Compiler Techniques for Protection of Critical Instructions on Faulty Architectures

Konstantinos Parasyris, Vassilis Vassiliadis, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas

Dept. Of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
CERTH, Center for Research and Technology, Hellas, Volos,Greece
$\{$*koparasy, vasiliad, cda, lalis, nbellas*$\}$@*inf.uth.gr*

Abstract—**Conventional computer systems are designed to deliver error-free operation. However, this strict correctness prequisite is threatened due to the continuous efforts towards denser structures which are vulnerable to voltage and temperature fluctuations. In such conditions, errors occur due to timing violations.**

In this paper, an LLVM-based, compile time analysis is described which categorizes instructions according to their criticality to program correctness. We quantify the positive effects on application resiliency when protecting critical instructions from producing erroneous results. Moreover, we study the effects of compiler optimizations on the number of critical instructions for Intel's x86 architecture. As a general rule, compiler optimization increase the number of instructions that are non-critical to program execution, and which can be executed on less-reliable hardware.

I. Introduction

Conventionally an application execution is considered as correct when all bits of the micro-architectural state are correct in every clock cycle. A more relaxed definition of correctness requires that only the architectural state of the CPU be correct in every clock cycle. In all cases though, the strict prerequisite is bit-wise correctness (in bibliography it is referred as architectural correctness).

Recent technology trends suggest that strict adherence to bit-level accurate execution may not only be unnecessary, but also redundant and wasteful. Namely, the significant energy cost of guard-bands on the operating frequency or circuit supply voltage to guarantee error-free operation even when subjected to worst-case combination of process, voltage and temperature (PVT) non-idealities, as well as the continued efforts towards even denser structures, has pushed researchers towards relaxing strict enforcement of precise hardware functionality. This push towards approximate computing is still in experimental phase, and has not yet been adopted by the industry.

While hardware unreliability can be handled via traditional fault-tolerance approaches, such as replication or checkpointing and replay [13], these methods have disadvantages. Running multiple replicas of the same task on different cores requires significantly more computing and energy resources. On the other hand, the construction of checkpoints and the replaying of tasks may slow down the execution of the computation substantially. Also, both approaches will not work if unreliable cores malfunction in a deterministic way, as recent work [11] suggests when trying to scale voltage below nominal V_{dd} values.

Interestingly, there are many application domains which appear to execute correctly from a user perspective, however the execution is not 100% correct when using the strict aforementioned correctness definition. This is referred to as application-level correctness. Such application domains include multimedia, applications with self-healing properties (e.g. iterative numerical applications), applications based on probabilistic computations (e.g. Monte Carlo, classification), etc. In multimedia applications, small errors in the pixels of an image are visually imperceptible. Likewise, iterative solvers can converge to the desired solution even in the presence of errors, albeit requiring additional iterations. Finally, in probabilistic applications the notion of error is embedded in the code and during execution the application adapts to soft errors.

Moreover, as shown by previous work on approximate computing [12], such applications may include computations or execution phases with an unequal contribution to the quality of the output result. In fact, the output may remain the same even if some parts of the computation produce incorrect results.

Nevertheless, all applications contain certain instructions which should always be executed correctly, even if they reside in an approximate part of the application. Pointer arithmetic instructions or instructions that may modify the control flow of the program are primary candidates. Such instructions are critical to the correct execution of the program, even when considering the relaxed definition of program correctness and should be protected to guarantee normal termination. Hardware mechanisms which are able to detect and correct faults due to timing violations have been proposed in [5], [6]. Those mechanisms try to contain hardware faults and to present an error-free execution engine to the software. On near threshold computing or even on sub threshold executions protecting critical instructions is a primary concern since any error on such instructions will almost certainly result to application failures.

In this paper, we perform an experimental evaluation on the effects of hardware faults on instructions that are critical to the execution of a program. We show that by deploying hardware mechanisms to protect the execution of critical instructions, we eliminate almost all failures due to these faults even in complex architectures such as Intel's x86.

The main contributions of this paper are: 1) a compile time analysis technique, which identifies and tags such instructions as critical. This work is based on the LLVM [8] compilation infrastructure. 2) Comparison of the resiliency of the application when critical instructions are not protected versus when they are protected by the hardware. 3) Experimental evaluation to identify the influence of compiler and manual optimizations to the number of critical instructions.

The rest of the paper is structured as follows. Section II briefly describes the compile time analysis. In Section III, we evaluate and discuss our findings. Section IV presents related work. Finally, section V concludes our paper and presents directions for future work.

