Journal of Instruction-Level Parallelism 10 (2008) 1-28 Submitted 6/08; published 9/08

Exploiting Application-Level Correctness for Low-Cost Fault Tolerance

Xuanhua Li XLI@ENG.UMD.EDU
Donald Yeung YEUNG@ENG.UMD.EDU
Department of Electrical and Computer Engineering
University of Maryland at College Park
College Park, MD 20742 USA

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 two lightweight fault recovery mechanisms, *stack recovery* and *hard state recovery*, that exploit the relaxed requirements of application-level correctness to reduce checkpoint cost. *Stack recovery* recovers 66.3% of crashes in soft computations with near-zero runtime overhead, and *hard state recovery* recovers 89.7% of crashes in soft computations with half the runtime overhead of conventional incremental checkpointing under application-level correctness.

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 [1], wearout [2], and process variations [3]. 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 [4], and developing low-cost fault detection [5], [6], [7], [8] and recovery [9] 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 execute 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 abstraction levels, *e.g.* the application level, compared to lower abstraction levels, *e.g.* 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 [10]. These programs belong to a class of computations which we call *soft computations* [11], [12].[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. It is an extension of our previous work on the same subject [13]. 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

1. The 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.

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. We provide 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. (This fault injection study was also presented in our earlier work [13]). 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%, are correct at the application level.

In addition to studying fault susceptibility, we also present two lightweight fault recovery techniques, *stack recovery* and *hard state recovery*, that exploit the relaxed requirements of application-level correctness to reduce checkpoint cost, trading off correctness for performance. In particular, stack recovery only checkpoints the minimum state required to restart a program after a crash. In contrast, hard state recovery takes more comprehensive checkpoints to provide higher fault protection. Currently, our stack recovery technique is automated, but our hard state recovery technique relies on manual code inspection to identify what to checkpoint. For the multimedia and AI workloads, stack recovery can recover 66.3% of program crashes to application-level correctness with near-zero runtime overhead. Hard state recovery can recover 89.7% of program crashes to application-level correctness with half the runtime overhead of a conventional incremental checkpointing technique. While stack recovery was studied in our previous work [13], hard state recovery is introduced for the first time in this paper.

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, Sections 5. and 6. describe and evaluate our lightweight recovery techniques. Finally, Section 7. presents related work, and Section 8. 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 Sections 5. and 6., we will study two lightweight fault recovery techniques that avoid checkpointing data that only contribute to soft program outputs, thus reducing 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 [14], 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 [15], a 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 [16]. SVM-L is the learning portion of a Support Vector Machine algorithm, called SVMlight [17]. 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 [18]. 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 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

Benchmark	Numerical Output	Qualitative Output	Fidelity Metric
Multimedia			
G.721-D	Decompressed audio datafile	Perceived audio	Segmental Signal-to-Noise Ratio (SNRseg)
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.

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 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 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.–6., we define the baseline to be the result obtained from a fault-free execution of a 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.

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

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 (*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 [19], [7], [20] 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.* [7]. 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 [21], 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 mispeculated instruction.

3. For 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.

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

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 mispeculated. 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 mispeculated. 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.

Tables 3 and 4 present detailed information about our fault injection experiments for each of our benchmarks described in Section 2.. In Table 3, 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

Bench	Input	Exec Time	Interval
G.721-D	clinton.pcm	77643471	7000
JPEG-D	lena.ppm	44520776	7000
MPEG-D	mei16v2.m2v	40457756	7000
LBP	WebKB [16]	2175526139	1000000
SVM-L	a1a [22]	53981768	7000
GA	r16-0.1.in [18]	127490411	15000
164.gzip	input.compress	93396309	15000
256.bzip2	input.compress	732651712	250000
175.vpr	test	800450837	250000

Table 3: Benchmarks. "Exec Time" reports execution time in cycles. "Interval" reports the average time between fault injections.

	Fault Injections			Architecturally Visible Faults		
Bench	Regfile	Fetch	Issue	Regfile	Fetch	Issue
G.721-D	10467	10449	10440	483 (0.046)	581 (0.056)	1183 (0.113)
JPEG-D	5950	5998	5922	542 (0.091)	4341 (0.724)	1483 (0.250)
MPEG-D	5413	5423	4506	713 (0.132)	434 (0.080)	803 (0.178)
LBP	2198	2164	2176	1317 (0.599)	946 (0.437)	589 (0.271)
SVM-L	7225	7176	7154	1138 (0.158)	2327 (0.324)	1564 (0.219)
GA	8491	8410	8471	479 (0.056)	626 (0.074)	1352 (0.160)
164.gzip	6693	6550	6654	467 (0.070)	829 (0.127)	861 (0.129)
256.bzip2	2941	2907	2903	264 (0.090)	1559 (0.536)	722 (0.249)
175.vpr	3177	3152	3195	968 (0.305)	166 (0.053)	330 (0.103)
Total	52555	52229	51421	6371 (0.121)	11809 (0.226)	8887 (0.173)

Table 4: Fault injection statistics. The "Fault Injections" columns report the total number of faults injected into the physical register file, fetch buffer, and issue queue, respectively. The "Architecturally Visible Faults" columns report the number of injected faults in these three hardware structures that become architecturally visible.

