

Application-Level Correctness and its Impact on Fault Tolerance

Xuanhua Li and Donald Yeung
Department of Electrical and Computer Engineering
University of Maryland at College Park
{xli, yeung}@eng.umd.edu

Abstract

Traditionally, fault tolerance researchers have required architectural state to be numerically perfect for program execution to be correct. However, in many programs, even if execution is not 100% numerically correct, the program can still appear to execute correctly from the user's perspective. Hence, whether a fault is unacceptable or benign may depend on the level of abstraction at which correctness is evaluated, with more faults being benign at higher levels of abstraction, i.e. at the user or application level, compared to lower levels of abstraction, i.e. at the architecture level.

The extent to which programs are more fault resilient at higher levels of abstraction is application dependent. Programs that produce inexact and/or approximate outputs can be very resilient at the application level. We call such programs soft computations, and we find they are common in multimedia workloads, as well as artificial intelligence (AI) workloads. Programs that compute exact numerical outputs offer less error resilience at the application level. However, we find all programs studied in this paper exhibit some enhanced fault resilience at the application level, including those that are traditionally considered exact computations—e.g., SPECInt CPU2000.

This paper investigates definitions of program correctness that view correctness from the application's standpoint rather than the architecture's standpoint. Under application-level correctness, a program's execution is deemed correct as long as the result it produces is acceptable to the user. To quantify user satisfaction, we rely on application-level fidelity metrics that capture user-perceived program solution quality. We conduct a detailed fault susceptibility study that measures how much more fault resilient programs are when defining correctness at the

application level compared to the architecture level. Our results show for 6 multimedia and AI benchmarks that 45.8% of architecturally incorrect faults are correct at the application level. For 3 SPECInt CPU2000 benchmarks, 17.6% of architecturally incorrect faults are correct at the application level. We also present a lightweight fault recovery mechanism that exploits the relaxed requirements on numerical integrity provided by application-level correctness to reduce checkpoint cost. Our lightweight fault recovery mechanism successfully recovers 66.3% of program crashes in our multimedia and AI workloads, while incurring minimum runtime overhead.

1 Introduction

Technology scaling—including feature size, voltage, and clock frequency scaling—has brought tremendous improvements in performance over the past several decades. Unfortunately, these same trends will make computer systems significantly more susceptible to hardware faults in the future, resulting in reduced system reliability. Sources of hardware faults include soft errors [23], wearout [29], and process variations [19]. In anticipation of the reduced reliability that further technology scaling will bring, computer architects have recently focused on several important fault tolerance issues. Areas of focus include characterizing fault susceptibility [18], and developing low-cost fault detection [2, 9, 25, 26] and recovery [32] techniques.

Fundamental to all such reliability research is the definition of correct program execution. In the past, researchers have made very strict assumptions about program correctness. Traditionally, a program's execution is said to be correct only if architectural state is numerically perfect on a cycle-by-cycle basis. A similar (though slightly looser) notion of correctness requires a program's visible architectural state—i.e., its output state—to be numerically perfect. In both cases, correctness requires precise numerical integrity at the architecture level, a fairly strict requirement.

An interesting question is: must we require strict numerical correctness for overall program execution to be correct? In many programs, even if execution is not 100% numerically correct, the program can still *appear* to exe-

This research was supported in part by NSF CAREER Award #CCR-0093110, and in part by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant #NBCH104009. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

cute correctly from the user’s perspective. Although such numerically faulty executions do not pass the muster of architecture-level correctness, they may be completely acceptable at the user or application level. Hence, whether a fault is intolerable or benign may depend on the *level of abstraction* at which correctness is evaluated. In general, more faults are acceptable at higher levels of abstraction, *i.e.* at the application level, compared to lower levels of abstraction, *i.e.* at the architecture level.

How much more fault resilient are programs at the application level? The answer to this question is application dependent, and primarily depends on how numerically exact a program’s outputs need to be. For instance, programs that process human sensory and perception information are highly fault resilient at the application level. An important example is multimedia workloads. Another example is artificial intelligence workloads (*e.g.*, reasoning, inference, and machine learning), which have become increasingly important recently [8]. These programs belong to a class of computations which we call *soft computations* [20, 10].¹ Soft computations compute on approximate data values associated with qualitative results, making them highly fault resilient because errors in numerical results seldom change the *user’s interpretation* of those numerical results. In contrast, programs whose correctness are tied directly to the numerical values they compute may offer little error resilience at the application level. Certain lossless data compression algorithms are examples of such programs. While the degree of error resilience at the application level varies across applications, we find *all* programs studied in this paper exhibit some enhanced fault resilience at the application level, including those that are traditionally considered as exact computations—*e.g.*, SPECInt CPU2000.

This paper explores definitions of program correctness that view correctness from the application’s standpoint rather than the architecture’s standpoint. Under *application-level correctness*, a program’s execution is deemed correct as long as the result it produces is acceptable to the user. In other words, correctness depends on the *user’s interpretation* of a program’s numerical result, not the numerical result itself. To quantify user satisfaction, we rely on application-level fidelity metrics that capture program solution quality as perceived by the user. Because the notion of solution quality is different across applications, our fidelity metrics are application specific, though applications from the same domain may share common fidelity metrics.

Our goal is to understand how application-level correctness impacts a system’s susceptibility to faults, especially transient faults or soft errors. The centerpiece of our work

¹The term “soft computation” is normally used to describe artificial intelligence algorithms. In this paper, we use the term to describe multimedia workloads as well because we find they exhibit similar inexact computing properties as the A.I. algorithms.

is a detailed fault injection study that quantifies how much more resilient programs are to soft errors at the application level compared to the architecture level. Our study injects 156,205 faults into a detailed architectural simulator, and performs 27,067 separate runs to program completion. For soft computations, we find 45.8% of fault injections that lead to architecturally incorrect execution produce acceptable results under application-level correctness. For SPEC programs, a smaller portion of architecturally incorrect faults, 17.6%, produce acceptable results at the application level. In addition to studying fault susceptibility, we also present a lightweight fault recovery mechanism that exploits the relaxed requirements on numerical integrity provided by application-level correctness to reduce checkpoint cost. Our technique checkpoints some minimum state needed to recover after a crash, but omits from checkpoints those data values for which the user can tolerate numerical imprecision. Although our lightweight fault recovery mechanism is not fail-safe, it successfully recovers 66.3% of program crashes in our multimedia and AI workloads, while incurring extremely low runtime overhead.

The remainder of this paper is organized as follows. Section 2 discusses our definitions of application-level correctness. Then, Section 3 presents our experimental methodology and Section 4 reports our fault susceptibility study. Next, Section 5 describes our lightweight recovery mechanism. Finally, Section 6 presents related work, and Section 7 concludes the paper.

2 Application-Level Correctness

This section presents our application-level correctness definitions. We begin by discussing soft program outputs, an important property for application-level correctness (Section 2.1). Then, we present fidelity metrics that quantify application-level correctness for the benchmarks studied in this paper (Section 2.2). Finally, we discuss limitations of our approach (Section 2.3).

