Appears in *Workshop on Architectural Support for Gigascale Integration*, Boston, MA. June 2006.

Exploiting Soft Computing for Increased Fault Tolerance

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

Abstract

Traditionally, fault tolerance researchers have made very strict assumptions about program correctness. Such strict notions of correctness are appropriate for workloads that are numerically oriented. However, a growing number of important workloads produce results that have a higher (often qualitative) user-level interpretation. We call these soft computations. *Examples of soft computations include multimedia processing, as well as cognitive information processing [1]. While data corruptions can change the numerical result of soft computations, they often do not change the user's interpretation of those numerical results. Hence, faults that would otherwise be deemed unacceptable from a strict numerical standpoint may in fact be tolerable (or even imperceptible) from the user's standpoint. Systems that can identify and exploit such error resiliency at the user level offer new opportunities for fault tolerance optimization.*

This paper makes several contributions in the context of fault tolerance for soft computations. First, we identify three characteristics of soft computations that make them resilient to error: redundancy, adaptivity, and reduced precision. Second, we quantify the extent to which soft computations are fault tolerant by conducting fault-injection experiments. Our results show that using relaxed notions of program correctness, 82% of injected faults are tolerable. Furthermore, when employing a lightweight recovery scheme enabled by soft computations, we find 96% of all injected faults are acceptably tolerated in our soft computing benchmarks. Finally, we present a method for identifying soft computations at the instruction level using dynamic slicing analysis. We find that in the benchmarks we study, 62% of all dynamic instructions participate in soft computations, and can tolerate single-bit errors up to a fault rate of 8×10^{-6}.

Keywords: Fault Tolerance; Soft Computing; Program Correctness; Fault Injection; Recovery.

1 Introduction

CMOS technology scaling along with voltage and clock frequency scaling have brought tremendous improvements in microprocessor performance. Unfortunately, these trends also make circuits more susceptible to transient faults due to high energy particle strikes (*i.e.*, soft errors), resulting in degraded system reliability [2]. Hence, there has been significant interest recently in developing techniques for improving reliability [3, 4, 5, 6, 7].

Fundamental to any reliability research is the definition of what constitutes correct program execution. In the past, fault tolerance researchers have made very strict assumptions about correctness. Typically, a program is said to execute correctly only when the produced architectural state is correct on a cycle-by-cycle basis. A looser (though still fairly strict) notion of program correctness commonly adopted by reliability researchers is that the visible memory state after program completion is correct in its entirety.

Such strict notions of program correctness are appropriate for traditional workloads that are numerically oriented. However, a growing number of important workloads produce results that have a higher (often qualitative) user-level interpretation. We refer to such computations as *soft computations*. An example of soft computation is processing of human sensory information common in multimedia workloads. Another example is cognitive information processing, an emerging application domain that applies artificial intelligence algorithms for reasoning, inference, and learning to commercial workloads [1].

While data corruptions can change the numerical result of soft computations, *they often do not* *change the user's interpretation of those numerical results.* Consequently, faults that would otherwise be deemed unacceptable from a numerical standpoint may in fact be tolerable (or even imperceptible) from the user's standpoint. Systems that can identify and exploit such error resiliency at the user level offer new opportunities for fault tolerance optimization.

In the past, researchers have observed soft computing characteristics and proposed to exploit them for reduced energy consumption [8, 9, 10] as well as for fault tolerance in ASIC design [11, 12]. In this paper, we study the potential for soft computations to increase fault tolerance in general-purpose CPUs. First, we identify three important characteristics of soft computations that make them resilient to error: *redundancy*, *adaptivity*, and *reduced precision*. Second, we quantify the extent to which soft computations are fault tolerant on general-purpose CPUs by conducting fault injection experiments. Our results show that 44% of the faults injected into our benchmarks are completely masked and do not alter the numerical results of the computations. However, an additional 38% of all injected faults are either imperceptible or tolerable from the user's standpoint. Third, we develop a lightweight recovery technique that tries to checkpoint and recover only "hard state" (*i.e.*, any state not involved in soft computations). Despite only recovering a small fraction of the program state (less than 1% of the entire program state in most benchmarks), our recovery technique can successfully recover 79% of fault injections leading to program crashes. With our lightweight recovery technique, our soft computations can successfully tolerate 96% of all injected faults. Finally, we present a method for identifying soft computations at the instruction level using dynamic slicing analysis. We find that in the benchmarks we study, 62% of all dynamic instructions participate in soft computations, and can tolerate single-bit errors up to a fault rate of 8×10^{-6}.