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 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. In Table 4, the three columns labeled "Fault Injections" report the total number of injected faults for the physical register file, fetch buffer, and issue queue. Across all three hardware structures and all benchmarks, 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 [23], [24]. 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. 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 [4] arises due to faults that attack idle hardware resources, or hardware resources occupied by mispeculated instructions. The three columns in Table 4 labeled "Architecturally Visible Faults" report the number of faults injected into the physical register file, fetch queue, and IQ, respectively, that become architecturally visible. In parentheses, we report the same data as a fraction of the total faults injected into the hardware structure (*i.e.*, from the "Fault Injections" columns). As Table 4 shows, the degree of masking varies

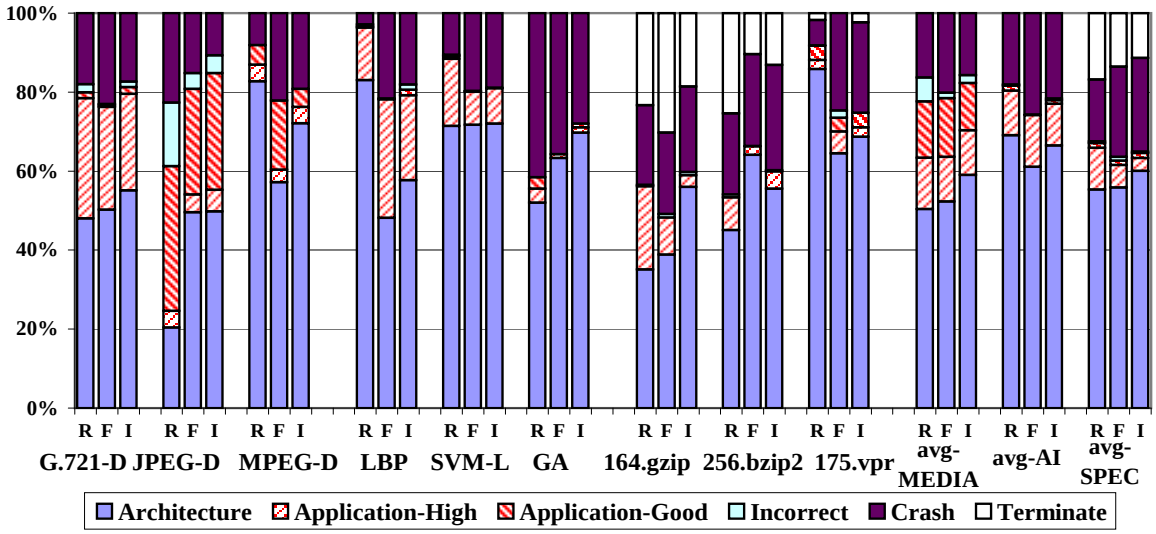

Figure 1: Program outcomes breakdown for architecturally visible fault injections: correct at the architecture level ("Architecture"), correct at the application level ("Application-High" and "Application-Good"), unacceptable ("Incorrect"), exception or program lockup ("Crash"), and early program exit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

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 [4]. 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 [25].

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 average bars in Figure 1 show, 27.3%, 26.1%, and 23.3% of all architecturally visible 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). In other words, 55.0%, 54.8%, and 56.8% of the architecturally incorrect faults (*i.e.*, excluding the "Architecture" components) are acceptable from the user's standpoint and achieve application-level correctness. As the second group of average bars show, 12.6%, 13.2%, and 11.4% of all architecturally visible faults for AI benchmarks occurring in the same three hardware structures, respectively, produce high or good fidelity program outputs as well. In other words, 40.4%, 33.8%, and 34.0% of architecturally incorrect faults 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, 11.7%, 6.8%, and 4.4% of all 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. In other words, 26.2%, 15.5%, and 11.1% of architecturally incorrect faults are correct at the application level. 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, LBP and SVM-L do not perform lossy operations. However, they are still highly error resilient due to the inexact nature of computing classification answers, as already discussed in Section 2.1.. While most soft computations are highly error resilient, one exception is GA. Upon closer examination, we found GA spends most of its time evaluating an objective function that reflects the cost of a given thread schedule. While GA's heuristic nature affords soft outputs (see Section 2.1.), unfortunately, its objective function evaluations are not soft computations, thus reducing the benefits of application-level correctness.

Our study also shows the SPEC benchmarks can tolerate faults. 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 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. Lightweight 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 system[4] while lockups can be flagged by a CPU watchdog timer. No additional hardware support nor runtime overhead need be incurred for detection. Recovery, on the other hand, requires checkpointing, a heavyweight operation that can incur significant runtime overhead.

In the remainder of this paper, we exploit application-level correctness to develop *lightweight fault recovery techniques*. Section 5.1. begins by introducing selective checkpointing of hard state, the main idea behind lightweight fault recovery. Then, Section 5.2. presents stack recovery, an extremely lightweight fault recovery technique, and Section 5.3. evaluates its performance. Later, Section 6. will extend stack recovery to improve its fault protection.

4. We 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.

5.1. Selective Hard State Checkpointing

Conventional fault recovery techniques use checkpointing to protect *all* program state. This comprehensive approach enables recovery to a numerically correct point in the program prior to any fault, thus unrolling the memory corruptions associated with the fault. While this is necessary for architecture-level correctness, it is overly conservative for application-level correctness. Under application-level correctness, fault recovery need only a). restart the program prior to the fault, and b). permit the program to complete with acceptable outputs; numerical correctness is not necessary. The key question is what minimum state must be protected to enable recovery for application-level correctness?

