders, where they enter the out-of-order cluster and are stored in the Reservation Station (RS). The Reorder Buffer (ROB) guarantees in-order retirement after out-of-order instruction execution. The Memory Reorder Buffer (MOB) interacts with the data cache in order to fetch the required information.

Our study focuses on the control units of the out-of-order cluster, namely the *Reservation Station* and the *Reorder Buffer*.

[1]A non-disclosure agreement between Yale University and Intel Corp. prevents us from providing further information about the actual microprocessor.

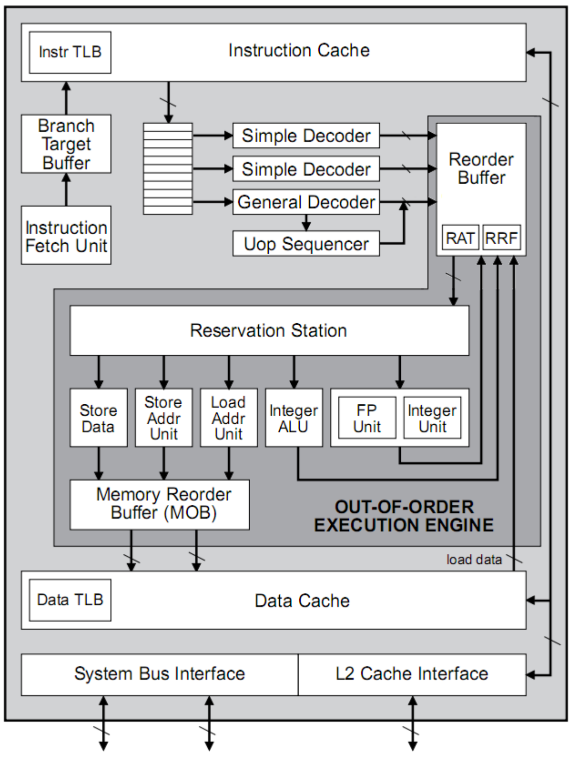

Fig. 2. P6 sample architecture [21]

Following the guidelines provided in Section III, our hierarchy includes 3 levels. Level 1 corresponds to the module (i.e., either the RS or the ROB). Level 2 corresponds to the partition (i.e., the out-of-order execution engine, shown as the dark gray area in Figure 2). Level 3 corresponds to the entire chip.

The flow of our simulation experiment is shown in Figure 3. Before starting the HFP algorithm, certain initialization steps are needed so that workload may be executed:

- **Workload execution:** A variety of workload should be selected for effective AVF analysis. A typical workload requires several million cycles to be completed, rendering full workload execution at the gate-level infeasible. Thus, after compiling the workload for the specific x86 architecture, we use pinLIT [22] to extract workloads in the object file format.
- **Vector generation:** The object file has to be simulated in order to generate appropriate vectors for use during fault simulation. A logic simulator is used to simulate the workload and Value Change Dumps (VCDs) are extracted at the input boundary of each level.
- **Fault list generation:** In order to apply SFI, a set of fault locations is needed, on which transient error injections are performed in order to calculate the AVF. Since our focus is the AVF of sequential elements, the design is parsed and the fault locations (latches) are enumerated. Given this latch list, transient errors are randomly generated and injected uniformly over time, ensuring that the same number of errors is injected in each latch.