II. Compiler Analysis

In Listing 1 we present a simple snapshot of a vector addition in *MIPS*. Line *6* of the assembly implements the actual addition. The *add* instruction does not perform any memory operations and does not alter the control flow. Lines *1, 3, 11*, and *12* correspond to either control flow instructions or to instructions that can modify the control flow. All remaining instructions perform pointer arithmetic or access the memory. Protecting all instructions from faults in hardware, at execution time, might be unreasonable due to significant performance and power overheads. Moreover not all instructions are created equal. Errors impacting pointer arithmetic instructions may result to program failures, (application fails to terminate due to harware or OS trap) more frequently than faults impacting processing instructions. The same applies for instructions controlling control flow.

Should someone compare the importance of instructions in relevance to application resiliency, instructions processing data should be the least important. Instructions operating between data might mask a fault, or in any case they rarely result to program failures. Therefore, protecting such instructions in the hardware may result to unnecessary waste of resources since errors might never manifest at the end result.

```
1    add $s1, $0, $0
2  for:
3    beq $s0, $s1, end
4    lw $t2, ($s2)
5    lw $t3, ($s3)
6    add $t4, $t3, $t2
7    sw $t4, ($s4)
8    addi $s2, $s2, 4
9    addi $s3, $s3, 4
10   addi $s4, $s4, 4
11   addi $s1, $s1, 1
12   j for
13 end:
```

Listing 1: Vector add used a simple example.

Distinguishing the instruction type in the hardware level might result to interesting research directions. For example, an opportunity would be to trade off the applications quality of output with performance and power saving by protecting only instructions performing pointer arithmetic and control flow information. This section presents an *LLVM* compiler pass that detects critical instructions in an application. Such instructions should be error-free.

A. Compiler Critical Instruction Identification Analysis

The analysis is similar to an *upward exposed uses* analysis[1]. Starting from the last basic block and traversing the instructions in reverse execution order we identify obvious *critical instructions*. Obvious critical instructions should meet one of the following criteria :

1) **Class I**: During the execution of the instruction an address calculation is performed. For example the *lw* instruction of the *MIPS* architecture. 2) **Class II**: The instruction has implicit or explicit impact on the control flow of the application. For example a branch instruction has explicit impact on control flow whereas a compare instruction has implicit impact if the result of is used in the branch.

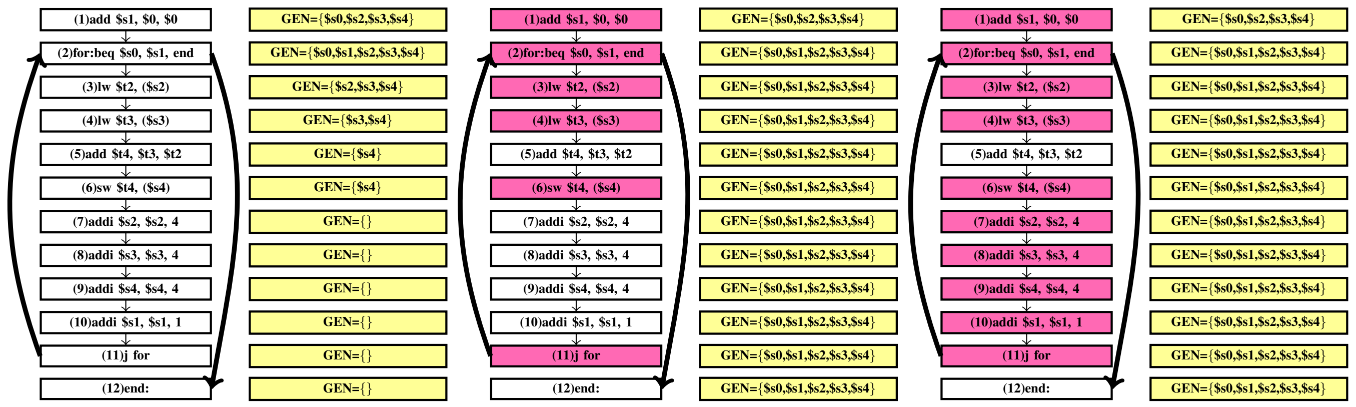
These instructions process critical information, such as memory addresses or control flow. Our analysis examines instructions producing the input operands of obvious *critical instructions* to identify sequences of critical instructions that need to be executed reliably.