2.1 Soft Program Outputs

Programs can exhibit enhanced error resilience at the application level compared to the architecture level for many reasons. However, the likelihood of this happening increases when a program permits *multiple valid outputs*. In this paper, we say such programs have “soft outputs.” Soft outputs commonly occur in programs computing results that are interpreted qualitatively by the user. Different numerical results can lead to the same or similar qualitative interpretation. Hence, multiple numerical outputs may be acceptable to the user. Another source of soft outputs is heuristic-based algorithms. Many programs solve complex problems for which optimal solutions are unachievable. Instead of the optimal, they try to find the best solutions possible given available computational resources. In practice,

many solutions are “good enough.” So, once again, multiple numerical outputs are acceptable to the user.

Soft outputs offer new opportunities for optimizing fault tolerance. In particular, faults that cause a program to simply generate one of its multiple valid outputs are completely benign. It is unnecessary to protect against such faults, allowing designers to reduce the cost of fault protection. For example, in Section 5, we will study a lightweight fault recovery technique that omits from checkpointing the data that only contribute to soft program outputs (since these data are highly error tolerant), thus reduces checkpoint cost.

To illustrate the soft output property, Table 1 lists 9 benchmarks used in our study—three from the multimedia domain, three from the artificial intelligence (AI) domain, and three from SPECInt CPU2000. The multimedia workloads, G.721-D, JPEG-D, and MPEG-D are taken from the Mediabench suite [15], and perform audio, image, and video decompression, respectively. All three decompression algorithms are lossy. The AI workloads are from various sources. LBP performs Pearl’s Loopy Belief Propagation [22], a well-known message-passing algorithm for approximate inference on large Markov networks. Our LBP implementation solves Taskar’s Relational Markov Network applied to a web-page classification problem [30]. SVM-L is the learning portion of a Support Vector Machine algorithm, called SVMlight [11]. SVM-L learns the parameters for a support vector (SV) model on a training dataset. GA is a genetic algorithm applied to multiprocessor thread scheduling [12]. Given a thread dependence graph, GA searches for a thread schedule that minimizes execution time. Finally, the SPECInt CPU2000 workloads are 164.gzip and 256.bzip2, two lossless data compression algorithms, and 175.vpr, a place-and-route program. (The data inputs we use for vpr only perform placement—see Table 3 in Section 3).

The second column of Table 1 reports the numerical outputs computed by each benchmark. As we will show, *all* of these numerical outputs are soft, so multiple valid outputs exist. In most cases, the soft outputs are due to the qualitative nature of the program results. When appropriate, we indicate this in the third column, labeled “Qualitative Output.” Many of our benchmarks also achieve soft outputs because they are heuristic-based; some examples of this are discussed below.

For the three multimedia programs, the numerical outputs are the decompressed datafiles, either in audio, image, or video format. Once decompressed, these datafiles can be played back to the user; hence, the qualitative output of these programs is the perceived playback, either aural or visual, of the numerical outputs. Because the user’s playback experience is qualitative in nature, it is possible for different numerical outputs to be acceptable (*i.e.*, valid) to the user.

Like the multimedia workloads, the AI workloads also

Bench	Num. Out	Qual. Out	Fidelity Metric
Multimedia			
G.721-D	Decompressed audio datafile	Perceived audio	Segmental Signal-to-Noise Ratio (SNR _{seg})
JPEG-D	Decompressed image datafile	Perceived image	Peak Signal-to-Noise Ratio (PSNR)
MPEG-D	Decompressed video datafile	Perceived video	Peak Signal-to-Noise Ratio (PSNR)
Artificial Intelligence			
LBP	Network belief values	Web Page Class Types	% Classification Change
SVM-L	Support Vector Model	Test Data Class Types	% Classification Change
GA	Thread Schedule	-	% Schedule Length Change
SPECInt CPU2000			
164.gzip	Compressed file	-	Compression Ratio
256.bzip2	Compressed file	-	Compression Ratio
175.vpr	Cell placement	-	Consistency Check

Table 1. Numerical and qualitative outputs computed by our benchmarks. The last column lists the fidelity metrics used to quantify solution quality.

exhibit soft program outputs. In LBP, nodes in the Markov network contain probability distribution functions (PDFs) over the possible class types inferred for web pages. Each PDF encodes how strongly we “believe” a particular web page belongs to each class type. The numerical output for LBP, hence, is the collective belief values across the entire Markov network. In SVM-L, the numerical output is the SV model parameters learned from the training dataset, as described earlier. Both LBP and SVM-L’s numerical outputs are soft because they are used to derive classification answers, the qualitative output for these programs. LBP selects a class type for each web page by choosing the most likely class indicated by the corresponding PDF. For SVM-L, extracting class types is more involved because SVM-L itself doesn’t perform classification. To obtain the class types we want, we run a separate SVM classifier (not listed in Table 1) that uses the SV model computed by SVM-L to perform classification on a test dataset. Computing the classification answers in both LBP and SVM-L is an extremely inexact process. Multiple numerical outputs (belief values for LBP and SV model parameters for SVM-L) can lead to the same (and hence, valid) classification answer. In GA, the numerical output is the thread schedule it computes. GA’s numerical output does not have a qualitative interpretation; however, users can still accept multiple numerical outputs because GA is a heuristic algorithm. Although it is infeasible to find the optimal thread schedule, in practice, there are many thread schedules that are adequate. Any one of these good enough answers represents a valid numerical

output from the user’s perspective.

Somewhat surprisingly, the three SPEC program outputs are also soft, though we do not call the SPEC benchmarks soft computations. As indicated in Table 1, none of the SPEC outputs have qualitative interpretations; nonetheless, multiple numerical outputs are valid. For the data compression algorithms, there is flexibility in how datafiles are compressed even though the compression algorithms themselves are exact. We will discuss the reasons for this in Section 4. The *vpr* benchmark tries to find a cell block placement for a chip design. Like GA, *vpr*’s algorithm is heuristic-based since finding an optimal placement (one that minimizes interconnect distance) is intractable. Hence, multiple cell block placements are valid.

Finally, while all the benchmarks in Table 1 exhibit soft outputs, it is important to note there are also programs for which multiple valid outputs do not exist. For example, sorting algorithms (*e.g.*, quicksort) permit only one correct answer. Thus, there is little or no additional error resilience that can be exploited at the application level. We do not consider such programs in this paper since our goal is to characterize and exploit application-level error resilience where it exists. Although studying the extent to which soft outputs occur in programs is certainly an important direction of research, it is beyond the scope of this work.

2.2 Solution Quality

Because the benchmarks in Table 1 permit multiple valid numerical outputs, their correctness is not simply “black or white;” hence architecture-level correctness (where all architectural values are either correct or wrong) is clearly too strict. An appropriate correctness definition should accommodate all valid numerical outputs. At the same time, it is important to recognize not all valid outputs are of equal value; instead, there are varying degrees of solution quality across our programs’ outputs.