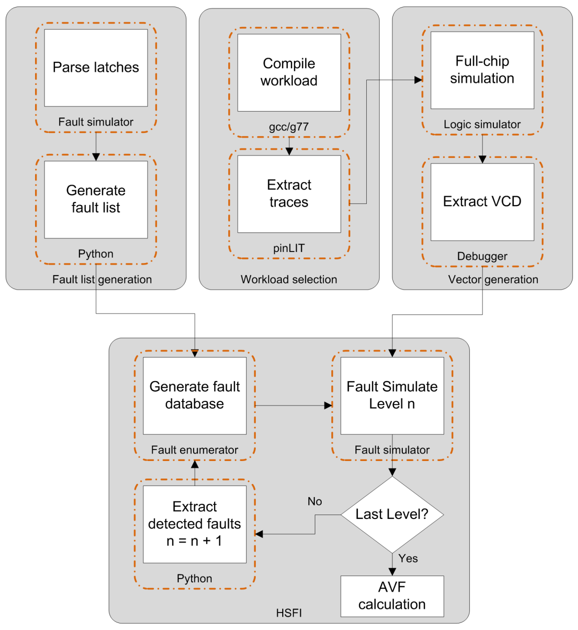

Fig. 3. Experimental flow

An in-house fault enumerator is used to convert the fault list to the internal fault input format. This database, along with the extracted VCDs, constitute the inputs to the in-house fault simulator. When the simulation is completed, the faults are analyzed and a new fault list is generated. This new list includes the faults that propagated to the primary outputs of the current level and is, once again, converted to the internal format in order to start simulation at the next level. When the top level is reached, the final fault list is analyzed and AVF numbers are calculated.

The workload used in our experiments comprises various SPEC benchmarks, namely `astar` (path-finding algorithms), `bzip` (compression), `h264` (video compression), `lucas` (primality checking) and `mcf` (combinatorial optimization). The benchmarks represent different workload types, which is needed for accurate AVF calculation. Workloads were executed to completion using minimal inputs. All experiments were performed within Intel's environment, on machines with similar capabilities. Moreover, the times reported in Section V are averages of several simulations to ensure fair comparison in terms of required resources.

V. Results

In this section, we demonstrate the AVF calculation acceleration obtained by the proposed HFP method over the Traditional SFI (TSFI) approach. Furthermore, we investigate possible options for further speed-up at the expense of some minor accuracy loss.

A. AVF calculation speed-up

Table I compares the simulation time of TSFI to that of HFP for our fault-injection campaign of 175K faults. The reported

TABLE I
Simulation time (in seconds) comparison between HFP and TSFI for 175K faults

		astar	bzip2	h264	lucas	mcf
RS	HFP$_1$	4,119	4,066	4,130	3,350	2,687
	HFP$_{1-2}$	9,984	8,245	7,837	5,365	9,197
	HFP$_{1-3}$	17,075	18,154	18,745	31,704	23,873
	TSFI	263,802	181,417	410,383	372,545	121,792
	Speed-up	**15x**	**10x**	**21x**	**12x**	**5x**
ROB	HFP$_1$	6,628	7,035	6,702	7,341	6,195
	HFP$_{1-2}$	7,883	8,533	7,983	8,708	7,549
	HFP$_{1-3}$	16,869	25,313	25,564	26,433	29,564
	TSFI	314,856	226,754	566,338	458,721	352,610
	Speed-up	**18x**	**9x**	**22x**	**17x**	**12x**

times are averages among 20 fault simulation runs on a single machine. The simulation time needed for each level in the HFP method is also shown in the table. Specifically, HFP$_1$ refers to the simulation time needed to obtain results at the boundary of the first level (i.e., the RS or the ROB module), HFP$_{1-2}$ refers to the cumulative simulation time to obtain results at the boundary of the second level (i.e., the out-of-order execution engine), and HFP$_{1-3}$ refers to the cumulative simulation time to obtain results at the boundary of the full chip. Evidently, HFP$_{1-3}$ $>$ HFP $_{1-2}$ $>$ HFP$_1$.

In order to calculate AVF with equal accuracy as with TSFI, the results of HFP$_{1-3}$ are needed, hence, the fair comparison is between the simulation time of these two approaches. The corresponding entries in Table I show that HFP outperforms TSFI by several orders of magnitude. For example, calculating AVF for the latches of RS while running the `astar` workload on 175K faults via TSFI requires 3 days (263,802 seconds), while the same results are obtained via HFP within only 5 hours (17,075 seconds), corresponding to a 15x speed-up. On average, across our simulations, HFP accelerated AVF calculation by 12.6x for RS and 15.6x for ROB. The key implication of the obtained speed-up is that we can now use HFP to generate statistically significant AVF numbers (i.e., 5 benchmarks, 175K faults per benchmark) for a module within a day (on a single machine), as opposed to the two weeks required by TSFI.