As discussed throughout this paper, faults that attack computations associated with soft program outputs can be tolerable from the user's standpoint. Not only are such computations error resilient, but the memory storing the soft program outputs these computations eventually corrupt are also error resilient. We refer to such memory as a program's "soft state." Moreover, we distinguish this soft state from all other program state, which we refer to as a program's "hard state."

Because soft state is highly resilient to data corruptions, in most cases, it can be omitted from checkpoints without sacrificing application-level correctness. On the other hand, using checkpoints to protect a program's hard state is necessary to permit program restart and completion after a fault. Hence, fault recovery can be made lighter weight by only checkpointing a program's hard state. By omitting soft state from checkpoints, checkpoint size and runtime overhead can be reduced.

5.2. Stack Recovery

To enable selective checkpointing, it is necessary to distinguish a program's hard state from its soft state. In general, this is a challenging task, but for one specific case, it is straight forward. As discussed in Section 5.1., one requirement of hard state checkpoints is to permit restart of the program prior to the fault. 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, an extremely naive (but effective) lightweight recovery technique is to periodically checkpoint the PC, architected register file, and program stack. While the register file and stack typically contain a mixture of both hard and soft state, it is unnecessary to further distinguish the hard state in these structures given the small amounts of data involved.

We call this simple technique *stack recovery*. Upon a crash, stack recovery restarts the program at the nearest checkpoint by rolling back the PC, register file, and stack only– stack recovery does not touch the program text, static data, or heap during rollback. To determine when checkpoints are taken, stack recovery identifies the main controlling loops in the benchmarks (usually the outer loops associated with major program phases), and instruments checkpointing at the top of each loop iteration. In this paper, we instrument the checkpoint calls manually, though it should be possible to automate this using compiler techniques [26]. Lastly, while checkpointing only needs to copy the state modified since the last checkpoint, stack recovery takes full checkpoints each time. This simple approach does not introduce significant overhead due to the very small size of the checkpoints.

Bench	Check	Interval	Size	Bench	Check	Interval	Size
G.721-D	261	1003622	566 (1.804%)	GA	300	1108510	1282 (0.003%)
JPEG-D	59	503137	3360 (0.397%)	gzip	252	964376	1108 (0.036%)
MPEG-D	60	2934834	664 (0.092%)	bzip2	1529	2019708	3462 (0.036%)
LBP	50	47197236	700 (0.012%)	vpr	2995	505182	3720 (1.869%)
SVM-L	430	404591	1944 (0.195%)				

Table 5: Stack checkpoint statistics. The columns labeled "Check," "Interval," and "Size" report total checkpoints taken, average interval size (in instructions), and average checkpoint size (in bytes).

Notice, stack recovery cannot successfully recover all crashes because it does not checkpoint any hard state outside of the PC, register file, and stack. Fortunately, as our results will show, stack recovery can still recover a significant number of crashes. In Section 6., we will revisit the topic of distinguishing a program's hard and soft state, and perform more comprehensive checkpointing of the hard state.

5.3. Stack Recovery Evaluation

We evaluate stack recovery 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 5 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 KB 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 using stack recovery. For every crash outcome in Figure 1, we rollback to the nearest checkpoint, as described in Section 5.2., 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 correctness, 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

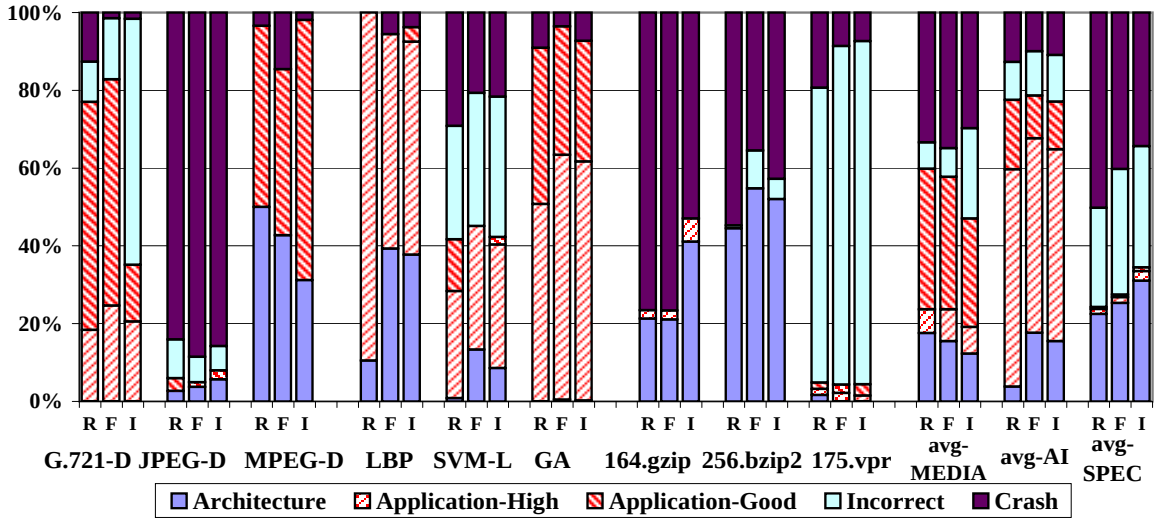

Figure 2: Program outcomes breakdown for recovery of crashes using stack recovery. The data is presented in a similar fashion to Figure 1.

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, stack 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 stack 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 stack recovery technique 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, stack 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, stack recovery allows 71.2% of all archi-

tecturally visible fault injections for SPEC to complete with correct outputs at either the architecture or application level.

