Characterization of Logical Masking and Error Propagation in Combinational Circuits and Effects on System Vulnerability

Nishant George and John Lach
Charles L. Brown Department of Electrical and Computer Engineering
University of Virginia, Charlottesville VA
[*niche, jlach*]@*virginia.edu*

***Abstract*—Among the masking phenomena that render immunity to combinational logic circuits from soft errors, logical masking is the hardest to model and characterize. This is mainly attributed to the fact that the algorithmic complexity of analyzing a combinational circuit for such masking is quite high, even for modestly sized circuits. In this paper, we present a hierarchical statistical approach to characterize the vulnerability of combinational circuits given logical masking and error propagation. By conducting detailed analyses and fault simulations for circuits at lower levels, initial assumptions of 100% vulnerability with single random output errors are refined. Fault simulations performed on the ISCAS85 benchmark circuits and Kogge-Stone adders of various widths demonstrate the varied nature of vulnerability for different circuits. The analysis performed at the circuit level for a 32-bit Kogge-Stone adder is applied to a microarchitecture simulation to examine impact on system-level vulnerability.**

***Keywords*-logical masking, soft error vulnerability, statistical fault injection**

I. Introduction

Combinational circuits have classically been considered more immune to high-energy particle strike induced soft errors than memory structures (SRAM, DRAM). While true for several past generations of semiconductor devices, this is no longer the case. As was predicted by Shivakumar et al. [1], aggressive technology scaling progressively reduces this immunity, effectively rendering combinational circuits as vulnerable as memory arrays in devices built using modern technologies. Combinational logic blocks are hence significant contributors to the unreliability of a semiconductor chip, necessitating the development of methods to accurately quantify their contribution to system-level soft error vulnerability.

When considering the total area of a general-purpose microprocessor chip, a large fraction is taken up by SRAM-based cache structures. Even though this makes combinational blocks relatively small targets for transient faults and soft errors, this might not be the case for application specific integrated circuits (ASICs), embedded processors or safety-critical applications, where cache structures are small or non-existent to ensure more deterministic operation. In addition, combinational blocks may contain a larger percentage of bits required for architecturally correct execution (ACE bits) than SRAM arrays. Therefore, the contribution of system vulnerability arising from faults in combinational blocks can be significant for many applications, and this forms an important component when characterizing total chip-vulnerability.

Unlike in SRAM where all soft errors at least affect system state (some of which may be unACE), soft errors in combinational circuits may or may not propagate to a storage element. This is due to a combination of three masking effects, namely electrical, latching-window and logical masking. While electrical and latching-window masking are fairly well-understood and can easily be quantified, logical masking is not. The former are explained by physical effects such as attenuation in a logic chain and temporal shielding respectively, whereas the latter is based on logical sensitivity of the circuit outputs to affected internal nodes for a given input. This makes modeling and characterizing logical masking a hard problem because analyzing so many input and fault combinations can be intractable, even for moderately sized circuits (especially if the circuit is treated as a flat design). In addition, soft errors at single internal nodes in a combinational circuit may propagate to multiple outputs in particular patterns, further complicating vulnerability assessment.

In this paper, we present a hierarchical approach to solving this problem. Without implementation details of underlying combinational blocks, block level vulnerability to soft errors must be assumed to be 100%. That is, if there is an error from a fault in a combinational block, it reaches the output of that block and propagates to higher levels. This is obviously a conservative estimate, as not all errors at a low level matter at higher levels. The fraction of errors that do matter is, after all, defined as vulnerability. In addition, this 100% vulnerability is typically assumed to arise from errors manifesting as *random*, *single* bit errors, which could also be inaccurate due to error propagation in combinational logic.

With implementation details of underlying combinational blocks, the accuracy of block vulnerability characterization can be improved using statistical fault injection. To enable this analysis, we developed a compiler capable of generating a fault injection simulator (FIsim) from the circuit

978-1-4244-9233-6/11/$26.00 ©2011 IEEE

netlist, and many execution iterations provide a statistically significant vulnerability metric for the circuit under test. The results of such campaigns on a set of ISCAS85 benchmarks reveal the widely varying vulnerability of combinational circuits to soft errors, further reinforcing the notion that logical masking must be carefully characterized on a per circuit basis. In addition, FIsim can be used to develop fault-to-error models that describe how faults and errors at a low level propagate through combinational logic to higher levels in a design hierarchy. We demonstrate how this error propagation model can be applied to a system-level model in order to obtain an estimate for the contribution of a particular combinational block to system-level vulnerability.

For fairly large circuits, such as a 32-bit Kogge-Stone adder, the total number of possible error combinations at its output can run into the billions. However, we show that this is not the case, and the error patterns that this circuit produces at its output for random sources of error within it comprise a very small fraction of all the possible patterns. This observation significantly simplifies the development of a fault-to-error model for applications at a higher level (e.g., how faults would be injected at combinational block outputs during a fault injection campaign to determine system-level vulnerability).

Through the use of extensive statistical fault injection at two levels in a design hierarchy, this paper makes the following contributions.

- A methodology to estimate vulnerability of combinational blocks arising from logical masking and error propagation. The FIsim compiler can easily be integrated into design flow to obtain early estimates of vulnerability.
- Results assessing the vulnerability of a number of combinational blocks, revealing the circuit-specific nature of vulnerability characteristics. This assessment includes the percentage of internal faults that result in combinational block output errors and the patterns of those errors.
- A methodology to use such combinational block vulnerability assessments to generate fault-to-error models that describe how faults at a low level propagate to higher levels. This simplifies the definition of a fault/error space for injections at the system level to better assess system vulnerability.

A summary of the remaining sections of the paper is as follows. Section II provides a brief background in the general area of transient faults and soft errors in logic circuits and the general definition of vulnerability. Section III presents a brief review of published literature related and relevant to that presented in this paper. FIsim compiler, the tool developed as part of this work, is elaborated in Section IV, as are the vulnerability assessments provided by FIsim for a number of combinational circuits. Section V defines a mathematical model to characterize vulnerability in a hierarchical manner. It also presents how results from FIsim are plugged into the model. Application of the fault-to-error model developed in Sections IV and V into a high-level system model to evaluate the impact of a combinational block on the system-level vulnerability is described in Section VI. Section VII provides discussion, and presents conclusions and summarizes this work.

II. Background: Transient Faults in Logic

Historically, transient faults in logic have been considered to be less malicious than transient faults in SRAM arrays. This is mainly because a transient fault causing a disturbance in a chain of logic in a circuit can be considered benign if its effects are not captured at an output sequential element. A number of factors are known to attenuate the effect of transient faults effectively to prevent errors from propagating [2][3][1]. Even though these masking phenomena effectively shielded logic from soft errors for many generations, many are known to reduce with technology scaling, effectively making logic circuits more vulnerable to soft errors. These effects are described below.