Obvious critical instructions are tagged as critical and depending on criteria met by each instructions some of the operands used (*uses*) to compute the definition (*def*) of this instructions are pushed to a bit vector, called *GEN*. The vector size is equal to the number of different registers supported by the architecture. If the instructions are in **Class I**, only the operands participating in the address calculation are pushed to the *GEN* vector. If the instruction is in **Class II**, all operands are pushed in the *GEN* vector.

When traversing an instruction we check whether it defines a value contained in the *GEN* vector. If this is the case, the instruction is tagged as critical, the definition is removed from the vector and the uses of the new critical instruction are pushed into the *GEN* vector.

When reaching the entry point of the basic block the *GEN* vector contains all the values x which are used by a critical instruction s inside the basic block, and there is no definition of x between s and the beginning of the basic block. After the procedure traverses the entire block, it propagates the information to all the predecessors of this block using the union operator. We apply this operator to the analyzed code iteratively until there are no changes in the *GEN* set. The analysis continues on the basic blocks of the function until there is no change between consecutive iterations.

[1]Upward exposed uses: For each definition of a variable, find all uses that it reaches



(a) Initial State (b) Iteration 1 (c) Iteration 2

Figure 1: A simple example of the compiler analysis pass for 3 iterations of the algorithm

Figure 1 shows a simple example. The analysis starts from the last basic block (node 12) and the GEN set is empty. The analysis continues by processing instruction 11 which is the last instruction of the next basic block. Instruction 11 is tagged as critical since it is a control flow instruction. Instruction 11 has no operands therefore the GEN set remains empty. Instruction 10-7 are not obvious critical. Instruction 6 performs address calculation since it is a store word. The instruction is tagged as critical and the operands of the instruction that contain addresses are stored inside the GEN vector. The next instruction is not an obvious critical one and does not define a register contained in the GEN vector, hence the instruction is not recognized as critical. Instructions 4,3 and 2 are identified as critical since they perform address calculations and branching. All operands used by these instructions are added in the GEN set. Finally the analysis moves to the first basic block and identifies instruction 1 as significant, because it sets a value to register $\$s1$ which is inside the GEN vector. After the instruction is processed register $\$s1$ is removed from the GEN set.

The second iteration identifies instructions 10-7 as critical because the registers defined by those instructions are in the *GEN* vector. Each selected instruction deletes the *uses* from *GEN*, and immediately adds back the *defs*. For example, instruction 9, deletes and adds register $\$s4$ to bit vector *GEN*. The analysis continues without any other additions. The analysis terminates when no additional instructions are identified as critical.

III. Evaluation

In this chapter we use three benchmarks, *dct, sobel*, and *blackscholes*, to validate the compilation analysis pass and to evaluate the following metrics: 1) the resiliency of the applications when critical instructions are protected from errors, and 2) the impact of compiler and programmer optimizations to the number of protected instructions.

Sobel is a high-pass filter for edge detection in images, which applies 3x3 pixel block filters to produce the pixel values of the output image. Discrete Cosine Transformation (DCT) is a module of video compression kernels, which transforms a block of 8x8 image pixels to a block of 8x8 frequency coefficients. Blackscholes is a benchmark of the PARSEC suite [1]. It implements a mathematical model for a market of derivatives, which calculates the buying and selling of assets so as to reduce the financial risk.

A. *Application Resiliency*

To evaluate the resiliency offered by protecting critical instructions at the hardware level we use GemFI [9], a fault injection tool that operates on top of Gem5 [2]. We exploit GemFI features which allow us to inject faults at different stages of the CPU pipeline which emulates the x86 Instruction Set Architecture. In the fetch stage, a fault corrupts a single bit of the instruction being fetched. In the decoding stage, the selection of registers is corrupted so that the instruction in question reads from or writes to a different register. In the execution stage, faults corrupt a single bit of the computed result value. Finally, faults in the memory stage corrupt a single bit of the value transferred from/to memory.