We use application-specific fidelity metrics to capture the quality of a program’s output as perceived by the user. Our fidelity metrics quantify how different a particular output is relative to a baseline output. (For the experiments in Sections 4 and 5, we define the baseline output to be the result obtained from a fault-free execution of a given benchmark). Outputs that are very similar to the baseline have high fidelity, whereas outputs that are very dissimilar have low fidelity. Whenever possible, we compute fidelity in terms of a benchmark’s qualitative outputs instead of its numerical outputs. This enables us to capture fidelity of the user’s qualitative experience, an important correctness consideration for many of our benchmarks.

The last column in Table 1 lists the fidelity metrics we use for our 9 benchmarks. For the multimedia workloads, we use signal-to-noise ratio (SNR). Specifically, we use segmental SNR (SNRseg) for G.271-D, and peak SNR

(PSNR) for JPEG-D and MPEG-D. For LBP and SVM-L, we use the percentage change in classification answers, and for GA, we use the percentage change in thread schedule length (*i.e.*, execution time). For the two data compression algorithms, we use the compression ratio.² Lastly, *vpr*’s fidelity metric is a consistency check provided by the code itself. This consistency check first determines whether a given cell block placement is valid (*i.e.*, doesn’t violate any design rules), and then computes a cost metric that reflects the degree to which interconnect distance is minimized. Placements that can’t pass the consistency check are incorrect.

Given the fidelity metrics in Table 1, application-level correctness can be defined by choosing the minimum fidelity that is “acceptable” to the user: outputs of equal or higher quality than the minimum fidelity satisfy the user’s requirement and are considered correct, while outputs of lower quality than the minimum fidelity are considered incorrect. An important question, then, is how do we determine the minimum fidelity threshold against which application-level correctness is measured? Unfortunately, minimum fidelity thresholds are extremely user-dependent. In practice, different users may desire different levels of solution quality (in fact, the *same* user may be able to live with varying levels of solution quality under different circumstances), so it is impossible to define one threshold that applies universally. Instead, users should be allowed to select the threshold that best fits their correctness requirements. As we will see in Section 4, this provides designers with the unique opportunity to tradeoff solution quality for fault tolerance, depending on how good a solution the user needs.

While minimum fidelity thresholds are user-dependent, nonetheless, we must choose a specific set of threshold values for the experiments conducted later in this paper. Section 3 will discuss how we choose minimum fidelity thresholds for our experiments.

2.3 Limitations

A limitation of application-level correctness is it only considers program outputs visible to the user. It does not account for other correctness issues unrelated to visible program outputs. For example, we do not consider real-time issues. Certain errors may not degrade solution quality appreciably, but they may alter *when* solutions become available. This is unacceptable for the correctness of real-time systems. In addition, we do not consider system-level issues. Errors that do not defeat individual benchmarks may still propagate to other programs in a multiprogrammed environment, causing them to crash or execute incorrectly. Lastly, it may still be necessary to provide architecture-level correctness in cases where architecture state is exposed to the user

²Note, due to their lossless nature, compressed outputs that cannot identically reproduce the original datafile are deemed as incorrect, regardless of the compression ratio.

Processor Parameters	
Bandwidth	8-Fetch, 8-Issue, 8-Commit
Queue size	64-IFQ, 40-Int IQ, 30-FP IQ, 128-LSQ
Rename reg/ROB	128-Int, 128-FP / 256 entry
Functional unit	8-Int Add, 4-Int Mul/Div, 4-Mem Port 4-FP Add, 2-FP Mul/Div
Branch Predictor Parameters	
Branch predictor	Hybrid
Meta table	8192-entry gshare/2048-entry Bimod
BTB/RAS	8192 entries 2048 4-way / 64
Memory Parameters	
IL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
DL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
UL2 config	1Mbyte, 64byte block, 4 way, 20 cycle lat
Mem config	300 cycle first chunk, 6 cycle inter chunk

Table 2. Parameter settings for the detailed architectural model into which we inject faults.

(*e.g.*, program debugging). In all these cases, application-level correctness is not strict enough and does not provide the desired correctness requirements.

3 Experimental Methodology

Having presented our definitions of application-level correctness, we now quantify how much more fault resilient programs are under application-level correctness compared to architecture-level correctness. This section discusses the experimental methodology used in our fault susceptibility study. Later, Section 4 will present the study’s results.

To analyze fault susceptibility, we conduct fault injection experiments [14, 25, 34] to observe the effects of faults on a CPU under different definitions of correctness. Each of our fault injection experiments injects a single bit flip into the execution of one of our benchmarks—*i.e.*, we assume a single event upset, or SEU, fault model. Our approach closely follows the methodology introduced by Reis *et al.* [25]. We initially inject faults into a detailed architectural simulator that models a modern out-of-order superscalar. After each fault is injected, we simulate the microarchitecture until the fault completely manifests itself in architectural state. Then, we checkpoint the simulator’s architectural state, and resume simulation from the checkpoint using a simple functional simulator. We try to run the benchmark to completion under the functional CPU model, and assuming the benchmark doesn’t crash, we evaluate the program’s outputs under both architecture- and application-level correctness.

In the detailed simulation phase, we use a modified version of the out-of-order processor model from SimpleScalar 3.0 for the PISA instruction set [5], configured with the simulator settings listed in Table 2. Compared to the original, our modified simulator models rename registers and issue queues separately from the Reservation Update Unit (RUU). Using this processor model, we inject faults into three hardware structures: the physical register file, the

fetch queue, and the issue queue (IQ).³ Faults injected into a physical register will appear in architectural state unless the register is idle or belongs to a mispredicted instruction. For the fetch queue, we allow faults to corrupt instruction bits, including opcodes, register addresses, and immediate specifiers. These faults manifest in architectural state as long as the injected instruction is not mispredicted. Lastly, for the IQ, we model 6 fields per entry: instruction opcode, 3 register tags (2 source and 1 destination), an immediate specifier, and a PC value. Like the fetch queue, faults in the IQ appear in architectural state for instructions that are not mispredicted. Corruptions to the IQ opcode and immediate fields behave similarly to corresponding corruptions in the fetch queue. Corruptions to the register tags alter instruction dependences, and corruptions to the PC value affect branch target addresses.

When simulating in detailed mode, two issues affect the collection of checkpoints for subsequent functional simulation. First, not all fault injections require functional simulation to program completion. Some faults are masked by the microarchitecture, and do not propagate to architectural state. Other faults incur exceptions or lockups. (We rely on a watchdog timer to detect lockups). In these cases, we simply record the outcome, and skip the functional simulation phase. Second, faults in the out-of-order portion of the processor pipeline (*i.e.*, the physical register file and IQ) can manifest in architectural state in an imprecise manner. For example, a corrupted register value may propagate to some instructions (those that haven’t issued yet) but not to others (those that have already issued). Our detailed simulator records these out-of-order effects. Then, when simulating the initial instructions in functional mode (*i.e.*, those that were in-flight at the time of the fault), we propagate the injected fault to exactly the same instructions that were affected during out-of-order simulation.