A. Electrical Masking

Electrical masking is the property by which an electrical pulse generated as a result of a particle strike within a combinational logic circuit gets attenuated by subsequent gates in the logic chain [2][3]. Shivakumar et al. [1] developed an electrical masking model using critical evaluation of pulse behavior and provided a method to predict the output characteristics of a pulse (e.g. amplitude) as it passes through a gate based on calibration with SPICE simulations. Their model serves to determine whether the magnitude of the generated pulse is strong enough to propagate an error. We do not focus further on this topic as it forms a self-contained derating factor for the raw error rate and can easily be incorporated into our model.

B. Latching-Window Masking

Of all the particle strikes that give rise to electrical pulses strong enough and capable of producing errors, only those that get captured at output latches are considered malicious. To propagate an error, the glitch must appear around the time that an input gets clocked into an output sequential element. For most sequential elements, the time during which an input needs to be steady in order for it to be captured into the output at the latching edge of the clock corresponds to setup time plus hold time. The fraction of this time relative to the total clock period gives a useful measure of temporal vulnerability to single event transients that the output sequential element experiences.

If the combinational logic chain that feeds into an output latch is longer, it would require a longer clock period and consequently presents a shorter relative vulnerable window.

However, the immunity of logic circuits to transient faults from latching window masking is progressively reducing with technology scaling as pipeline stages shorten and operating frequencies increase [1]. The total amount of time that a circuit is vulnerable (irrespective of the clock period and the length of the combinational logic chain that feeds it) is largely a function of the design of the sequential element, specifically its setup and hold times. We do not discuss this further because it is not the primary focus of this work. Nonetheless, it is an important component of vulnerability characterization of logic circuits and its effects can be easily factored into our framework.

C. Logical Masking

The third effect that renders immunity to combinational logic circuits is logical masking. It refers to the situation where an erroneous value held by a circuit node is logically irrelevant for certain input combinations of the circuit [2][3][1]. For example, an incorrect logic value of 0 at one input of a two-input OR gate does not produce an error at its output if the other input is at logic level 1. Characterization of logical masking is often left out of vulnerability estimation methods because of the inherent algorithmic complexity of evaluating even modestly sized combinational circuits for their large number of input and fault combinations (unlike the trivial complexity of electrical and latching-window masking characterization).

D. Soft Error Vulnerability

We define *vulnerability* as the fraction of faults that leads to errors at the output of a system. Soft error vulnerability can be defined and evaluated at different levels in the design hierarchy. For example the vulnerability of a standard cell (e.g. 2-input OR) may be referred to as *cell* vulnerability. Similarly, at a higher level of the hierarchy, say at the microarchitecture level (e.g. adder block, pipeline stage), this may be referred to as *block* vulnerability, and at the highest level, *architectural* vulnerability. In all cases, the definition of vulnerability remains the same, and it is the probability that a fault occurring in that level manifests as an error at *its* output. At the standard cell level, an error would be an incorrect signal level of the gate that the cell represents. Similarly, an error at the output of a microarchitecture structure arising from a fault in that structure can be on one or more bits that comprise its output.

These definitions are emphasized here because of the following reasons. Evaluating the enormous number of input and fault combinations when attempting to estimate system-level vulnerability of a structurally flat design will be prohibitively expensive. Credibly abstracting the propagation of faults between different levels in a design hierarchy as fault-to-error models is a tractable way of managing this complexity and is the approach followed in this paper.

III. Related Work: Evaluation of Error Propagation and Logical Masking

Massengill et al. [4] first described a VHDL-based simulation approach to characterizing logical masking for a small circuit and the effect of errors at one circuit node on the high-level system. In this paper, we present a generalized hierarchical methodology to quantify and model the propagation of errors from a combinational circuit to its output and how such errors are fed to higher levels to determine their impact at the system level. The main approach to accomplish this goal is hierarchical fault simulation and fault injection, from the circuit-level to the system-level.

Automatic Test Pattern Generation (ATPG) is a similar approach used to algorithmically determine input vectors that exercise known inherent fault conditions in a circuit with the main goal being to reduce time-on-tester. The goal of the work presented in this paper, on the other hand, is to find the fraction of input cases to a circuit that make the outputs sensitive to random internal faults (i.e. vulnerability). Further, this work models the propagation of errors to higher levels in the design hierarchy and estimates the resulting impact.

Krishnaswamy et al. [5] present a signature-based soft error analysis methodology to quantify the impact of logical masking in a combinational block. A similar approach is presented in [6]. Both works use heuristics to identify vulnerable nodes in a given circuit, which are then selectively strengthened by restructuring in order to increase the effect of logical masking at those nodes. The intent of their approach is to determine at the circuit-level, which nodes are vulnerable in order to redesign and strengthen them. While [5] and [6] assess the vulnerability of a combinational block, they do not provide detailed analyses about how faults at a lower level propagate to higher levels and impact system-level vulnerability, as is demonstrated by the work presented in this paper.

IV. Methodology for Circuit-Level Vulnerability Estimation

This section describes the tool that was developed as part of this work to estimate the inherent fault masking characteristics of combinational circuits. At the heart of the tool chain is a compiler that takes as input the description of a combinational circuit in the form of a netlist and converts it into a functional simulator capable of performing random statistical fault injection. Multiple iterations of the simulator for each circuit produce a statistically significant estimate of the circuit's vulnerability.

A. FIsim Compiler

A tool that is capable of evaluating a vulnerability metric for a circuit can be extremely valuable for directing early design time decisions about inclusion or exclusion of fault tolerance in a design. While circuits at early stages in a

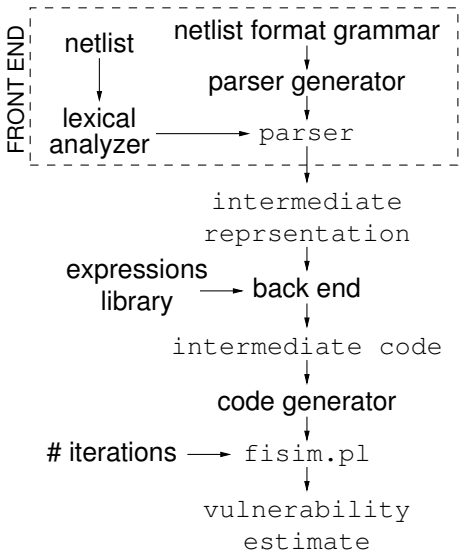


Figure 1. FIsim compiler components and operation

design flow are often represented as netlists, there is no single accepted netlist format (e.g. EDIF, SPICE (Cadence)). In order for such a tool to be useful to a broad community it should have the capability to interpret different netlist formats and be easily extendable to new netlist formats. This tool was therefore designed using principles of complier design so that the only effort required to enable use with a different netlist format would be the development of a new front-end that takes as input the netlist grammar and produces an input-format-independent intermediate representation of the netlist. The rest of the compiler tool chain can be reused.