The remainder of this paper is organized as follows. Section 2 discusses characteristics of soft computations that make them particularly resilient to error. Then, Section 3 reports on our fault injection experiments, and presents our lightweight fault recovery technique. Next, Section 4 introduces our methodology for identifying soft computations at the instruction level. Finally, Section 5 presents related work, and Section 6 concludes the paper.

2 Soft Computations

This section discusses why soft computations are inherently resilient to faults (Section 2.1), and then presents a detailed illustrative example, loopy belief propagation (Section 2.2).

2.1 Sources of Fault Resiliency

Soft computations, in comparison to traditional numerically-oriented computations, exhibit an increased resilience to faults for two major reasons. First, soft computations permit a less strict definition of program correctness due to the qualitative nature of their results. Consider the following five definitions of program correctness, listed below in increasing strictness.

I. Architectural state is numerically correct on a per-cycle (or per multiple-cycle) basis.

II. Output state (*i.e.*, computation results visible at program completion or during system calls) is numerically correct.

III. Output state is numerically correct within some tolerance.

IV. Output state is qualitatively correct based on higher-level interpretation.

V. Output state is qualitatively correct based on higher-level interpretation within some tolerance.

Definitions I and II employ precise numerical measures of program state, and are commonly used to evaluate program correctness in existing fault tolerance research. The remaining definitions are less strict, and are appropriate for soft computations. Like the first two, definition III is numerical, but allows for some error. (The error may be tolerable because the results are computed at a greater precision than necessary). Definition IV applies to programs producing numerical results that have a higher-level interpretation. For example, an audio application may produce a signal consisting of a stream of numbers that is heard by an end-user. Or, an inference algorithm may compute a probability distribution function from which the program draws a conclusion. Finally, definition V also assumes a higher-level interpretation of numerical results (like IV), but allows for some error in the interpretation itself.

Soft computations exhibit increased error resilience because their results are not directly tied to precise numerical values. The results are qualitative (definitions IV and V) or imprecise (definitions III and V). Hence, errors in numerical computations may be tolerable, affording new opportunities for fault tolerance. Moreover, under definitions III and V, correctness ceases to be black or white; instead, we can speak of a "degree of correctness" determined by the amount of tolerable error. This opens the possibility to trade off answer quality for error resiliency.

One caveat is our correctness definitions consider output integrity only, disregarding other important factors. For example, we do not consider real-time constraints. While errors may not degrade solution quality, they might alter *when* solutions become available which can be unacceptable under certain circumstances. In addition, we do not consider system-level correctness. Errors that do not defeat individual soft computations may still propagate to other programs (either related or unrelated to the soft computations), causing them to crash or execute incorrectly. These are important correctness issues that we hope to address in future work.

In addition to relaxed program correctness, soft computations can also be error resilient due to special algorithmic properties. We have identified the following properties in several soft computations:

I. **Redundancy**. Soft computations that are iterative or that exhibit reduced precision (see below) often contain some degree of redundancy. Unlike dead code, these redundant computations contribute to the application result, but may not improve answer quality appreciably. Programs with redundant computations are more error resilient because the redundancy can mask faults.

II. **Adaptivity**. Many soft computing algorithms are already designed with error in mind. This is particularly common in algorithms that compute on noisy or probabilistic data. Such soft computations include code to detect certain forms of error, and adapt the computation accordingly. Due to their self-healing nature, adaptive codes are naturally error resilient.

III. **Reduced Precision**. Soft computations often have precision requirements that are lower than the datatypes supported by the programming environment / hardware architecture. These soft computations are resilient to errors that modify data values within the precision tolerance, as described earlier. Also, they are tolerant to errors whose magnitude decay as the errors propagate through the computation.

2.2 Loopy Belief Propagation

Loopy Belief Propagation (LBP) is a leading algorithm for approximate inference on graphical models. It is an extension of belief propagation (BP) [13], and is widely used in applications such as coding theory and combinatorial optimization. Our implementation of LBP, which we use for our experiments in Sections 3 and 4, performs classification.

