

Characterization of Error-Tolerant Applications when Protecting Control Data

Darshan D. Thaker †, Diana Franklin ‡, John Oliver †,
Susmit Biswas ★, Derek Lockhart ‡, Tzvetan Metodi †, Frederic T. Chong★

†*University of California, Davis*

★*University of California, Santa Barbara*

‡*California Polytechnic State University, San Luis Obispo*

contact: ddthaker@ucdavis.edu

Abstract

Soft errors have become a significant concern and recent studies have measured the “architectural vulnerability factor” of systems to such errors, or conversely, the potential that a soft error is masked by latches or other system behavior. We take soft-error tolerance one step further and examine when an application can tolerate errors that are not masked. For example, a video decoder or approximation algorithm can tolerate errors if the user is willing to accept degraded output. The key observation is that while the decoder can tolerate error in its data, it can not tolerate error in its control.

We first present static analysis that protects most control operations. We examine several SPEC CPU2000 and MiBench benchmarks for error tolerance, develop fidelity measures for each, and quantify the effect of errors on fidelity. We show that protecting control is crucial to producing error-tolerance, for without this protection, many applications experience catastrophic errors (infinite execution time or crashing).

Overall, our results indicate that with simple control protection, the error tolerance of many applications can provide designers with considerable added flexibility when considering future challenges posed by soft errors.

1 Introduction

As the minimum feature size of process technologies continues to decrease, microprocessor designers are faced with new reliability challenges. Feature sizes of less than $0.25\mu\text{m}$ result in an increased likelihood of noise-related faults that are the result of electrical disturbances in the logic values held in circuits and on wires [1, 2]. Natural radiation such as neutrons produced by cosmic rays and alpha particles generate electron-hole pairs as they pass through a

semiconductor device. This may lead to transient faults that cause single bit upsets, which in turn may introduce a logical fault in the circuit. In addition, as the number of transistors increases, so does the complexity, which makes verification a much harder process, thus increasing the chance of undetected errors.

A considerable amount of recent research has focused on understanding how errors in low-level circuits manifest themselves in the architecture. Much of this research in error-tolerance has focused on preventing any errors from affecting the running program. One can run two copies of the program, utilizing simultaneous redundant multithreading to detect and correct errors. At the hardware level, reliable circuits can be constructed from error-prone components, but at the cost of increased circuit size and latency [3]. Unfortunately, this cost is not sustainable when applied uniformly [4].

This paper focuses on applications that exhibit some tolerance to reduced accuracy. In the embedded domain, such tolerance is often used to accommodate variations in quality of service in communication and network performance. We suggest that trends in technology and usage shall motivate “pushing” this tolerance into the microarchitecture. To do so, we must understand what effect errors due to the microarchitecture have on the application.

To manage this interaction between the microarchitecture and applications, we leverage the following key observation: computations involving control are much more sensitive to inaccuracy than others [5]. We propose using static analysis to identify instructions leading to control decisions. We present a set of applications that, when protected with our compiler, perform better in the face of errors. We quantify the benefit of this protection by defining application specific fidelity measures for relaxed accuracy requirements. Note that, although our focus is error-tolerant applications, our solutions are not application-specific.

We continue by describing the applications in Section 2.

Application	Description	Fidelity Measure
Susan	edge detection	Imagemagick comparison
MPEG	video encoding	% frames not dropped
MCF	vehicle scheduler	% extra time in schedule
Blowfish	encryption	% bytes correct from original
GSM	speech encoding/decoding	signal-to-noise difference
ART	image recognition	error in confidence of match

Table 1: Summary applications and their fidelity measures

We then present our static analysis in Section 3. Sections 4 and 5 describe our methodology and present our results. We place this in the context of recent work in Section 6. We conclude in Section 7.

2 Applications

We focus on application classes that do not require full accuracy to get their intended results. This can occur in several ways. First, applications that interact with human senses are very tolerant to slight inaccuracies. For example, phone lines do not carry sound perfectly, yet are sufficient for human perception. Second, there are numerical and search algorithms that expect to iterate until an adequate answer is attained.