While stack recovery addresses a significant number of crashes, Figure 2 also shows a drawback. As already mentioned in Section 5.1., stack recovery cannot recover all crashes since its checkpoints are not comprehensive. 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 stack 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, stack 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 stack recovery come at the expense of a modest increase in the number of incorrect outcomes.

6. Hard State Recovery

We now present our second lightweight fault recovery technique, *hard state recovery*, that extends stack recovery to provide higher fault protection. Like stack recovery, hard state recovery also performs selective hard state checkpointing. However, instead of only checkpointing the PC, register file, and stack, hard state recovery checkpoints hard state across the *entire program*. The following presents our technique in 3 parts. First, Section 6.1. discusses issues associated with checkpointing hard state program-wide. Then, Section 6.2. describes how we identify hard state to drive the selective checkpointing. Finally, Section 6.3. evaluates our technique.

6.1. Program-Wide Hard State Checkpointing

Similar to stack recovery, hard state recovery acquires full checkpoints of the PC, register file, and stack. While full checkpoints are fine for these small structures, they are impractical for other parts of a program (*e.g.*, static data and heap) given the potentially large amounts of data involved. For these other memory regions, hard state recovery employs *incremental checkpointing* [27]. In incremental checkpointing, a list of objects modified since the last checkpoint is maintained. At checkpoint time, only the dirty objects are checkpointed, thus eliminating redundant copies. After each checkpoint, the modified list is cleared to initiate tracking of dirty objects for the next checkpoint.

Due to fragmentation, the granularity of dirty object tracking impacts the size of incremental checkpoints. For simplicity, modifications can be tracked at page granularity, thus

Bench	Soft Program State
G.721-D	N/A
JPEG-D	cinfo->coef->MCU_buffer[], cinfo->upsample[]
MPEG-D	backward_reference_frame[], forward_reference_frame[], axframe[], substitute_frame[], llframe0[], llframe1[], lltmp[]
LBP	nodes[].belief[], nodes[].potential[], nodes[].messages[]
SVM-L	a_fullset[], xi_fullset[], lin[], learn_parm->svm_cost, model->alpha, last_suboptimal_at, selcrit, aicache, qp, qp.opt_ce, qp.opt_ce0, qp.opt_g, qp.opt_g0, qp.opt_xinit, qp.opt_low, qp.opt_up, weights
GA	ga_NewPop[].chrom[], ga_NewPop[].schedule[], ga_OldPop[].chrom[], ga_OldPop[].schedule[], A[]

Table 6: Data structures that store soft program state in the multimedia and AI benchmarks.

leveraging the hardware in TLBs for tracking dirty pages.[5] While this is cost effective, it incurs some overhead since most data objects are smaller than a page. Section 6.3. will quantify the impact of such fragmentation effects.

In addition to incremental checkpoints, another issue with hard state recovery is the hard state itself must be identified program-wide. Unfortunately, there is no simple heuristic–like checkpoint the stack–that can automatically identify all the hard state in a program. In this work, we identify hard state *manually* via code inspection. While this is impractical for most programmers, it permits us to conduct a preliminary study of hard state recovery, and quantify the *potential* benefits it can provide. Further research is needed to determine whether these benefits can be achieved in a more automated fashion. The next section describes how we acquire the hard state information.

6.2. Hard State Information

Because we adopt a manual approach to identify hard state, we do not expect our analysis to be complete, especially given the large size of some of our benchmarks. Hence, rather than identify hard state directly, we instead identify soft state, and assume all other program state is hard. That way, incompleteness in our analysis impacts performance (less soft state omitted from checkpoints) rather than correctness (hard state omitted from checkpoints). Also, we only analyze the soft computations since the SPEC benchmarks provide very little opportunities for selective checkpointing.

For the soft computations, we considered the soft program outputs discussed in Section 2.1., and inspected the code to identify the data structures associated with these soft outputs. We only inspected heap data structures since this is the main source of soft program state across our benchmarks. Table 6 lists the data structures we identified for each benchmark. For the multimedia benchmarks, the soft state are the arrays used to compute the decompressed datafiles listed in Table 1. Specifically, we identified the arrays in JPEG-D associated with its DCT computation, and in MPEG-D, we identified several arrays that store frame buffer data. (G.721-D does not contain heap data structures due to its very

5. Hard state recovery requires a separate dirty bit in the TLB since we must clear the bits after each checkpoint. Providing a separate dirty bit prevents interference with the operating system's ability to track dirty pages for virtual memory.

small memory footprint, so we did not identify any soft state for this benchmark). For the AI benchmarks, the soft state consists of data structures associated with the benchmarks' soft outputs listed in Table 1. In particular, we identified the arrays in LBP associated with its message-passing computation; in SVM-L, we identified several data structures associated with its quadratic programming algorithm; and in GA, we identified the arrays that store each population's chromosomes as well as the thread schedules the chromosomes encode.