LBP is a graph-based iterative message passing algorithm. Graph nodes represent *belief values*, while graph edges connect beliefs that are statistically related. At each iteration, neighboring graph nodes exchange messages based on their mutual belief values. Then, each node integrates all its received messages into its local belief to compute a new belief value. This process iterates until the entire graph converges.

Many of the soft computing characteristics discussed in Section 2.1 can be found in LBP. Foremost, LBP exhibits a loose definition of program correctness. For example, our LBP implementation computes a final set of belief values; however, its ultimate goal is to derive the class type for each belief value in the graph. Because different belief values can lead to the same class type, faulty executions of LBP can produce the same classification answer even though the belief values themselves are corrupted. Furthermore, because LBP performs approximate inference, the classification answer it computes may not be correct (compared to ground truth) even when the algorithm executes error-free. Hence, the user must be prepared to accept some baseline number of miss-classifications. Any additional miss-classifications the user is willing to accept beyond this baseline could be traded off to tolerate errors during belief computation.

LBP also exhibits all three of the algorithmic properties discussed in Section 2.1. LBP is redundant because many of the messages passed between graph nodes across different iterations are identical, especially when the graph nears convergence. Our experience shows roughly 2/3 of all messages can be dropped without affecting numerical convergence. These redundant messages enhance LBP's ability to absorb or recover from transient faults. LBP is also adaptive due to its iterative nature. Errors that are serious enough to prevent convergence cause the algorithm to simply execute additional iterations; these extra iterations often correct the errors and enable proper convergence. Lastly, LBP exhibits reduced precision. Beliefs, implemented using high-precision floating point values, only require a small amount of precision to numerically converge and yield the desired classification answer. Errors that propagate beyond this minimum level of precision do not cause miss-classifications.

3 Fault Resilience Results

In this section, we study the fault resilience of soft computations (Section 3.1), and then we present and evaluate a lightweight recovery technique enabled by soft computing (Section 3.2).

3.1 Fault Injection Experiments

We first study the overall fault resilience of programs by injecting a single fault into a program's execution and examining the resulting program behavior (*i.e.*, we assume a single event upset, or SEU, fault model). We conduct our fault-injection experiments using the in-order functional simulator from Simplescalar 3.2b. A single fault is simulated by picking a dynamic instruction and one of its source operands, and then flipping one of the operand's bits–all chosen randomly. The program is then run to completion, unless exceptions or program lockups occur. Program lockups are detected via expiration of watchdog timers that are set at the beginning of major loops in each program. If the program completes, result accuracy is assessed. We tried hundreds or thousands of such single-fault injection experiments for each program, depending on the size of the program. The number of faults injected appears beneath each benchmark in Figure 1, which will be explained momentarily.

By injecting faults directly into instructions' source operands, our faults have a high likelihood of flipping the bits necessary for architecturally correct execution, or ACE bits [14]. (The only unACE bits we flip are those associated with dynamically dead code). This effectively exposes program-level fault masking characteristics, the focus of our study. However, it is well-known that many faults can also be masked at the microarchitecture or architecture level [14]. Since our simulations do not account for these effects, our results tend to underestimate the number of errors that result in numerically correct execution when observing SEU faults on real processors. In the future, we plan to account for hardware-level masking.

Table 1 lists the benchmarks used in our experiments, consisting of two benchmarks representing cognitive information processing–LBP and SVM-L–and two benchmarks representing multimedia–G.721-D and JPEG-D. LBP is our loopy belief propagation algorithm described in Section 2.2. SVM-L is the learning portion of a Support Vector Machine algorithm, called SVMlight [15]. SVM-L learns the parameters for a support vector (SV) model on a training dataset. G.721-D and JPEG-D are audio and image decompression algorithms, respectively, from the Mediabench suite [16]. Both algorithms are lossy.

benchmarks	Numerical results	Qualitative results
LBP	Belief values	Node Classification
SVM-L	Support Vector (SV) Model	Input Dataset Classification using SV Model
G.721-D	Decompressed audio datafile	Segmental Signal-to-Noise Ratio (SNRseg)
JPEG-D	Decompressed image datafile	Peak Signal-to-Noise Ratio (PSNR)

Table 1. Benchmarks used in our experiments. Both the numerical and qualitative results computed by each benchmark are listed.