Table 3 presents detailed fault injection information for each of our benchmarks described in Section 2. The column labeled “Input” specifies the input dataset used for each benchmark, and the column labeled “Exec Time” reports each benchmark’s measured execution time in cycles on our detailed out-of-order simulator. We inject faults only after program initialization, so “Exec Time” does not include the benchmarks’ initialization phase. After program initialization, we run each benchmark to completion in our detailed simulator, performing all fault injections and checkpoints for a single hardware structure in the same run. We perform 3 such injection runs on each benchmark to inject faults into the 3 hardware structures (*i.e.*, physical register file, fetch queue, and IQ). In each run, faults are randomly injected

³For both the physical register file and issue queue, our simulator models separate integer and floating point versions of the structures. However, when injecting faults, we distribute the faults uniformly across both versions as if they formed a unified structure.

Bench	Input	Exec Time	Interval	Injects	Regfile	Fetch	Issue
G.721-D	clinton.pcm	77643471	7000.0	10467	483 (0.05)	581 (0.06)	1183 (0.11)
JPEG-D	lena.ppm	44520776	7000.0	5950	542 (0.09)	4341 (0.73)	1483 (0.25)
MPEG-D	mei16v2.m2v	40457756	7000.0	5413	713 (0.13)	434 (0.08)	803 (0.15)
LBP	WebKB [30]	2175526139	1000000.0	2198	1317 (0.60)	946 (0.43)	589 (0.27)
SVM-L	a1a(a1a) [6]	53981768	7000.0	7225	1138 (0.16)	2327 (0.32)	1564 (0.22)
GA	r16-0.1.in(a1a) [12]	127490411	15000.0	8491	479 (0.06)	626 (0.07)	1352 (0.16)
164.gzip	input.compressed	93396309	15000.0	6693	467 (0.07)	829 (0.12)	861 (0.13)
256.bzip2	input.compressed	732651712	250000.0	2941	264 (0.09)	1559 (0.53)	722 (0.25)
175.vpr	test	800450837	250000.0	3177	968 (0.30)	166 (0.05)	330 (0.10)

Table 3. Fault injection statistics. “Exec Time” reports execution time in cycles. “Interval” reports the average time between fault injections. “Injects” reports the total number of faults injected into the physical register file. The last 3 columns report the number of functional simulation runs for each of the 3 hardware structures.

into a single hardware structure one after another using a uniformly distributed inter-fault arrival time.

It is crucial to limit the total number of fault injections since each fault potentially requires functional simulation to program completion. Our methodology limits the number of injected faults in two ways. First, we choose program inputs that do not result in exceedingly long execution times. Second, we set the inter-fault arrival time based on each benchmark’s execution time. We use larger arrival times for longer-running benchmarks, thus reducing the number of injected faults for benchmarks with longer execution times. The column labeled “Interval” in Table 3 reports the inter-fault arrival time used for each benchmark, while the column labeled “Injects” reports the total number of injected faults for the physical register file. (The number of injected faults for the other two hardware structures is very similar since they use the same inter-fault arrival time. More specifically, the total number of injected faults is 52,555 for physical register file, 52,229 for fetch buffer, and 51,421 for IQ). Across all 3 hardware structures, our fault injection campaign performs 156,205 fault injections.

In addition to how we inject faults, another important methodology issue is what standard do we use to assess application-level correctness? As discussed in Section 2.2, application-level correctness is defined by the minimum fidelity threshold that is “acceptable” to the user. In our experiments, we define two fidelity thresholds for this purpose: “high” and “good.” The high threshold corresponds to program outputs of extremely high quality, with no noticeable solution quality degradation compared to a fault-free execution. The good threshold corresponds to program outputs with only slightly (barely noticeable) degraded solution quality compared to a fault-free execution. Although we define two separate thresholds, in our analysis, we consider any program output that meets the good threshold as being correct under application-level correctness (*i.e.*, the good threshold is our minimum fidelity threshold).

We quantify the high and good thresholds for each fidelity metric in Table 1 as follows. For the SNRseg and PSNR metrics associated with our multimedia benchmarks,

we define high and good outputs to be greater than 90dB and between 50dB and 90dB, respectively, when compared to outputs from fault-free execution. We aurally and visually compared faulty and fault-free outputs to select these thresholds so that they conform qualitatively to the high and good standards described above. Also, we confirmed quantitatively that the good threshold is equal to or better than what is accepted by the signal processing community as constituting a “barely noticeable” difference [3, 7]. For all other fidelity metrics, we define high and good outputs to be within 1% and 5%, respectively, of the program outputs obtained via fault-free execution. Unfortunately, we were unable to find any standards in the literature against which to compare these thresholds, so we chose them to be conservative. For our AI benchmarks, the fault-free outputs themselves are erroneous (the AI benchmarks only compute approximate solutions). In all cases, the fault-free outputs are off by 15% or more compared to perfect solutions obtained off-line. Hence, 1% and 5% represent small additional errors on top of the benchmarks’ baseline errors. For the SPEC benchmarks, there is no quantitative justification for our high and good thresholds; we chose 1% and 5% because we believe these represent small increases in file size (for gzip and bzip) and average wire length (for vpr).

4 Fault Susceptibility

This section discusses our fault susceptibility study in two parts. First, Section 4.1 presents the fault injection results. Then, Section 4.2 analyzes the sources of increased error resilience at the application level.

4.1 Fault Injection Results

Our first result is only a portion of fault injections manifest themselves in architectural state because many faults are masked by the microarchitecture. Microarchitecture-level masking [18] arises due to faults that attack idle hardware resources, or hardware resources occupied by mispredicted instructions. The last three columns in Table 3 report the number of faults injected into the physical register

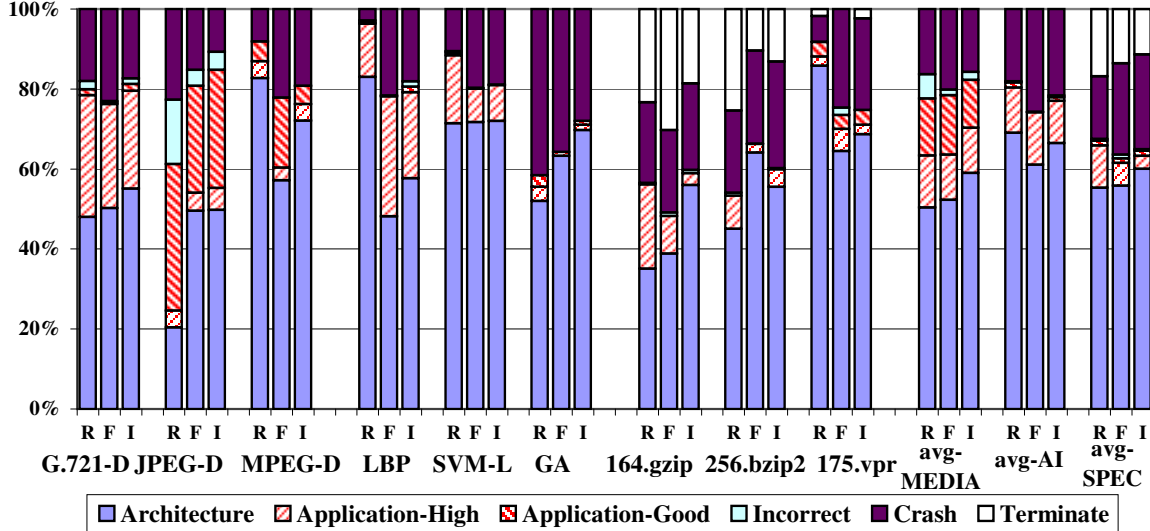


Figure 1. Program outcomes breakdown for architecturally visible fault injections: correct at the architecture level (“Architecture”), correct at the application level (“Application-High” and “Application-Good”), unacceptable (“Incorrect”), exception or program lockup (“Crash”), and early program exit (“Terminate”).