We also point out that the speed-up obtained by HFP is larger as the initial fault list size increases. To demonstrate this point, Figure 4 shows the impact of the initial fault list size on the HFP and TSFI simulation time. More faults per simulation pass require more processing and more memory, thus greatly increasing the simulation time required for TSFI. HFP, on the other hand, is almost unaffected because the first level is extremely small compared to the full chip, allowing many faults to be injected during a single pass. And given the large amount of error masking, as explained in Section III, only a very small percentage of faults ends up being simulated at the entire chip level. The key implication of Figure 4 is that the HFP speed-up would be much higher than what is reported in Table I, if a larger initial fault list was utilized (Table I compares the two methods for a fault list of 175K faults).

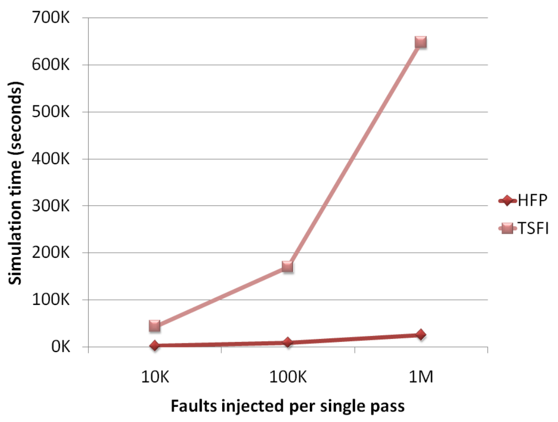

Fig. 4. Simulation time (in seconds) comparison between TSFI and HFP for various numbers of injected faults

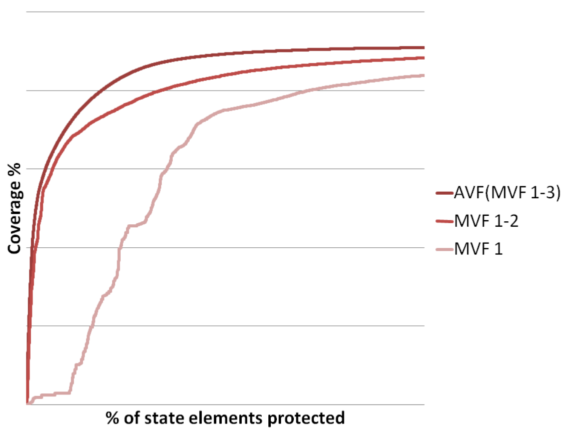

Fig. 5. Correlation between AVF and MVF calculated at different levels

B. AVF to MVF correlation

Further AVF calculation speed-up can be obtained, at the cost of some minor accuracy loss, by leveraging an interesting observation regarding the correlation of AVF to MVF. Evidently, since a fault may be masked at a subsequent level in an n-level hierarchy, $\text{MVF}_1 \geq \text{MVF}_{1-2} \geq \ldots \geq \text{MVF}_{1-n} = \text{AVF}$ for every latch. However, as we approach the full-chip level, masking becomes less likely and depends mostly on the application behavior (application masking) rather than the properties of the injected latch; therefore, it is likely that the MVF will become increasingly correlated to the AVF. For example, the correlation coefficient between MVF_{1-3} = AVF and MVF_{1-2} in our experiment is 97%. This does not come as a surprise, due to the nature of the partitioning chosen. A fault escaping Level 2 guarantees incorrect execution of an instruction, since Level 2 contains all the execution functionality.

The implication of this observation is that we can speed-up AVF calculation by using the –much faster to compute– MVF of preceding levels instead. For example, suppose that AVF is used to rank the state elements of a module in order to select and protect the most vulnerable ones. The top line (MVF_{1-3}=AVF) in Figure 5 reports the coverage (Y-axis) that would be achieved by protecting the corresponding percentage of the state elements shown in the X-axis[2]. This is the point of reference, reflecting an optimal state element ranking. Suppose now that the same state elements are ranked based on MVF_{1-2} instead. In this case, the coverage achieved by protecting the corresponding percentage shown in the X-axis is given by the middle line on the plot of Figure 5. Evidently, the accuracy lost when selecting based on MVF_{1-2} instead of AVF is fairly small, while the computational gains are very large. For example, the 15x speed-up reported in Table 4 for `astar` becomes 27x if the simulation stops at HFP_{1-2}.