In this study, we identify several applications that are tolerant of errors to varying degrees. All applications are part of SPEC CPU2000 [6] or MiBench [7]. Perceptual applications are more tolerant than decision-making applications. In order to evaluate each application, we define a fidelity measure for this application. This is typically some sort of distance from the optimal solution. For some applications, we have also defined a *fidelity threshold*, which is a subjective measure on how much inaccuracy a user would tolerate. We will briefly describe the applications we studied and the fidelities we defined for each one. The information is summarized in Table 1.

Susan

Susan is an application, from the MiBench suite, that implements the Smallest Univalued Segment Assimilating Nucleus (Susan) Principle, which performs edge and corner detection and structure preserving noise reduction. The Susan Principle is implemented using digital approximation of circular masks, (sometimes known as windows or kernels). If the brightness of each pixel within a mask is compared with the brightness of that mask’s nucleus, then an area of the mask can be defined which has the same (or similar) brightness as the nucleus.

We use Imagemagick [8] to determine the fidelity of the output. The *fidelity threshold* is 10dB PSNR, which means that the output after error insertion is considered bad if Imagemagick, after comparing the corrupted and correct images, returns a difference of 10dB or greater.

MPEG:

The MPEG standard, first proposed by the Moving Picture Experts Group in 1993, currently forms the most popular compression method for video and audio, and for streaming applications. Instead of encoding each frame individually, frames are encoded as the difference between themselves and the previous frame, allowing for much denser encoding.

There are three types of frames in an MPEG file, in order of importance they are: I frames, P frames and B frames. In general, the loss of B and P frames can be compensated for by the decoder, while the loss of an I frame will result in very noticeable quality degradation.

Our fidelity measure is the number of bad frames. A frame is considered bad if the SNR value compared to the correct frame is more than 2dB for I frames, 4 dB for P frames and 6 dB for B frames. The *fidelity threshold*, or the acceptable quality for viewers, is 10% of bad frames.

MCF is a benchmark from the SPEC 2000 integer benchmark suite. It is a single-depot vehicle scheduler for public mass transportation. Based on routes and desired frequencies of service, a schedule is determined. The schedule is determined using a network simplex algorithm which is a specialized version of the well known simplex algorithm for network flow problems. We measure the fidelity of the MCF schedule with errors inserted by comparing the schedules of an optimal schedule.

Blowfish is a symmetric block cipher with a variable length key. It was developed in 1993 by Bruce Schneier. Its key length can range from 32 to 448 bits. The

input data used is ASCII text file. The fidelity measure is the percent of bytes that match between the input(original) and the output data. The output data was obtained by decrypting the data obtained by encrypting input. The fidelity measure is the percent of similarity of the input ASCII data and the output ASCII data.

Adpcm or Adaptive Differential Pulse Code Modulation (ADPCM) is a variation of the well-known standard Pulse Code Modulation (PCM). The ADPCM encode/decode package included in this benchmark was implemented by Jack Jansen. The ADPCM encoder takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1. The decoder program converts the adpcm data to pcm data. The input data are small and large speech samples. ADPCM encode/decode have approximately 80% integer ALU operations and fewer than 10% branch operations for a very computation intensive operation. The fidelity measure is the percent of similarity of the output PCM data when errors were inserted with the output PCM data with no errors inserted. Though the input data is sample speech file, this metric was used instead of SNR as this benchmark does not treat data and header separately. Hence the generated output was not speech files. Instead they were ADPCM encoded data of sound file.

GSM is a benchmark from the MiBench suite in the telecommunications group. GSM communications is the communications standard that operates the majority of cell phones in Europe. A large speech sample is first encoded, then decoded. The fidelity measurement used is the signal-to-noise difference between the decoded output with errors inserted into the decoder, and the decoded output without error insertion. Typically, a 6 dB loss in signal for voice communications does not distort voice communications beyond recognition.

ART is a benchmark from the SPEC 2000 floating point benchmark suite. ART is a neural network utilized to identify items within an image. After the neural net is first trained on objects, it is provided with a thermal image. The thermal image is scanned with a window corresponding to the size of the learned objects. From this windowed imageage, the neural net attempts to identify the objects it has learned.