The number of fault injection experiments for each application and the type and target of faults are generated statistically to achieve a 99% confidence level and 1% error margin. Based on these simulation campaigns, we statistically estimate how each type of fault impacts the application, using the following categorization: 1) **Program Failure**: The application failed to terminate normally, for example decoding a corrupted opcode could result to an *Illegal Instruction* violation. 2) **Program Corruption**: The application succeeds to terminate, however the result is not acceptable by the end user. 3) **Correct**: The application produces a result which is acceptable by the end user however it is not exactly the same as an error-free execution. 4) **Bitwise exact**: The execution resulted at the exact same output as an error-free execution. 5) **Protected**: The fault corrupted a critical instruction but it was corrected by the hardware. The

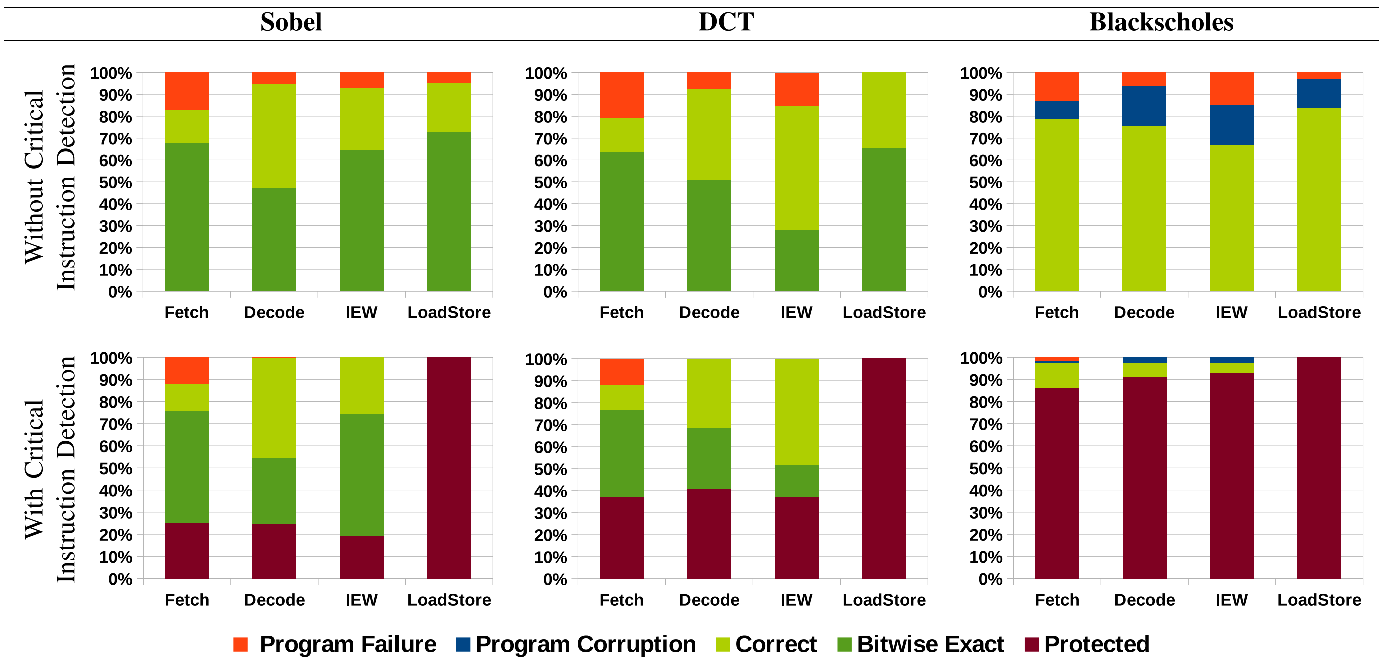


Figure 2: Application resiliency when fault injecting different architectural components with/without critical instruction protection

end result is the same as an error-free execution.

For each benchmark we opt to execute 2 campaigns, the first one does not protect critical instructions whereas the second one does. By doing so we can see the additional resiliency offered by identifying critical instructions.

Figure 2 depicts the results of the fault injection campaigns. Protecting critical instructions offers significant application resiliency since in most pipeline stages program failures are completely eliminated. In the Fetch stage, failures are not eliminated. Protecting only critical instructions is not sufficient. x86 instruction set has variable length instructions. Should a fault corrupt the opcode of the fetched instruction, it may result in decoding another type of instruction, with an opcode length different than that of the correct one. In such a case the binary alignment is corrupted, which results to a program failure.

Interestingly, injecting faults in the decoding stage does not cause any failures when critical instructions are protected. This partially correlates with criticality information. If an error takes place during the decoding stage, it may corrupt the selection of a read register used by the instruction. In turn, if this register does not store any address the fault will quite probably not manifest as a program failure. On the other hand if the selection corrupts the selection of a destination register, and the new faulty destination does contain a memory location, the location will be overwritten by the faulty instruction. Therefore an upcoming instruction which uses this register will fail. In reality though, the fault does not manifest during the execution of this instruction. Consequently it cannot be corrected by the hardware.