Given the data structures listed in Table 6, we then identified the store instructions from each benchmark that write to these data structures. We refer to these as a program's "soft stores." We marked all soft store instructions in our benchmarks' binaries so they can be identified at runtime. As we will explain in the next section, this enables our hard state recovery technique to track dirty pages containing only soft state, and omit such dirty pages from checkpoints.[6]

6.3. Hard State Recovery Evaluation

Our evaluation of hard state recovery considers both runtime cost as well as recovery rate. To evaluate runtime cost, we modified our detailed out-of-order simulator from Section 3. to support incremental checkpointing. As described in Section 6.1., incremental checkpointing uses the TLB and OS to track dirty pages. Since our simulator does not model TLBs nor the OS, we track dirty pages in the simulator instead. For every executed store instruction, our simulator observes which page the store writes to (assuming a 4 KB page size). On a page's first write, the simulator appends the corresponding page number to a modified page list. So, at any given time, this modified page list specifies the state that must be incrementally checkpointed. (Although updates to the modified page list are not simulated, we do not expect this to impact our results. In an actual system, these updates would occur during TLB faults. Given the relatively low frequency of TLB faults and the low overhead for manipulating the modified page list in the TLB fault handler, we believe this bookkeeping overhead is negligible.)

When a checkpoint is taken, the modified page list is traversed and each dirty page is checkpointed using a copy function. (We instrument checkpoints in the same program loops identified for stack recovery, as described in Section 5.2.). We also checkpoint the PC, register file, and stack using the same copy function. While updates to the modified page list are performed in the simulator, the checkpoint copy function is performed on-line and fully simulated. In particular, we simulate all processor loads and stores executed within the copy function, including their associated cache misses. On average, we find it takes 1,560 cycles to checkpoint a single 4 KB page.

In addition to incremental checkpointing, we also modified our simulator to support soft store instructions. These instructions are created from unused opcodes in the PISA instruction set, and then inserted into our benchmarks' binaries everytime we determine a store writes to soft program state, as described in Section 6.2.. At runtime, our simulator recognizes the soft store instructions, and does not update the modified page list for such

6. It may be possible for some store instructions, particularly those associated with indirect writes, to store to both hard and soft state at different times in a program. Marking such "partially soft stores" as soft can compromise fault resilience since it can cause some hard state to be left out of checkpoints. Fortunately, we did not find any partially soft stores associated with the data structures we identified in Table 6.

Bench	Dirty Pages	Dirty Hard Pages		Dirty Blocks	Dirty Hard Blocks	
G.721-D	3	3	(100%)	13	13	(100%)
JPEG-D	14	7	(50%)	369	146	(40.0%)
MPEG-D	8	2	(25%)	216	20	(9.3%)
LBP	1444	1	(0.07%)	33633	1	(0.003%)
SVM-L	34	19	(55.9%)	578	359	(62.1%)
GA	46	38	(82.6%)	349	46	(13.2%)

Table 7: Selective hard state checkpoint statistics. The last 4 columns report each benchmarks' dirty pages, dirty pages containing hard data, dirty memory blocks, and dirty memory blocks containing hard data. The percentages indicate the relative size of selective hard state checkpoints compared to incremental checkpoints.

stores even if they are the first write to a given page. Hence, dirty pages that are written solely by soft stores are not checkpointed, thus facilitating selective hard state checkpointing.

6.3.1. Runtime Overhead

As in Section 5.3., we first run checkpoint-instrumented versions of our benchmarks to acquire all the checkpoints. But this time, instead of running them on our functional simulator, we run them on our detailed out-of-order simulator to measure the checkpointing cost. And instead of running our benchmarks only once, we run each of them twice: first to acquire checkpoints assuming no soft store instructions (*i.e.*, conventional incremental checkpoints), and then to acquire checkpoints with soft store instructions (*i.e.*, selective hard state checkpoints). For our checkpoint runs, we use the same checkpoint instrumentation described in Section 5.3., so the number of checkpoints acquired as well as the number of instructions between checkpoints is the same as in Table 5.

Table 7 shows the size of our checkpoints. The columns labeled "Dirty Pages" and "Dirty Hard Pages" report the average number of 4 KB pages in each conventional incremental checkpoint and selective hard state checkpoint, respectively, for the soft computations. As Table 7 shows, our selective hard state checkpoints contain 11.7 pages on average, or 46.7 KB, so they are larger than the stack checkpoints in Table 5. However, compared to conventional incremental checkpoints, our selective hard state checkpoints are 48% smaller on average. This demonstrates omitting soft state from checkpoints does significantly reduce their size.

Figure 3 shows the performance of our checkpointing techniques. In particular, we report the execution time of conventional incremental checkpointing and selective hard state checkpointing normalized to no checkpointing. The last group of bars reports the average across all the benchmarks. As Figure 3 shows, selective hard state checkpointing incurs 3.3% overhead on average over no checkpointing. This overhead is worse than stack recovery, which we estimate to be no more than 1% (see Section 5.3.). However, Figure 3 also shows conventional incremental checkpointing incurs a much higher overhead–10.5% on average. This demonstrates the reduced size of selective hard state checkpoints does translate into tangible performance benefits compared to conventional incremental checkpointing.