All of these applications have in common the fact that they can tolerate errors in just certain areas of the algorithms. For example, if the approximation algorithm were to exit the loop too early, that would constitute failure, not just increased execution time. Likewise, if the simulated annealing problem ended early, it might exit with a local max-

ima, not a global maxima. Worse, if the decision itself were corrupted, it might give the wrong answer altogether. It is important in our work that we analyze where the algorithms are tolerant to errors. Somewhat surprisingly, we find that we can perform our protection at the assembly level with an automatic compiler. The programmer identifies which functions can tolerate some error to their data, and the compiler tags instructions that do not affect the control operations.

3 Static Analysis

We found in a previous study[5] that protecting data used for control increases the fidelity of MPEG dramatically. In this section, we describe a simple data flow analysis technique for identifying Def-Use chains that lead to control flow decisions.

Our goal for performing data flow analysis is to identify *arithmetic instructions* that lead to a change in control flow. The technique we employ is used in contemporary compilers to determine *reaching definitions* [9], which enable optimizations such as loop-invariant code motion and copy propagation. We start at the last instruction of a basic block and move in the direction opposite program flow, tagging arithmetic instructions that *do not* influence control flow. We assume inter-procedural analysis.

We define elements of set **CVar** as variables likely to influence control flow. Any arithmetic instruction whose destination variable is not \in **CVar** is tagged. In addition, each instruction may *add* and/or *remove* elements from **CVar**. Instructions that directly influence control flow will add elements to **CVar**. Instructions that define (write to) variables \in **CVar** will both remove those defined variables and add to **CVar** the variables used in the definition. The process completes when set **CVar** becomes empty. Note that **CVar** may not be empty even after we consider all instructions in the current basic block. As a result, to complete this analysis, it may be necessary to cross basic block boundaries and even procedure boundaries.

The following example will demonstrate the ideas described above.

```
BB 0:
.
.
I0:  $2 = $4 + 1          *
I1:  LD $3, addr         []
I2:  $2 = $3 + 2         [$3]
I3:  $3 = $3 + 8         [$3, $2]
I4:  $10 = $8 - $4       [$3, $2] *
.
.
BB 1:
```

Algorithm	Errors Introduced	Total Instructions	% Failures With Protection	% Failures Without Protection
Susan	2200	144M	0%	10%
MPEG	20	2.74B	0%	100%
	120		0%	100%
MCF	1	201M	0%	100%
	340		6%	100%
Blowfish	2	507M	0%	10%
	20		19%	48%
GSM	10	892M	0%	100%
	40		0%	100%
ART	4	42.77B	0%	0%
ADPCM	3	324M	2%	8.5%
	56		8%	53.5%

Table 2: The percentage of catastrophic failures (infinite runs or crashes) with and without protecting control data.

```

I5:  $10 = $3 << $2      [$3, $2]
I6:  $4 = $3 + $6        [$3, $10] *
I7:  $3 = $3 + 1        [$3, $10]
I8:  BNE $3, $10, label [$3, $10]

```

The above code shows two basic blocks, BB0 and BB1. We wish to determine what instructions, from those shown, will affect the outcome of the branch that ends BB1. The square brackets after each instruction show the contents of set **CVar** after processing each instruction. (*BNE: branch if not equal, LD: load from address*) To start the analysis, we begin at the bottom of basic block 1, at instruction I8, and set **CVar** is empty. I8 generates elements \$3 and \$10 for **CVar**. I7 defines \$3, thus it will remove \$3. But I7 also uses \$3 thus adding it back to **CVar**. I6 does not change **CVar** and may safely be tagged. I5 will remove \$10 while adding \$2. I4 may be safely tagged. I3 behaves like I7. I2 defines \$2, thus removing it, finally set **CVar** becomes empty as a result of instruction I1. The instructions we tag as not influencing the branch in instruction I8 are I6, I4 and I0.

4 Methodology

There were two major elements to our experiments. The first is the static analysis, and the second is the simulation.

Static analysis was performed at the MIPS assembly level. Only functions that were user-identified as eligible were tagged. In order to be eligible, the data in the function must be in some way tolerable to error. A memory allocation function, for example, would not be eligible for tagging. A function that manipulates the data in an image would be.

Our compiler generated tagged executables that were then run on on SimpleScalar[10] environment for func-

tional simulation.