file, fetch queue, and IQ, respectively, that become architecturally visible. In parentheses, we report the same data as a fraction of the total injected faults (*i.e.*, the column labeled “Injects”). As Table 3 shows, the degree of masking varies considerably across different benchmarks and hardware structures. But on average, only 17.3% of injected faults (27,067 out of 156,205) become architecturally visible, with the fetch queue exhibiting the most fault sensitivity (22.6%) and the register file and IQ exhibiting less sensitivity (12.1% and 17.3%, respectively). Faults that are masked by the microarchitecture produce correct program outputs under both architecture- and application-level correctness.

Next, we examine the architecturally visible faults in more detail. Figure 1 breaks down the outcome of all architecturally visible fault injections when they are simulated to program completion. For each benchmark, we report the fault injections into the physical register file, fetch queue, and IQ separately in a group of 3 bars labeled “R,” “F,” and “I,” respectively. Each bar contains 6 categories. The first category, labeled “Architecture,” indicates the program outputs that pass architecture-level correctness (these outputs are also correct at the application level). The next two categories, labeled “Application-High” and “Application-Good,” indicate the additional program outputs that are acceptable under application-level correctness only, assuming the “high” and “good” thresholds described in Section 3. The category labeled “Incorrect” indicate outcomes that are either invalid or unacceptable under both architecture- and application-level correctness. Finally, the last two categories indicate experiments that fail to complete during functional simulation due to an exception or a program lockup (labeled “Crash”) or early program exit with an error

(labeled “Terminate”). The last 3 groups of bars in Figure 1 report the average breakdowns for the multimedia, AI, and SPEC benchmarks, respectively.

Looking at Figure 1, we see a large portion of architecturally visible faults lead to correct program outputs under architecture-level correctness (*i.e.*, the “Architecture” components). The last 3 groups of bars in Figure 1 show architecture-level correctness is achieved in 50.4% to 60.1% of program outputs on average across the 3 hardware structures for the multimedia and SPEC benchmarks, and in 61.0% to 68.8% on average for the AI benchmarks. Similar to microarchitecture-level masking, many fault injections attack architectural state unnecessary for maintaining numerical integrity in our computations, and hence, become architecturally masked [18]. In our benchmarks, the primary source of architecture-level masking is logical and inequality instructions. These instructions rarely change their outputs despite corruptions to their input operands; thus, they are highly resilient to faults. Other (less significant) sources of architecture-level masking include dynamically dead code, NOP instructions, and Y-branches [33].

The remaining fault injections that are not masked at the microarchitecture or architecture levels do not produce numerically correct program outputs. These fault outcomes have traditionally been considered *incorrect* under architecture-level correctness. Across all benchmarks and all hardware structures, 41.2% of architecturally visible fault injections on average are architecturally incorrect. However, we find a significant portion of architecturally incorrect outcomes produce high-quality solutions. This is particularly true for the multimedia and AI benchmarks, our soft computations. As the first group of aver-

age bars in Figure 1 show, 55.0%, 54.8%, and 56.8% of architecturally incorrect faults for multimedia benchmarks occurring in the physical register file, fetch queue, and IQ, respectively, produce program outputs with either high or good fidelity (*i.e.*, the “Application-High” or “Application-Good” components). As the second group of average bars show, 40.4%, 33.8%, and 34.0% of architecturally incorrect faults for AI benchmarks occurring in the same three hardware structures, respectively, produce high or good fidelity program outputs as well. While these program outputs are incorrect numerically, they are completely acceptable from the user’s standpoint—*i.e.*, they are correct at the application level. Overall, 45.8% of architecturally incorrect faults in our soft computations achieve application-level correctness.

In addition to soft computations, we find the SPEC benchmarks also exhibit enhanced fault resilience at the application level. As the last group of bars in Figure 1 shows, 26.2%, 15.5%, and 11.1% of architecturally incorrect faults for the SPEC benchmarks occurring in the physical register file, fetch queue, and IQ, respectively, produce program outputs with either high or good fidelity. These gains are much more modest than those for our multimedia and AI benchmarks. However, we believe the fact that application-level correctness provides any additional fault resilience in SPEC is a positive result given these benchmarks are traditionally considered as exact computations.

4.2 Error Resilience Analysis

The majority of faults leading to the “Application-High” and “Application-Good” categories in Figure 1 occur on computations related to soft outputs. As discussed in Section 2.1, such computations are error resilient since they still have a high likelihood of generating acceptable answers in the face of faults. For example, JPEG-D and MPEG-D perform inverse DCT and quantization, while G.721-D performs adaptive prediction and quantization. Even in the absence of faults, these computations incur small errors due to rounding and their lossy nature. To such computations, faults act like additional errors, and are often tolerable. Compared to the multimedia workloads, our AI programs do not perform lossy operations. Nevertheless, their computations are still highly resilient to faults. As discussed in Section 2.1 for LBP and SVM-L, very little precision in the output data (*i.e.*, belief values or SV model parameters) is needed to derive the correct qualitative answers (*i.e.*, classifications). Therefore, a large number of faults on belief and SV parameter computations can be tolerated as long as they do not affect the most significant bits of the data. Although many soft computations are highly error resilient, one notable exception is GA. As discussed in Section 2.1, GA exhibits soft outputs due to its heuristic nature. However, upon closer examination, we found GA spends most of its time evaluating an objective function that reflects the

cost of a given thread schedule. Unfortunately, this objective function is not a soft computation, thus reducing the benefits of application-level correctness.

Our study also shows the SPEC benchmarks can tolerate faults. As mentioned in Section 2.1, gzip and bzip2’s program outputs are soft due to flexibility in how datafiles can be compressed. We found certain faults cause these compression algorithms to emit different output tokens compared to a fault-free execution. While these output tokens do not achieve as high a compression ratio, they still correctly encode their corresponding input tokens. Hence, a numerically different (slightly larger) compressed file is created, but the exact original file can still be recovered via decompression. In vpr, as already discussed in Section 2.1, the source of soft program outputs is multiple valid cell block placements. Some of our fault injections cause vpr to produce these different cell block placements, and are thus acceptable.