A schematic of the FIsim compiler is shown in Figure 1. The entire tool chain was written using PERL. The front end comprises the grammar definition of the chosen netlist format and a parser generated using YAPP (yet another PERL parser compiler) from the grammar. The parser together with a lexical analyzer parses the input netlist and produces an intermediate representation. Two front-ends capable of interpreting SPICE and ISCAS85 netlist formats were developed for the results presented in this paper. The only assumption FIsim makes is that the netlist provided as input to it is one that is mapped to a synthesis-ready library of gates. This requirement is complemented by a user input – the expressions library. This is a collection of logical expressions (in C-like format) that correspond to various operations that gates in the library evaluate. Such a library can easily be compiled from user manuals of standard cell libraries.

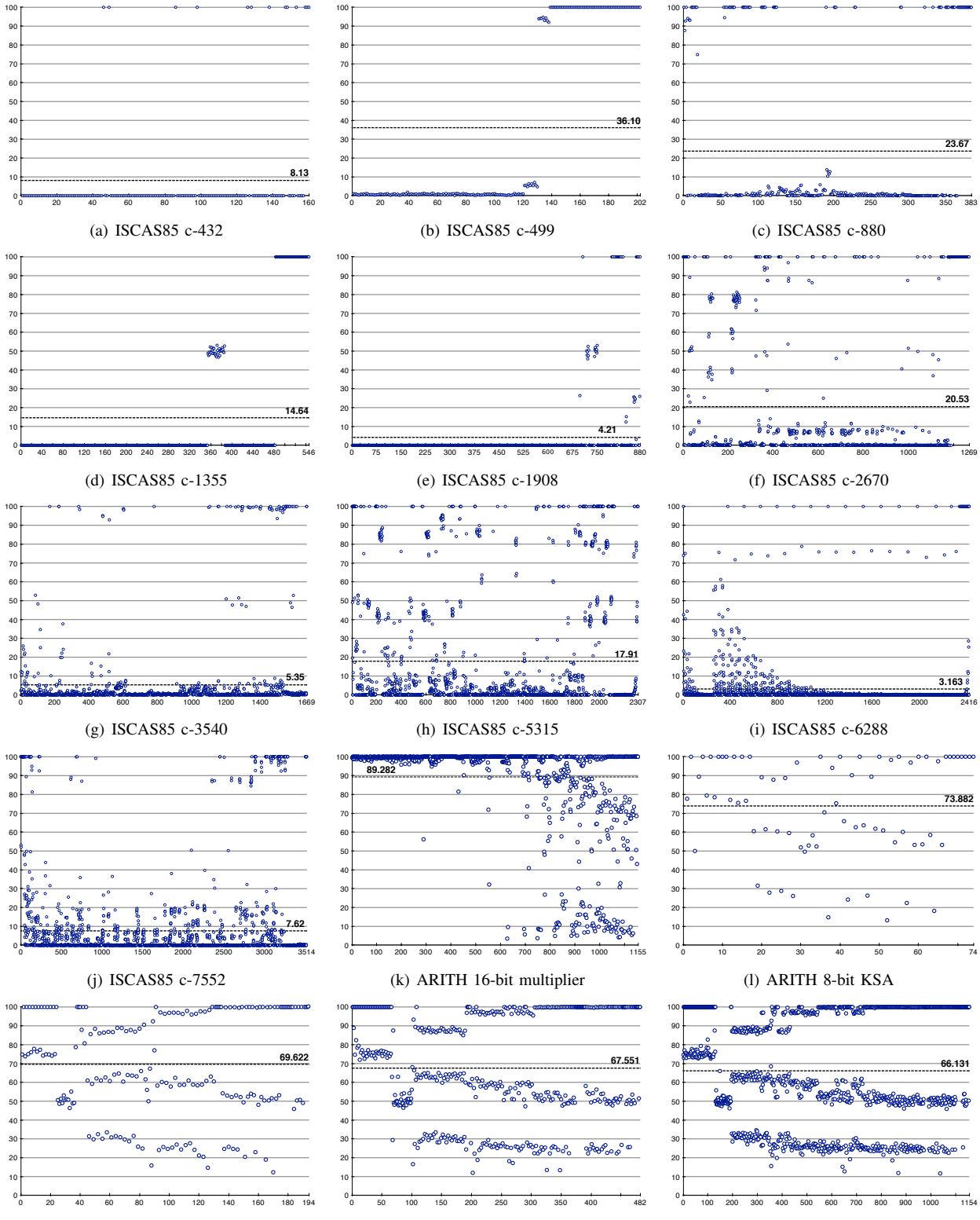
After simple translation of gates in the intermediate representation into logical expressions using rules in the expressions library, the back end reorders the expressions to maintain sequential consistency and to avoid hazards. This representation is referred to as the intermediate code and is a language syntax independent sequential list of logical expressions. The code generator converts the intermediate code into a PERL program (`fisim.pl`) that sets up random inputs to the circuit and evaluates the logical expressions to produce the output of the circuit for the chosen random input combination.

The code generator also inserts saboteur modules (benign by default) at locations corresponding to every internal net that represents an internal gate output. One execution of `fisim.pl` corresponds to one random pattern applied to the circuit inputs and evaluation of the circuit output in the presence of an error at an internal net in the circuit, introduced by activation of a single saboteur module that corresponds to the location where an error is to be injected. This emulates the situation where a fault occurring somewhere within the circuit produces an incorrect logic level at an internal net. Statistical significance may be achieved by running a long campaign that iterates `fisim.pl` a sufficient number of times, because each iteration chooses a different random input sequence and picks a random saboteur to activate a fault in a random location within the circuit. `fisim.pl` may be executed in a second mode where the chosen saboteur is fixed for a campaign. This mode can be used to study the effect of errors on specific nets or regions of nets so that the vulnerability of different regions in the circuit may be characterized.

B. FIsim Results

The results obtained from FIsim simulations for various combinational circuits are shown in Figure 2. Each plot shows all the internal nets in a circuit on the x-axis and corresponding vulnerability on the y-axis (i.e. the probability that there is an error at the output of the combinational block, given that there is a soft error on that net). The nets in each graph shown in Figure 2 do not include input nets as they are assumed to be fed by sequential elements such as latches or flip flops. The experiments presented here emulate faults in combinational logic gates and their output nets. Therefore, the nets shown in Figure 2 include output nets, all of which have 100% vulnerability. The nets are ordered on the x-axis based on their *distance* in terms of number of gate hops from input. Nets that have equal distance from the input are ordered arbitrarily. Therefore, all of the output nets are not necessarily grouped together on the right.