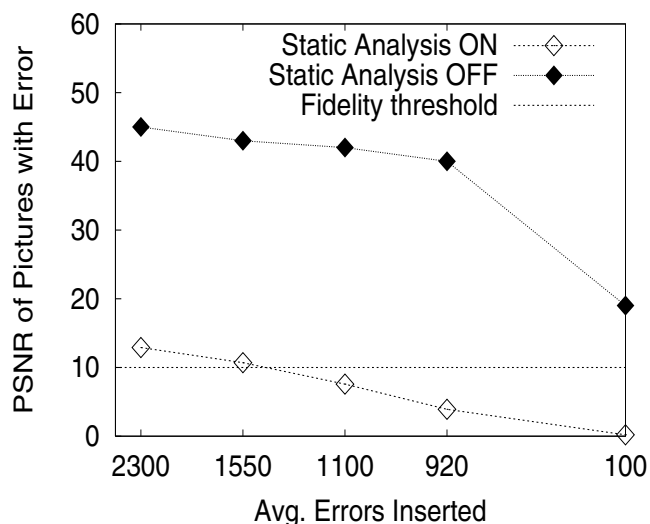


Figure 1: Susan Results

Error Insertion: In this paper, we do not attempt to study the masking effects of circuits or of the microarchitecture on soft-errors. We are interested in the behavior of the application when an error, a bit-flip, becomes visible to it. To model this, we flip a bit in the result of an instruction that was tagged as not influencing a control decision. We assume that untagged instructions will be protected in some way (e.g. redundant execution or extra hardware).

Single bit-flip errors were randomly inserted with a uniform distribution. Once an error was introduced in any instruction, it would propagate to all dependent instructions. The number of errors introduced for all applications was

much higher than current soft error rates. This was necessary to evaluate the change in fidelity as the error rate increased.

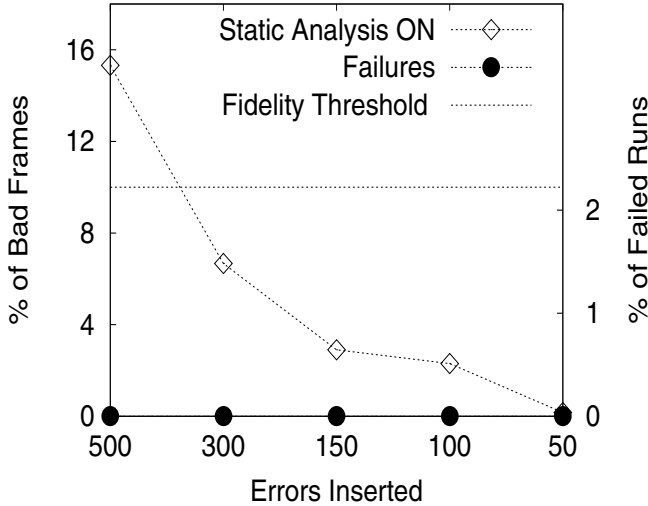


Figure 2: MPEG Results

5 Experimental results

Our purpose is to show that several benchmark applications are error-tolerant. We begin by showing that without protecting control data, even applications that were designed to tolerate errors experience frequent catastrophic failure. We continue by showing the degradation in fidelity as errors are introduced into the applications. Finally, we look at the potential in performance and/or cost by exploiting this error tolerance.

5.1 Protecting Control Data

Ideally, we would show the difference in fidelity between running the unchanged application in the presence of errors and running the application that has been tagged by our compiler. We found that without protecting control data, a very high percentage of the simulations failed, making such a comparison was unfeasible. Table 2 shows the percentage of simulations that ended in catastrophic failures for each application. Two data points are shown, one on each end of the number of errors introduced to that application. We show the lowest error rate for which the unchanged application failed for all simulations.

We find that without protecting control data, there is little or no error tolerance. This true even for applications like the MPEG decoder, which can work around inconsistencies in data such as loss of packets. We also see that even with our