5 Fault Recovery

Section 4 demonstrates many architecturally incorrect faults are acceptable when evaluated at the application level. However, even after considering application-level correctness, a large number of faults still lead to incorrect program outcomes—*i.e.*, the “Incorrect,” “Crash,” and “Terminate” components in Figure 1. Of these, by far the most significant is the “Crash” component. Across all experiments, crashes account for 80.8% of faults on average that are incorrect at both the architecture and application levels.

Addressing crashes requires detecting the corresponding faults, and recovering from them. Since crashes consist of exceptions and program lockups, detection is straightforward: exceptions are intercepted by the operating system⁴ while lockups can be flagged by a CPU watchdog timer. No significant hardware support nor runtime overhead need be incurred for detection. Recovery, on the other hand, can be more costly. Normally, recovery is performed via checkpoints. However, checkpoints incur runtime overhead for copying, either at pre-determined checkpoint locations, or upon first writes (*e.g.*, copy-on-write schemes).

5.1 Lightweight Recovery Mechanism

Under application-level correctness, we do not need to checkpoint the whole program state. Instead, only data that is necessary to restart program execution after a crash needs to be saved. Program state that only contribute to soft outputs do not need to be saved, thus reducing both checkpoint size and runtime overhead. The key question, however, is how do we identify the state that requires check-

⁴We assume all terminating exceptions are due to soft errors (*i.e.*, programs are assumed to be bug free), so we initiate recovery for all of them. In addition, we assume the OS will not trigger recovery for non-fatal exceptions, but instead will process them normally.

pointing to satisfy application-level correctness? We examined several program crashes, and found in most cases program restart can occur simply with a valid program counter (PC) plus the correct stack state at the associated program control point. Hence, we developed a lightweight recovery mechanism that periodically checkpoints the PC, architected register file, and program stack. Upon a crash, we restart the program at the nearest checkpoint, rolling back its PC, register file, and stack only—we do not touch the program text, static data, or heap during rollback. To determine when checkpoints are taken, we identify the main controlling loops in our benchmarks (usually the outer loops associated with major program phases), and instrument checkpointing at the top of each loop iteration. In this paper, we instrument checkpointing manually, though it should be possible to insert the checkpointing code automatically using compiler techniques [16].

Notice, our lightweight recovery mechanism cannot successfully recover all crashes because it does not guarantee all the state necessary for program restart is checkpointed. A fail-safe version of our technique would need to precisely identify the state associated with soft program outputs, and only omit these data from checkpoints. Nonetheless, as the next section will show, our lightweight recovery mechanism can still recover a significant number of crashes.

5.2 Recovery Results

We evaluate our lightweight recovery mechanism using the functional simulator from our two-phase simulation methodology (see Section 3). First, we run checkpoint-instrumented versions of our benchmarks on the functional simulator once to acquire all the checkpoints. Table 4 reports statistics from these checkpoint runs. The columns labeled “# Check,” “Interval,” and “Size” report the total number of checkpoints, the average number of instructions between checkpoints (excluding instrumentation code), and the average checkpoint size, respectively. In parenthesis, we also report the average checkpoint size as a fraction of the total program size. Because we only checkpoint the PC, register file, and stack, our checkpoints are extremely lightweight. On average, our checkpoints are roughly 2 Kbytes in size, with consecutive checkpoints separated by 400,000 instructions or more. Since we acquire our checkpoints on the functional simulator, we have not measured the actual runtime cost of our checkpoints; however, we estimate a 1% runtime overhead at worst.

After acquiring all the checkpoints, we perform recovery experiments. For every crash outcome in Figure 1, we rollback to the nearest checkpoint, as described in Section 5.1, and restart execution in our functional simulator. Then, we try to run the benchmark to completion, and assuming the benchmark doesn’t crash again, we evaluate the program’s outputs under both architecture- and application-level cor-

Bench	# Check	Interval	Size
G.721-D	261	1003622	566 (0.01804)
JPEG-D	59	503137	3360 (0.00397)
MPEG-D	60	2934834	664 (0.00092)
LBP	50	47197236	700 (0.00012)
SVM-L	430	404591	1944 (0.00195)
GA	300	1108510	1282 (0.00003)
164.gzip	252	964376	1108 (0.00036)
256.bzip2	1529	2019708	3462 (0.00036)
175.vpr	2995	505182	3720 (0.01869)

Table 4. Checkpoint statistics. The last 3 columns report total checkpoints taken, average interval size (in instructions), and average checkpoint size (in bytes).

rectness, just as we did in Section 4. Figure 2 breaks down the outcome of our recovery experiments. For each benchmark, we report the recovery outcome for crashes from the physical register file, fetch queue, and IQ fault injections separately in a group of 3 bars labeled “R,” “F,” and “I,” respectively. Each bar is broken down into the same categories as Figure 1 minus the “Terminate” category (none of our recoveries end in early program exit). The last 3 groups of bars in Figure 2 report the average breakdowns for the multimedia, AI, and SPEC benchmarks, respectively.

Looking at Figure 2, we see some recoveries lead to correct program outputs even under architecture-level correctness (*i.e.*, the “Architecture” components). The 3 groups of average bars in Figure 2 show architecture-level correctness is achieved in 3.8% to 17.7% of recoveries on average for the multimedia and AI benchmarks, and in 22.5% to 31.0% of recoveries on average for the SPEC benchmarks. In these cases, there are no corruptions to uncheckpointed state between the rollback checkpoint and the crash; hence, lightweight recovery can enable program completion with numerically perfect outputs.

However, Figure 2 also shows that under application-level correctness, a significant number of additional crashes can be recovered (*i.e.*, the “Application-High” and “Application-Good” components), especially for soft computations. The first 2 groups of average bars in Figure 2 show application-level correctness permits an additional 34.8% to 73.8% of recoveries on average to be correct for the multimedia and AI benchmarks. Averaged across all hardware structures, an additional 52.6% of recoveries are correct under application-level correctness for the soft computations. G.721-D, LBP, and GA respond particularly well to lightweight recovery, with as many as 90% of crash recoveries achieving application-level correctness. In combination with numerically correct recoveries, these additional application-level correct recoveries allow 66.3% of all crashes on average to complete with acceptable results for soft computations. Furthermore, when combined with the results from Figure 1, our lightweight recovery mecha-