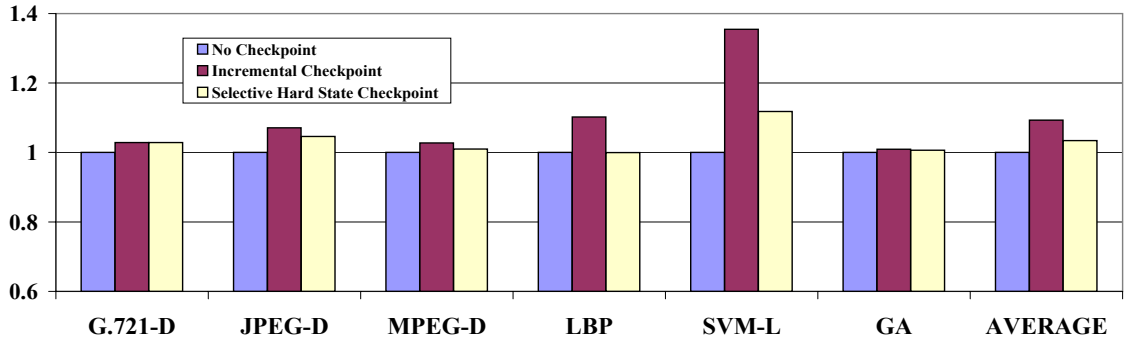

Figure 3: Normalized execution time of the original code, and the original code instrumented with incremental checkpointing and selective hard state checkpointing.

As discussed in Section 6.1., acquiring checkpoints at page granularity can suffer fragmentation. To quantify the impact of this problem, we modified our simulator to track and checkpoint dirty objects at *cache block granularity* (64 bytes). In Table 7, the columns labeled "Dirty Blocks" and "Dirty Hard Blocks" report the average number of cache blocks checkpointed under conventional incremental checkpointing and selective hard state checkpointing, respectively. Comparing these two columns, we see the selective hard state checkpoints are 63% smaller on average than the conventional incremental checkpoints. These results show an even greater potential exists for selective hard state checkpointing to reduce checkpoint size if checkpoints can be acquired at finer granularity.

6.3.2. Recovery Rate

After acquiring all the checkpoints, we perform recovery experiments using hard state recovery (we will discuss recovery using conventional incremental checkpointing at the end of this section). These experiments are performed on the functional simulator from Section 3. and are identical to the experiments in Section 5.3., except we use the selective hard state checkpoints for recovery instead of the stack checkpoints. Figure 4 breaks down the outcome of our recovery experiments. The format of Figure 4 is identical to Figure 2, except Figure 4 only evaluates the soft computations.

As in Figure 2, Figure 4 shows some recoveries lead to architecturally correct program outputs (*i.e.*, the "Architecture" components). Averaged across all hardware structures, architecture-level correctness is achieved in 48.3% and 31.3% of recoveries for the multimedia and AI benchmarks, respectively. Figure 4 also shows a number of additional crashes can be recovered to application-level correctness (*i.e.*, the "Application-High" and "Application-Good" components). Averaged across all hardware structures, application-level correctness is achieved in 33.0% and 66.7% of recoveries for the multimedia and AI benchmarks, respectively. In combination with numerically correct recoveries, these additional application-level correct recoveries allow 81.3% and 98.0% of all crashes for the multimedia and AI benchmarks, respectively, to complete with acceptable results. Furthermore, when combined with the results from Figure 1, hard state recovery allows 96.4% of all architecturally visible fault

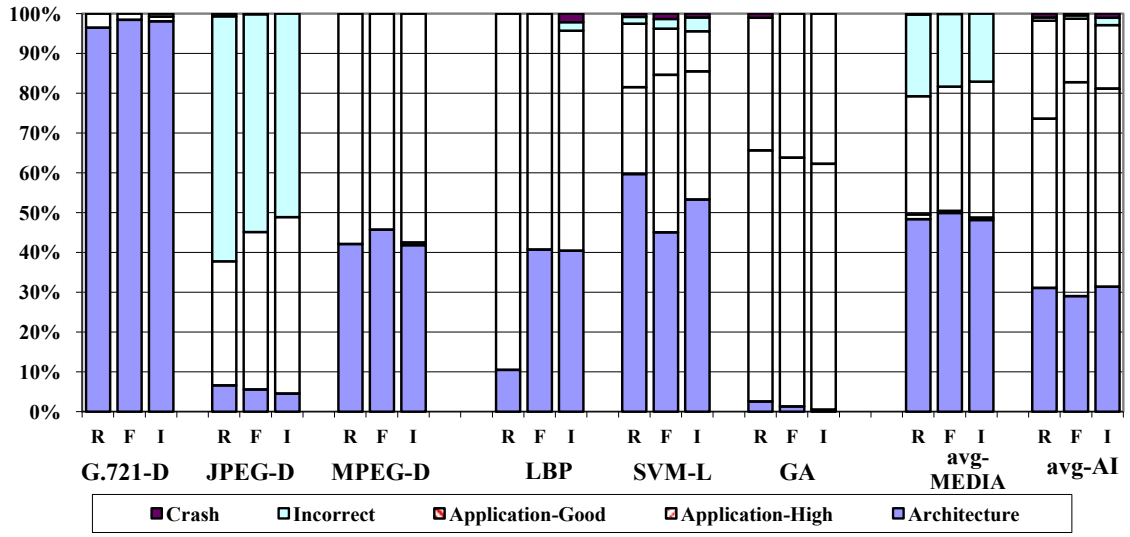

Figure 4: Program outcomes breakdown for recovery of crashes using hard state recovery. The data is presented in a similar fashion to Figures 1 and 2

.

injections for soft computations to complete with correct outputs at either the architecture or application level.

These results show hard state recovery is highly effective at recovering crashes, much more so than stack recovery. Comparing Figures 2 and 4, we see hard state recovery achieves application-level correctness for 89.7% of crashes averaged across all soft computations compared to only 66.3% for stack recovery. It is important to note, however, this increased fault resilience comes at the expense of some performance, as demonstrated in Section 6.3.1.. Hence, rather than claim hard state recovery is superior to stack recovery or vice versa, it is more accurate to say the two techniques simply achieve a different tradeoff between performance and recovery rate.