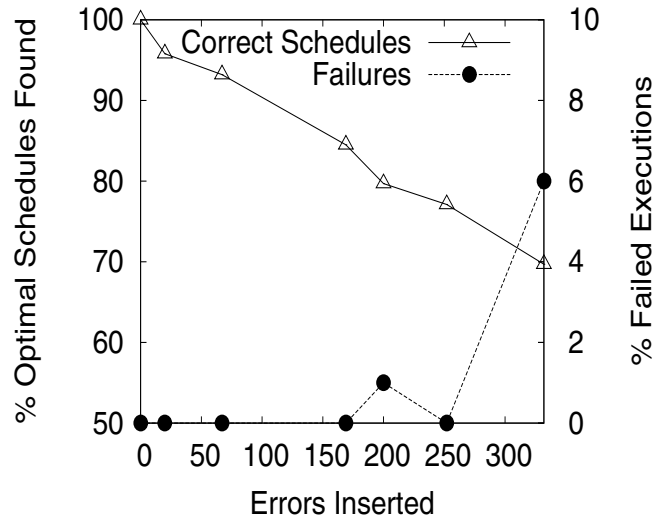


Figure 3: MCF Results

static analysis, some failures do occur. Because we perform no memory disambiguation, it is possible for an a value, tagged as not influencing control, to be written to memory and later to be read back from memory and used for a control calculation. If an error where introduced into such a value, it could lead to a catastrophic failure. So although we make great strides in protecting control data, we do not protect everything.

5.2 Error Tolerance with Control Protection

We begin with MPEG and Susan, whose results are shown in Figures 2 and 1. These applications were simulated with very high error rates because no fidelity was lost at lower error rates. We see that it takes more than 100 errors per second before Susan shows any frame loss due to the SNR being too low. In addition, although the unprotected execution suffers no catastrophic errors, the fidelity is substantially lower than with protection. MPEG has about 2% loss at 10 errors per second. It has no results without protection, since all simulations crashed. For Susan, disabling protection leads to very poor fidelity of output, however it does not crash the application. This can be attributed to the relatively small number of instructions (fewer than 9%) that are intolerant to error as shown in table 3.

In Figure 3 we see the results of the SPEC 2k benchmark MCF. The MCF benchmark performs quite well with errors inserted. Over 95% of schedules are still correct with 20 errors inserted into a simulation run of over 201M instructions. The incorrect schedules were all noticeably incorrect - although the application completed and printed results,

the schedules were not just inoptimal, but incomplete. So someone using the application would know immediately to rerun the application.

Figure 4 shows the effect errors had on the encryption application Blowfish. At 10 errors, the output is identical to the correct output. As we add more errors, however, the application begins experiencing catastrophic failures as well as a loss in precision. At 40 errors inserted, it fails 17% of the time, and its accuracy is only 84%.

The GSM application, shown in Figure 5, performs quite well when subjected to errors. Only a 95% of SNR, or 2dB loss of signal, is heard if 20 errors are inserted over a run of 892M instructions. Seven dB are lost with the insertion of 40 errors.

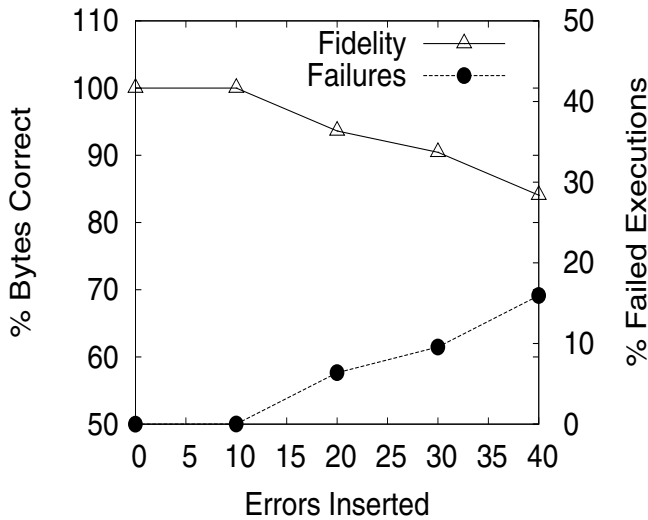


Figure 4: Blowfish Results

The ART application from SPEC 2k suite is more susceptible to errors. With only two errors inserted, only three quarters of the time did the benchmark correctly identify the object it was looking for in the thermal image. It never suffers from catastrophic error, however.