We note, however, that the correlation between MVF_{1-3}=AVF and MVF_1 is only 61%, implying that a fault escaping the boundary of a small module such as the RS or the ROB is not a good indication of whether it will actually affect execution. As a result, ranking and protecting state elements based on MVF_1, as shown on the bottom line of Figure 5, yields very sub-optimal results.

[2]Values on the axis are omitted and results are reported only for a subset of the latches due to the confidential nature of the actual data.

C. Fault injection window

AVF calculation may also be accelerated, again at the cost of minor accuracy loss, by limiting the number of simulation cycles after a fault is injected. A similar approach is taken in the RT-Level AVF calculation method described in [20], where faults are not simulated until the end of the workload due to limited resources. Instead, a fault is injected during a limited number of cycles (up to 10,000). Indeed, fault simulation of RT- or Gate-Level models is feasible only for a few thousand cycles.

In the results presented herein, complete workloads were executed exclusively at the Gate-Level, yet with minimal inputs in order to keep the clock cycles below 1 million. Figure 6 shows how many cycles, after injection of a fault, the fault appears at the primary outputs of the corresponding partition. As can be observed, over 96% of the faults appear within 1,000 cycles after injection, and almost all faults (i.e, 99.96%) reach the outputs within 10,000 cycles. Thus, we can safely stop the simulation 10,000 cycles after fault injection and still compute very accurate AVF numbers with the data available at that point in time. In our experiments, this approach would result in an additional 10x AVF calculation speed-up, since we would fault simulate 10,000 cycles instead of 100,000+ typically used as a safety margin.

D. Fault list size

Finally, additional AVF calculation speed-up may be obtained by moderating the fault list size, possibly at the expense of minor accuracy loss. In our experiments, we used a sample size of 2,500 injections per latch (5 benchmarks, 500 injections per latch) to compute the AVF numbers, which we use as our baseline. In this section, we examine the impact of reducing this sample size on the accuracy of the computed AVF.

Figure 7 shows how the AVF computed using smaller fault list sizes (i.e., 1,000, 500, 250, 125, 60 and 30 injections per

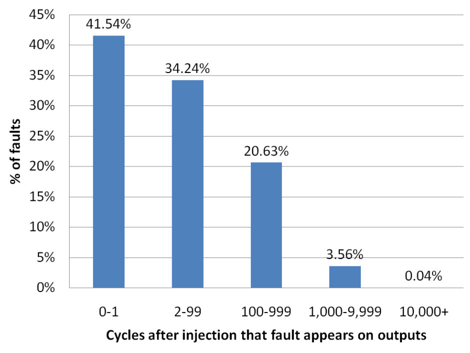

Fig. 6. Statistics driving fault injection window selection

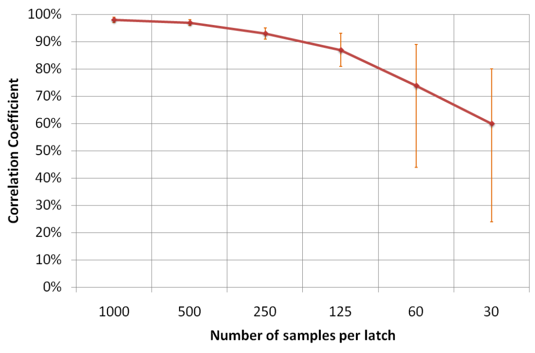

Fig. 7. Correlation between AVFs produced by different number of samples

latch) correlates to the baseline AVF. For each sample size, the average over 10 different fault lists is reported (along with the deviation). For example, a sample size of 60 injections per latch correlates, on average, 78% with the optimal list, yet this number may be as low as 40%. The key take-away point from this graph is that accurate AVF estimations (i.e., with both minimum and average correlation coefficient >90%) can be obtained using a fault list size of as few as 250 samples. This leads to an additional 10x speed-up, since 250 instead of 2,500 injections are performed.

VI. Conclusion

The Hierarchical Fault Pruning method proposed herein exploits the high masking factors of modern microprocessors in order to accelerate AVF analysis. By hierarchically partitioning the design and incrementally fault simulating each level only for the subset of faults that evade masking in previous levels, the number of simulation cycles required is drastically reduced. Experimental results on a modern microprocessor demonstrate that, on average, HFP speeds-up AVF calculation by 14.1x as compared to the traditional SFI approach, without sacrificing any accuracy. Additional speed-up is obtained, at the cost of minor accuracy loss, by leveraging the correlation between intermediate and final results of HFP, by limiting the fault injection window and by moderating the fault list size.