While hard state recovery is effective, it is not perfect. In particular, Figure 4 shows a very small number of recoveries (1.1% or less for all benchmarks) result in a second crash. For these rare cases, the fault and its recovery straddle a checkpoint; hence, the checkpoint itself is corrupted (more on this in a moment). In addition to a very small number of crashes, some recoveries lead to incorrect outcomes. Averaged over all hardware structures, Figure 4 shows 18.7% and 1.1% of recoveries for the multimedia and AI benchmarks, respectively, result in incorrect outcomes. Fortunately, similar to Figure 2, most of these incorrect outcomes–between 80% and 90%–still exhibit good solution quality, and fall short of application-level correctness by only a small amount. Nevertheless, this result shows comprehensive checkpointing of hard state alone does not guarantee correct execution. Faults to unprotected soft state can, in a few cases, degrade solution quality sufficiently to make the result unacceptable. Hence, by omitting soft state from checkpoints, hard state recovery cannot recover all crashes successfully, even under application-level correctness.

Compared to hard state recovery (and stack recovery), recovery using conventional incremental checkpointing will achieve a higher recovery rate as well as recovery to higher

	G.721-D	JPEG-D	MPEG-D	LBP	SVM-L	GA
Total Crashes	569	1147	304	174	891	806
Straddle	0	2	0	1	10	2
% Straddle	0%	0.2%	0%	0.6%	1.1%	0.2%

Table 8: Results for the number of crashes in which the fault and crash straddle a checkpoint.

solution quality. In this work, we do not perform recovery experiments for conventional incremental checkpointing. However, because incremental checkpointing checkpoints *all* modified state, it is guaranteed to recover a crash to architecture-level correctness unless the checkpoint itself is corrupted. As described above, this can occur when a fault and its subsequent crash and recovery straddle a checkpoint. This problem arises because our fault detection mechanism (waiting until a crash occurs before initiating recovery) has non-zero latency. Fortunately, we find the fault-to-crash latency is typically very small compared to the inter-checkpoint time, so corrupted checkpoints are extremely rare. Table 8 shows how rare. In Table 8, the row labeled "Total Crashes" reports the total number of crashes we try to recover in our hard state recovery and stack recovery experiments across the soft computations. For these crashes, the row labeled "Straddle" reports the number in which the fault and crash straddle a checkpoint, and the row labeled "% Straddle" reports the same as a percentage of the crashes from each benchmark. As Table 8 shows, at worst only 1.1% (and usually much fewer) of the crashes occur with the fault and crash straddling a checkpoint. In other words, when considering all the benchmarks, over 99% of the crashes are assured recovery to an uncorrupted checkpoint, and are thus guaranteed to complete with architecture-level correctness when using conventional incremental checkpointing. From this data, we conclude that conventional incremental checkpointing will successfully recover practically all the crashes.

7. Related Work

As mentioned in Section 1., this paper is an extension of our earlier work on application-level correctness [13]. Compared to our previous paper, this paper is the first to introduce hard state recovery, and to evaluate its potential benefits.

Besides our own work, this paper is also 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* [20] inject faults into a CPU similar to the Alpha 21264 or AMD Athlon; and Kim and Somani [19] 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. Other researchers have demonstrated many faults are *masked* and never become visible to software. Shivakumar *et al* [1] study *circuit level masking*; Kim *et al* [29] study *logical masking*; Mukherjee *et al* [4] identify *microarchitecture-level* and *architecture-level masking*; and Wang *et al* [25] 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* [30] 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 [23], [31] 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) [31]. 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 [32] 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 [33] 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 [34] modifies program predicates to force execution down different control paths, thus correcting software control flow bugs. 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* [35] which observes certain image processing and tracking algorithms are inexact, and exploits this to improve task schedulability in real-time systems. Palem [36] exploits probabilistic algorithms to build randomized circuits that are extremely energy efficient. And Alvarez *et al* [37] exploit the resilience to precision loss in multimedia applications to develop 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.

Finally, there have been studies in exploring ways to remove unnecessary state from checkpoints. Feldman *et al* [27] propose incremental checkpointing to only copy memory pages that have been modified since the previous checkpoint; Plank *et al* [38] propose memory exclusion techniques to remove dead and read-only memory regions from checkpoints.

8. Conclusion

This paper explores definitions of program correctness that view correctness from the user's standpoint rather than the architecture's standpoint. To quantify user satisfaction, we rely on application fidelity metrics to capture 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 SPEC 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.

Our work also presents lightweight fault recovery techniques that exploit application-level correctness to selectively checkpoint hard state. Our first technique, stack recovery, recovers 66.3% of program crashes in our soft computations. For the SPECInt CPU2000 benchmarks, stack recovery recovers fewer crashes, 24.3% to 34.5%, of which 2.5% represent additional recoveries allowed by application-level correctness. These recoveries are achieved with near-zero runtime overhead and no programmer intervention (*i.e.*, stack recovery is fully automated). Our second technique, hard state recovery, recovers 89.7% of program crashes in our soft computations with half the runtime overhead of conventional incremental checkpointing. Unfortunately, hard state recovery cannot be applied to the SPEC benchmarks because it requires manual identification of the selective checkpoints which we could not do for SPEC.