Table 1 also lists the numerical and qualitative results we evaluate to validate program correctness after each fault-injection experiment. As described in Section 2.2, the numerical results of LBP are the belief values, while the qualitative results are the class types derived from the belief values. The SV model parameters learned by SVM-L are its numerical results. SVM-L's qualitative result is the classification answer one would get using the learned SV model. To obtain this classification answer, we run the classifier provided by the SVMlight distribution using the learned SV model on an input dataset. (Note, the classifier is used only to obtain the qualitative result; we don't inject errors into the classifier). Lastly, the numerical results for G.721-D and JPEG-D are the decompressed datafiles each benchmark produces. The qualitative result is the signal-to-noise ratio (SNR) relative to the datafiles produced by the original lossy, but fault-free, decompression. We use the segmented SNR and peak SNR metrics for G.721-D and JPEG-D, respectively.

Figure 1 breaks down the outcome of all fault-injection experiments performed on each benchmark into 5 categories: program finishes with correct numerical results, program finishes with correct qualitative results, program finishes with "acceptable" qualitative results, program finishes with "unacceptable" qualitative results, and program crashes or locks up. Qualitatively acceptable is defined to be within 10% of the actual qualitative result; otherwise, the outcome is qualitatively unacceptable. Note that the first 3 outcomes represent "correct"

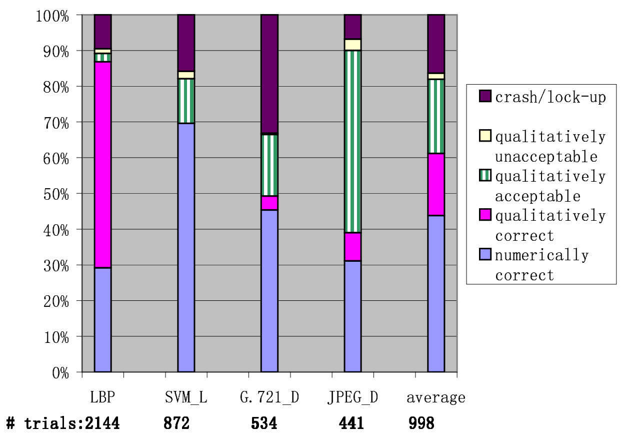

Figure 1. Breakdown of fault-injection experiment outcomes.

outcomes, and correspond to definitions II, IV, and V of correct program execution, respectively, from Section 2.1.

In Figure 1, we see 44% of all experiments on average finish with the correct numerical results. Another 17% on average finish with the correct qualitative results, and an additional 21% on average finish with qualitatively acceptable results. These experiments demonstrate the potential for increased fault tolerance in soft computations. 38% of injected faults (qualitatively correct + qualitatively acceptable) are tolerable thanks to the relaxed notions of program correctness afforded by our benchmarks. In addition, a great portion of the numerically correct outcomes (44% of injected faults) is due to the redundant, adaptive, and reduced precision properties from Section 2.1 exhibited by our benchmarks. For example, we observe that most of the numerically correct outcomes in LBP are due to these properties. We are currently in the process of quantifying the impact of these properties in the remaining benchmarks.

3.2 Lightweight Recovery

The last 2 outcome categories in Figure 1 (qualitatively unacceptable + crash/lock-up) represent "incorrect" outcomes, accounting for 18% of the fault-injection experiments. Of the two, the more significant is crashes and lock-ups (16%). Fortunately, soft computations can help simplify recovery of these terminating faults. Traditional approaches require checkpointing program state to enable unrolling corrupt data modifications that occur after a fault but before detection. Soft computations relax the need for heavyweight checkpoints since they are resilient to data corruptions, as demonstrated in Section 3.1. We still need to checkpoint noncorruptible data (*i.e.*, "hard state"), but this is usually much less than the entire program state.

We developed a lightweight fault recovery mechanism that exploits soft computing properties. Specifically, we checkpoint program execution periodically–immediately prior to certain function calls or loops. Rather than checkpoint all program state, we only checkpoint the processor state (program counter and register file) and the stack; we do not checkpoint the heap or static data. (We find that a majority of "hard state" resides in these memory structures, so checkpointing them is sufficient in most cases. We are currently studying ways to more precisely identify "hard state.") For the crash and lock-up outcomes in Figure 1, we rollback the processor state and stack to the most recent partial checkpoint and restart the computation. Notice we do not employ any fault detection mechanisms other than looking for crashes and lock-ups. Consequently, many faults may propagate before triggering recovery. This increases the potential for data corruption, but we rely on the soft computing properties of our benchmarks to tolerate these corruptions.