5.3 Future Potential

Fault-tolerance in itself is not our goal. We would like to exploit this property to provide faster or cheaper reliability. We could employ well-known reliability implementations to protect control data while running the rest of the instructions, labeled in Table 3 as low-reliability instructions, on cheaper or faster hardware. In order for this to be beneficial, a sufficient percentage of the execution must be on low-reliability instructions. The table above shows that Susan, our most error-tolerant application, needs to protect

Algorithm	Instructions	% Low Reliability Instr
Susan	144.3M	91.3%
MPEG	2.74B	50.3%
MCF	201.0M	8.9%
Blowfish	507.1M	62.4%
ADPCM	324.1M	93.26%
GSM	892.0M	19.6%
ART	42.77B	70.8%

Table 3: The number of instructions and percentage of dynamic instructions that our static analysis identified as not leading to control instructions. These instructions could be run in a low-reliability environment.

less than 10% of its instructions, making it the most promising application for exploiting this property.

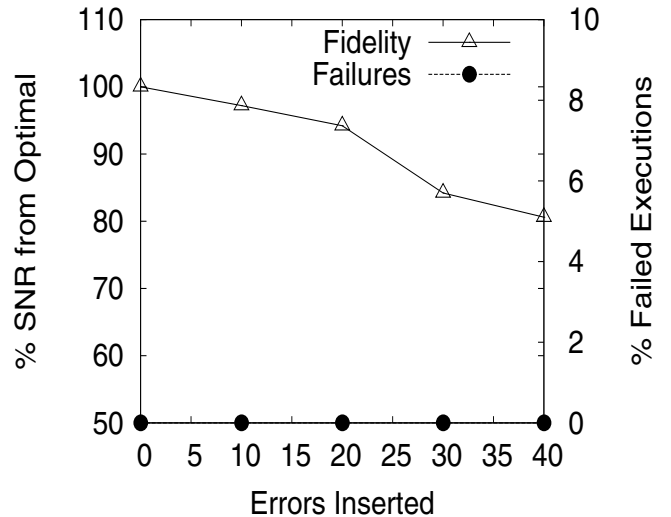


Figure 5: GSM Results

6 Related Work

Our work is related to many areas of error-tolerance. There has been recent work in reducing the cost of reliability, more accurately modeling soft errors, and providing applications that can tolerate errors.

Many groups have developed techniques to provide correctness in the face of failures. The RAMP architecture [11] dynamically adapts the reliability of the processor depending on the device properties through time. As different components fail, the architecture adjusts parameters to maintain the overall level of reliability desired. Weaver et al [12] developed techniques to reduce the probability of errors and

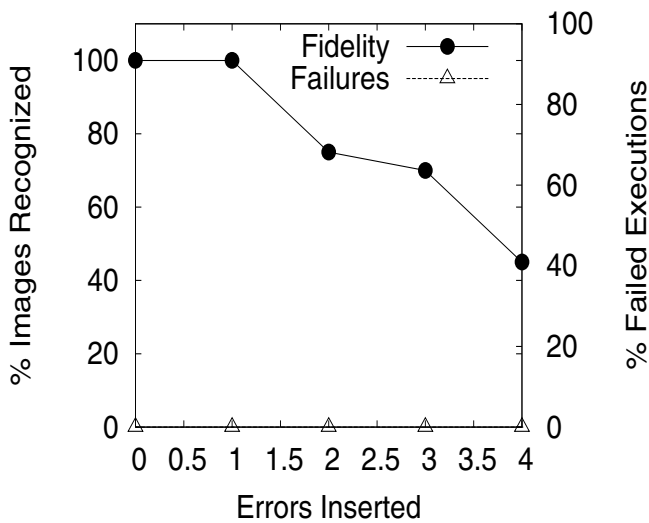


Figure 6: Art Results

reducing the impact of errors. They analyzed which structures were less reliable and reduced the time spent in those structures. Mukherjee et al [13] studied how to more accurately model errors in microarchitectures. Austin et al propose the Razor architecture [14], which uses “shadow latches” to check pipeline paths in the presence of voltage overscaling.

There is a growing movement to curtail the costs of reliability. Reis et al [15] propose allowing the user to switch between levels of reliability at the software level. That way, unimportant applications like web-surfing will not pay the costs, whereas banking applications would. We focus on allowing lower reliability *within* an application, as opposed to across applications. Kumar et al [16] reduce the resources necessary to redundantly execute instructions.

All of the above work is complementary to ours. Since we have shown that applications are more tolerant to errors than previously believed, we could selectively apply the techniques described above.