1) ISCAS85 Circuits: The ISCAS85 benchmark circuits [7] comprise ten combinational circuits that include functions such as decode, multiply, ALU operations, ALU control, and error detection and correction. Table I shows the number of internal nets and output nets in each of the circuits as determined by the FIsim compiler. The number of internal nets can be used as a relative measure of circuit size. A front end for the FIsim compiler capable of interpreting the ISCAS85 netlist format was written as a formal grammar



(a) ISCAS85 c-432 (b) ISCAS85 c-499 (c) ISCAS85 c-880

(d) ISCAS85 c-1355 (e) ISCAS85 c-1908 (f) ISCAS85 c-2670

(g) ISCAS85 c-3540 (h) ISCAS85 c-5315 (i) ISCAS85 c-6288

(j) ISCAS85 c-7552 (k) ARITH 16-bit multiplier (l) ARITH 8-bit KSA

(m) ARITH 16-bit KSA (n) ARITH 32-bit KSA (o) ARITH 64-bit KSA

Figure 2. Results from fault injection on various combinational circuits. X-axis for all graphs indicates internal nets of each circuit (see Section IV-B for details). The corresponding vulnerability (i.e. the % of fault injections performed on that net that resulted in errors at the circuit output) is plotted on the y-axis, as is the average circuit vulnerability. All graphs have been rendered using scalable graphics; details may be visualized when enlarged.

Circuit	#Nets	#Outs	Circuit	#Nets	#Outs
c-432	160	7	c-499	202	32
c-880	383	26	c-1355	546	32
c-1908	880	25	c-2607	1269	140
c-3540	1669	22	c-5315	2307	123
c-6288	2416	32	c-7552	3513	108

Table I
ISCAS85 CIRCUITS TESTED

Circuit	Input width (# bits)	# Nets
Kogge-Stone adder	8+8	74
Kogge-Stone adder	16+16	194
Kogge-Stone adder	32+32	482
Kogge-Stone adder	64+64	1154
Multiplier	16×16	1154

Table II
VERILOG-SPICE CIRCUITS TESTED

combined with a lexical analyzer. Using the fault injection simulator generated by the compiler, 1000 fault injections per net were conducted for each circuit. For example, a total of ∼3.5 million fault simulations were run for the circuit c-7552. Results of these experiments are shown in Figures 2(a) – 2(j).

A fault simulation on a particular net is counted vulnerable if the output of the circuit had an error given that there was an error on that net for a random input combination. The height of each point (net) on the graphs corresponds to the percentage of observed vulnerable experiments among the 1000 conducted on that net. The dashed line in each plot shows an aggregate estimate of vulnerability assuming that an error on each net is equally likely. This is obviously not the case, but simply a generalization. This can further be refined based on area of each gate that feeds each net and the inherent vulnerability of the gate. An approach to incorporate this into the analysis process is demonstrated in Section V.

It can be seen that different circuits exhibit vulnerability in very different ways. For example, in Figures 2(d) and 2(e), most of the nets have a vulnerability of 0 and a few nets towards the output of the circuit have a high vulnerability. Closer examination of these circuits revealed that they contained a number of *wide* AND gates (e.g. 8 or 9 inputs). These can be considered *sinks* with respect to error propagation because for most input combinations, the AND gate produces an output of 0 (i.e. if at least one input is 0). This could imply that conducting just 1000 fault experiments on these nets is not sufficient to encounter experiments where input conditions are suitable to propagate errors. So these circuits were reevaluated with 10,000 fault experiments per net. The results, however, largely remained the same except for a few nets, which produced a few (1-2) vulnerable cases per 10,000 experiments conducted. Therefore, those nets that show a vulnerability of 0 do not imply that their vulnerability is exactly 0 (indeed, the net would then serve no purpose), but is very small, precisely, less than one in one or ten thousand.

2) Cadence SPICE Circuits: Verilog descriptions of circuits such as adders of various input widths and a multiplier obtained from the Arithmetic Module Generator (AMG, ARITH) [8] were used to demonstrate utility of the FIsim compiler tool chain in a design flow. These example circuits (downloaded as structural Verilog) were converted to a design implementation using the Cadence RTL compiler by mapping onto gates in a 120nm process standard cell library. The mapped verilog circuits were then imported into Cadence schematic editor and exported into SPICE netlists by associating the circuit with a technology library.

A second front end to the FIsim compiler capable of interpreting SPICE netlists was developed to demonstrate practical application in the design flow. The netlists were provided as inputs to the FIsim compiler to generate fault injection simulators for these circuits. The number of nets identified by the FIsim compiler for the different circuits is shown in Table II. Each circuit's fault simulator was iterated 1000 times for each net in the circuit. The total amount of time required to run this experiment for the largest circuit tested, comprising 1.154M experiments was about 25 hours. Results of this experiment for Kogge-Stone adders of various input widths and a 16-bit multiplier are shown in Figures 2(k) – 2(o).

It is clear that the general position of nets in Kogge-Stone adders of different widths exhibit similar vulnerabilities. This is evident from the fact that Figures 2(l) – 2(o) progressively appear to be more crowded versions of each other. Even though the vulnerability reduces with increasing bit width, it quickly settles.

V. DEVELOPMENT OF A FAULT TO ERROR MODEL

As was defined in Section II-D, the vulnerability of a top-level circuit is the probability that there is an error at its output in the presence of faults that are capable of producing errors within it. To evaluate it, we propose a hierarchical approach, which defines it recursively in terms of the vulnerabilities of its constituent components, thus avoiding the tremendous combinational complexity of a flat design. Even though we demonstrate the model using two levels in the design hierarchy (top and $block$; $cell$ vulnerability is not refined here), it can be extended to an arbitrary number of levels. The following equations define the model.

$$P(E_{top}) = \sum_{blocks} P(E_{top}|E_{block}) \times P(E_{block}) \frac{A_{block}}{A_{top}} \quad (1)$$

$$P(E_{block}) = \sum_{cells} P(E_{block}|E_{cell}) \times P(E_{cell}) \frac{A_{cell}}{A_{block}} \quad (2)$$

$P(E_{top})$ in Equation 1 corresponds to the total system-level vulnerability. The term $P(E_{top}|E_{block})$ stands for the system-level vulnerability arising from errors at the output of a particular block in the system. Estimation of this term using fault injection experiments at the system level is demonstrated in Section VI. To aggregate top-level vulnerabilities (arising from errors in different blocks) into a single system-vulnerability metric, each term needs to be derated using two factors. First, the vulnerability of the block itself $(P(E_{block}))$, as not all errors originating within the block manifest at *its* output, and second, an area derating factor $(\frac{A_{block}}{A_{top}})$, which gives the probability of the specific block being the target of a particle strike, given that the top-level is struck.