The cost of our lightweight checkpoints is very small: on average, our partial checkpoints are less than 1% of the entire program state (i.e., including the heap and static data). One exception is G.721-D. G.721-D has a small memory footprint; hence, the processor state and stack make up a significant portion of the overall program state. Each checkpoint in G.721-D saves 1/3 of the program state.

Figure 2 breaks down the outcome of all crash/lock-up cases from Figure 1 for each benchmark into 4 categories: program recovers with correct qualitative results, program recovers with "acceptable" qualitative results, program recovers with "unacceptable" qualitative results, and program cannot be recovered. In Figure 2, we see 79% of all crashes/lock-ups can be successfully recovered on average (qualitatively correct + qualitatively acceptable) by our lightweight recovery mechanism.

Combining the successfully recovered faults in Figure 2 with the successfully tolerated faults in Figure 1, we see our benchmarks can execute correctly (with acceptable results) in almost 96% of the fault-injection experiments. As discussed in this and the previous section, the high fault resilience comes mostly from the soft computing properties in our benchmarks.

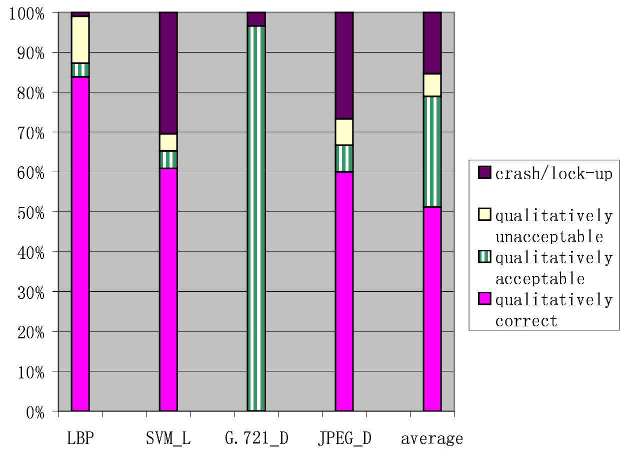

Figure 2. Breakdown of lightweight recovery outcomes for the crash/lock-up cases from Figure 1.

4 Soft Instruction Results

Section 3 demonstrates that soft computations are highly resilient to faults. In this section, we study the source of this fault resiliency *quantitatively* (in contrast to the qualitative discussion in Section 2.1). First, we identify fault resilient computations at the instruction level, and then we demonstrate how fault resilient specifically identified instructions are.

4.1 Identifying Soft Instructions

Soft computations permit looser definitions of program correctness because they compute on data values that are interpreted qualitatively, as described in Section 2.1. Such "soft data values" are approximate, inexact, or probabilistic in nature. However, not all code within a soft program deal with soft data values. This is why, for example, some of the fault injections in Section 3.1 lead to crashes. Soft computations are rarely entirely soft; instead, they consist of a mix of soft and exact computations.

We analyzed program execution at the instruction level to separate the instructions that participate in soft and exact computations. Starting from the soft data values in our computations, which we identified by hand, we perform dynamic backward slicing [17] to identify the subset or "slice" of instructions that participate in soft computations. Our dynamic backward slicer analyzes the following three instruction classes:

I. **Arithmetic**. All arithmetic instructions that compute on soft data are included in the slice.

II. **Memory**. Loads or stores of soft data are included in the slice. Instructions computing load addresses of soft data are also included in the slice because corrupted soft load addresses cannot corrupt exact state. However, instructions computing store addresses of soft data are not included in the slice because corrupted soft store addresses can corrupt exact state.

III. **Control**. Instructions computing conditional branch outcomes are included in the slice if the branch controls computations affecting soft data values only. However, instructions computing branch/jump target addresses are not included in the slice to ensure execution never deviates from the program's control-flow graph.

In addition to the above instruction classes, our slicer also includes dynamically dead instructions into the slice.

We modified the Simplescalar in-order functional simulator to perform backward slicing analysis at runtime. Table 2 presents the results of our slicing analysis. For each benchmark, Table 2 reports the size of the slice containing the soft instructions as a percentage of all dynamically executed instructions. We find that soft instructions are significant in the benchmarks we examine, accounting for about 62% of the dynamic instruction stream on average.

LBP	SVM-L	G.721-D	JPEG-D	average
64%	52%	57%	77%	62%

Table 2. Size of slice containing soft instructions in our benchmarks.