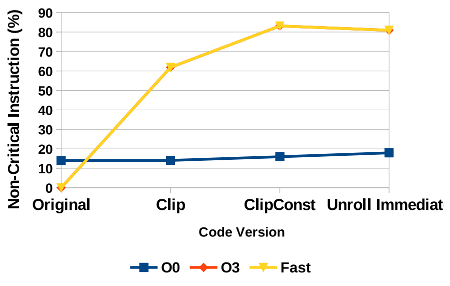
Finally errors manifested during load store instructions were all protected by the hardware, since they explicitly impact address calculation computations.

B. *Optimization effects on Critical Instructions*

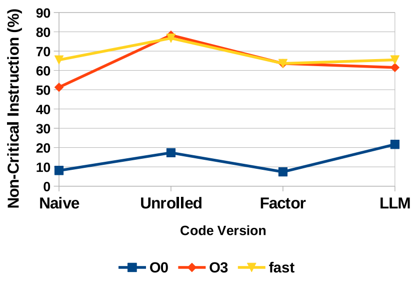
Our goal is to evaluate the correlation of manual and compiler optimizations to the number of critical instructions. Our compile time analysis identifies critical instructions in the application code only and not in the external libraries. We limit the extent of dependencies to external libraries to reduce the number of non-analyzed instructions for the applications under test.

Each benchmark is analyzed under a variety of compiler optimization scenarios. We gradually increase the extent of manual optimizations and, thus, programmer's effort to optimize the code. Each version is compiled three times with the extended version of the LLVM: once with compiler optimizations turned off *-O0*, once with the optimizations turned on *-O3* and once using the *-O3 fast-math* flags.

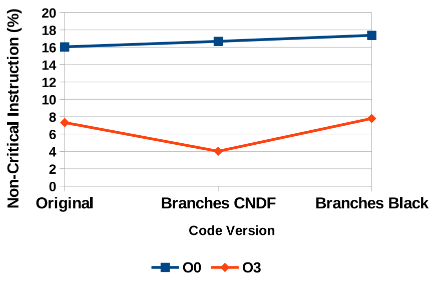
Figure 3 shows the effects of these compiler optimizations on the number of critical intructions. The x-axis shows different versions of a benchmark with incremental manual optimization effort as we move to the right of the x-axis. All version are compiled with different compiler optimization flags (*O0, O3, O3 –fast-math*). In the case of blackscholes we did not use the *–fast-math* option since it results to erroneours executions. The y-axis shows the percentage of non-critical instructions.



(a) Sobel



(b) DCT



(c) Blackscholes

Figure 3: Percentage of critical instructions for different versions of each benchmark when compiled with different compiler optimizations

Compiler and manual optimizations significantly reduce the number of critical instructions. In the case of *Sobel* in the *original* version has only 0% non-critical instructions because the application uses branches within a clipping function. The data used in the clipping function are computed by the previous statements. Our analysis detects such a dependency between the statements and identifies them as critical. The other versions perform the clipping function using masking operations, therefore the percentage of non-critical instructions is increased.

C. *Instruction Set Characterization*

Complex Instruction Set Computers (CISC) often break a more complex instruction to several low level operations. For example, a single CISC instruction may load from the memory, perform an arithmetic operation and store the result back to the memory. The difference compared with RISC architectures is that the latter uses a uniform instruction length for all instructions and employs strictly separate load/store instructions. For example, in Intel's x86, the best known example of CISC architecture, most instructions have one or more input operands that they operate on. The majority of instructions may operate on registers and memory locations.

An architecture such as x86 tends to categorize more instructions as critical compared with a RISC architecture. Any instruction which incudes memory addressing is considered by the analysis as obvious critical. This is because these instructions are translated during the decoding stage to multiple micro-operations. Some of the micro-operations load or store values to the memory, therefore they should be protected by the hardware since they calculate memory addresses. The analysis identifies the entire instruction as critical and effectively all the micro-operations are protected by the hardware. This restriction imposed by the instruction set may result to identifying more instructions as critical than a RISC implementation of the same source code.