Block vulnerability is similarly defined in Equation 2 in terms of its constituent components – standard cells or gates. $P(E_{block}|E_{cell})$ corresponds to the block vulnerability arising from errors at the output of a specific gate in the block. Results of fault injection experiments using FIsim shown in Figure 2 corresponds precisely to this term. Each point in the figures corresponds to the block-vulnerability when errors were injected at a specific cell-output (net). An aggregation rule similar to that of Equation 1 can be used in Equation 2. Block vulnerability estimates (arising from errors in different cells) need to be derated when they are aggregated to estimate total block vulnerability. This derating is captured by the terms $P(E_{cell})$, the cell-vulnerability and $\frac{A_{cell}}{A_{block}}$.

The main advantage of such an approach is that when additional information about lower-level implementation is not available, it can conservatively be assumed vulnerable 100% of the time ($P(E_{lower-level}) = 1$). For example, if there isn't sufficient information to estimate the vulnerability of a standard cell, $P(E_{cell})$ can be assigned 1 until it can be further refined using implementation details.

Similarly, the area derating factor in Equation 2 can conservatively be assumed to be $\frac{1}{\#cells}$ until specific layout details are available. For example, each graph in Figure 2 shows vulnerability of the total combinational block as a numerical average of vulnerabilities of all nets in the block. This estimate makes the following assumptions. First, all nets are equally vulnerable to a fault, i.e. $\frac{A_{cell}}{A_{block}} = \frac{1}{\#cells}$. Further, any fault occurring in a cell propagates an error to its output, i.e. $P(E_{cell}) = 1$. These assumptions are obviously incomplete representations of the actual scenario, but when more detailed information becomes available, the model can further be refined.

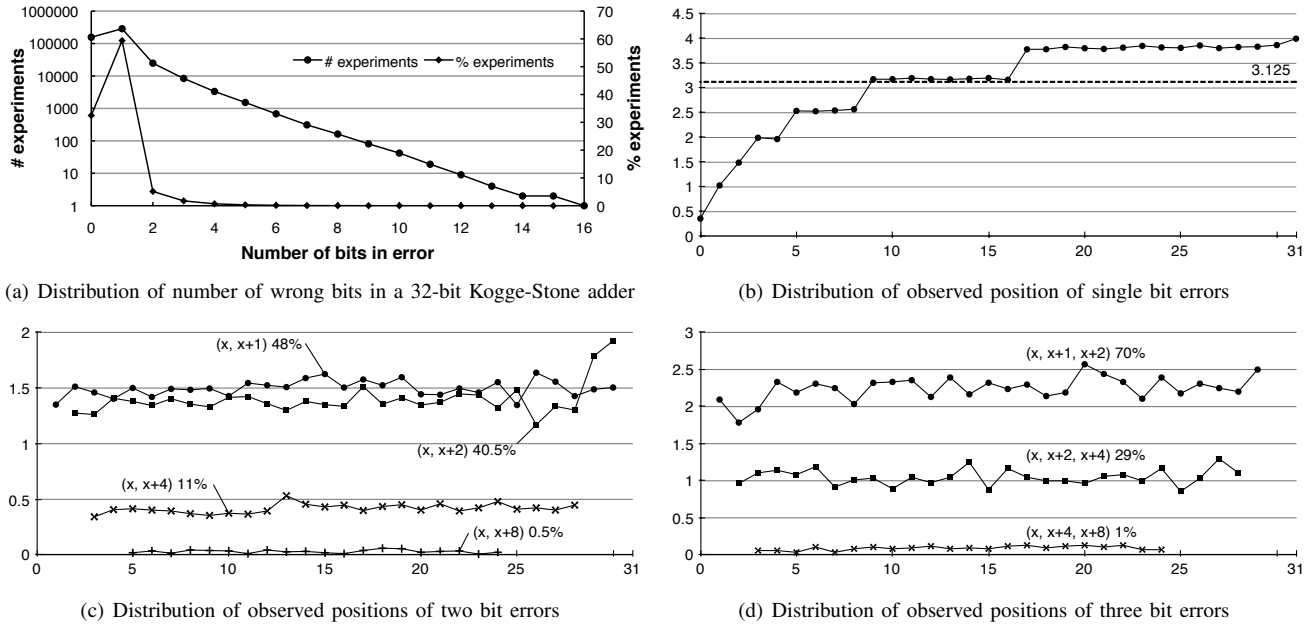
At the system level the area derating factor can play a more significant role (Equation 1). That is, when estimating the impact that a particular microarchitecture structure has on the vulnerability of a whole device such as a processor, its area relative to the total area can give rise to significant derating. For example, the area of an adder relative to the total area of a microprocessor can be quite insignificant, given that a very large on-chip area is taken up by cache and other SRAM-based array structures. This makes the adder circuit a relatively small target within the whole processor for a soft error from, for example a transient fault. However, applications such as custom ASICs and embedded processors employed in safety-related applications may not include large cache structures in order to provide more deterministic operation. In such cases, the contribution of the otherwise small microarchitecture structures that implement combinational blocks towards the computation of total vulnerability becomes significant.

A. Results from FIsim

The results obtained from the fault injection simulator generated by the FIsim compiler contain a wealth of information about the way errors manifest at circuit outputs in the presence of internal errors. For example, Figure 3(a) shows a distribution of the different error multiplicities observed from a fault campaign conducted on a 32-bit Kogge-Stone adder. 1000 fault simulations were run on each of the 482 internal nets. It shows that of the 482000 total fault simulations conducted, about 32% experiments resulted in no response (0 bits in error) and ∼60% produced error on exactly one bit. The remaining ∼8% comprise the remaining fault multiplicities, i.e. 2-16. No fault injection in the campaign produced errors in more than 16 bits. Figure 3(a) shows the same distribution on two scales – as a count of the number of experiments on the logarithmic scale and as a percentage on the linear scale.

If the fault-to-error propagation in an adder circuit were truly random, errors originating within the adder could, in the worst case, propagate to all output bits of the adder (e.g. error in carry-in bit). Enumerating all such output error combinations can quickly get intractable with the width of the adder. For example, the possible number of random error combinations at the output of a 32 bit adder can be evaluated by the expression $\sum \binom{32}{i}$ and is approximately 4.3 billion. However, after 482000 fault simulations (482 nets $\times$ 1000 faults per net) on the 32-bit Kogge-Stone adder, just 501 unique patterns were observed. This comprises 0.000012% of the possible combinations. This observation is important when designing fault injection experiments at a higher level in the design hierarchy. It simplifies the definition of the fault-to-error model and therefore how errors can credibly be reproduced at a higher level simulation. The fault space from which to choose a fault in a campaign is significantly smaller that a random selection of wrong bits.