4.2 Selective Fault Injection

Now that we have identified the soft instructions, we next measure how fault tolerant they are. We conduct a similar set of random fault-injection experiments as Section 3.1; however, we limit our fault injections only to soft instructions, *i.e.* those identified by our slicer. During fault injection, two types of exceptions can occur: arithmetic exceptions, and memory exceptions (because we include load address computations in our slices). We modified our simulator to ignore exceptions, and to return a pre-defined constant value into any output registers whenever exceptions occur. Lastly, instead of injecting a single fault per program run as in Section 3.1,

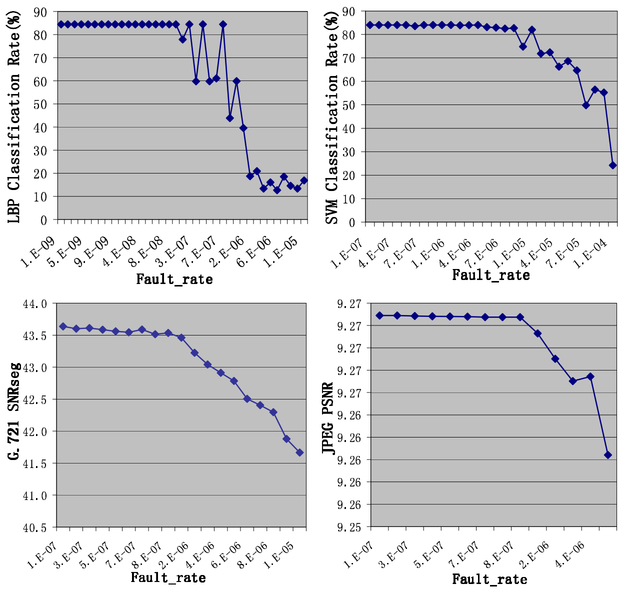

Figure 3. Fault resiliency of soft instructions for our benchmarks.

we inject multiple faults at some pre-defined fault rate.

Figure 3 plots correctness of the qualitative results from each benchmark (see Table 1) as a function of fault rate, with each point representing the average over 10 trials. All 4 benchmarks exhibit the same behavior. Correctness remains high despite numerous faults being injected into the benchmarks until some critical fault rate. The critical fault rate is different for each benchmark, but varies from 10^{-7} to 8×10^{-6}. Beyond the critical fault rate, correctness degrades. Figure 3 shows the soft instructions from each benchmark are highly resilient to faults. Moreover, we note that all of the experiments in Figure 3 complete, thus demonstrating that our slicer does in fact identify the soft instructions from each benchmark.

5 Related Work

This work is related to four categories of research. First, previous researchers have identified soft computations and observed their resilience to error. Specifically, Krishna Palem's work [8, 9] tries to reduce energy consumption in probabilistic algorithms, Carlos Alvarez's work [10] exploited soft computations in multimedia applications for value reuse and energy improvement, and Melvin Breuer's work [11, 12] tries to tolerate manufacturing defects and trasient faults for ASICs in multimedia algorithms. Compared to Palem's or Alvarez's work, we are focused on system fault tolerance rather than energy or performance. Compared to Breuer's work, we are interested in general-purpose CPUs rather than application-specific hardware. To our knowledge, we are the first to quantify the extent to which general-purpose systems are fault tolerant when exploiting soft computing characteristics.

The second related area is characterizing programs' susceptibility to transient faults. Kim and Somani [18] injected faults into an RTL model of Sun's picoJava-II. Wang et al. [19] injected faults into an RTL model of a CPU similar to the Alpha 21264 or AMD Athlon. In both of these previous studies, program correctness requires architectural state to exactly match with known error-free execution. In contrast, our work explores workloads that permit a looser definition of correctness, enabling new opportunities for fault tolerance optimization.

The third related area is identifying sources of fault-tolerance in programs. Previous research has noticed that not all transient faults are visible externally. Wang et al. [20] noticed that certain conditional branch outcomes can be wrong without affecting program correctness. Mukherjee et al. [14] found that certain bit corruptions at the microarchitectural and architectural levels are not visible. They found that NOP, performance-enhancing, and dynamically dead instructions lead to the fault resilience they observed. Our work differs in that we take algorithm-level effects into consideration, such as high-level definitions of correctness and redundancy, adaptivity, and reduced precision. Previous work considered only microarchitectural- or architectural-level effects. Our approach exposes more fault resilience by exploiting opportunities at the algorithm level.