IV. Related Work

In [7] the authors categorize instructions in classes, depending on their expected behavior under the presence of transient faults. Instructions with negative impact on the application output are duplicated by the compiler. The application resiliency is studied in [15] when using some sort of protection on control flow instructions whereas in [14], branch instructions are replicated to guarantee correct execution. Although replication of instructions is considered as a fault tolerance method [11], when operating below nominal V_{dd} values, replicating the same code block under the same circumstances will deterministically result to the same faulty behavior. Therefore, replicating an instruction will not guarantee correct execution when facing timing violations since both instructions will probably face corruptions. Multimedia workloads, which are inherently error tolerant in errors are analyzed in detail in [3]. Based on their observations the authors address common manufacturing defects. In [10] the authors use Dynamic Dependence Graphs (DDG) to identify critical instructions. During static analysis instructions affecting critical instructions are also considered as critical. These methods are input dependent, therefore these approaches do not guarantee identification of all critical instructions. In [4] a profiling-guided static program analysis technique and runtime approach is presented. On compilation instructions are classified as static critical and non-static critical: the static critical instructions are further classified into likely critical and likely non-critical instructions.

All these approaches focus on error coverage and error resiliency. In our work we study the correlation of application resiliency with compiler or hand-made code optimizations. Our goal is to activate hardware error detection and correction mechanisms only when application failure is expected. The remaining errors are ignored and allowed to surface to the application level. Application failure is expected when errors corrupt critical instructions. Using a compiler analysis we identify such instructions and we try to reduce their numbers using compiler optimizations.

V. Conclusions

We introduce a compiler analysis technique on the *x86* instruction set which identifies critical instructions. Such instructions are those which perform pointer arithmetic or control flow. The remaining instructions are non-critical. Using GemFI, a fault injection tool we simulated a non-reliable execution environment. In such an environment errors occur at the different pipeline stages. We quantify the extra resiliency offered by protecting a subset of the total number of instruction in three benchmarks, *Sobel, DCT, Blackscholes* is quantified.

The results indicate that protecting a subset of the instructions certainly provides extra fault tolerance against program failures. We should mention that all failures of the protected version are observed when errors are injected during the fetch stage. The fetch stage is vulnerable to faults regardless the context of the instructions being processed at that point. Therefore the entire stage should be protected. Moreover, compiler optimizations in general significantly reduce the number of critical instructions. Moreover, manual code optimizations decrease even more the number of critical instructions. Although in the context of this paper we do not study the performance and power overhead of protecting the instructions, we qualitatively assume that the less the protected instructions the less the overhead.

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[2] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[3] M. A. Breuer. Multi-media applications and imprecise computation. In *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, pages 2–7, Piscataway, NJ, USA, Aug. 2005. IEEE Press.

[4] J. Cong and K. Gururaj. Assuring application-level correctness against soft errors. In *Computer-Aided Design, Proceedings of the International Conference on*, ICCAD '11, pages 150–157, Piscataway, NJ, USA, 2011. IEEE Press.

[5] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18, Piscataway, NJ, USA, Dec. 2003. IEEE, IEEE Press.

[6] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society.

[7] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. *SIGPLAN Not.*, 45(3):385–396, Mar. 2010.

[8] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, Piscataway, NJ, USA, Mar. 2004. IEEE Press.

[9] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas. Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 622–629, Washington, DC, USA, 2014. IEEE Computer Society.

[10] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Automated derivation of application-aware error detectors using static analysis. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, pages 211–216, Piscataway, NJ, USA, July 2007. IEEE Press.

[11] A. Rahimi, L. Benini, and R. K. Gupta. Analysis of Instruction-level Vulnerability to Dynamic Voltage and Temperature Variations. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 1102–1105, Piscataway, NJ, USA, Mar. 2012. IEEE Press.

[12] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. A Variability-aware OpenMP Environment for Efficient Execution of Accuracy-configurable Computation on shared-FPU Processor Clusters. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 35:1–35:10, Piscataway, NJ, USA, 2013. IEEE Press.

[13] B. Randell, P. Lee, and P. C. Treleaven. Reliability Issues in Computing System Design. *ACM Comput. Surv.*, 10(2):123–165, June 1978.

[14] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin. Efficient Fault Tolerance in Multi-media Applications Through Selective Instruction Replication. In *Radiation Effects and Fault Tolerance in Nanometer Technologies, Proceedings of the 2008 Workshop on*, WREFT '08, pages 339–346, New York, NY, USA, 2008. ACM.

[15] D. 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 *Workload Characterization, 2006 IEEE International Symposium on*, pages 142–149, Piscataway, NJ, USA, Oct. 2006. IEEE Press.