When considering one bit upsets, which comprise 88% of all experiments counted vulnerable, the position at which the error appears can either be considered random and uniformly distributed or according to a distribution such as the one shown in Figure 3(b). It shows the frequency of error at each output bit position for experiments that produced an error at exactly one output bit. If the error at each bit position was



(a) Distribution of number of wrong bits in a 32-bit Kogge-Stone adder

(b) Distribution of observed position of single bit errors

(c) Distribution of observed positions of two bit errors

(d) Distribution of observed positions of three bit errors

Figure 3. Various error distributions in a 32-bit Kogge-Stone adder. For Figures 3(b)–3(d), the x-axis shows bit position and y-axis shows frequency of observation (%).

considered to be equally likely, the expected frequency of error at each position would be 3.125% (1/32, dashed line in Figure 3(b)).

All two bit errors appeared to be confined to one of four error combination patterns. Each error combination pattern is distinguished by the difference between the positions of bits in error, specifically being equal to 1, 2, 4 or 8 bits. A similar pattern can be observed for three bit error combinations – the difference in bit position between the first and second wrong bits was always the same as that between the second and third wrong bits. Further, this difference was confined to one of three values – 1, 2 and 4 bit positions. Such regularity of error patterns can be attributed to the fact that the topology and fan-out of Kogge-Stone adders are very regular and well-defined. Error patterns for other circuits may be significantly different.

Distributions of each of these error patterns are shown in Figures 3(c) and 3(d). The relative frequency of each error pattern within each multiplicity is also indicated next to each series. For example, in Figure 3(d), the patterns with a difference of 1 between the bit position in error (wrong bits are x, x+1, x+2) form 70% of all three bit error patterns. Observations from such analyses are valuable in that they provide error patterns that can be emulated to reproduce errors at higher levels. For example, when reproducing two bit error patterns at the high level, an experiment where the error patterns (x, x+1) and (x, x+2) each appear with 50% probability would be fairly representative of the observation in Figure 3(c), as the occurrence of the patterns (x, x+4) and (x, x+8) are relatively rare. Further, the largely *flat* profiles of each series in Figures 3(c) and 3(d) indicate that the position of occurrence of two and three bit errors appear more or less uniformly distributed when compared to the position of one bit errors shown in Figure 3(b). This implies that a random selection of x to introduce a two or three bit error at the system level is valid. Again, these results can vary significantly from circuit to circuit.

VI. Methodology for System-Level Vulnerability Estimation

System-level vulnerability may be estimated in a number of ways. Irrespective of the approach, the basic methodology is to estimate the fraction of faults that will affect correctness of the output. One approach for processors is to estimate the fraction of instructions that is required for architecturally correct execution (ACE) [9][10]. Another approach is to use fault injection, where an error arising from a fault is reproduced within a workload and its effects on the output are statistically quantified [11][12]. ACE methodologies are not suitable to study the effects of different fault models on system vulnerability because correctness of output is estimated by detailed analyses of the behavior of instructions; i.e., how they manipulate data stored in the resources that they use. Fault injection, on the other hand, is more conducive to studying system effects of different fault models because an estimate obtained from fault injection will be more precise than ACE methodologies if the fault-to-error model is well defined [12]. Fault injection into a high-fidelity system simulation running workloads in the presence of errors was therefore chosen as the approach to estimating

system-level vulnerability. In this paper, general purpose processors (with a focus on embedded architectures, although high performance architectures are not excluded) are used as the system model, but ASICs, field programmable gate arrays (FPGAs), graphics processing units (GPUs), application specific instruction processors (ASIPs), etc. can all employ a generalized version of the methodology detailed here.

A. Fault Injection Infrastructure

For the experiments designed to estimate system-level vulnerability, fault injections were performed on a microarchitecture simulator running benchmark applications as the workload. Fault injection in simulation has a number of advantages, including the observability and controllability to track execution cycles and to inject faults into precise locations at precise points in time. Another valuable feature that simulation provides is the ability to switch execution to native hardware in order to speed up long fault injection campaigns and possibly run workloads to completion (the importance of which is discussed below).

1) Choice of Simulator: PTLsim [13] is a cycle accurate x86-64 architecture simulator that has all of the advantages mentioned above to enable easy augmentation for fault injection. The main advantage among which is its ability to easily switch to native hardware execution when executed on any x86-64 hardware. This capability is very valuable to realize *fast forwarding* to speed up simulation times. PTLsim, by default, is an x86-64 microarchitecture simulator and is therefore representative of modern general purpose microprocessors. It was operated in a mode that assumed a *perfect cache* to emulate behavior that is more representative of embedded processors that lack cache structures, than general purpose microprocessors. The impact of a perfect cache is that accesses to memory are completed in a single cycle during the simulation, but it does not affect the system vulnerability to combinational logic errors.

2) Choice of Workload: The impact that any microarchitecture structure has on system-level vulnerability can be highly dependent on the workload that is running on it. For this reason, workload forms an important component of the system model. The choice of workload is therefore driven by a number of factors, including suitability as a representative workload in the system model. From a practical point of view, if the workload is too large, simulation times can be prohibitively long for running large fault campaigns. Given that fault injection is a statistical experiment, a large number of *samples* are needed for a high confidence in the estimated parameters.

Further, when using fault injection to estimate vulnerability, it is imperative to be able to quickly and precisely determine the outcome of an experiment. This is because a vulnerability estimate is an estimate of the correctness of output in the presence of faults. The easiest approach to this would be to run the simulation of a workload to completion after a fault has been injected and to determine outcome by comparing output with a *gold* output (fault-free). While this approach may be precise, it can be practical only if the workloads are small enough to allow the execution of long campaigns to satisfy the constraint imposed by statistical confidence. For these reasons, the widely used SPEC benchmark applications are not suitable workloads for fault injection on microarchitectures. While prior work in the literature does make use of SPEC benchmarks as workloads, they are usually not run to completion as they typically consist of billions of simulated cycles. Consequently, a small snapshot of each benchmark is what is often simulated in the presence of faults. This approach is not precise when making decisions about the outcome of a workload in the presence of a fault and can give rise to significant error in the estimated parameters.