Based on these results, we make several observations. First, our lightweight fault recovery techniques are impractical for fail-safe systems since we cannot recover 100% of the faults. However, because our results show selective checkpointing can recover a significant number of faults (especially for soft computations), we conclude our techniques are useful for systems where simply improving reliability is desirable. In addition, our results clearly show a tradeoff exists between the degree of reliability improvement our techniques provide and the cost they incur in terms of performance loss and program analysis effort. (Stack recovery optimizes the cost side of this tradeoff, while hard state recovery optimizes the reliability side of this tradeoff). We believe stack recovery represents a more desirable point in this tradeoff space because it provides a non-trivial reliability benefit at very low cost to the user. In contrast, while hard state recovery provides a larger reliability benefit, it is useful only for soft computations and only in systems where the performance benefit justifies the programmer effort to identify the hard state. Hard state recovery would become more desirable if techniques to automate the identification of hard state were developed.

Finally, although it is beyond the scope of this work, an important question is to what extent do real workloads exhibit soft program outputs? Real workloads are typically more complex than the programs studied in this paper. One concern is whether the soft outputs our techniques exploit are as prevalent in real workloads as they are in the programs we study. If not, our lightweight fault recovery techniques may be less applicable to real systems. This is an important direction for future work.

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. 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.

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 combinatorial logic," in *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pp. 389–398, June 2002.

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Lifetime Reliability: Toward an Architectural Solution," *IEEE Micro*, vol. 25, pp. 70–80, May-June 2005.

[3] S. R. Nassif, "Design for Variability in DSM Technologies," in *Proc. of the 1st Int'l Symp. on Quality of Electronic Design*, pp. 451–454, March 2000.

[4] 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*, pp. 29–40, Dec. 2003.

[5] 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*, pp. 196–207, Nov. 1999.

[6] M. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, pp. 172–183, June 2005.

[7] 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 32st Annual Int'l Symp. on Computer Architecture*, pp. 148–159, June 2005.

[8] 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*, pp. 243–254, March 2005.

[9] 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*, pp. 87–98, May 2002.

[10] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology @ Intel Magazine*, pp. 1–10, Feb. 2005.

[11] U. of California at Berkeley, "The Berkeley Initiative in Soft Computing.".

[12] Y. Jin, "A Definition of Soft Computing.".

[13] X. Li and D. Yeung, "Application-Level Correctness and its Impact on Fault Tolerance," in *Proc. of the 13 Int'l Conf. on High-Performance Computer Architecture*, pp. 181–192, Feb. 2007.

[14] 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*, pp. 330–335, Dec. 1997.

[15] J. Pearl, "Probabilistic reasoning in intelligent systems: networks of plausible inference," Morgan Kaufmann Publishers Inc., 1988.

[16] B. Taskar, M.-F. Wong, P. Abbeel, and D. Koller, "Link Prediction in Relational Data," in *Advances in Neural Information Processing Systems 16*, MIT Press, 2004.

[17] T. Joachims, "Making Large-Scale Support Vector Machine Learning Practical," in *Advances in Kernel Methods: Support Vector Learning*, pp. 169–184, MIT Press, 1998.

[18] V. Kianzad and S. S. Bhattacharyya, "Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, pp. 667–680, July 2006.

[19] 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*, pp. 416–425, Sept. 2002.

[20] 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*, pp. 61–72, June 2004.

[21] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: the simplescalar tool set," Tech. Rep. CS-TR-1996-1308, Univ. of Wisconsin - Madison, July 1996.

[22] C. Chang and C. Lin, "LIBSVM : a library for support vector machines.".

[23] M. A. Breuer, "Multi-media Applications and Imprecise Computation," in *Proc. of the 8th Euromicro Conf. on Digital System Design*, pp. 2–7, Sept. 2005.

[24] 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*, pp. 523–531, Oct. 2005.

[25] 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*, pp. 56–67, Sept. 2003.

[26] 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*, pp. 74–81, June 1990.

[27] S. I. Feldman and C. B. Brown, "Igor: A System for Program Debugging via Reversible Execution," in *Proc. of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pp. 112–123, Jan. 1989.

[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*, pp. 760–769, June 2005.

[29] 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.

[30] 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*, pp. 142–149, Oct. 2006.

[31] 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*, pp. 216–227, May-June 2004.

[32] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee, "Enhancing server availability and security through failure-oblivious computing," in *Proc. of the 6th Symp. on Operating Systems Design and Implementation*, pp. 303–316, Dec. 2004.

[33] 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*, pp. 235–248, Oct. 2005.

[34] X. Zhang, N. Gupta, and R. Gupta, "Locating Faults Through Automated Predicate Switching," in *Proc. of the 28th Int'l Conf. on Software Engineering*, pp. 272–281, May 2006.

[35] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise Computations," *Proc. of the IEEE*, vol. 82, pp. 83–94, Jan. 1994.

[36] K. V. Palem, "Energy Aware Computing Through Probabilistic Switching: A Study of Limits," *IEEE Transactions on Computers*, vol. 54, pp. 1123–1137, Sept. 2005.

[37] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy Memoization for Floating-Point Multimedia Applications," *IEEE Transactions on Computers*, vol. 54, pp. 922–927, July 2005.

[38] G. K. James S. Plank, Micah Beck and K. Li, "Libckpt: Transparent checkpointing under unix," in *Proc. of the Usenix Winter 1995 Technical Conf.*, pp. 213–224, Jan 1995.