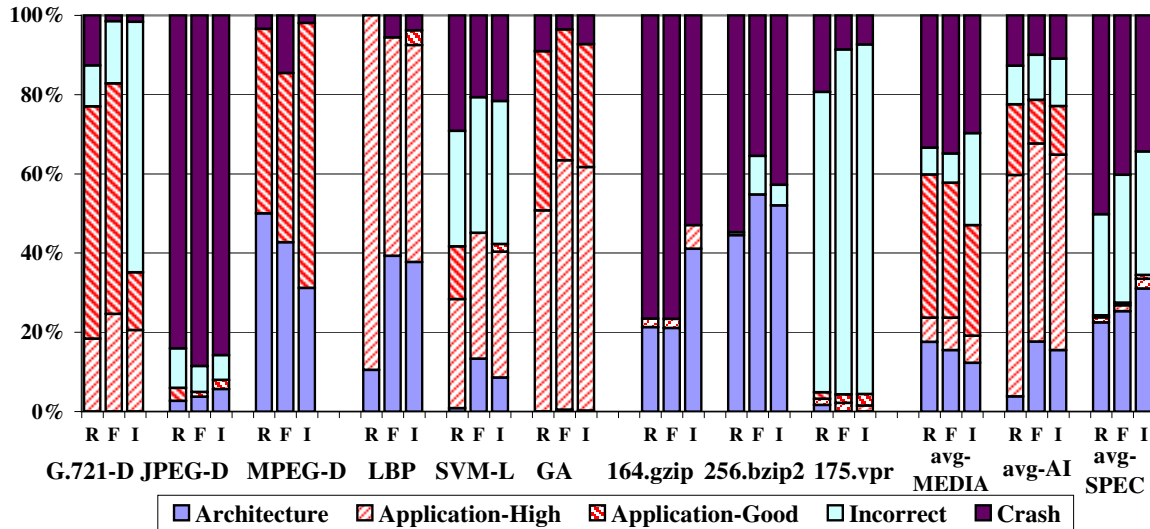


Figure 2. Program outcomes breakdown for lightweight recovery of crashes. The data is presented in a similar fashion to Figure 1.

nism allows 92.4% of all architecturally visible fault injections for soft computations to complete with correct outputs at either the architecture or application level.

Compared to soft computations, a much smaller number of crashes are recoverable for the SPEC benchmarks. The last group of average bars in Figure 2 show application-level correctness provides only 2.5% more correct outputs on top of the numerically correct recoveries. Nonetheless, when combined with the numerically perfect outputs, lightweight recovery still permits 28.7% of all crashes in SPEC on average to complete with acceptable results. And in combination with the results from Figure 1, lightweight recovery allows 71.2% of all architecturally visible fault injections for SPEC to complete with correct outputs at either the architecture or application level.

While lightweight recovery addresses a significant number of crashes, Figure 2 also shows a drawback. As already mentioned in Section 5.1, lightweight recovery cannot recover all crashes since it does not checkpoint all state necessary for program restart. The last 3 groups of average bars in Figure 2 show our technique fails to recover 45.1%, 22.2%, and 71.3% of crashes for the multimedia, AI, and SPEC benchmarks, respectively. Many of these failed recoveries lead to crashes; in these cases, we’re no worse off than we were without lightweight recovery. However, some failed recoveries complete and produce incorrect program outputs. The last 3 groups of average bars in Figure 2 show our technique leads to incorrect outcomes in 12.4%, 11.0%, and 29.6% of recoveries for the multimedia, AI, and SPEC benchmarks, respectively. Unfortunately, incorrect outcomes are potentially more problematic than crashes since they are harder to detect. For the “Incorrect” cases in Figure 2, lightweight recovery is arguably worse than no

recovery at all.

Although our recovery mechanism leads to some incorrect outputs, the incorrect cases are not that bad. We found for soft computations (*i.e.*, multimedia and AI), a significant number of the incorrect outcomes—between 80% and 90%—still exhibit good solution quality, and fall short of application-level correctness by only a small amount. In addition, for vpr, almost all the “Incorrect” cases are invalid solutions that are caught by the consistency check (as described in Section 2.2). Hence, they do not go undetected. Nevertheless, the successful recoveries provided by lightweight recovery come at the expense of a modest increase in the number of incorrect outcomes.

6 Related Work

Our work is related to the significant body of prior research on characterizing soft error susceptibility. Several researchers have injected faults into detailed CPU models to investigate soft error effects. Saggese *et al* [28] inject faults into a DLX-like embedded processor; Wang *et al.* [34] inject faults into a CPU similar to the Alpha 21264 or AMD Athlon; and Kim and Somani [14] inject faults into Sun’s picoJava-II. All of these fault susceptibility studies use gate- or RTL-level models, and inject faults into the entire CPU. In contrast, our study uses a high-level architecture model, and focuses fault injections on the register file, fetch queue, and IQ only. Additionally, some researchers have demonstrated many faults are *masked* and never become visible to software. Shivakumar *et al* [23] study masking at the *circuit level*; Kim *et al* [13] study logical masking; Mukherjee *et al* [18] identify *microarchitecture-level* and *architecture-level masking*; and Wang *et al* [33] study Y-branches, another source of architecture-level masking.

The main difference between our work and all previous studies on soft error susceptibility is the definition of correctness used to judge soft error impact. Previous work requires architectural state to be numerically correct for program execution to be correct. In contrast, our work only requires program outputs to be acceptable to the user. By evaluating correctness at a higher level of abstraction, we measure the additional soft errors that can lead to acceptable program outputs.

In addition to studying soft error susceptibility, several researchers have also exploited application-level error resilience. Like us, Thaker *et al* [31] observe many approximate algorithms can tolerate soft errors with only minimal solution quality degradation. They also show control computations are more vulnerable to faults than data computations, and develop tools to automatically distinguish the two. In comparison, we provide a more complete characterization of application-level error resilience through detailed architectural simulation. Also, while Thaker *et al* exploit error resilience to reduce redundant protection in the context of fault detection, we exploit the same to reduce checkpointing in the context of fault recovery. Breuer [3, 4] also recognizes multimedia workloads can tolerate errors, and proposes exploiting this to address manufacturing defects. Application-level correctness is similar to Breuer’s notion of “error tolerance” (ET) [4]. The main difference is Breuer exploits ET to tolerate hardware defects for higher chip yield, whereas we exploit application-level correctness to tolerate soft errors on functionally correct hardware.

Moreover, researchers have also exploited application-level error resilience to address security attacks and software bugs. Failure-oblivious computing [27] relies on bounds-checking code to catch memory errors due to security attacks before they can corrupt program state. Rather than throw an exception, execution is allowed to proceed past errors in the hope that the program can continue correctly. Rx [24] recovers failures due to software bugs, and re-executes them from checkpoints in a modified environment. By removing environmental factors that exercise bugs, Rx can run faulty programs to completion. Automated predicate switching [35] modifies program predicates to force execution down different control paths, thus correcting software bugs in program control flow. Similar to our work, these previous works observe programs can achieve acceptable results in the face of errors. However, while these previous works catch and/or correct errors, our work permits program corruptions to occur but tolerates them.

Other application-level error resilience research includes Liu *et al* [17] which observes certain image processing and tracking algorithms are inexact, and exploits this to improve task schedulability in real-time systems. Palem [21] exploits probabilistic algorithms to build randomized circuits

that are extremely energy efficient. Lastly, Alvarez *et al* [1] exploit the resilience to precision loss exhibited by multimedia applications to develop novel value reuse and energy reduction techniques for floating point operations. Compared to our work, none of these previous studies exploit error resilience for reliability purposes.