For these reasons MiBench [14] was the workload of choice. It is a free and commercially representative set of applications directed towards embedded applications. MiBench applications are considerably smaller when used with the *small* data set in comparison to SPEC benchmarks (tens of millions vs. billions of simulated cycles). SPEC benchmarks are typically used for performance measurements, where having large applications is beneficial. However, for the experiments presented here, performance metrics are of little significance because it is the functional correctness of a microarchitecture in the presence of faults that is sought. It is more important to be able to *quickly* and *precisely* determine the outcome of the output of a representative workload in the presence of faults. The applications in the MiBench suite served to fulfill both of these requirements, but the appropriate workload must always be determined for the system and application under consideration.

3) Experiment Control: Automating fault campaigns is another key aspect of practical fault injection. PERL scripts were used to automate the various steps in each fault injection experiment and over long fault campaigns. A fault-free execution of each workload was carried out to determine the execution time and the *gold* output to compare to after each fault injection. Each fault injection experiment consisted of setup and start of the simulator with the appropriate parameters and workload. When the chosen instruction/cycle was reached, the saboteur module injected an error and resumed execution.

After completion of execution, the automation scripts compared the output of the workload with the gold output. If the workload ran to completion but produced an error in the output, that experiment was flagged as silent data corruption (*SDC*). In a number of experiments, an exception (*EXCP*) was flagged by the microarchitecture, which prematurely terminated execution. These are potentially detectable, but because it has an impact on the outcome of the experiment, it is counted as required for architecturally correct execution

and the experiment is therefore counted vulnerable. In a few experiments per campaign, the injected error resulted in the workload to enter an infinite (*INF*) loop and would not terminate in the expected simulation time. These experiments were killed if the simulation time exceeded ten times the time required to simulate the fault-free case. Experiments in which the injected fault had no effect on the outcome were flagged *unACE*.

B. Emulating Errors in Combinational Blocks

This subsection describes how a fault-to-error model developed using FIsim (elaborated in Section V-A) for a combinational block was applied as injected errors in the high-level system model. This is demonstrated assuming that a 32-bit Kogge-Stone adder is the adder circuit implemented in the ALU block in the microarchitecture of PTLsim. PTLsim was augmented with a saboteur module, which when enabled introduced an error at the output of the ALU. The error that it chooses to inject was based on the fault-to-error model and fault space evaluated using FIsim for a 32-bit Kogge-Stone adder. The specific adder topology was chosen for the purpose of demonstration. Nonetheless, any adder topology may be used.

Errors were injected at the output of randomly chosen instructions that make use of the adder circuit (e.g. ADD, ADDA, SUB, ...). This is because not all randomly selected instructions make use of the ALU and, specifically, the adder, and would result in wasted simulation effort (i.e., the bits would already be known to be unACE). This assumption requires an additional derating factor, which represents a fraction of instructions that make use of the adder relative to the total number of instructions. This term would depend heavily on the workload and can easily be obtained from a profile of the instruction stream. For the benchmarks tested here, a distribution of the instruction-mix is available in [14] and is around 60% on an average across all benchmarks.

C. Results

This section describes the results obtained from micro-architecture fault injection experiments, where errors in an adder circuit were reproduced to estimate its impact on system-level vulnerability. Section V-A alluded to the fact that a large fraction ($\sim$60%) of errors within an adder manifest as 1 bit errors at its output. About 5% appear as 2 bit errors and $\sim$1.7% as 3 bit errors. These error multiplicities were tested at the system level by introducing error patterns from Figures 3(b)–3(d). A uniform distribution was assumed for the position of single errors at the adder output. 50% probability for the occurrence of each of the patterns (x, x+1) and (x, x+2) was assumed for two bit errors. A uniform distribution of the position, with the pattern (x, x+1, x+2), was used for three bit error patterns.

1000 error patterns for each multiplicity were applied to each of the fourteen benchmarks in the MiBench suite. Figure 4 shows results from these experiments. It shows the total vulnerability for the microarchitecture in the presence of errors at the adder output for various benchmark applications as the sum of three different vulnerable outcomes (SDC, INF and EXCP). The 90% confidence interval for the total vulnerability estimate is shown as error bars for each fault campaign (1000 experiments per benchmark-multiplicity combination). The average confidence interval at 90% confidence level is $\pm$2.56%. It is clear from Figure 4 that there is little impact by introducing the notion of error multiplicity at the adder output. For this microarchitecture and the adder circuit, it is apparent that rather than the magnitude of error, it is the fact that any error is present at the adder output that contributes to vulnerability at the system level. This is evident from the fact that the vulnerability estimate under the assumption of each fault multiplicity is within the 90% confidence interval for the vulnerability estimates for most benchmarks.

As stated above, this analysis assumed a uniform distribution for the bit position of single errors at the adder output. However, the results in Figure 3(b) show that not to be the case. Therefore, analyses were also run on these workloads to determine the sensitivity of system vulnerability to single bit error *position* at the adder output. The results for three benchmarks are shown in Figure 5. As might be expected, the trend shows that the system is somewhat more vulnerable to errors at more significant bit positions, but the differences are not large. As a result, a reevaluation of system vulnerability to single bit adder errors using the bit position distribution in Figure 3(b) does not reveal any significant changes from the results in Figure 4. In fact, the revised vulnerabilities all fall within the 90% confidence interval for each benchmark – *string*: 48.52$\Rightarrow$50.54, *bmath*: 61.38$\Rightarrow$62.51, *dijkstra*: 47.80$\Rightarrow$50.02. As was the case for the number of bit errors, this particular system's vulnerability to adder errors under these workloads is not very sensitive to the magnitude of those errors (directly related to the bit position of the error). However, this may be different for other systems, and the methodology presented here can be used to provide the most accurate vulnerability assessment.

VII. Discussion and Conclusions

Electrical and latching window masking describe how soft errors physically remain *contained* within combinational blocks. These effects are fairly well-understood and their corresponding derating factors can easily be evaluated when assessing system vulnerability to such errors. The third masking effect that shields combinational blocks from propagating errors is logical masking, which describes the phenomenon where an error is logically irrelevant for certain input conditions. This effect is comparatively harder to model and quantify because the sheer number of input and