Acknowledgement

This work was supported by a generous gift from Intel Corp. The first author performed part of this research during a summer internship with Intel Corp. in Santa Clara, CA.

References

[1] L.R. Rockett Jr, "An SEU-hardened CMOS data latch design," *IEEE Transactions on Nuclear Science*, vol. 35, no. 6, pp. 1682–1687, 1988.
[2] M. Zhang, S. Mitra, T.M. Mak, N. Seifert, N.J. Wang, Q. Shi, K.S. Kim, N.R. Shanbhag, and S.J. Patel, "Sequential element design with built-in soft error resilience," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 12, pp. 1368–1378, 2006.
[3] Q. Zhou and K. Mohanram, "Transistor sizing for radiation hardening," in *International Reliability Physics Symposium*, 2004, pp. 310–315.
[4] N. Miskov-Zivanov and D. Marculescu, "MARS-C: modeling and reduction of soft errors in combinational circuits," in *Design Automation Conference*, 2006, pp. 767–772.
[5] C. Zhao, X. Bai, and S. Dey, "A scalable soft spot analysis methodology for compound noise effects in nano-meter circuits," in *Design Automation Conference*, 2004, pp. 894–899.
[6] M. Zhang and N.R. Shanbhag, "Soft-error-rate-analysis (SERA) methodology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2140–2155, 2006.
[7] R. Garg, N. Jayakumar, S.P. Khatri, and G. Choi, "A design approach for radiation-hard digital electronics," in *Design Automation Conference*, 2006, pp. 773–778.
[8] Q. Zhou and K. Mohanram, "Gate sizing to radiation harden combinational logic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 1, pp. 155–166, 2006.
[9] S. Almukhaizim, Y. Makris, Y. Yang, and A. Veneris, "Seamless Intergration of SER in Rewiring-based Design Space Exploration," in *International Test Conference*, 2006, vol. 2, pp. 29.3.1–29.3.9.
[10] C.G. Zoellin, H.J. Wunderlich, I. Polian, and B. Becker, "Selective Hardening in Early Design Steps," in *European Test Symposium*, 2008, pp. 185–190.
[11] S. Krishnaswamy, S.M. Plaza, I.L. Markov, and J.P. Hayes, "Signature-based SER analysis and Design of Logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 74–86, 2009.
[12] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *International Symposium on Microarchitecture*, 2003, pp. 29–40.
[13] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," *SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 460–469, 2007.
[14] E.W Czeck and D.P Siewiorek, "Effects of transient gate-level faults on program behavior," in *International Symposium on Fault-Tolerant Computing*, 1990, pp. 236–243.
[15] K. Seongwoo and A.K. Somani, "Soft error sensitivity characterization for microprocessor dependability enhancement strategy," in *International Conference on Dependable Systems and Networks*, 2002, pp. 416–425.
[16] M. Maniatakos and Y. Makris, "Workload-driven selective hardening of control state elements in modern microprocessors," in *VLSI Test Symposium*, 2010, pp. 159–164.
[17] S.S. Mukherjee, J. Emer, and S.K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 243–247.
[18] A. Biswas, P. Racunas, J. Emer, and S. Mukherjee, "Computing accurate AVFs using ACE analysis on performance models: A rebuttal," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 21–24, 2008.
[19] G.P. Saggese, N.J. Wang, Z.T. Kalbarczyk, S.J. Patel, and R.K. Iyer, "An experimental study of soft errors in microprocessors," *IEEE Micro*, pp. 30–39, 2005.
[20] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks*, 2004, pp. 61–70.
[21] L. Gwennap, "Intels P6 uses decoupled superscalar design," *Microprocessor Report*, vol. 9, no. 2, pp. 9–15, 1995.
[22] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder, "Automatic logging of operating system effects to guide application-level architecture simulation," in *International Conference on Measurement and Modeling of Computer Systems*, 2006, pp. 227–238.