7 Conclusion

This paper explores definitions of program correctness that view correctness from the application’s standpoint rather than the architecture’s standpoint. Under *application-level correctness*, a program’s execution is deemed correct as long as the result it produces is acceptable to the user. To quantify user satisfaction, we rely on application-level fidelity metrics to capture program solution quality as perceived by the user. We conduct a detailed fault susceptibility study to quantify how much more fault resilient programs are at the application level compared to the architecture level. Across 6 multimedia and AI benchmarks, we find 45.8% of fault injections that lead to architecturally incorrect execution are correct under application-level correctness. Across 3 SPECInt CPU2000 benchmarks, we find 17.6% of architecturally incorrect faults produce acceptable results at the application level. Based on these results, we conclude a significant number of faults that were previously thought to cause erroneous execution are in fact completely acceptable to the user, especially for soft computations. In addition to studying fault susceptibility, we also present a lightweight fault recovery mechanism that exploits the relaxed requirements of application-level correctness to reduce checkpoint cost. Even though our lightweight recovery mechanism only copies 2 Kbytes of data per checkpoint on average, it successfully recovers 66.3% of program crashes in our multimedia and AI workloads. For SPECInt CPU2000, our technique recovers 24.3% to 34.5% of crashes, of which only 2.5% represent additional recoveries allowed by application-level correctness. This lower recovery rate is due to the fewer soft program outputs permitted by SPEC programs compared to soft computations.

8 Acknowledgements

The authors would like to thank Hameed Badawy, Steve Crago, Vida Kianzad, Wanli Liu, Janice McMahon, Priyanka Rajkhowa, and Meng-Ju Wu for insightful discussions on soft computing. The authors would also like to thank Ming-Yung Ko for help with the SVM-L benchmark.

References

- [1] C. Alvarez, J. Corbal, and M. Valero. Fuzzy Memoization for Floating-Point Multimedia Applications. *IEEE Transactions on Computers*, July 2005.
- [2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd annual IEEE/ACM Int’l Symp. on Microarchitecture*, Nov. 1999.

- [3] M. A. Breuer. Multi-media Applications and Imprecise Computation. In *Proc. of the 8th Euromicro Conf. on Digital System Design*, Sept. 2005.
- [4] M. A. Breuer, S. K. Gupta, and T. M. Mak. Defect and Error Tolerance in the Presence of Massive Numbers of Defects. *IEEE Design and Test Magazine*, May-June 2004.
- [5] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Tech. Report CS-TR-1996-1308, Univ. of Wisconsin - Madison, July 1996.
- [6] C. Chang and C. Lin. LIBSVM : a library for support vector machines. <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [7] I. S. Chong and A. Ortega. Hardware Testing For Error Tolerant Multimedia Compression based on Linear Transforms. In *Proc. of the 20th IEEE Int'l Symp. on Defect and Fault Tolerance in VLSI Systems*, Oct. 2005.
- [8] P. Dubey. Recognition, Mining and Synthesis Moves Computers to the Era of Tera. *Technology @ Intel Magazine*, Feb. 2005.
- [9] M. Gomaa and T. N. Vijaykumar. Opportunistic Transient-Fault Detection. In *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, June 2005.
- [10] Y. Jin. A Definition of Soft Computing. <http://www.soft-computing.de/def.htm>.
- [11] T. Joachims. Making Large-Scale Support Vector Machine Learning Practical. In *Advances in Kernel Methods: Support Vector Learning*. MIT Press, Cambridge, MA, 1998.
- [12] V. Kianzad and S. S. Bhattacharyya. Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, July 2006.
- [13] J. S. Kim, C. Nicopoulos, N. Vijaykrishnan, Y. Xie, and E. Lattanzi. A Probabilistic Model for Soft-Error Rate Estimation in Combinational Logic. In *Proc. of the Int'l Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems*, Italy, Sept. 2004.
- [14] S. Kim and A. K. Somani. Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy. In *Proc. of the 2002 Int'l Conf. on Dependable Systems and Networks*, Sept. 2002.
- [15] C. Lee. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of the 30th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 1997.
- [16] C.-C. J. Li and W. K. Fuchs. Catch – Compiler-Assisted Techniques for Checkpointing. In *Proc. of the 20th Int'l Symp. on Fault Tolerant Computing*, June 1990.
- [17] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise Computations. *Proc. of the IEEE*, Jan. 1994.
- [18] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factor for a High-Performance Microprocessor. In *Proc. of the 36th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2003.
- [19] S. R. Nassif. Design for Variability in DSM Technologies. In *Proc. of the 1st Int'l Symp. on Quality of Electronic Design*, March 2000.
- [20] U. of California at Berkeley. The Berkeley Initiative in Soft Computing. <http://www-bisc.cs.berkeley.edu/>.
- [21] K. V. Palem. Energy Aware Computing Through Probabilistic Switching: A Study of Limits. *IEEE Transactions on Computers*, Sept. 2005.
- [22] J. Pearl. Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann Publishers Inc., 1988.
- [23] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinatorial logic. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, June 2002.
- [24] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs As Allergies- A Safe Method to Survive Software Failures. In *Proc. of the 12th ACM Symp. on Operating systems principles*, Oct. 2005.
- [25] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. Aug. Design and Evaluation of Hybrid Fault-Detection Systems. In *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, June 2005.
- [26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. Aug. SWIFT: Software implemented fault tolerance. In *Proc. of the 3rd Int'l Symp. on Code Generation and Optimization*, March 2005.
- [27] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebe. Enhancing server availability and security through failure-oblivious computing. In *Proc. of the 6th Symp. on Operating Systems Design and Implementation*, Dec. 2004.
- [28] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer. Microprocessor Sensitivity to Failures: Control vs. Execution and Combinational vs. Sequential Logic. In *Proc. of the 2005 Int'l Conf. on Dependable Systems and Networks*, June 2005.
- [29] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Lifetime Reliability: Toward an Architectural Solution. In *Proc. of the 38th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Nov. 2005.
- [30] B. Taskar, M.-F. Wong, P. Abbeel, and D. Koller. Link Prediction in Relational Data. In *Proc. of Neural Information Processing Systems*, Dec. 2003.
- [31] D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. T. Chong. Characterization of Error-Tolerant Applications when Protecting Control Data. In *Proc. of the IEEE Int'l Symp. on Workload Characterization*, Oct. 2006.
- [32] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proc. of the 29th annual Int'l Symp. on Computer Architecture*, May 2002.
- [33] N. Wang, M. Fertig, and S. J. Patel. Y-branches: When you come to a fork in the road, take it. In *Proc. of the 12th Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2003.
- [34] N. Wang, J. Quek, T. M. Rafacz, and S. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proc. of the 2004 Int'l Conf. on Dependable Systems and Networks*, June 2004.
- [35] X. Zhang, N. Gupta, and R. Gupta. Locating Faults Through Automated Predicate Switching. In *Proc. of the 28th Int'l Conf. on Software Engineering*, May 2006.