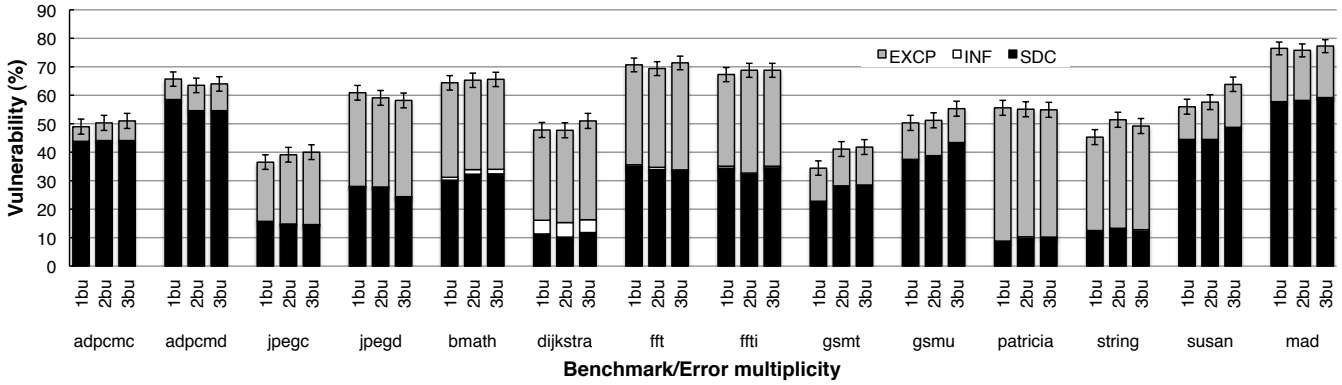


Figure 4. System-level vulnerability to ALU adder errors for benchmarks and error multiplicities

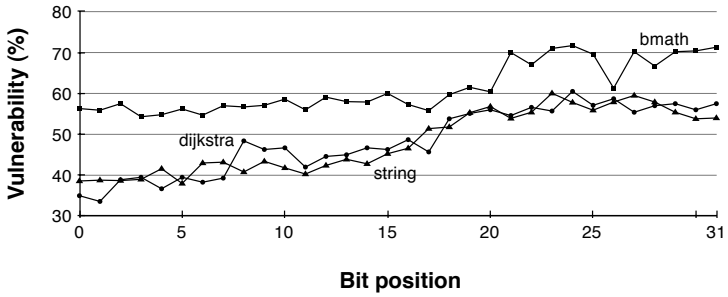


Figure 5. System-level vulnerability to ALU adder errors at each bit position. Results from 1000 fault injections per bit position.

fault combinations in even moderately large circuits can make evaluating them prohibitively expensive.

This paper described a hierarchical approach combined with statistical fault injection as an approach to tackling this problem. At the system level, a conservative estimate of 100% is initially assumed for the vulnerability of the underlying combinational blocks, if implementation details are unavailable. When implementation details become available, a novel FIsim compiler is used to convert the circuit description of a combinational block into a fault injection simulator, which when iteratively executed, a statistical estimate of the circuit's vulnerability is obtained assuming a uniform distribution of inputs. Further, FIsim can be used to generate a model that describes how faults originating within the combinational block manifest at its output in terms of distributions of observed error patterns.

Utility of the FIsim compiler is demonstrated on all ten circuits in the ISCAS85 suite, Kogge-Stone adders of widths ranging from 4 to 64 bits, and a 16-bit multiplier. It is evident from the results in Figure 2 that different circuits exhibit vastly different vulnerability characteristics. Vulnerability estimates ranged from as low as 4% to as high as 90% for the circuits tested. This emphasizes the importance of this type of analysis when designing a system for reliability.

Fault injection is often disregarded as a useful tool because of its statistical nature and overhead involved in collecting large amounts of data in order to ensure statistical significance. However, the time required to iterate FIsim 1000 times for each net in these circuits (3513 and 1154 nets for the ISCAS85 and SPICE circuit sets, respectively) did not exceed two days ($\sim$3.5M, 1.1M total runs), even though the FIsim compiler generated simulator code is largely unoptimized. Moreover, these experiments are data-parallel and can easily be separated onto multiple high-performance hardware units if speed up is desired. Similar arguments hold true even for the fault injection simulations at the system level.

A fault-to-error model developed using FIsim for a 32-bit Kogge-Stone adder was used as *input* to fault injection at a higher-level simulation to estimate its impact on the system-level vulnerability. This model revealed that the total number of observed error patterns forms a very small fraction of all possible error patterns at an adder output. This observation simplifies the set of errors patterns that need to be applied at the higher level and confines it to one, two and three bit errors that follow a regular pattern for the Kogge-Stone adder. When evaluating the vulnerability of the system to different multiplicities, their relative contributions are given by distributions such as the one shown in Figure 3(a) and the vulnerability to a specific multiplicity is obtained from fault injections at the high level (e.g. Figure 4). For the microarchitecture and combinational blocks demonstrated in

this paper, the effects of error multiplicity and bit position have little impact on the vulnerability at the system level. Nonetheless, it could be different for other block-system-application combinations, and the approach in this paper can be used to accurately estimate it, thereby the informing the development of potential fault tolerance techniques to mitigate such faults.

ACKNOWLEDGEMENTS

The authors thank Sudhanva Gurumurthi for his invaluable feedback and the anonymous reviewers for their comments and suggestions. The authors also acknowledge the USNRC for enabling completion of this work.

REFERENCES

[1] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *International Conference on Dependable Systems and Networks*, 2002, pp. 389–398.

[2] P. Lidén, P. Dahlgren, R. Johansson, and J. Karlsson, "On Latching Probability of Particle Induced Transients in Combinational Networks," in *International Symposium on Fault-Tolerant Computing*, 1994, pp. 340–349.

[3] M. Baze and S. Buchner, "Attenuation of Single Event Induced Pulses in CMOS Combinational Logic," *IEEE Transactions on Nuclear Science*, vol. 44, no. 6, pp. 2217–2223, 1997.

[4] L. Massengill, A. Baranski, D. Van Nort, J. Meng, and B. Bhuva, "Analysis of Single-Event Effects in Combinational Logic-Simulation of the AM2901 Bitslice Processor," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2609–2615, 2002.

[5] S. Krishnaswamy, S. Plaza, I. Markov, and J. 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.

[6] I. Polian, S. Reddy, and B. Becker, "Scalable Calculation of Logical Masking Effects for Selective Hardening Against Soft Errors," in *IEEE Computer Society Annual Symposium on VLSI*, 2008, pp. 257–262.

[7] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *International Symposium on Circuits and Systems*, 1985, pp. 695–698.

[8] "www.aoki.ecei.tohoku.ac.jp/arith, Arithmetic Module Generator, Aoki Laboratory, Tohoku University."

[9] 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.

[10] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," in *International Symposium on Computer Architecture*, 2005, pp. 532–543.

[11] 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.

[12] N. George, C. Elks, B. Johnson, and J. Lach, "Transient Fault Models and AVF Estimation Revisited," in *International Conference on Dependable Systems and Networks*, 2010, pp. 477–486.

[13] M. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *International Symposium on Performance Analysis of Systems and Software*, 2007, pp. 23–34.

[14] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *International Workshop on Workload Characterization*, 2001, pp. 3–14.