Finally, in exploiting fault-tolerance information, Wang et al. [20] presented the performance speedup obtained by removing mispredictions on outcome-tolerant branches. Weaver et al. [21] developed techniques to avoid signaling false errors which occur on instructions unrelated to final results. Instead, we propose that soft computations, which provide significant fault tolerance, also enable lightweight recovery. Experimental results demonstrate that our recovery technique is very effective, despite the fact that only a small fraction of the program state is saved at each checkpoint.

6 Conclusion

This work investigates the impact of soft computations on fault tolerance. We propose that understanding the algorithms and how users interpret program results can provide significant opportunities for increasing fault tolerance. With relaxed definitions of program correctness, programs present much more resilience to transient faults. We quantify their fault resilience by conducting fault injection experiments. Our experiments show that by integrating the user's interpretation into the evaluation of program correctness, the number of trials that can generate acceptable results are increased by roughly a factor of two. We also develop a lightweight recovery technique that tries to checkpoint and recover only a minimal subset of program state. Overall, with our lightweight recovery mechanism, soft computations can successfully tolerate 96% of all injected faults. Lastly, we identify soft computations at the instruction level using dynamic slicing analysis, and find that soft instructions account for 62% of all dynamic instructions in the benchmarks we examine. Furthermore, these soft instructions can tolerate random single-bit errors up to a fault rate of 8×10^{-6}.

This paper is the initial study on exploiting soft computations for increased fault tolerance in general-purpose systems. In the future, we plan to continue our study on soft computations using more detailed processor models, for example RTL models. We are also interested in studying how soft computations can be exploited for performance in addition to fault tolerance.

7 Acknowledgements

The authors would like to thank Hameed Badawy, Steve Crago, Vida Kianzad, Wanli Liu, Janice McMahon, and Priyanka Rajkhowa 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. 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.

References

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

[2] 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 *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 389–398, June 2002.

[3] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 87–98, May 2002.

[4] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pp. 243–254, March 2005.

[5] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple instruction issue in the NonStop Cyclone processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 216–226, May 1990.

[6] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in *IEEE Transactions on Reliability*, pp. 63–75, March 2002.

[7] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and Evaluation of Hybrid Fault-Detection Systems," in *Proceedings of the 32st Annual International Symposium on Computer Architecture*, pp. 148–159, June 2005.

[8] K. V. Palem, "Energy Aware Algorithm Design via Probabilistic Computing: From Algorithms and Models to Moore's Law and Novel (Semiconductor) Devices," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 113–116, October 2003.

[9] K. V. Palem, "Energy Aware Computing Through Probabilistic Switching: A Study of Limits," tech. rep., September 2005.

[10] C. Alvarez and M. Valero, "A Fast, Low-Power Floating Point Unit for Multimedia," in *2nd Workshop on Application Specific Processors*, pp. 17–24, January 2003.

[11] M. A. Breuer, "Multi-media Applications and Imprecise Computation," in *Proceedings of the 8th Euromicro Conference on Digital System Design*, pp. 2–7, September 2005.

[12] 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.

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

[14] 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 *36th Annual International Symposium on Microarchitecture*, pp. 29–40, December 2003.

[15] T. Joachims, "Making Large-Scale Support Vector Machine Learning Practical," in *Advances in Kernel Methods: Support Vector Learning*, pp. 169–184, MIT Press, 1999.

[16] C. Lee, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proceedings of the 30nd Annual International Symposium on Microarchitecture*, pp. 330–335, December 1997.

[17] C. B. Zilles and G. S. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 172–181, June 2000.

[18] S. Kim and A. K. Somani, “Soft error sensitivity characterization for microprocessor dependability enhancement strategy,” in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 416–425, September 2002.

[19] N. Wang, J. Quek, T. M. Rafacz, and S. Patel, “Characterizing the effects of transient faults on a high-performance processor pipeline,” in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pp. 61–72, June 2004.

[20] N. Wang, M. Fertig, and S. J. Patel, “Y-branches: When you come to a fork in the road, take it,” in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pp. 56–67, September 2003.

[21] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, “Techniques to reduce the soft error rate of a high-performance microprocessor,” in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 264–275, June 2004.