The theoretical foundations of this work are based in approximate signal processing [17] and flexible computation [18] which deals with systematic tradeoff between accuracy of results and the utilization of resources in their implementation. Video compression is a good application of approximate signal processing because the human perception system can inherently tolerate some inaccuracy. This can be exploited to create a hierarchy of importance in terms of the underlying data, in terms of the impact on distortion caused by a particular piece of data, which in turn can be used to make dynamic resource trade-offs, which in the case of video is typically the bit rate. Li and Yeung examine such error tolerance for two multimedia and two decision support

applications in [19]. Our work is similar in spirit, but we focus on compiler support to protect control data, which we first proposed in [5] and find in this study to be critical when examining broader application domains.

7 Conclusions

Our results indicate that error-tolerance in some applications offer significant potential for architectural optimization. However, it is imperative to protect control structures when executing in an unreliable environment. Even the most fault-tolerant applications, mpeg and susan, are very sensitive to errors that affect their control. We find that with static analysis, many applications can be protected such that their tolerance to errors is greatly improved. Moreover, the fraction of dynamic instructions related to control structures is often small when compared to overall execution. This indicates that only moderate effort is necessary for an architecture to protect these instructions through redundancy.

References

- [1] N.Cohen, T.S.Sriram, N.Leland, D.Moyer, S.Butler, and R.Flatley, “Soft error considerations for deep-submicron cmos circuit applications,” *IEEE International Electron Devices Meeting: Technical Digest*, pp. 315–319, December 1999.
- [2] J.F.Ziegler, “Terrestrial cosmic rays,” *IBM Journal of Research and Development*, vol. 40, pp. 19–39, January 1996.
- [3] J. V. Neumann, “Probabilistic logic and the synthesis of reliable organisms from unreliable components,” *Automata Studies, Ann. of Math. Studies*, vol. 34, pp. 43–98, 1956.
- [4] D. D. Thaker, F. Impens, I. L. Chuang, R. Amirtharajah, and F. T. Chong, “Recursive TMR: Scaling fault tolerance in the nanoscale era,” *IEEE Des. Test*, vol. 22, no. 4, pp. 298–305, 2005.
- [5] D. D. Thaker, D. Franklin, V. Akella, and F. T. Chong, “Reliability requirements of control, address, and data operations in error-tolerant applications,” *Proceedings of the Workshop on Architectural Reliability, held in conjunction with MICRO-2005*, December 2005.
- [6] J. L. Henning, “SPEC CPU2000: measuring CPU performance in the new millennium,” *IEEE Computer*, vol. 33, July 2000.

- [7] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown, "Mibench: A free, commercially representative embedded benchmark suite," pp. 3–14, December 2001.
- [8] www.imagemagick.com
- [9] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [10] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
- [11] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," *Proceedings of the 31st Annual International Symposium on Computer Architecture*, December 2004.
- [12] C. T. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Reducing the soft-error rate of a high-performance microprocessor," *IEEE Micro*, vol. 24, pp. 30–37, Nov 2004.
- [13] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 29, IEEE Computer Society, 2003.
- [14] T. M. Austin, D. Blaauw, T. N. Mudge, and K. Flautner, "Making typical silicon matter with razor," *IEEE Computer*, vol. 37, no. 3, pp. 57–65, 2004.
- [15] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled fault tolerance," *ACM Transactions on Architecture and Code Optimization*, vol. 2, pp. 366–396, Dec 2005.
- [16] S. Kumar and A. Aggarwal, "Reduced resource redundancy for concurrent error detection techniques in high performance microprocessors," in *HPCA '06: Proceedings of International Conference on High Performance Computer Architecture*, IEEE Computer Society, 2006.
- [17] S. Nawab, A. Oppenheim, A. Chandrakasan, J. Winograd, and J. Ludwig, "Approximate signal processing," 1997.
- [18] E.J. Horvitz, "Reasoning about beliefs and actions under computational resource constraints," *Third-Workshop on Uncertainty in Artificial Intelligence*, pp. 429–439, 1987.
- [19] X. Li and D. Yeung, "Exploiting soft computing for increased fault tolerance," in *Proceedings of the 2006 Workshop on Architectural Support for Gigascale Integration*, June 2006.